Imperative Core (Impcore)

is an interpreted language

Translation of compiled programs

Characters (file)	
	} lexer
Tokens	
	} parser
Abstract-Syntax Trees (ASTs)	
	} static semantics
Intermediate Forms	
	} optimizer?
Quadruples	
	} code generator
Assembly Instructions	
	} assembler
Machine Instructions	

Supporting cast

Compile time

- **Symbol table (implements *environments*)**

Run time

- **Instrumentation**
  - **profiling, tracing, testing**
- **Run-time system**
  - **dynamic typing**
  - **memory management**
  - **exceptions**
- **Debugging**

An Imperative Core

Models heart of most languages

Trivial syntax

parenthesized prefix expressions (LISP-like)

Two kinds of inputs

- function definitions

```
(define mod (m n) (- m (* n (/ m n))))
```

like the C function

```
int mod (int m, int n) {
  return m - n * (m / n);
}
```

- expressions

An expression-oriented language

Expressions include control flow (no "statements")

```
(if e1 e2 e3)          if (e1) e2; else e3;
(while e1 e2)          while (e1) e2; 0
(set x e)              x = e
(begin e1 ... en)      { e1; ... ; en }
(f e1 ... en)          f(e1, ... , en)
```

`f` may be primitive or defined with `(define f ...)`

primitives include:

```
+ - * / = < > print
```

More Impcore

Datatypes: integer

(we have functions, but they aren't values)

Scopes (name spaces, environments):

2 levels only: globals, formals

***no* local variables:**

use excess formals (as in awk)

functions live in their own name space

(not shared with variables)

Separate name spaces at work

```
-> (val f 33)
33
-> (define f (x) (+ x x))
f
-> (f f)
66
```

Impcore concrete syntax

toplevel ⇒ *expression* | *fundef* | *val-binding* | `(use` *filename*`)`

fundef ⇒ `(define` *function-name formals expression*`)`

formals ⇒ `(`{*parameter-name*}`)`

val-binding ⇒ `(val` *variable-name expression*`)`

expression ⇒ *literal-value*

| *variable-name*

| `(if` *expression expression expression*`)`

| `(while` *expression expression*`)`

| `(set` *variable-name expression*`)`

| `(begin` *expression* {*expression*}`)`

| `(`*op* {*expression*}`)`

op ⇒ *function-name* | `+` | `-` | `*` | `/` | `=` | `<` | `>` | `print`

More syntax

literal-value $\Rightarrow$ ***integer***

integer $\Rightarrow$ **digits, with optional `-` sign**

****-name*** $\Rightarrow$ **characters, but not `();` or blank**

Abstract syntax

Input translated into efficient internal representation: abstract-syntax tree exp

- **LITERAL (integer)**
- **VAR (name)**
- **SET (name, exp)**
- **IFX (exp, exp, exp)**
- **WHILEX (exp, exp)**
- **BEGIN (explist)**
- **APPLY (name, explist)**

Both built-in and user-defined functions are "application"

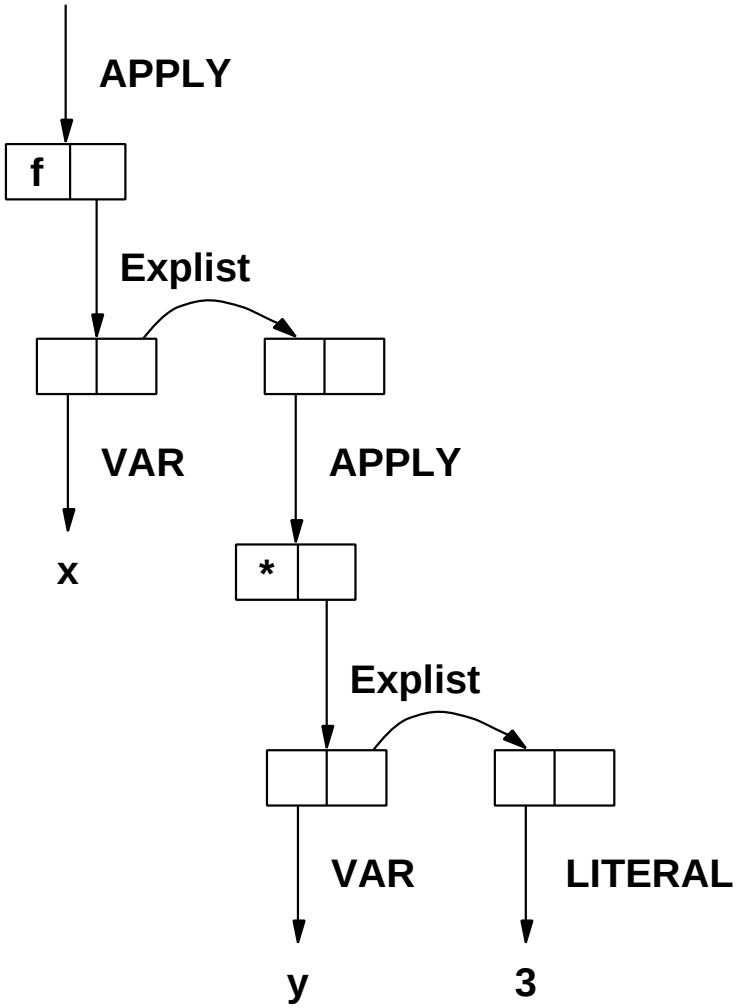
Representing abstract syntax

A recursive type: `typedef struct Exp Exp;`

```
struct Exp {
  enum
  { LITERAL, VAR, SET, IFX, WHILEX, BEGIN, APPLY } ty;
  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;
};
```

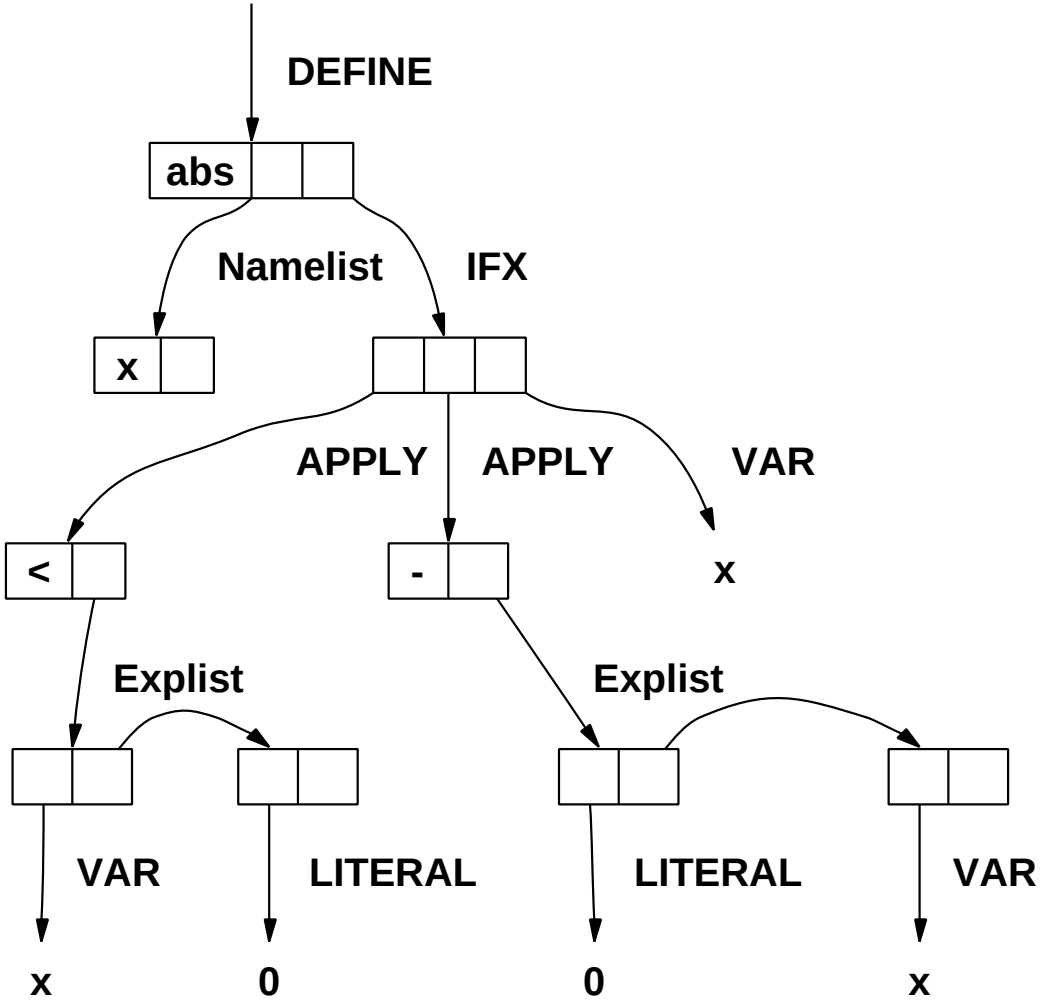
Example AST

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



Another AST

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



Meanings, part I: names

Focus on *environments*:

associate values with variables

Environment ρ is mapping $\{x_1 \mapsto n_1, \ldots x_k \mapsto n_k\}$, which associates variables x_i with values n_i.

$\rho(x)$ denotes value associated with variable x in environment ρ

Environments as abstract type

Declaration:

```
typedef struct Valenv Valenv;
```

Creator:

```
Valenv *mkValenv(Namelist *vars, Valuelist *vals);
```

Observers:

```
int isvalbound(Name *name, Valenv *env);
Value fetchval(Name *name, Valenv *env);
```

Mutator:

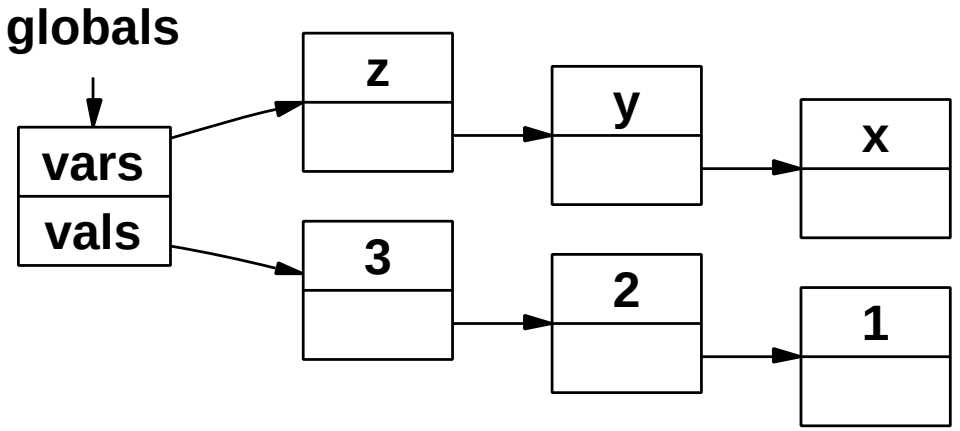
```
void bindval(Name *name, Value val, Valenv *env);
```

Implementing environments

Use pair of lists, e.g., after

```
(val x 1)
(val y 2)
(val z 3)
```

global environment:



Desire for efficient representation, allocation, and deallocation of environments often drives language design

Meanings, part II: expressions

Expressions evaluated w.r.t. environment

(composition of formal, global, function environments)

Heart of the interpreter

- **structural recursion on `Exp`s**
- **environment provides meanings of names**

How do we explain evaluation?

Answer three questions:

1. **What are the expressions?**
2. **What are the values?**
3. **What are the rules for turning expressions into values?**

Combined: *operational semantics*

Operational semantics

Specify executions of programs on an abstract machine

Typical uses

- **Very concise and precise language definition**
- **Direct guide to implementor**
- **Prove things like "well-typed programs don't go wrong"**

Operational Semantics

Loosely speaking, an interpreter

More precisely, formal rules for interpretation

- **Set of expressions, also called terms**
- **Set of values**
- **Full state of abstract machine**
  (e.g., $\langle e, \xi, \phi, \rho \rangle$, $\equiv$ expression + 3 environments)
- **Well specified initial state**
- **Transition rules for the abstract machine**
  - **Good programs end in an accepting state**
  - **Bad programs get stuck ($\equiv$ "go wrong")**

Operational semantics for Impcore

You've seen expressions: ASTs

All values are integers.

State $\langle e, \xi, \phi, \rho \rangle$ is

- e **Expression being evaluated**
- ξ **Values of global variables**
- ϕ **Definitions of functions**
- ρ **Values of formal parameters**

Rules form a proof system for making judgments:

$$\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle$$

Formal semantics

We'll use natural semantics.

This is a specific kind of operational semantics.

It consists of rules of inference (or judgments).

Each rule has the following format:

$$\frac{\text{premises}}{\text{conclusion}} \qquad \text{(NAME OF RULE)}$$

Impcore semantics: Literals

$$\frac{}{\langle \textsf{LITERAL}(v), \xi, \phi, \rho \rangle \Downarrow \langle v, \xi, \phi, \rho \rangle} \qquad \textbf{(LITERAL)}$$

Impcore semantics: Variables

Parameters hide global variables.

$$\frac{x \in \mathrm{dom}\,\rho}{\langle \mathbf{VAR}(x), \xi, \phi, \rho \rangle \Downarrow \langle \rho(x), \xi, \phi, \rho \rangle}$$

(FORMALVAR)

$$\frac{x \notin \mathrm{dom}\,\rho \qquad x \in \mathrm{dom}\,\xi}{\langle \mathbf{VAR}(x), \xi, \phi, \rho \rangle \Downarrow \langle \xi(x), \xi, \phi, \rho \rangle}$$

(GLOBALVAR)

Impcore semantics: Assignment

$$\frac{x \in \mathrm{dom}\,\rho \qquad \langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle}{\langle \mathsf{SET}(x, e), \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho'\{x \mapsto v\} \rangle}$$

(FORMALASSIGN)

$$\frac{x \notin \mathrm{dom}\,\rho \qquad x \in \mathrm{dom}\,\xi \qquad \langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle}{\langle \mathsf{SET}(x, e), \xi, \phi, \rho \rangle \Downarrow \langle v, \xi'\{x \mapsto v\}, \phi, \rho' \rangle}$$

(GLOBALASSIGN)

Evaluation code

```
Value eval(Exp *e, ξ, ϕ, ρ) {
  switch(e->ty) {
  case LITERAL: return e->u.literal;
  case VAR: ... /* look up in ρ and ξ */
  case SET: ... /* modify  ρ or ξ */
  case IFX: ...
  case WHILEX: ...
  case BEGIN: ...
  case APPLY: f = fetchfun(e->u.apply.name, ϕ);
              ... /* user fun or primitive */
  }
}
```

Evaluation cases

`VAR` find binding for variable and use value

`SET` rebind variable in `formals` or `globals`

`IFX` (recursively) evaluate condition, then `t` or `f`

`WHILEX` (recursively) evaluate condition, body

`BEGIN` (recursively) evaluate each `Exp` of body

`APPLY` look up function in `functions`

- built-in `PRIMITIVE` — do by cases
- `USERDEF` function —
  - use arg values to build `formals` env
  - recursive `eval` using fun body

Evaluation — Variables

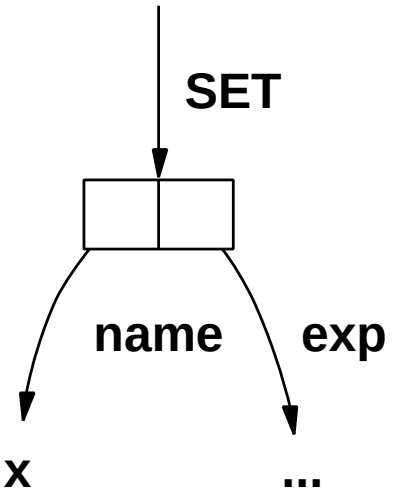
To evaluate x, find binding $\rho(x)$, get value

Conceptually, *one* environment, composed of `formals+globals`

Composition implemented in `eval`, not in `Env` type:

```
case VAR:
  if (isvalbound(e->u.var, formals))
    return fetchval(e->u.var, formals);
  else if (isvalbound(e->u.var, globals))
    return fetchval(e->u.var, globals);
  else
    error("unbound variable %n", e->u.var);
```

Assignment



(set x e)

Means change $\rho(x)$:

- **change parameter or**
- **change global**

```
case SET: {
  Value v = eval(e->u.set.exp,globals,functions,formals);
  if(isvalbound(e->u.set.name, formals))
    bindval(e->u.set.name, v, formals);
  else if(isvalbound(e->u.set.name, globals))
    bindval(e->u.set.name, v, globals);
  else
    error("set: unbound variable %n", e->u.set.name);
  return v; }
```

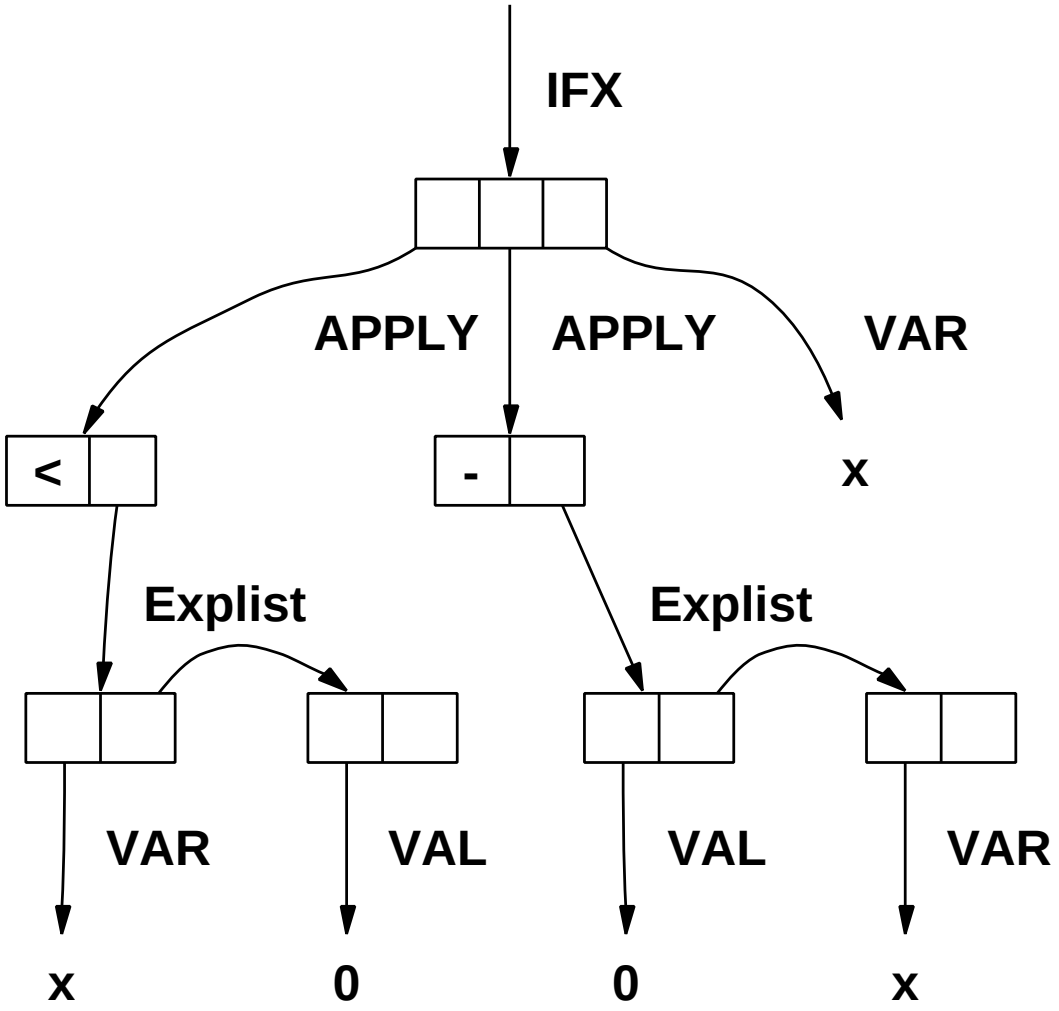
Impcore semantics: "If" Expression

$$\frac{\langle e_1, \xi, \phi, \rho\rangle \Downarrow \langle v_1, \xi', \phi, \rho'\rangle \qquad v_1 \neq 0 \qquad \langle e_2, \xi', \phi, \rho'\rangle \Downarrow \langle v_2, \xi'', \phi, \rho''\rangle}{\langle \mathsf{IF}(e_1, e_2, e_3), \xi, \phi, \rho\rangle \Downarrow \langle v_2, \xi'', \phi, \rho''\rangle} \quad \textbf{(IfTrue)}$$

$$\frac{\langle e_1, \xi, \phi, \rho\rangle \Downarrow \langle v_1, \xi', \phi, \rho'\rangle \qquad v_1 = 0 \qquad \langle e_3, \xi', \phi, \rho'\rangle \Downarrow \langle v_3, \xi'', \phi, \rho''\rangle}{\langle \mathsf{IF}(e_1, e_2, e_3), \xi, \phi, \rho\rangle \Downarrow \langle v_3, \xi'', \phi, \rho''\rangle} \quad \textbf{(IfFalse)}$$

“If” Example

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



Impcore semantics: "While" Expression

$$\frac{\langle e_1,\xi,\phi,\rho\rangle \Downarrow \langle v_1,\xi',\phi,\rho'\rangle \qquad v_1 \neq 0 \qquad \langle e_2,\xi',\phi,\rho'\rangle \Downarrow \langle v_2,\xi'',\phi,\rho''\rangle \qquad \langle \textsf{WHILE}(e_1,e_2),\xi'',\phi,\rho''\rangle \Downarrow \langle v_3,\xi''',\phi,\rho'''\rangle}{\langle \textsf{WHILE}(e_1,e_2),\xi,\phi,\rho\rangle \Downarrow \langle v_3,\xi''',\phi,\rho'''\rangle}$$

(WHILEITERATE)

$$\frac{\langle e_1,\xi,\phi,\rho\rangle \Downarrow \langle v_1,\xi',\phi,\rho'\rangle \qquad v_1 = 0}{\langle \textsf{WHILE}(e_1,e_2),\xi,\phi,\rho\rangle \Downarrow \langle 0,\xi',\phi,\rho'\rangle}$$

(WHILEEND)

Impcore semantics: "Begin" Expression

$$\frac{}{\langle \textsf{BEGIN}(), \xi, \phi, \rho \rangle \Downarrow \langle \mathbf{0}, \xi, \phi, \rho \rangle} \qquad \textsc{(EmptyBegin)}$$

$$\frac{\begin{array}{c}\langle e_1, \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v_1, \xi_1, \phi, \rho_1 \rangle \\ \langle e_2, \xi_1, \phi, \rho_1 \rangle \Downarrow \langle v_2, \xi_2, \phi, \rho_2 \rangle \\ \vdots \\ \langle e_n, \xi_{n-1}, \phi, \rho_{n-1} \rangle \Downarrow \langle v_n, \xi_n, \phi, \rho_n \rangle \end{array}}{\langle \textsf{BEGIN}(e_1, \ldots, e_n), \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v_n, \xi_n, \phi, \rho_n \rangle} \qquad \textsc{(Begin)}$$

Evaluation — Application

1. Find function in old environment

```
f = fetchfun(e->u.apply.name, functions);
```

2. Evaluate actuals to get list of values (also in old ρ)

```
vl = evallist(e->u.apply.actuals, globals, functions, formals);
```

Note: actuals evaluated in the current environment

3. Make new env, binding formals to actuals

```
new_formals = mkValenv(f.u.userdef.formals, vl);
```

4. Evaluate body in new environment

```
return eval(f.u.userdef.body, globals, functions, new_formals);
```

Application — binding parameters

Actuals evaluated in the current environment

Result is `Valuelist` — "half of an environment"
(reason why pair of lists, not list of pairs)

Formals are bound to actuals in a new environment
`mkValenv` builds an environment from two lists

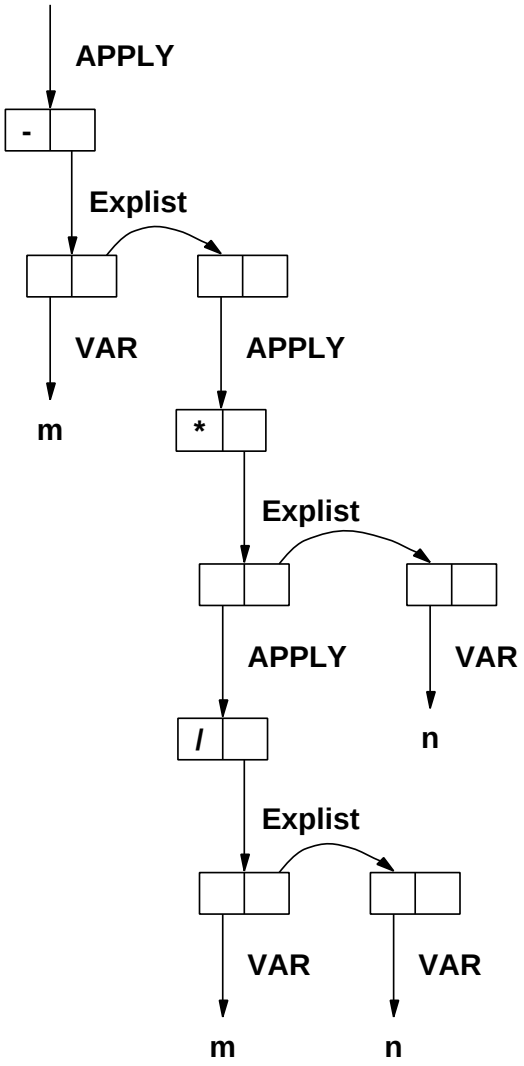
Application semantics

$$\frac{\begin{array}{c}\phi(f) = \textsc{user}(\langle x_1, \ldots, x_n \rangle, e) \\ x_1, \ldots, x_n \text{ all distinct} \\ \langle e_1, \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v_1, \xi_1, \phi, \rho_1 \rangle \\ \langle e_2, \xi_1, \phi, \rho_1 \rangle \Downarrow \langle v_2, \xi_2, \phi, \rho_2 \rangle \\ \vdots \\ \langle e_n, \xi_{n-1}, \phi, \rho_{n-1} \rangle \Downarrow \langle v_n, \xi_n, \phi, \rho_n \rangle \\ \langle e, \xi_n, \phi, \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\} \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle \end{array}}{\langle \textsc{apply}(f, e_1, \ldots, e_n), \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v, \xi', \phi, \rho_n \rangle}$$

(ApplyUser)

Application Example

```
(define mod (m n) (- m (* (/ m n) n)))
```



Semantics for applying primitives

Addition (+) is a typical example.

$$\frac{\begin{array}{c}\phi(f) = \textsc{primitive}(+) \\ \langle e_1, \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v_1, \xi_1, \phi, \rho_1 \rangle \\ \langle e_2, \xi_1, \phi, \rho_1 \rangle \Downarrow \langle v_2, \xi_2, \phi, \rho_2 \rangle \end{array}}{\langle \textsc{apply}(f, e_1, e_2), \xi_0, \phi, \rho_0 \rangle \Downarrow \langle v_1 + v_2, \xi_2, \phi, \rho_2 \rangle} \quad (\textsc{ApplyAdd})$$

Other primitives are similar.

Things to notice about Impcore

Lots of environments:

- **global variables**
- **functions**
- **parameters**
- **local variables?**

More environments $=$ more name spaces

$\Rightarrow$ more complexity

Typical of many programming languages.

Questions to remember

Abstract syntax: what are the terms?

Values: what do terms evaluate to?

Environments: what can names stand for?

Evaluation rules: how to evaluate terms?

Initial basis (primitives): what's built in?