Little (but Hard) Theorems About Big Systems: Some Case Studies

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

Marktoberdorf Summer School 2004

Lecture 1

Thesis

The size and complexity of today's artifacts make it hard to prove theorems about them.

This is not an argument *against* formal methods.

It is an argument *for* formal methods!

We focus on proving theorems of interest to the industrial designers of commercial hardware and software artifacts.

These theorems *do not* ensure every interesting property of the design. In this sense they are “little” theorems.

But they deal with very precise models of extremely complex designs. So their proofs are often “big.”

How Important Are These Theorems?

An elusive circuitry error is causing a chip used in millions of computers to generate inaccurate results

— *NY Times, "Circuit Flaw Causes Pentium Chip to Miscalculate, Intel Admits," Nov 11, 1994*

Intel Corp. last week took a $475 million write-off to cover costs associated with the divide bug in the Pentium microprocessor's floating-point unit — *EE Times, Jan 23, 1995*

In 1995, we proved that FDIV works correctly (IEEE 754 compliance) on the AMD K5 – months before the K5 was fabricated.

All elementary arithmetic ops in the Athlon’s FPU were verified before fab.

Four RTL-level bugs were found and fixed.

AMD Austin has 5 people using theorem proving on AMD designs.

One person keeps up with the FPU design team.

AMD's design process (for FPUs) has changed!

RTL-level circuit descriptions of modules in the FPU are routinely verified (mechanically) to be IEEE compliant.

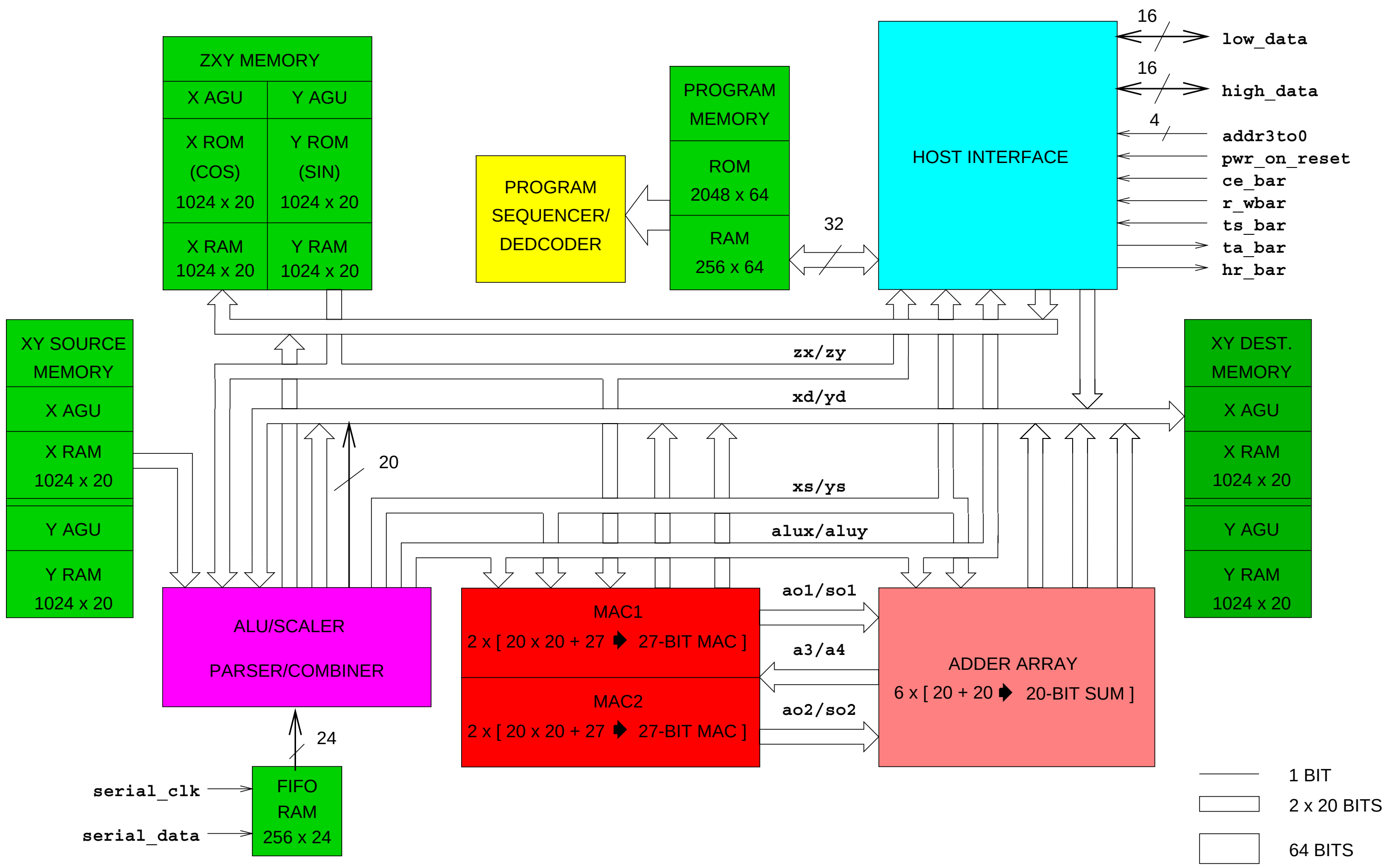
"AMD will never again produce an unverified floating-point unit."

How Big is “Big”?

AMD work involves thousands of lemmas.

Models involve 4000 mutually recursive functions.

Motorola CAP Digital Signal Processor



Our formal model of the Motorola CAP digital signal processor was bit- and cycle-accurate.

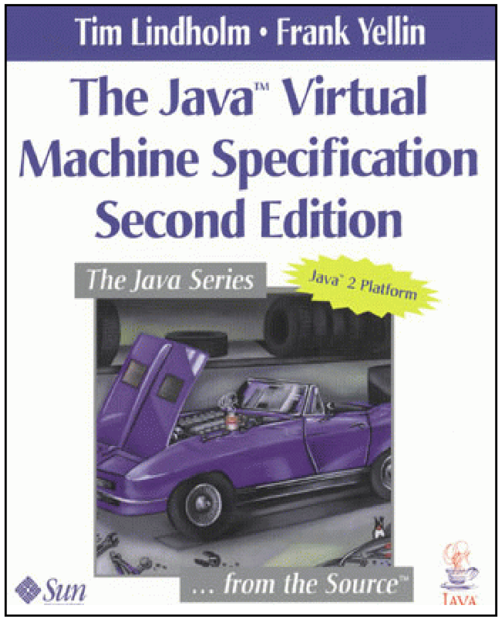
We proved that the microarchitecture implemented the specified microcode engine.

We proved microcode correct.

Some proofs involved subgoals of 25 MB.

Our formal model of aJile System's silicon JVM was bit- and cycle-accurate and

was used in place of the traditional C-simulator for requirements testing


Tim Lindholm • Frank Yellin

The Java™ Virtual Machine Specification Second Edition

The Java Series

Java™ 2 Platform

Sun

… from the Source™

Java

```
; JVM in ACL2


(defun make−state (tt hp ct)
   ...)

(defun step (th s)
   ...)

(defun run (sched s)
  (if (endp sched)
      s
    (run
       (cdr sched)
       (step (car sched) s))))
```

Our model of the JVM executes most J2ME Java programs and includes

- almost all bytecodes
- multi-threading
- synchronization
- class loading
- initialization, and
- byte-code verification.

The model is 10K lines of formalism (about 175 pages).

Additionally, about 200MB of formal constants are involved (representing the entire Sun CLDC API library)

How Do You Construct Big Proofs?

(1) Learn to express your conjectures formally.

How Do You Construct Big Proofs?

(1) Learn to express your conjectures formally.

(2) Learn to prove little theorems formally.

How Do You Construct Big Proofs?

(1) Learn to express your conjectures formally.

(2) Learn to prove little theorems formally and simply.

How Do You Construct Big Proofs?

(1) Learn to express your conjectures formally.

(2) Learn to prove little theorems formally and simply.

(3) Learn how to get a machine to do the rote work.

How Do You Construct Big Proofs?

(1) Learn to express your conjectures formally.

(2) Learn to prove little theorems formally and simply.

(3) Learn how to get a machine to do the rote work.

(4) Practice, practice, practice . . .

The Plan for These Lectures

- a trivial formal system (1 lecture and homework)
- how to prove little theorems (2 lectures and homework)
- demo with (slightly) larger problems (1 lecture)

- proof of an algorithm on a Motorola DSP (1 lecture)
- modeling the JVM and verifying JVM bytecoded methods (1 lecture)

How To Get the Most Out of This

Read the orange booklet associated with these lectures.

The paper in the booklet is titled “How to Prove Theorems Formally” and gives lots of advice.

Do the $\sim$ 30 exercises in the booklet!

Check your answers against mine on the web.

Or, use a mechanical theorem prover.

A Simple Formal System

We will use a tiny subset of pure Lisp, both as a programming language and a first-order mathematical logic.

The subset is from

A Computational Logic for
Applicative Common Lisp

or

ACL2

Poll

How many have heard of Lisp (or Scheme)?

How many have written at least one Lisp program?

How many *know* Lisp?

About Lisp and ACL2

Lisp is *untyped*.

Lisp is *strict* (not lazy).

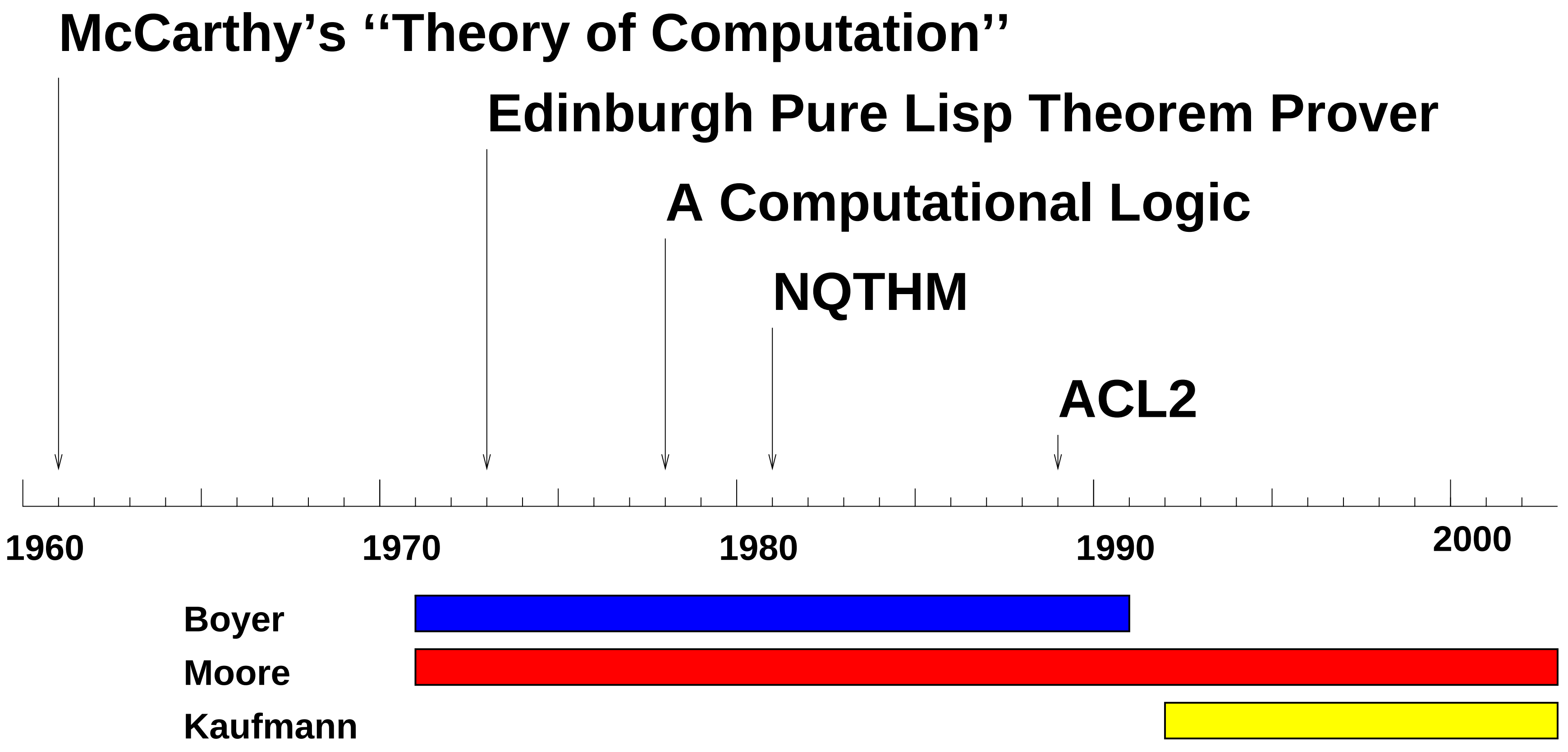
ACL2 is a subset of Lisp.

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

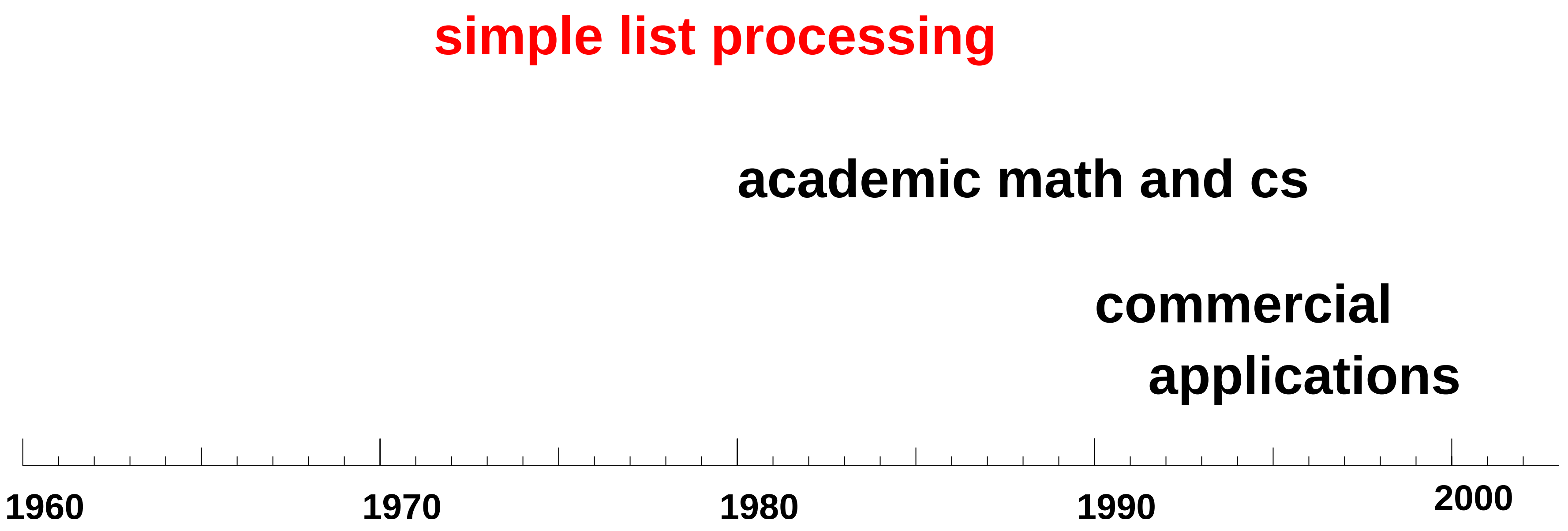
ACL2 is *applicative* (functional).

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

The Boyer-Moore Project



The Boyer-Moore Project



Lisp Syntax

$$\begin{aligned} < term > := \;& < var > \mid \\ & \texttt{(quote } < const > \texttt{)} \mid \\ & \texttt{(}< fn > \quad < term >_1 \\ & \qquad\qquad\quad \ldots \\ & \qquad\qquad\quad < term >_n\texttt{)} \end{aligned}$$

Example Terms

```
(CONS (CAR X) REST))
```

e.g., $\mathcal{CONS}(\mathcal{CAR}(x), rest)$

```
(ASSOC-EQUAL KEY ASSOCIATION-LIST)

(assoc-equal key association-list)

(assoc-equal key
             association-list)
```

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
- accessing the constituents
- recognizing instances of each type
- other expected operations (e.g., addition of numbers)

Every instance of each type can be printed.

In these lectures we won't have much use for numbers (other than the integers), characters, strings, or many symbols.

Case will be irrelevant here, e.g., `NIL`, `Nil` and `nil` are the same.

Here are some printed instances . . .

```
    65    66  #c(3  2)    #\Space
97        22/7    #\A   #\B
 -33  "ABC"  "Hello, World!"
 T NIL  ABC   COLOR    RED
(65 . RED)  JONES::FN   SMITH::FN
  (#\A . (#\B . (#\C . NIL)))
      ((0 . 1) . (2 . 4))
    ((COLOR . RED) (SIZE . 7))
```

About T and NIL

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

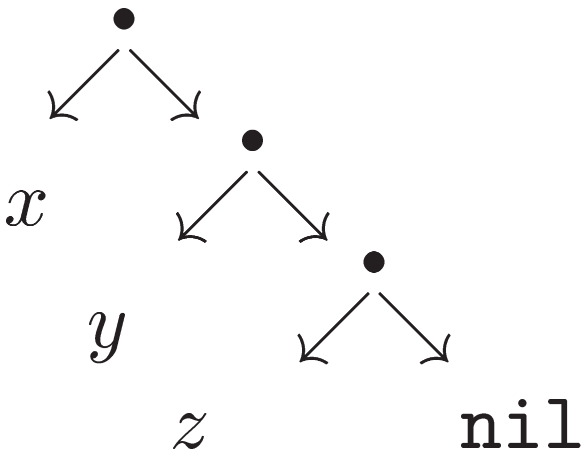
`NIL` is used as the “terminal marker” on nested pairs representing lists.

Informally, “`NIL` is the empty list.”

But `T` and `NIL` are *symbols*!

About Pairs

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



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

About Pairs

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

$$\begin{tikzcd}
& \bullet \arrow[dl] \arrow[dr] & & & \\
x & & \bullet \arrow[dl] \arrow[dr] & & \\
& y & & \bullet \arrow[dl] \arrow[dr] & \\
& & z & & \texttt{nil}
\end{tikzcd}$$

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

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

About Pairs

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

$$\begin{tikzcd}
 & \bullet \arrow[dl] \arrow[dr] & & & \\
x & & \bullet \arrow[dl] \arrow[dr] & & \\
 & y & & \bullet \arrow[dl] \arrow[dr] & \\
 & & z & & \mathtt{nil}
\end{tikzcd}$$

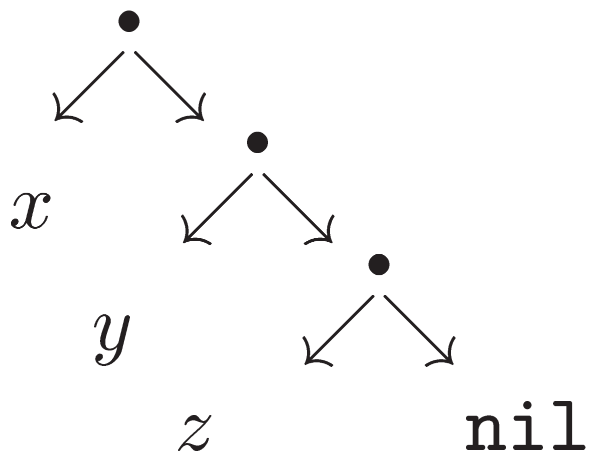
$(x\ \ .\ \ (y\ \ .\ \ (z\ \ .\ \ \mathtt{nil})))$

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

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

About Pairs

$< x, \ < y, \ < z,$ `nil`$>>>$



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

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

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

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

About Pairs

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

$$\begin{tikzcd}
& \bullet \arrow[dl] \arrow[dr] & & & \\
x & & \bullet \arrow[dl] \arrow[dr] & & \\
& y & & \bullet \arrow[dl] \arrow[dr] & \\
& & z & & \mathtt{nil}
\end{tikzcd}$$

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

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

$$
\begin{array}{llllll}
(x & . & (y & . & (z & . \ \texttt{nil}))) \\
(x & . & (y & . & (z & \phantom{. \ \texttt{nil}}))) \\
(x & . & (y & & \ z & \phantom{. \ \texttt{nil})}))\\
(x & & \ y & & \ z & \phantom{. \ \texttt{nil}))})
\end{array}
$$

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

`123`

`0123`

`+123`

`01111011`$_2$

`0x7B`

About Pairs

$<< 1,\ 2 >,\ < 3,\ 4 >>$

$$\begin{tikzcd}
 & & & \bullet \arrow[dll] \arrow[drr] & & & \\
 & \bullet \arrow[dl] \arrow[dr] & & & & \bullet \arrow[dl] \arrow[dr] & \\
1 & & 2 & & 3 & & 4
\end{tikzcd}$$

```
((1 . 2) . (3 . 4))
((1 . 2)   3 . 4 )
```

$$
\begin{aligned}
< term > := \; & < var > \mid \\
& (\texttt{quote} \; < const >) \mid \\
& (< fn > \quad < term >_1 \\
& \qquad\qquad\quad \ldots \\
& \qquad\qquad\quad < term >_n)
\end{aligned}
$$

Every instance of each type can be written as a constant term.

'$const$ is an abbreviation for `(quote` $const$`)`.

About Quote

`(CAR X)` is a term denoting the application of the function symbol `CAR` to the value of the variable symbol `X`.

`'(CAR X)` is a term denoting a list constant.

Which do you mean?

```
(OK OK 'OK '(OK OK 'OK))
```

$\mathcal{OK}(ok, \mathbf{OK}, [\mathbf{OK}, \mathbf{OK}, [\mathbf{QUOTE}, \mathbf{OK}]])$

$\mathcal{F}(x, \mathbf{3}, [\mathbf{3}, \mathbf{3}, [\mathbf{7}, \mathbf{3}]])$

Primitive Functions

- `(cons` x y`)`
  construct the ordered pair $\langle x, y \rangle$

- `(car` x`)`
  left component of x, if x is a pair; `nil` otherwise

- `(cdr` x`)`
  right component of x, if x is a pair; `nil` otherwise

- `(consp` x`)`
  `t` if x is a pair; `nil` otherwise
- `(if` x y z`)`
  z if x is `nil`; y otherwise
- `(equal` x y`)`
  `t` if x is y; `nil` otherwise

Defining Functions

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

(defun and (p q) (if p q nil))

(defun or (p q) (if p p q))

(defun implies (p q)
  (if p (if q t nil) t))

(defun iff (p q)
  (and (implies p q) (implies q p)))
```

We make the conventions

“term”	*term*
`(and p q r)`	`(and p (and q r))`
`(or p q r)`	`(or p (or q r))`
`(+ i j k)`	`(+ i (+ j k))`
`(list x)`	`(cons x nil)`
`(list x y)`	`(cons x (cons y nil))`
`(list x y z)`	`(cons x (cons y (cons z nil)))`

```
(list x y z) (cons x
                (cons y
                  (cons z nil)))
```

Homework

Problem 1.0 If you haven't already, read "How to Prove Theorems Formally!"

Do the exercises in it!

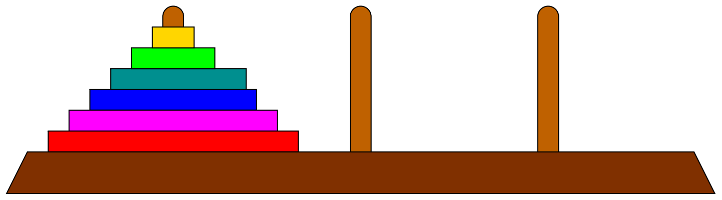
Some of you may want some new exercises. I will not collect or grade these. But if you want something to think about . . .

Problem 1.1. We say e “is a twin” in the true-list x iff e occurs exactly twice in x. Define `(twins x)` so that it returns a list of all the twins in x, with no duplications. *Avoid using arithmetic* in your definition(s)!

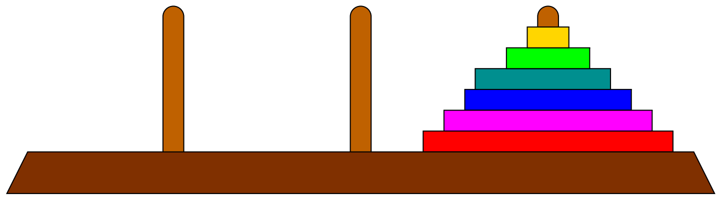
Problem 1.2. Define `(perm x y)` so that it returns `t` or `nil` according to whether the lists `x` and `y` are permutations of one another. You may assume both are true-lists (terminated by `nil`).

Problems 1.3 and 1.4. Skipped.

Problem 1.5. You know the "Towers of Hanoi" puzzle:



start



finish

Define (`hanoi` n) so that it solves the puzzle for an arbitrary natural number n.

(To subtract 1 from `n`, use `(- n 1)`. To test whether `n` is 0, use `(zp n)`.)

Problem 1.6. Formalize what it means for `hanoi` to be "correct." That is, write a term whose value is always true precisely if (`hanoi` n) solves the n-puzzle.

Why Am I Being So Vague?

. . . because a crucial skill you must learn is how to write formulas that “capture” imprecise, informal, intuitive ideas.