

CS378 - A Formal Model of The Java Virtual Machine

<http://www.cs.utexas.edu/users/moore/classes/cs378-jvm/>

Semester	Spring, 2007
Unique Id	55095
Instructor	J Strother Moore
Email	moore@cs.utexas.edu
Office	TAY 4.140A
Office Hours	TT 2:30–3:30
TA	Alex Spiridonov - alexsp@cs.utexas.edu

Abstract

In this course we'll study Sun's Java Virtual Machine – but in a very unusual way: we'll build a model of it in a functional programming language and we'll use an automatic theorem prover to reason about the model.

So you'll learn about four different things:

- how to build a mathematical model of something complicated like the JVM,
- how to program in a simple functional language,
- how to reason about such models, and
- how to use a powerful mechanized reasoning tool.

This course is a mixture of

CS313K Logic, Sets and Functions

CS336 Analysis of Programs

CS343 Artificial Intelligence

CS352 Computer Systems Architecture

CS375 Compilers

CS321H Functional and
Symbolic Programming

Outline

- Formal Methods
- ACL2 Background
- Elementary Examples
- A Closer Look at a Big JVM Model

Formal Methods ...

is the use of formal mathematics to model hardware and software and the use of mechanized mathematical tools to reason about those models.

Formal Methods ... Why Bother?

Formal Methods ... Why Bother?

- formal models often clarify the designers' intentions
- formal models can be used as simulation engines
- today's digital artifacts are so complicated, people cannot reason about

them accurately without mechanized help

- verified properties of a module often provide the best documentation of its intended behavior
- verified systems can be changed with only incremental verification costs

Boyer-Moore Project

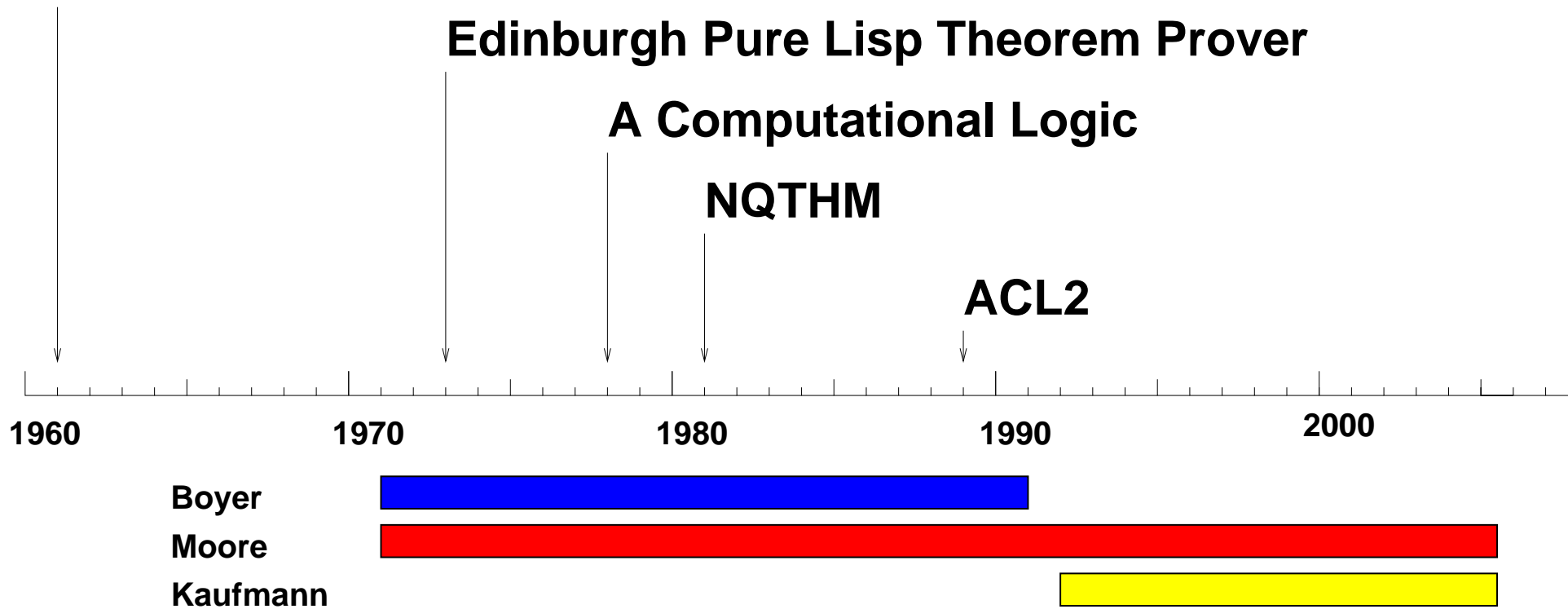
McCarthy's "Theory of Computation"

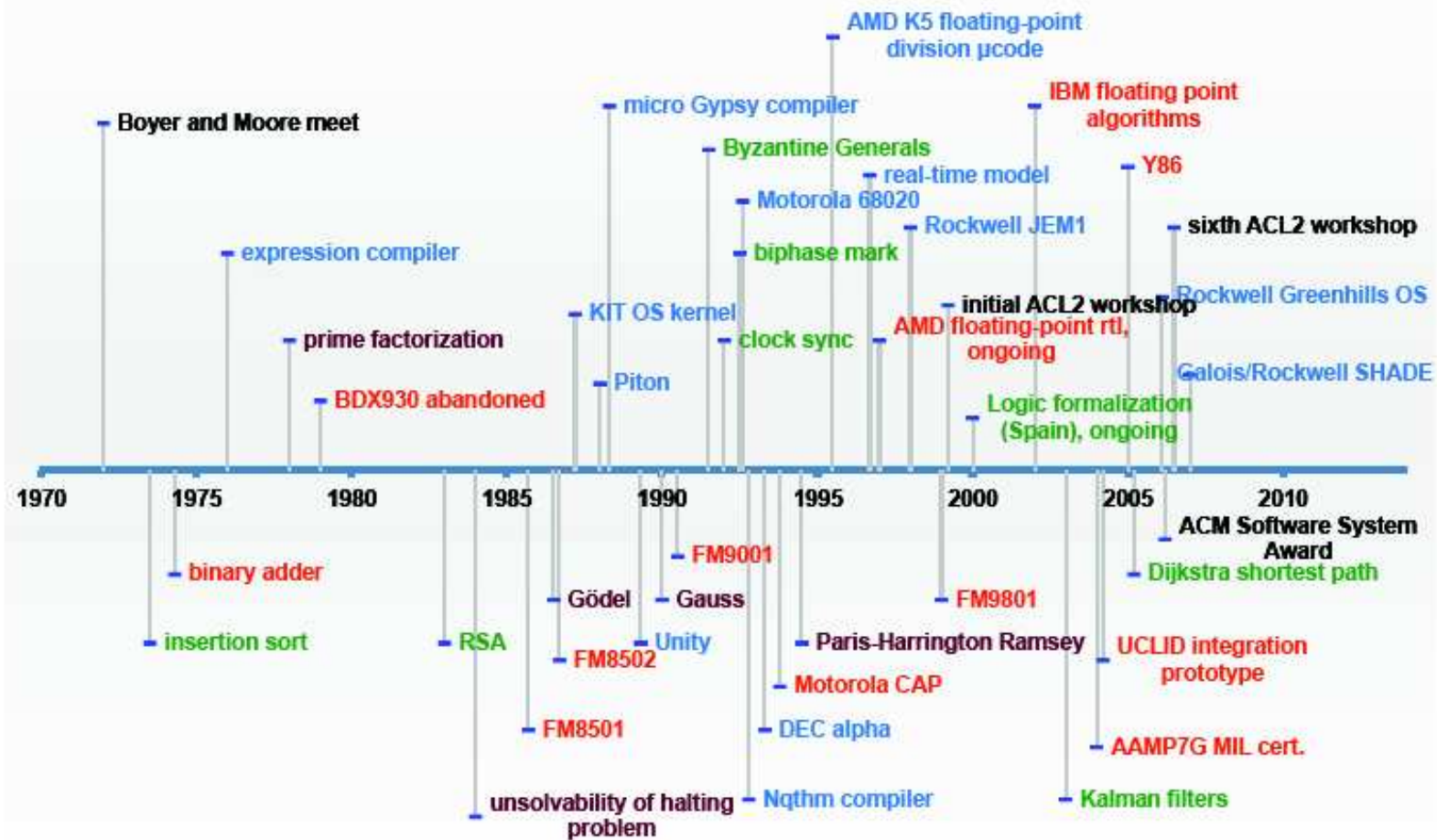
Edinburgh Pure Lisp Theorem Prover

A Computational Logic

NQTHM

ACL2





ACL2 =

A Computational **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).

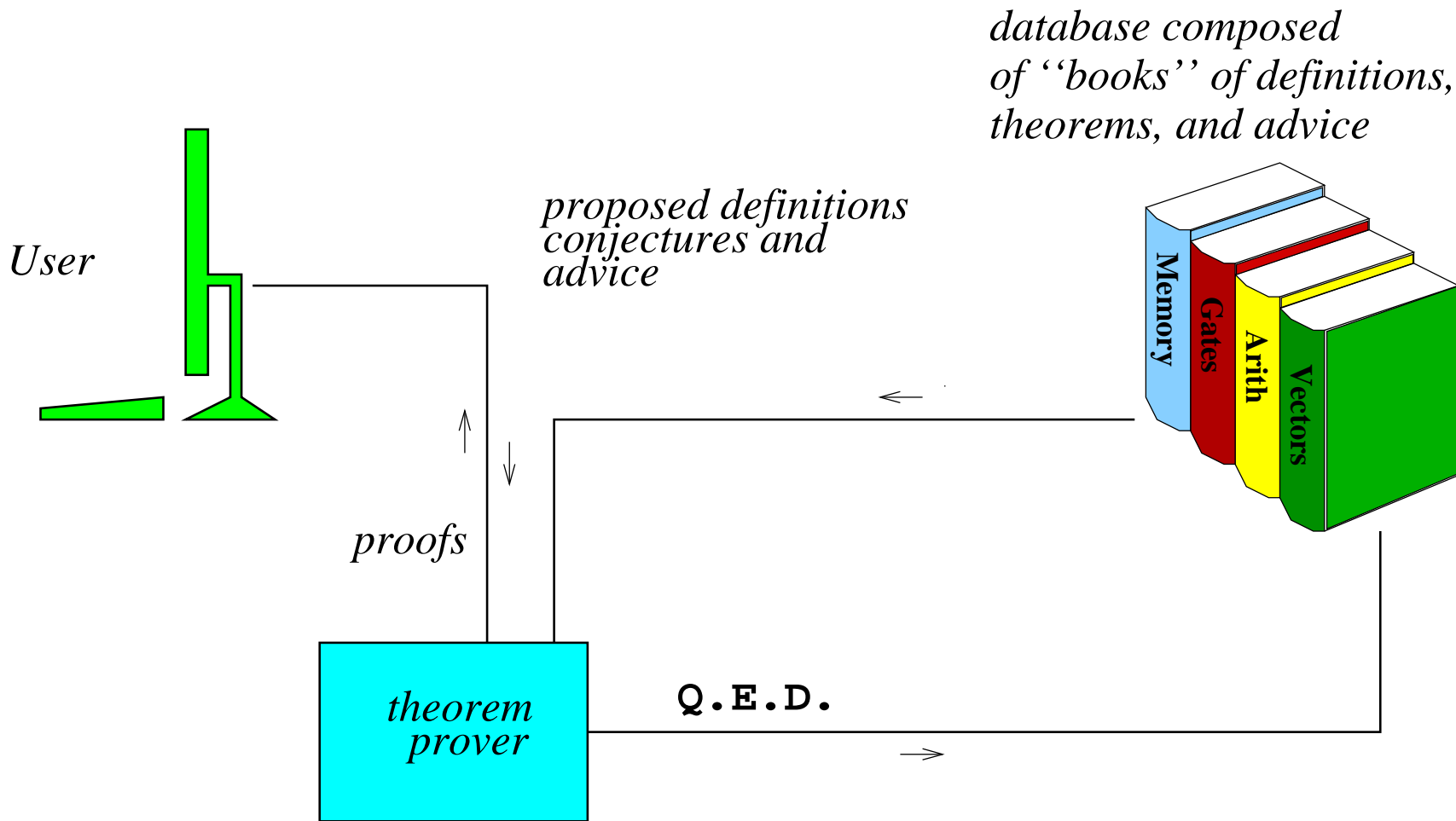
All ACL2 functions are *total* (always terminate on all arguments).

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

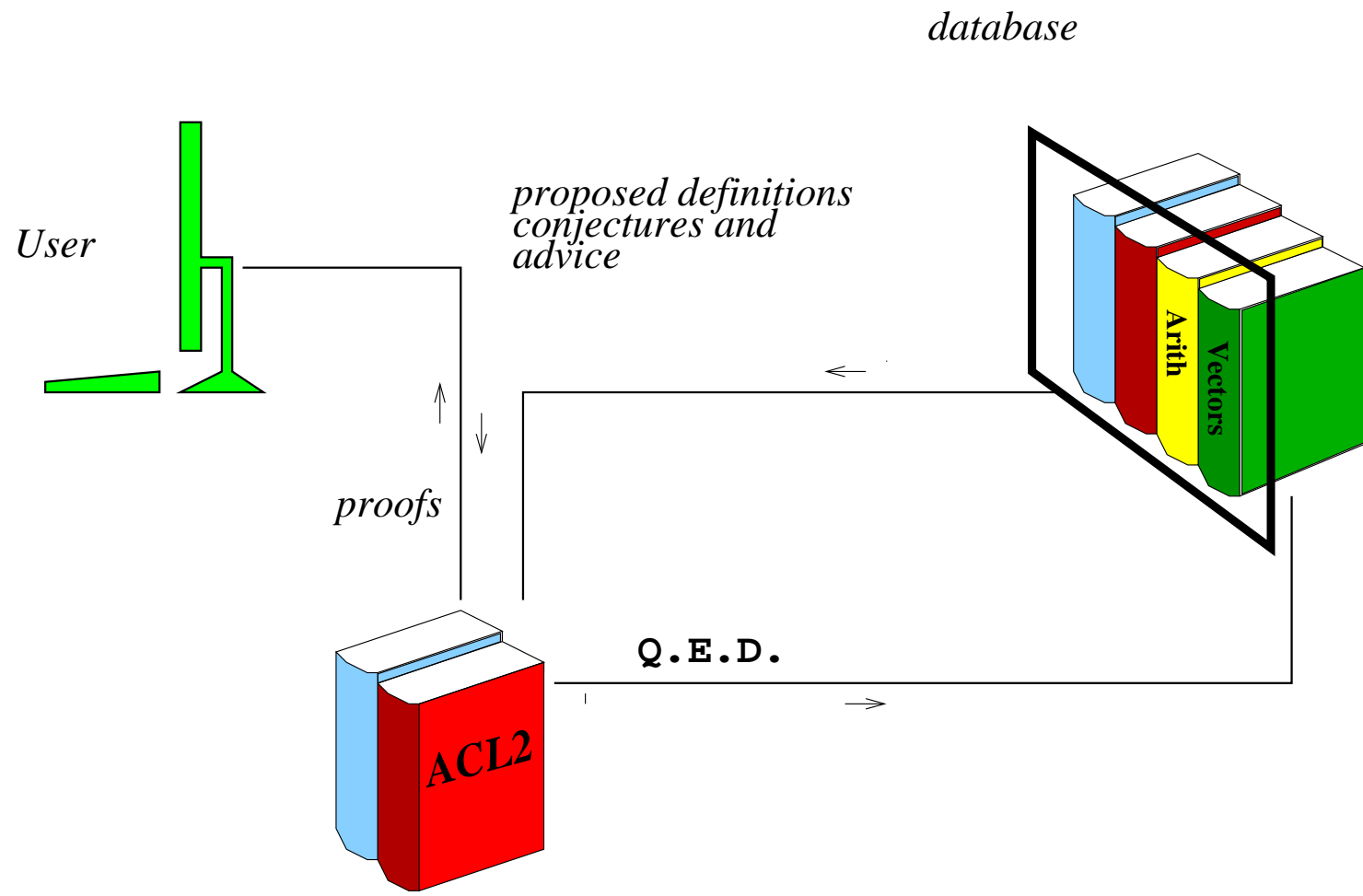
ACL2 is *quantifier-free* – but has the expressive power of full first-order logic thanks to Skolemization.

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.



ACL2 is coded in 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 Stop Reload

http://awards.acm.org/software_system/ Search

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:

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 Stop Reload

http://awards.acm.org/software_system/ Search

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](#)

About ACL2s

ACL2 has been compared to a finely tuned high-powered race car.

The IDE preferred by experts is Emacs.

When novices try to drive ACL2, they often crash and burn.

ACL2s is an experimental ACL2 “sedan”

with an Eclipse front end.

We'll use ACL2s in here.

Alex is the expert.

I've never used ACL2s for real and so I'll be learning too.

Lisp Syntax

$$\begin{aligned} \langle term \rangle := & \langle var \rangle \mid \\ & ' \langle const \rangle \mid \\ & (\langle fn \rangle \langle term \rangle_1 \\ & \quad \dots \\ & \quad \langle term \rangle_n) \end{aligned}$$
$$\begin{aligned} \langle const \rangle := & \langle number \rangle \mid \langle char \rangle \mid \\ & \langle string \rangle \mid \langle symbol \rangle \mid \\ & \langle pair \rangle \end{aligned}$$

Example Constants

123, 22/7

\#Newline, #\A, #\a

"Hello world!"

x, world, pt, PT, Pt, :pc

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

Example Terms

`(cons (car x) rest)`

e.g., `cons(car(x), $rest$)`

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

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

Packages

Logically, each symbol has two components: its “package” and its “name”.

For example `ACL2::PUSH` is a symbol: its package is "ACL2" and its name is "PUSH".

There is always a “default package” and initially it is "ACL2".

When you write a symbol without an explicit package, the default package is used.

Thus, `PUSH` is the same symbol as `ACL2::PUSH` – when the default package is "ACL2".

When you define a new package, you can “import” certain symbols into it.

Suppose "ACL2" is current package.

Suppose we define "M1" to be a new package.

Suppose we import CONS into "M1" but we do not import PUSH.

Suppose we select "M1" to be the default package.

Then PUSH is the same as M1::PUSH.

M1::PUSH is different from ACL2::PUSH!

But CONS is the same as ACL2::CONS.

Thus, even if `ACL2::PUSH` is already defined, we can define `M1::PUSH` – because it is a different symbol!

And since we're in the "M1" package, we can write `PUSH` to mean `M1::PUSH` and `CONS` to mean `ACL2::CONS`.

So

```
(defun push (x y) (cons x y))
```

About Keywords

Keywords are just symbols in the "KEYWORD" package.

`KEYWORD::NAME` is a keyword.

It can always be shortened to `:NAME`.

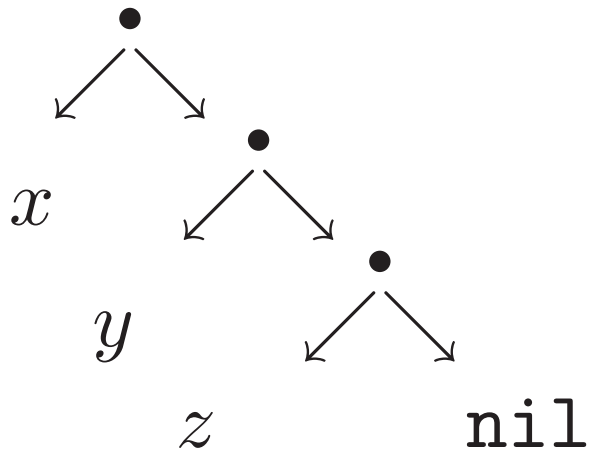
Keywords cannot be used as function names.

They cannot be used as variables.

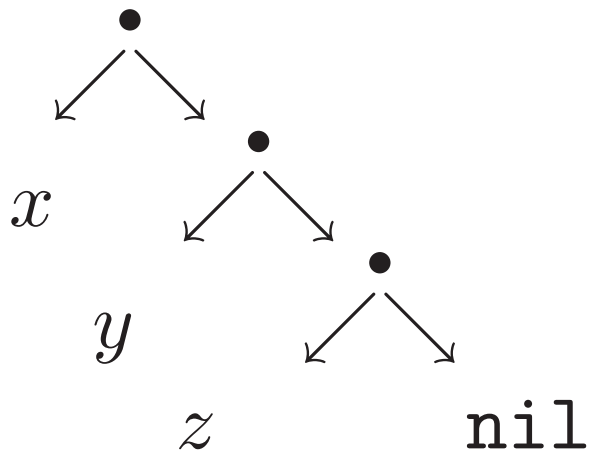
They evaluate to themselves.

About Pairs

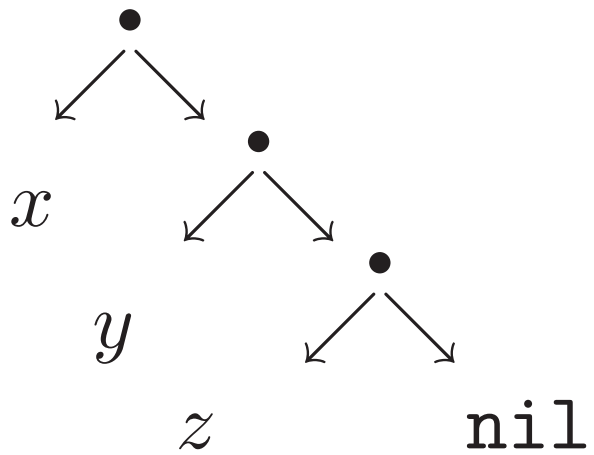
$\langle x, \langle y, \langle z, \text{nil} \rangle \rangle \rangle$



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

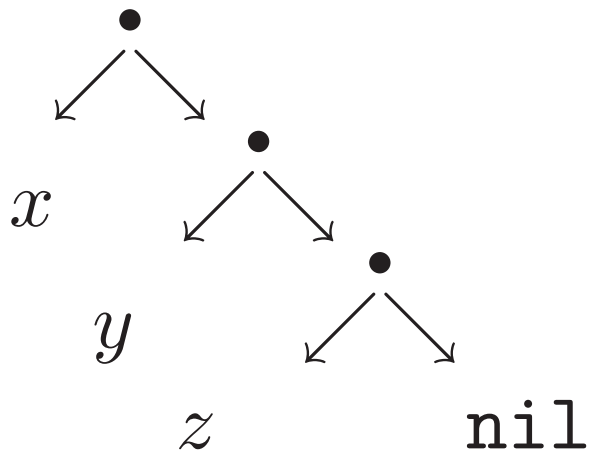


$(x \ . \ (y \ . \ (z \ . \ nil)))$



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

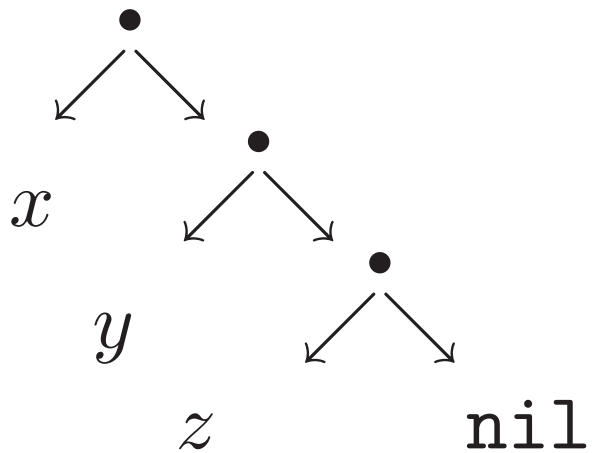
$(x \ . \ (y \ . \ (z \ \ \ \)))$; *may erase* ‘‘. nil’’



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

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

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

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

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

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

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

123

0123

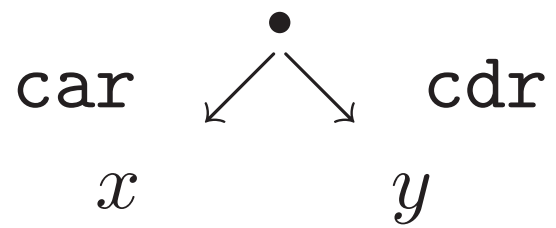
+123

01111011₂

0x7B

About Pairs

`(cons x y)`



`← (consp ●) = t`

Data Types

ACL2 supports five disjoint data types:

- numbers (integers, non-integer rationals, complex rationals)
- characters
- strings
- symbols (including keywords)
- 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)) ⇒ t`

`(consp 1) ⇒ nil`

`(consp nil) ⇒ nil`

A Few Axioms

$t \neq \text{nil}$

$x = \text{nil} \rightarrow (\text{if } x \ y \ z) = z$

$x \neq \text{nil} \rightarrow (\text{if } x \ y \ z) = y$

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

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

`(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

$$(\text{not } x) = (\text{if } x \text{ nil } t)$$

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))
```

Recursive Definition

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

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

```
(equal (app (app a b) c)
       (app a (app b c)))
```

(equal (app (app a b) c)
 (app a (app b c))))

Proof: by induction on a.

(equal (app (app a b) c)
 (app a (app b c))))

Proof: by induction on a.

Base Case: (endp a).

(equal (app (app a b) c)
 (app a (app b c))))

```
(equal (app (app a b) c)
       (app a (app b c)))
```

Proof: by induction on a.

Base Case: (endp a).

```
(equal (app b c)
       (app a (app b c)))
```

```
(equal (app (app a b) c)
       (app a (app b c)))
```

Proof: by induction on a.

Base Case: (endp a).

```
(equal (app b c)
       (app a (app b c)))
```

```
(equal (app (app a b) c)
       (app a (app b c)))
```

Proof: by induction on a.

Base Case: (endp a).

```
(equal (app b c)
       (app b c))
```

(equal (app (app a b) c)
 (app a (app b c)))

Proof: by induction on a.

Base Case: (endp a).

(equal (app b c)
 (app b c))

```
(equal (app (app a b) c)
       (app a (app b c)))
```

Proof: by induction on a.

Base Case: (endp a).

T

(equal (app (app a b) c)
 (app a (app b c))))

Proof: by induction on a.

Induction Step: (not (endp a)).

(equal (app (app a b) c)
 (app a (app b c))))

```
(equal (app (app a b) c)
       (app a (app b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

```
(equal (app (cons (car a)
                (app (cdr a) b)) c)
       (app a (app b c)))
```

```
(equal (app (app a b) c)
       (app a (app b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

```
(equal (app (cons (car a)
           (app (cdr a) b)) c)
       (app a (app b c)))
```

```
(equal (app (app a b) c)
       (app a (app b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

```
(equal (cons (car a)
       (app (app (cdr a) b) c))
       (app a (app b c)))
```

```
(equal (app (app a b) c)
       (app a (app b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

```
(equal (cons (car a)
             (app (app (cdr a) b) c))
       (app a (app b c)))
```

```
(equal (app (app a b) c)
       (app a (app b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

```
(equal (cons (car a)
            (app (app (cdr a) b) c))
       (cons (car a)
          (app (cdr a) (app b c))))
```

```
(equal (app (app a b) c)
       (app a (app b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

```
(equal (cons (car a)
       (app (app (cdr a) b) c))
       (cons (car a)
        (app (cdr a) (app b c))))
```

```
(equal (app (app a b) c)
       (app a (app b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

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

```
(equal (app (app a b) c)
       (app a (app b c)))
```

Proof: by induction on a.

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

```
(equal (app (app a b) c)
       (app a (app b c)))
```

Proof: by induction on a.

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

```
(equal (app (app a b) c)
       (app a (app b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

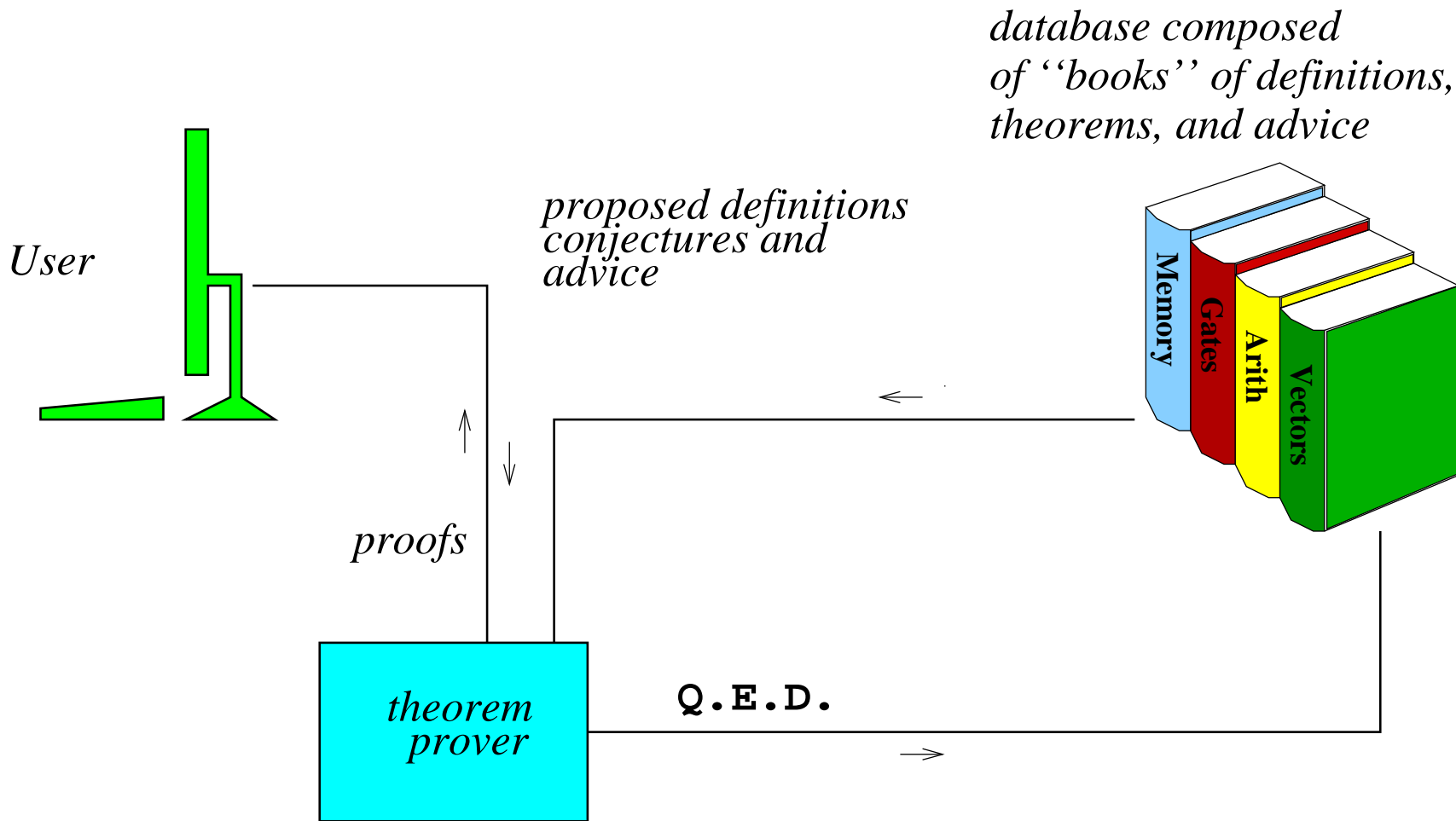
T

(equal (app (app a b) c)
 (app a (app b c))))

Proof: by induction on a.

Q.E.D.

Demo 1

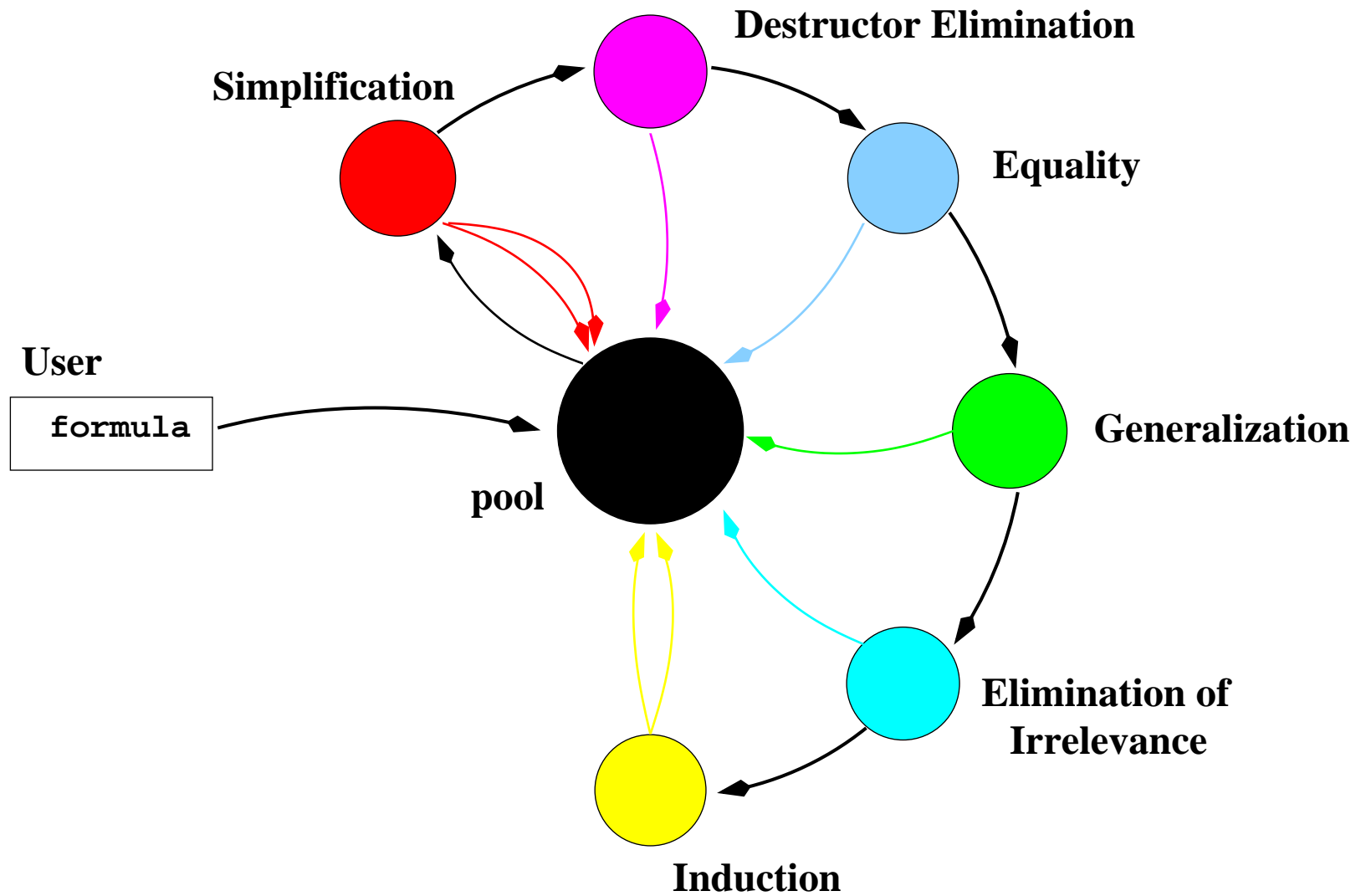


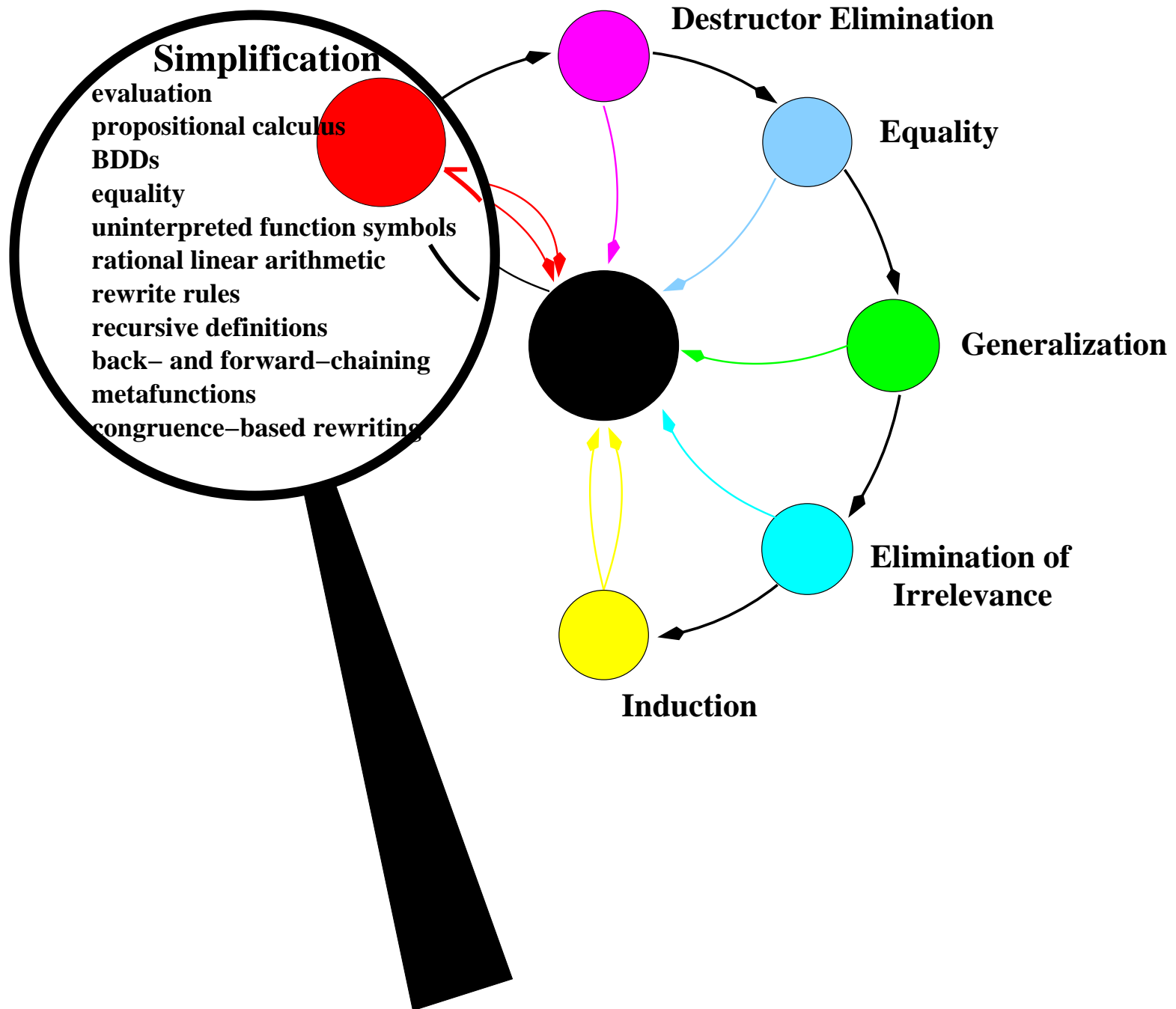
database composed of "books" of definitions, theorems, and advice

proposed definitions conjectures and advice

proofs

Q.E.D.





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 data types (except floats), multi-threading, dynamic class loading,

class initialization and synchronization via
monitors.

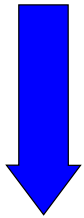
We have translated the entire Sun CLDC API library implementation into our representation with 672 methods in 87 classes. We provide implementations for 21 out of 41 native APIs that appear in Sun's CLDC API library.

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.

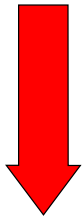
This work is supported by a gift from Sun Microsystems.

.java



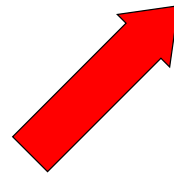
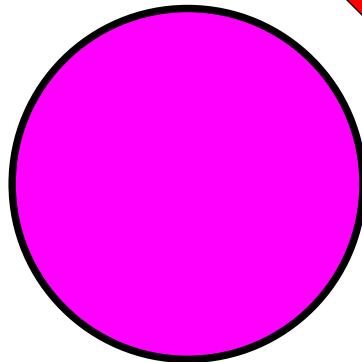
javac

.class



jvm2acl2

.lisp



Theorems

“pi(246)=123”



“pi(n)=n/2”

Demo 2

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>