

$(\text{power } x \ 0) \quad == \ 1$   
 $(\text{power } x \ (+ \ n \ 1)) \quad == \ (* \ (\text{power } x \ n) \ x)$

$(\text{power } x \ 0) \quad == \ 1$   
 $(\text{power } x \ (+ \ (* \ 2 \ m) \ 0)) \quad == \ (\text{square } (\text{power } x \ m))$   
 $(\text{power } x \ (+ \ (* \ 2 \ m) \ 1)) \quad == \ (* \ x \ (\text{square } (\text{power } x \ m)))$

$(\text{all-fours? } d) \quad == \ (= \ d \ 4), \ [d \ \text{is digit}]$   
 $(\text{all-fours? } (+ \ (* \ 10 \ m) \ d)) \quad ==$   
 $(\text{and } (= \ d \ 4) \ (\text{all-fours? } m)), \ \text{where } m \ != \ 0$

$(\text{has-digit? } d \ d) \quad == \ 1$   
 $(\text{has-digit? } d \ d') \quad == \ 0, \ \text{where } d \ \text{differs from } d'$   
 $(\text{has-digit? } (+ \ (* \ 10 \ m) \ d) \ d') \quad ==$   
 $(\text{or } (= \ d \ d') \ (\text{has-digit? } m \ d')), \ \text{where } m \ != \ 0$

# **Bloom's taxonomy (Metacognition 10)**

## **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

## Definitions and expressions:

```
def ::= (define f (x1 ... xn) exp)    ;; "true" defs
      | (val x exp)
      | exp
      | (use filename)                ;; "extended" defs
      | (check-expect exp1 exp2)
      | (check-assert exp)
      | (check-error exp)
```

```
exp ::= integer-literal
     | variable-name
     | (set x exp)
     | (if exp1 exp2 exp3)
     | (while exp1 exp2)
     | (begin exp1 ... expn)
     | (function-name exp1 ... 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 specification

What's different? What's the same?

```
x = 3;
```

```
(set x 3)
```

```
while (i * i < n)
```

```
(while (< (* i i) n)
```

```
  i = i + 1;
```

```
  (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 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 `ExprList`)

A definition:

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

# Behaviors of ASTs, part I: Atomic forms

**Numeral:** stands for a **value**

**Name:** stands for what?

# In Impcore, a name stands for a value

**Environment** associates each **variable** with one **value**

Written  $\rho = \{x_1 \mapsto n_1, \dots, x_k \mapsto n_k\}$ ,  
associates variable  $x_i$  with value  $n_i$ .

**Environment** is **finite map**, aka **partial function**

$x \in \text{dom } \rho$      $x$  is defined in environment  $\rho$

$\rho(x)$             the value of  $x$  in environment  $\rho$

$\rho\{x \mapsto v\}$     extends/modifies environment  $\rho$  to map  $x$  to  $v$

# Environments in C, abstractly

An abstract type:

```
typedef struct Valenv *Valenv;  
  
Valenv mkValenv(Namelist vars, Valuelist vals);  
bool isvalbound(Name name, Valenv env);  
Value fetchval (Name name, Valenv env);  
void bindval   (Name name, Value val, Valenv env);
```

# **“Environment” is pointy-headed theory**

**You may also hear:**

- Symbol table**
- Name space**

**Influence of environment is “scope rules”**

- In what part of code does environment govern?**

# Find behavior using environment

## Recall

`(* y 3) ; ;` what does it mean?

Your thoughts?

# Impcore uses three environments

Global variables  $\xi$

Functions  $\phi$

Formal parameters  $\rho$

There are **no local variables**

- Just like `awk`; if you need temps, use extra formal parameters
- For homework, you'll add local variables

Function environment  $\phi$  not shared with variables—just like Perl