(power x 0) == 1

(power x (+ n 1)) == (* (power x n) x)

(power x 0) == 1

(power x (+ (* 2 m) 0)) == (square (power x m))

(power x (+ (* 2 m) 1)) == (* x (square (power x m)))

(all-fours? d) == (= d 4), [d is digit]
(all-fours? (+ (* 10 m) d)) ==

  (and (= d 4) (all-fours? m)), where m != 0

(has-digit? d d) == 1

(has-digit? d d’) == 0, where d differs from d’
(has-digit? (+ (* 10 m) d) d’) ==

  (or (= d d’) (has-digit? m d’)), where m != 0
Bloom’s taxonomy

Cognitive actions:
1. Remember
2. Understand
3. Apply
4. Analyze
5. Evaluate
6. Create
Operational semantics

Cognitive actions:
1. Remember
2. Understand
3. Apply
4. Analyze
5. Evaluate
6. Create
Concrete syntax for Impcore (again)

Definitions and expressions:

$$\text{def ::= } (\text{define } f \ (x_1 \ldots x_n) \ \text{exp}) \ ;; \ "true" \ \text{defs}$$

| (val \ x \ \text{exp}) |
| exp |
| (use filename) \ ;; \ "extended" \ \text{defs} |
| (check-expect \ \text{exp1} \ \text{exp2}) |
| (check-assert \ \text{exp}) |
| (check-error \ \text{exp}) |

$$\text{exp ::= } \text{integer-literal}$$

| \ \text{variable-name} |
| (set \ x \ \text{exp}) |
| (if \ \text{exp1} \ \text{exp2} \ \text{exp3}) |
| (while \ \text{exp1} \ \text{exp2}) |
| (begin \ \text{exp1} \ldots \ \text{expn}) |
| (function-name \ \text{exp1} \ldots \ \text{expn}) |
How to define behaviors inductively

Expressions only

Base cases (plural): numerals, names

Inductive steps: compound forms
  • To determine behavior of a compound form, look at behaviors of its parts
First, simplify the task of definition

What’s different? What’s the same?

\[
x = 3; \quad \text{(set } x 3)\]
\[
\text{while } (i \times i < n) \quad \text{(while } (< (* i i) n)\)
\quad i = i + 1; \quad \text{(set } i (+ i 1))\]

Abstract away gratuitous differences

(See the bones beneath the flesh)
Abstract syntax

Same inductive structure as BNF grammar (related to proof system)

More uniform notation

Good representation in computer

Concrete syntax: sequence of symbols

Abstract syntax: ???
The abstraction is a tree

The abstract-syntax tree (AST):

Exp = LITERAL (Value)
    | VAR (Name)
    | SET (Name name, Exp exp)
    | IFX (Exp cond, Exp true, Exp false)
    | WHILEX (Exp cond, Exp exp)
    | BEGIN (Explist)
    | APPLY (Name name, Explist actuals)

One kind of “application” for both user-defined and primitive functions.
In C, trees are a bit fiddly

typedef struct Exp *Exp;
typedef enum {
    LITERAL, VAR, SET, IFX, WHILEX, BEGIN, APPLY
} Expalt;     /* which alternative is it? */

struct Exp { // only two fields: 'alt' and 'u'!
    Expalt alt;
    union {
        Value literal;
        Name var;
        struct { Name name; Exp exp; } set;
        struct { Exp cond; Exp true; Exp false; } ifx;
        struct { Exp cond; Exp exp; } whilex;
        Explist begin;
        struct { Name name; Explist actuals; } apply;
    } u;
};
Let’s picture some trees

An expression:

(f x (* y 3))

(Representation uses Explist)

A definition:

(define abs (n)
    (if (< n 0) (- 0 n) n))