Recall Impcore concrete syntax

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)
```

Define behaviors inductively

We'll focus on expressions

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

Abstract syntax

Same inductive structure as concrete syntax

But,

- More uniform representation
- Designed for compiler, not programmer

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 (One syntax, two behaviors)

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 {  // 'alt' combines with any field in union
    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;
    };
};
```

Let's picture some trees

An expression:

```
(f x (* y 3))
```

Let's picture some trees

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

Environment is finite map, aka partial function

$x \in \mathrm{dom}\,\rho$	**x is defined in environment ρ**
$\rho(x)$	**the value of x in environment ρ**
$\rho\{x \mapsto v\}$	**extends/modifies environment ρ 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 formal term

You may also hear:

- **Symbol table**
- **Name space**

So-called “Scope rules” determine which environment governs where

Find behavior using environment

Consider the program fragment

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

Key Idea: Look up the meaning of variables in an environment

Impcore uses three environments

Global variables ξ ("ksee")

Functions ϕ ("fee")

Formal parameters ρ ("roe")

There are no local variables

- **Just like `awk`; if you need temps, use extra formal parameters**

Function environment ϕ not shared with variables—just like Perl