Meaningsof syntax

Norman Ramsey
Geoffrey Mainland

COMP 105—Programming Languages
Tufts University

January 26, 2015

What is the meaning of a while loop?

How do we represent a while loop?

- Code that has a condition and a body.
- The condition can be any expression.
- The body can be any expression, which is executed for its side effects.

Meanings, part I: Names

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);
int isvalbound(Name name, Valenv env);
Value fetchval(Name name, Valenv env);
void bindval(Name name, Value val, Valenv env);
```

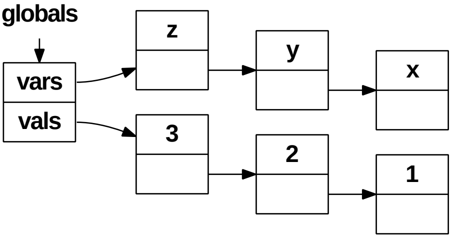
Implementing environments

Uses pair of lists.

Example: after

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

global environment:



Environment costs can drive language design (e.g., Exercise 22).

Concrete syntax for Impcore

Definitions and expressions, as strings

```
def ::= (val x exp)
     |  exp
     |  (define f (formals) e)

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

op ::= function-name | primitive-name
```

Abstract syntax for Impcore

Definitions and expressions as data structures

```
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.

Abstract syntax in C

```
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;
};
```

Analysis and examples

Example AST for

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

(Example uses Explist)

Example Ast for

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

(Example uses Namelist)

Syntax and environments combine to produce meaning

Trick question:

> *What's the value of* `(* y 3)`?

OK, what's its meaning?

Meanings, part II: expressions

Expression evaluation

- Expressions are evaluated in an environment to produce values.
- An environment consists of formal, global, and function environments.

Heart of the interpreter

- structural recursion on Exps
- 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 "environments can be kept on a stack"

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 judgment:

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

(This is a big-step judgment form.)

Impcore semantics: Literals

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

Impcore semantics: Variables

Parameters hide global variables.

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

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

Impcore semantics: Assignment

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

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

Rules of semantics play two roles

- Code: Each rule implemented in interpreter
- Math: Compose rules to make proofs

Interpreter succeeds if and only if a proof exists

Code: Cases to implement evaluation rules

`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, recursively evaluate fun body

Code to implement evaluation

```
Value eval(Exp *e, ξ, φ, ρ) {
  switch(e->alt) {
  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: if (!isfunbound(e->u.apply.name, φ))
                  error("call to undefined function %n",
                        e->u.apply.name);
              f = fetchfun(e->u.apply.name, φ);
              ... /* user fun or primitive */
  }
}
```

Impcore semantics – Variables

$$\frac{x \in \text{dom}\,\rho}{\langle \text{VAR}(x), \xi, \phi, \rho \rangle \Downarrow \langle \rho(x), \xi, \phi, \rho \rangle} \quad \text{FORMALVAR}$$

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

Evaluation — Variables

- To evaluate x, find x in ξ or ρ, 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);
```

Impcore semantics – Assignment

$$\frac{x \in \operatorname{dom} \rho \qquad \langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle}{\langle \textsc{set}(x, e), \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho'\{x \mapsto v\} \rangle} \ \textsc{FormalAssign}$$

$$\frac{x \notin \operatorname{dom} \rho \qquad x \in \operatorname{dom} \xi \qquad \langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle}{\langle \textsc{set}(x, e), \xi, \phi, \rho \rangle \Downarrow \langle v, \xi'\{x \mapsto v\}, \phi, \rho' \rangle} \ \textsc{GlobalAssign}$$

Evaluation — Assignment

`(set x e)` means change ρ or ξ, depending on where `x` is bound.

```
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 – Application

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

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);
   ```

   N.B. 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

Return to math

Use rules to create syntactic proofs
Valid proof is a derivation $\mathcal{D}$
Compositionality again:

- Rule with no premises above the line?
  A derivation by itself
- Rule with premises?
  Build derivations from smaller derivations

Build derivation from conclusion up, left to right

In Impcore, `(+ 2 3)` evaluates to 5 in an environment where $\phi(\texttt{+}) = \textsc{primitive}(\texttt{+})$.

To construct the derivation:

1. Start with $\langle \textsc{apply}(+, \textsc{literal}(2), \textsc{literal}(3)), \xi, \phi, \rho \rangle$ on left side of bottom
2. Find applicable rule $\textsc{ApplyAdd}$ and work up
3. Construct derivations for $\textsc{literal}(2)$ and $\textsc{literal}(3)$ recursively (Notice that ξ and ρ don't change.)
4. Finish with $\langle 5, \xi, \phi, \rho \rangle$ on right side of bottom

Build derivation from conclusion up, left to right

In Impcore, `(+ 2 3)` evaluates to 5 in an environment where $\phi(\texttt{+}) = \textsc{primitive}(\texttt{+})$.

$$\textsc{ApplyAdd}\ \dfrac{\textsc{Literal}\ \dfrac{}{\langle \textsc{literal}(2), \xi, \phi, \rho\rangle \Downarrow \langle 2, \xi, \phi, \rho\rangle} \qquad \dfrac{}{\langle \textsc{literal}(3), \xi, \phi, \rho\rangle \Downarrow \langle 3, \xi, \phi, \rho\rangle}\ \textsc{Literal}}{\langle \textsc{apply}(+, \textsc{literal}(2), \textsc{literal}(3)), \xi, \phi, \rho\rangle \Downarrow \langle 5, \xi, \phi, \rho\rangle}$$

To construct the derivation:

1. Start with $\langle \textsc{apply}(+, \textsc{literal}(2), \textsc{literal}(3)), \xi, \phi, \rho\rangle$ on left side of bottom
2. Find applicable rule $\textsc{ApplyAdd}$ and work up
3. Construct derivations for $\textsc{literal}(2)$ and $\textsc{literal}(3)$ recursively (Notice that ξ and ρ don't change.)
4. Finish with $\langle 5, \xi, \phi, \rho\rangle$ on right side of bottom

Build derivation from conclusion up, left to right

In Impcore, `(+ 2 3)` evaluates to 5 in an environment where $\phi(\texttt{+}) = \text{PRIMITIVE}(\texttt{+})$.

$$\text{ApplyAdd}\ \dfrac{\text{Literal}\ \dfrac{}{\langle \text{LITERAL}(2), \xi, \phi, \rho\rangle \Downarrow \langle 2, \xi, \phi, \rho\rangle} \qquad \dfrac{}{\langle \text{LITERAL}(3), \xi, \phi, \rho\rangle \Downarrow \langle 3, \xi, \phi, \rho\rangle}\ \text{Literal}}{\langle \text{APPLY}(+, \text{LITERAL}(2), \text{LITERAL}(3)), \xi, \phi, \rho\rangle \Downarrow \langle 5, \xi, \phi, \rho\rangle}$$

To construct the derivation:

1. Start with $\langle \text{APPLY}(+, \text{LITERAL}(2), \text{LITERAL}(3)), \xi, \phi, \rho\rangle$ on left side of bottom
2. Find applicable rule ApplyAdd and work up
3. Construct derivations for $\text{LITERAL}(2)$ and $\text{LITERAL}(3)$ recursively (Notice that ξ and ρ don't change.)
4. Finish with $\langle 5, \xi, \phi, \rho\rangle$ on right side of bottom

Build derivation from conclusion up, left to right

In Impcore, (+ 2 3) evaluates to 5 in an environment where $\phi(+) = \text{PRIMITIVE}(+)$.

$$\text{ApplyAdd}\ \dfrac{\text{Literal}\ \dfrac{}{\langle \text{LITERAL}(2), \xi, \phi, \rho\rangle \Downarrow \langle 2, \xi, \phi, \rho\rangle} \qquad \dfrac{}{\langle \text{LITERAL}(3), \xi, \phi, \rho\rangle \Downarrow \langle 3, \xi, \phi, \rho\rangle}\ \text{Literal}}{\langle \text{APPLY}(+, \text{LITERAL}(2), \text{LITERAL}(3)), \xi, \phi, \rho\rangle \Downarrow \langle 5, \xi, \phi, \rho\rangle}$$

To construct the derivation:

1. Start with $\langle \text{APPLY}(+, \text{LITERAL}(2), \text{LITERAL}(3)), \xi, \phi, \rho\rangle$ on left side of bottom
2. Find applicable rule ApplyAdd and work up
3. Construct derivations for LITERAL(2) and LITERAL(3) recursively (Notice that ξ and ρ don't change.)
4. Finish with $\langle 5, \xi, \phi, \rho\rangle$ on right side of bottom

Build derivation from conclusion up, left to right

In Impcore, `(+ 2 3)` evaluates to 5 in an environment where $\phi(+) = \textsc{primitive}(+)$.

$$\textsc{ApplyAdd}\ \dfrac{\textsc{Literal}\ \dfrac{}{\langle \textsc{literal}(2), \xi, \phi, \rho\rangle \Downarrow \langle 2, \xi, \phi, \rho\rangle} \qquad \dfrac{}{\langle \textsc{literal}(3), \xi, \phi, \rho\rangle \Downarrow \langle 3, \xi, \phi, \rho\rangle}\ \textsc{Literal}}{\langle \textsc{apply}(+, \textsc{literal}(2), \textsc{literal}(3)), \xi, \phi, \rho\rangle \Downarrow \langle 5, \xi, \phi, \rho\rangle}$$

To construct the derivation:

1. Start with $\langle \textsc{apply}(+, \textsc{literal}(2), \textsc{literal}(3)), \xi, \phi, \rho\rangle$ on left side of bottom
2. Find applicable rule $\textsc{ApplyAdd}$ and work up
3. Construct derivations for $\textsc{literal}(2)$ and $\textsc{literal}(3)$ recursively (Notice that ξ and ρ don't change.)
4. Finish with $\langle 5, \xi, \phi, \rho\rangle$ on right side of bottom

Build derivation from conclusion up, left to right

In Impcore, `(+ 2 3)` evaluates to 5 in an environment where $\phi(+) = \textsc{primitive}(+)$.

$$\textsc{ApplyAdd}\ \dfrac{\textsc{Literal}\ \dfrac{}{\langle \textsc{literal}(2), \xi, \phi, \rho\rangle \Downarrow \langle 2, \xi, \phi, \rho\rangle} \qquad \dfrac{}{\langle \textsc{literal}(3), \xi, \phi, \rho\rangle \Downarrow \langle 3, \xi, \phi, \rho\rangle}\ \textsc{Literal}}{\langle \textsc{apply}(+, \textsc{literal}(2), \textsc{literal}(3)), \xi, \phi, \rho\rangle \Downarrow \langle 5, \xi, \phi, \rho\rangle}$$

To construct the derivation:

1. Start with $\langle \textsc{apply}(+, \textsc{literal}(2), \textsc{literal}(3)), \xi, \phi, \rho\rangle$ on left side of bottom
2. Find applicable rule $\textsc{ApplyAdd}$ and work up
3. Construct derivations for $\textsc{literal}(2)$ and $\textsc{literal}(3)$ recursively (Notice that ξ and ρ don't change.)
4. Finish with $\langle 5, \xi, \phi, \rho\rangle$ on right side of bottom

A syntactic proof (derivation) is a data structure

Things to notice about Impcore

Lots of environments:

global variables
functions
parameters
local variables?

More environments = more name spaces
⇒ 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?