Traditional Operational Semantics

Define semantics by explaining how a computation is performed.

Informal operational semantics is the normal means of describing programming languages

Examples of Operational Semantics

- Lambda calculus β-reduction (Chapter 5)
- Metacircular interpreters (Chapter 6)
- Translational semantics via attribute grammars (Chapter 7)

Formal Methods

- Vienna Definition Language (VDL)
- SECD Machine
- Structural Operational Semantics

SECD: An Abstract Machine

For Evaluating Lambda Calculus Expressions

S for **Stack**

Structure for storing partial results awaiting subsequent use.

E for **Environment**

Collection of bindings of values (actual parameters) to variables (formal parameters).

C for **Control**

Stack of lambda expressions yet to be evaluated plus a special symbol "@" meaning that an application can be performed; the top expression on the stack is the next to be evaluated.

D for **Dump**

Stack of complete states corresponding to evaluations in progress but suspended while other expressions (inner redexes) are evaluated.

Representation

SECD state: cfg(S, E, C, D)

S and C Stacks: Lists

Empty Stack: [ ]

Top: *head*([a,b,c,d]) = a

Pop: *tail*([a,b,c,d]) = [b,c,d]

Push X onto S: [X|S]

Environment E: List of bindings

E = [x ↦3, y ↦8]

Empty environment: nil

Lookup: E(x) returns 3

Update: [y ↦21]E

Failed lookup: E(z) returns z

Dump D: A stack of states

cfg(S_1,E_1,C_1,cfg(S_2,E_2,C_2,cfg(S_3,E_3,C_3,nil)))

SECD Evaluation

State = Configuration = cfg(S,E,C,D)

Initial Configuration:

cfg([ ],nil,[exp],nil)

Transition Function:

transform : State → State

Final State:

Empty control stack C, *and*

Empty dump D

Transition Function

transform cfg(S,E,C,D) =

(1) if *head*(C) is a constant,
then cfg([*head*(C)|S], E, *tail*(C), D)

If the next expression to be evaluated is a constant, move it from the control stack to the partial result stack as is.

(2) else if *head*(C) is a variable,
then cfg([E(*head*(C))|S], E, *tail*(C), D)

If the next expression is a variable, push its binding in the current environment onto S.

If no binding exists, push the variable itself.

(3) else if *head*(C) is an application, (Rator Rand),
then cfg(S, E, [Rator,Rand,@|*tail*(C)], D)

If the next expression is an application, (Rator Rand), decompose it and reenter it onto the control stack C,

with the Rator at the top,
the Rand next, and
the special application symbol @ following the Rand.

(4) else if *head*(C) is a lambda abstraction, $\lambda V . B$,
then cfg([closure(V,B,E)|S], E, *tail*(C), D)

If the next expression is a lambda abstraction, form a closure incorporating the current environment and add that closure to the partial result stack.

The use of a closure ensures that when the lambda abstraction is applied, its free variables are resolved in the environment of its definition, thereby providing static scoping.

(5) else if *head*(C) = @ and *head*(*tail*(S)) is a predefined function f,
then cfg([f(*head*(S))|*tail*(*tail*(S))], E, *tail*(C), D)

If the next expression is @ and the function in the second place on the S stack is a predefined function,

apply that function to the evaluated argument at the top of the S stack, and replace the two of them by the result.

(6) else if *head*(C) = @ and
head(*tail*(S)) = closure(V,B,E_1),
then cfg([ ], [V $\mapsto$ *head*(S)]E_1, [B],
cfg(*tail*(*tail*(S)),E,*tail*(C),D))

If the next expression is @ and the function in the second place of the S stack is a closure,

push current configuration onto the dump, and
initiate a new computation to evaluate the body of the closure in the closure's environment augmented with the binding of the bound variable in the closure to the argument at the top of the S stack.

(7) else if C = [ ],
then cfg([*head*(S)|S_1], E_1, C_1, D_1)
where D = cfg(S_1,E_1,C_1,D_1)

If the control stack is empty, that means the current evaluation is completed and its result is on the top of the partial result stack.

Pop the configuration on the top of the dump and make it the new current state with the result of the previous computation appended to its partial result stack.

Example: (fun 4)

where *fun* = (λm . (sqr ((λn . (add m n)) 2)))

S	E	C	D	
[ ]	nil	[*fun* 4]	nil	
[ ]	nil	[*fun*, 4, @]	nil	*(3)*
[cls_1]	nil	[4, @]	nil	*(4)*
where cls_1 = closure(m, (sqr ((λn . (add m n)) 2)), nil)				
[4, cls_1]	nil	[@]	nil	*(1)*
[ ]	[m→4]	[(sqr ((λn . (add m n)) 2))]	d_1	*(6)*
where d_1 = cfg([ ], nil, [], nil)				
[ ]	[m→4]	[sqr, ((λn . (add m n)) 2), @]	d_1	*(3)*
[sqr]	[m→4]	[((λn . (add m n)) 2), @]	d_1	*(1)*
[sqr]	[m→4]	[(λn . (add m n), 2), @, @]	d_1	*(3)*
[cls_2, sqr]	[m→4]	[2, @, @]	d_1	*(4)*
where cls_2 = closure(n, (add m n), [m→4])				
[2, cls_2, sqr]	[m→4]	[@, @]	d_1	*(1)*
[ ]	[n→2, m→4]	[(add m) n]	d_2	*(6)*
where d_2 = cfg([sqr], [m→4], [@], d_1)				

[ ]	[n→2, m→4]	[(add m), n, @]	d_2	*(3)*
[ ]	[n→2, m→4]	[add, m, @, n, @]	d_2	*(3)*
[add]	[n→2, m→4]	[m, @, n, @]	d_2	*(1)*
[4, add]	[n→2, m→4]	[@, n, @]	d_2	*(2)*
[add_4]	[n→2, m→4]	[n, @]	d_2	*(5)*
where add_4 = the function that adds its parameter to 4				
[2, add_4]	[n→2, m→4]	[@]	d_2	*(2)*
[6]	[n→2, m→4]	[ ]	d_2	*(5)*
[6, sqr]	[m→4]	[@]	d_1	*(7)*
[36]	[m→4]	[ ]	d_1	*(5)*
[36]	nil	[ ]	nil	*(7)*

Example: (λn . λf . λx . f (n f x)) (λg . λy . g (g y))

S	E	C	D	
[ ]	nil	[(λn.λf.λx. f (n f x)) (λg.λy. g (g y))]	nil	
[ ]	nil	[(λn.λf.λx.f (n f x)), (λg.λy.g (g y)), @]	nil	*(3)*
[cls_1]	nil	[(λg . λy . g (g y)), @]	nil	*(4)*
where cls_1 = closure(n, λf . λx . f (n f x)), nil)				
[cls_2, cls_1]	nil	[@]	nil	*(4)*
where cls_2 = closure(g, λy . g (g y), nil)				
[ ]	[n→cls_2]	[λf . λx . f (n f x))]	d_1	*(6)*
where d_1 = cfg([ ], nil, [ ], nil)				
[cls_3]	[n→cls_2]	[ ]	d_1	*(4)*
where cls_3 = closure(f, λx . f (n f x), [n→cls_2])				
[cls_3]	nil	[ ]	nil	*(7)*

The answer is the closure cls_3, which represents the function

λf . λx . f (n f x))

with the environment

n→cls_2 or n = (λg . λy . g (g y))

Since the SECD machine follows pass by value semantics, the reduction stops here.

If we choose to continue the evaluation inside of the lambda abstraction using β-reductions, we get:

λf . λx . f ((**λg** . λy . g (g y)) **f** x)

$\Rightarrow_\beta$ λf . λx . f ((**λy** . f (f y)) **x**)

$\Rightarrow_\beta$ λf . λx . f (f (f x))

This is the reduction from Chapter 5:

Succ 2 ⇒ 3

Notes

- SECD machine implements pass by value semantics—β-redexes inside a lambda abstraction may not be reduced.

- SECD rules are defined to provide static scoping—the nonlocal environment of a function is the environment in which it is defined, not the calling environment.

  We get dynamic scoping by changing rule (6):

  else if *head*(C) = @ and
  head(*tail*(S)) = closure(V,B,E_1)
  then cfg([ ], [V ↦*head*(S)]E, [B],
  cfg(*tail*(*tail*(S)),E,*tail*(C),D))

Implementing the SECD Machine

S and C Stacks: Prolog lists

head and tail implemented by pattern matching

Environments: Prolog structures

Structures of the form: env(x, 3, env(y, 8, nil))

```
extendEnv(Env,Ide,Val,env(Ide,Val,Env)).

applyEnv(env(Ide,Val,Env),Ide,Val).
applyEnv(env(Ide1,Val1,Env),Ide,Val) :-
                    applyEnv(Env,Ide,Val).
applyEnv(nil,Ide,var(Ide)).
```

Dumps: Prolog structures

cfg(S_1,E_1,C_1,cfg(S_2,E_2,C_2,cfg(S_3,E_3,C_3,nil)))

Transform Predicate

(1) if *head*(C) is a constant,
then cfg([*head*(C)|S], E, *tail*(C), D)

becomes

```
transform(cfg(S,E,[con(C)|T],D),
                    cfg([con(C)|S],E,T,D)).      % 1
```

(2) else if *head*(C) is a variable,
then cfg([E(*head*(C))|S], E, *tail*(C), D)

becomes

```
transform(cfg(S,E,[var(X)|T],D),
    cfg([Val|S],E,T,D)) :- applyEnv(E,X,Val).    % 2
```

(3) else if *head*(C) is an application, (Rator Rand),
then cfg(S, E, [Rator,Rand,@|*tail*(C)], D)

becomes

```
transform(cfg(S,E,[comb(Rator,Rand)|T],D),
              cfg(S,E,[Rator,Rand,@|T],D)).      % 3
```


(4) else if *head*(C) is a lambda abstraction, λV . B,
then cfg([closure(V,B,E)|S], E, *tail*(C), D)

becomes

```
transform(cfg(S,E,[lamb(X,B)|T],D),
          cfg([closure(X,B,E)|S],E,T,D)).        % 4
```

(5) else if *head*(C) = @ and *head*(*tail*(S)) is a predefined function f,
then cfg([f(*head*(S))|*tail*(*tail*(S))], E, *tail*(C), D),

becomes

```
transform(
    cfg([con(Rand),con(Rator)|T],E,[@|T1],D),
    cfg([Val|T],E,T1,D)) :-
                compute(Rator,Rand,Val).         % 5
```

(6) else if *head*(C) = @ and
head(*tail*(S)) = closure(V,B,E_1),
then cfg([ ], [V |→*head*(S)]E_1, [B],
cfg(*tail*(*tail*(S)),E,*tail*(C),D))

becomes

```
transform(
    cfg([Rand,closure(V,B,E1)|T],E,[@|T1],D),
    cfg([ ],E2,[B],cfg(T,E,T1,D))) :-
                    extendEnv(E1,V,Rand,E2).   % 6
```

(7) else if C = [ ]
then cfg([*head*(S)|S_1], E_1, C_1, D_1),
where D = cfg(S_1,E_1,C_1,D_1)

becomes

```
transform(cfg([H|S],E,[ ],cfg(S1,E1,C1,D1)),
            cfg([H|S1],E1,C1,D1)).          % 7
```


Driver

```
interpret(cfg([Result|S],Env,[ ],nil), Result).

interpret(Config,Result) :-
                transform(Config,NewConfig),
                interpret(NewConfig,Result).

go :- nl, write('>>> SECD: Interpreting the
                        Lambda Calculus <<<'), nl,
      write('Enter name of source file:  '), nl,
      readfile(File),nl, see(File), scan(Tokens),
      write('Successful Scan'), nl, !,
%     write(Tokens), nl, nl,
      seen, program(prog(Exp),Tokens,[eop]),
      write('Successful Parse'),nl,nl, !,
%     write(prog(Exp)), nl, nl,
      interpret(cfg([ ],nil,[Exp],nil), Result), nl,
      write('Result =  '), pp(Result), nl.
```

Try It: cp ~slonnegr/public/plf/secd .
cp ~slonnegr/public/plf/twice .

Structural Operational Semantics

Abstract machine is a system of logical inference rules.

Premises, a conclusion, and possibly a condition.

$$\frac{\text{premise}_1 \quad \text{premise}_2 \quad \ldots \quad \text{premise}_n}{\text{conclusion}} \quad \text{condition}$$

An axiom is a rule without premises.

This modus ponens rule is an example of natural deduction.

$$\frac{p \qquad p \supset q}{q}$$

Abstract Syntax of Wren

Ignore declarations.

Assume program is syntactically correct, including context-sensitive syntax.

Syntactic Categories (with metavariables)

- n ∈ Num = Set of numerals
- b ∈ { **true**, **false** } = Set of boolean values
- id ∈ Id = Set of integer identifiers
- bid ∈ Bid = Set of boolean identifiers
- iop ∈ Iop = { +, −, *, / }
- rop ∈ Rop = { <, ≤, =, ≥, >, <> }
- bop ∈ Bop = { **and**, **or** }
- ie ∈ Iexp = Set of integer expressions
- be ∈ Bexp = Set of boolean expressions
- c ∈ Cmd = Set of commands

Some Abstract Syntax Rules:

$n : iexp \quad n \in Num \qquad id : iexp \quad id \in Id$

$$\frac{ie_1 : iexp \quad ie_2 : iexp}{ie_1 \ iop \ ie_2 : iexp} \ iop \in Iop$$

$$\frac{ie : iexp}{-\ ie : iexp}$$

skip : cmd

read id : cmd id∈Id

$$\frac{ie : iexp}{id \ \textbf{:=} \ ie : cmd} \ id \in Id$$

$$\frac{be : bexp \quad c_1 : cmd \quad c_2 : cmd}{\textbf{if} \ be \ \textbf{then} \ c_1 \ \textbf{else} \ c_2 : cmd}$$

$$\frac{be : bexp \quad c : cmd}{\textbf{while} \ be \ \textbf{do} \ c : cmd}$$

$$\frac{c_1 : cmd \quad c_2 : cmd}{c_1 \ ; \ c_2 : cmd}$$

Derivation in the Abstract Syntax

Show that

if m<0 **then** m := -m; **write** m+2 **else skip**

is a command.

$$\frac{\dfrac{m : iexp \quad \dfrac{\dfrac{m : iexp \quad 2 : iexp}{m+2 : iexp}}{\textbf{write} \ m+2 : cmd}}{m := -m : cmd} \quad \textbf{skip} : cmd}{\dfrac{m : iexp \quad 0 : iexp}{m<0 : bexp} \quad \textbf{if} \ m<0 \ \textbf{then} \ m := -m \ ; \ \textbf{write} \ m+2 \ \textbf{else skip} : cmd}$$

Structural Induction

Abstract syntax is defined inductively.

Base Cases:

$n \in Iexp$ for each $n \in Num$

$id \in Iexp$ for each $id \in Id$

Induction Cases:

If $ie_1, ie_2 \in Iexp$ and $iop \in Iop$,
then $(ie_1 \ iop \ ie_2) \in Iexp$

If $be \in Bexp$ and $c \in Cmd$,
then (**if** be **then** c) $\in Cmd$

Principle of Structural Induction

To prove that a property holds for all phrases in some syntactic category,

confirm two conditions:

Basis: The property must be established for each atomic (nondecomposable) syntactic element produced by an axiom.

Induction step: The property must be proved for any composite element given that it holds for each of its immediate subphrases (the induction hypothesis).

Example: Lists of Integers

Definition

[ ] : intList

$$\frac{\text{tail : intList}}{\text{m :: tail : intList}}\ \text{m}\in\text{num}$$

Functions on Lists:

length([ ]) = 0

length(m :: tail) = 1 + length(tail)

sum([ ]) = 0

sum(m :: tail) = m + sum(tail)

max([ ]) = "-maxint"

max(m :: tail) = "larger of"(m,max(tail))

concat([ ],L) = L

concat(m :: tail, L) = m :: concat(tail,L)

Property P(L):

sum(L) ≤ max(L) * length(L)

Proof by structural induction

Basis: P([ ]) because

sum([ ]) = 0 ≤ max([ ]) * 0 = max([ ])*length([ ])

Induction Step:

Suppose P(tail) as the induction hypothesis

sum(tail) ≤ max(tail) * length(tail)

for some tail : intList

Note that m ≤ max(m :: tail)

and max(tail) ≤ max(m :: tail)

So

sum(m :: tail) = m + sum(tail)

≤ m + max(tail) * length(tail)

≤ max(m :: tail) + max(m :: tail) * length(tail)

= max(m :: tail) * (1 + length(tail))

= max(m :: tail) * length(m :: tail)

That is, P(m :: tail).

Semantics of Expressions in Wren

Store

- Models memory of a computer
- A finite function
- *dom*(sto) = set of variables bound in sto

if sto = { m ↦13, p ↦false, z ↦-34 },

dom(sto) = { m, p, z }.

Abstract operations manipulate store:

emptySto

represents a store with no binding (all identifiers are undefined).

updateSto(sto,id,n) and *updateSto*(sto,bid,b)

represent the store that agrees with sto but contains one new binding: id ↦n or bid ↦b.

applySto(sto,id) and *applySto*(sto,bid)

return the value associated with id or bid; if no binding exists, the operation fails blocking the deduction.

Utility function for binary operations

compute(op,arg_1,arg_2)

compute(+,5,8) is 13
compute(/,n,0) is undefined

A Configuration: a pair
an expression and a store

Final Configuration:

<n,sto> or <b,sto>
where n ∈ Num and b ∈ { **true**, **false** }

Transition Function: →

$\langle e_1,sto\rangle \rightarrow \langle e_2,sto\rangle \rightarrow \ldots \rightarrow \langle e_{n-1},sto\rangle \rightarrow \langle e_n,sto\rangle$.

Inference System for Expressions

$$(3)\ \frac{\langle be_1,sto\rangle \rightarrow \langle be_1',sto\rangle}{\langle be_1 \text{ bop } be_2,sto\rangle \rightarrow \langle be_1' \text{ bop } be_2,sto\rangle}$$

$$(6)\ \frac{\langle be_2,sto\rangle \rightarrow \langle be_2',sto\rangle}{\langle b \text{ bop } be_2,sto\rangle \rightarrow \langle b \text{ bop } be_2',sto\rangle}$$

(9) $\langle b_1 \text{ bop } b_2, sto\rangle \rightarrow$
$\langle$*compute*(bop,b_1,b_2), sto$\rangle$

$$(10)\ \frac{\langle be,sto\rangle \rightarrow \langle be',sto\rangle}{\langle \textbf{not}(be),sto\rangle \rightarrow \langle \textbf{not}(be'),sto\rangle}$$

(11a) <**not**(**true**),sto> → <**false**,sto>
(11b) <**not**(**false**),sto> → <**true**,sto>

(13) <bid,sto> → <*applySto*(sto,bid),sto> bid∈*dom*(sto)

$$(14)\ \frac{\langle e_1,sto\rangle \rightarrow \langle e_2,sto\rangle \quad \langle e_2,sto\rangle \rightarrow \langle e_3,sto\rangle}{\langle e_1,sto\rangle \rightarrow \langle e_3,sto\rangle}$$

e_i∈Iexp or e_i∈Bexp

Sample Derivation: p **or** z<0

Given sto = { m ↦13, p ↦false, z ↦-34 },

a) <p, sto> → <**false**, sto> 13
since *applySto*(sto,p)=**false**

b) <p **or** z<0, sto> → <**false or** z<0, sto> 3,a

c) <z, sto> → <-34, sto> 12
since *applySto*(sto,z)=-34

d) <z<0, sto> → <-34<0, sto> 2,c

e) <-34<0, sto> → <**true**, sto> 8
since *compute*(<,-34,0)=**true**

f) <z<0, sto> → <**true**, sto> 14,d,e

g) <**false or** z<0, sto> → <**false or true**, sto> 6,f

h) <p **or** z<0, sto> → <**false or true**, sto> 14,b,g

i) <**false or true**, sto> → <**true**, sto> 9
since *compute*(**or**,**false**,**true**)=**true**

j) <p **or** z<0, sto> → <**true**, sto> 14,h,i

Stuck Derivations

Conditions on rules prohibit dynamic (semantic) errors:

(iop≠/) or (n≠0)	for rule (7),
id∈*dom*(sto)	for rule (12), and
bid∈*dom*(sto)	for rule (13).

The **read** command allows another semantic error.

Derivations also stop when a final configuration is reached, since no inference rule applies.

View this normal form as "normal" termination.

Normal form is the goal of a derivation.

Completeness Theorem

1) For any ie∈(Iexp – Num) and sto∈Store with *var*(ie)⊆*dom*(sto) and no occurrence of the division operator in ie, there is a numeral n∈Num such that <ie,sto> → <n,sto>.

Proof: The proof is by structural induction following the abstract syntax of expressions in Wren.

1) Let ie∈Iexp – Num and sto∈Store with *var*(ie)⊆*dom*(sto), and suppose ie has no occurrence of the division operator. By the definition of abstract syntax, Figure 8.4, ie must be of the form id∈Id or (ie_1 iop ie_2) where iop∈Iop – {/} and ie_1,ie_2 : iexp have no occurrence of /.

Case 1: ie = id∈Id.
Then id∈dom(sto) and <id,sto> → <n,sto> where n = *applySto*(sto,id) using rule (12).

Case 2: ie = ie_1 iop ie_2 where iop∈Iop – {/} and ie_1,ie_2 : iexp, and for i=1,2, var(ie_i)⊆*dom*(sto) and ie_i contains no occurrence of /.

Subcase a: ie_1 = n_1∈Num and ie_2 = n_2∈Num.

Then <ie,sto> = <n_1 iop n_2,sto> → <n,sto> where n = compute(iop,n_1,n_2) by rule (7) whose condition is satisfied since iop≠/.

Subcase b: ie_1 = n_1∈Num and ie_2∈(Iexp – Num).
By the induction hypothesis,
<ie_2,sto> → <n_2,sto> for some n_2∈Num.

Then using rule (4), we get
<ie,sto> = <n_1 iop ie_2,sto> → <n_1 iop n_2,sto>, to which we can apply subcase a.

Subcase c: ie_1∈(Iexp – Num) and ie_2∈Iexp.

By the induction hypothesis,
<ie_1,sto> → <n_1,sto> for some n_1∈Num.

Then using rule (1), we get
<ie,sto> = <ie_1 iop ie_2,sto> → <n_1 iop ie_2,sto>, to which we can apply subcase a or b.

Therefore, the conditions for structural induction on integer expressions are satisfied and the theorem holds for all ie∈(Iexp – Num).

Semantics of Commands in Wren

Commands may modify the store.

Need to model input and output to provide semantics for the **read** and **write** commands.

Configurations: <c, st(in,out,sto)>

where "in" and "out" are finite lists of integers representing input and output streams,
and
st(in,out,sto) is called the state.

A Computation:

$\langle c_0, st(in_0,out_0,sto_0)\rangle \rightarrow$
$\langle c_1, st(in_1,out_1,sto_1)\rangle \rightarrow$
$\langle c_2, st(in_2,out_2,sto_2)\rangle \rightarrow \ldots.$

Computations may continue forever.

Auxiliary Operations for Lists

head, *tail*, and *affix* where *affix*([ ],5) = [5]
and *affix*([13,21],34) = [13,21,34]

Goal of a Derivation:

Starting with an initial configuration,
reduce the command c
comprising a Wren program
to an empty command, **skip**.

Initial State:

Given input list $[n_1, n_2, n_3, \ldots, n_k]$ and command c,

$st_0 = st([n_1, n_2, n_3, \ldots, n_k], [\,], emptySto)$

Successful Derivation:

$\langle c, st([n_1, n_2, n_3, \ldots, n_k], [\,], emptySto)\rangle \rightarrow$
$\ldots \rightarrow \langle$**skip**, st(in,$[k_1, k_2, \ldots, k_i]$,sto)$\rangle$

Inference System for Commands

Assignment Commands

(1) $$\frac{\langle ie,sto\rangle \rightarrow \langle ie',sto\rangle}{\langle id := ie,st(in,out,sto)\rangle \rightarrow \langle id := ie',st(in,out,sto)\rangle}$$

$$\frac{\langle be,sto\rangle \rightarrow \langle be',sto\rangle}{\langle bid := be,st(in,out,sto)\rangle \rightarrow \langle bid := be',st(in,out,sto)\rangle}$$

(2) $\langle id := n,st(in,out,sto)\rangle \rightarrow$
$\langle$**skip**,st(in,out,*updateSto*(sto,id,n))$\rangle$

$\langle bid := b,st(in,out,sto)\rangle \rightarrow$
$\langle$**skip**,st(in,out,*updateSto*(sto,bid,b))$\rangle$

If Commands

(3) $$\frac{\langle be,sto\rangle \rightarrow \langle be',sto\rangle}{\langle \textbf{if } be \textbf{ then } c_1 \textbf{ else } c_2,st(in,out,sto)\rangle \rightarrow \langle \textbf{if } be' \textbf{ then } c_1 \textbf{ else } c_2,st(in,out,sto)\rangle}$$

(4) $\langle$**if true then** c_1 **else** c_2,state$\rangle \rightarrow \langle c_1$,state$\rangle$

(5) $\langle$**if false then** c_1 **else** c_2,state$\rangle \rightarrow \langle c_2$,state$\rangle$

(6) $\langle$**if** be **then** c,state$\rangle \rightarrow$
$\langle$**if** be **then** c **else skip**,state$\rangle$

While Command

(7) $\langle$**while** be **do** c,state$\rangle \rightarrow$
$\langle$**if** be **then** (c ; **while** be **do** c)
else skip,state$\rangle$

Command Sequencing

$$(8)\quad \frac{\langle c_1,state\rangle \rightarrow \langle c_1',state'\rangle}{\langle c_1 ; c_2,state\rangle \rightarrow \langle c_1' ; c_2,state'\rangle}$$

(9) $\langle$**skip** ; c,state$\rangle \rightarrow \langle$c,state$\rangle$

Read Command

(10) $\langle$**read** id,st(in,out,sto)$\rangle \rightarrow$ in≠[ ]
$\langle$**skip**,st(*tail*(in),out,
updateSto(sto,id,*head*(in)))$\rangle$

Write Command

$$(11)\quad \frac{\langle ie,sto\rangle \rightarrow \langle ie',sto\rangle}{\langle \textbf{write}\ ie,st(in,out,sto)\rangle \rightarrow \langle \textbf{write}\ ie',st(in,out,sto)\rangle}$$

(12) $\langle$**write** n,st(in,out,sto)$\rangle \rightarrow$
$\langle$**skip**,st(in,*affix*(out,n),sto)$\rangle$

Possible Outcomes

1. After a finite number of transitions, we reach a configuration $\langle$**skip**,state$\rangle$ in normal form.

2. After a finite number of transitions, we reach a configuration that is not in normal form but for which no further transition is defined.

   Semantic Errors:

   - an undefined (uninitialized) identifier
   - division by zero
   - empty input list when executing a **read** command

3. A computation sequence continues forever (a **while** command that never terminates).

   Notation: "$\langle$**c,state**$\rangle \rightarrow \infty$ "

A Sample Computation: Good luck

Semantic Equivalence

Definition: Commands c_1 and c_2 are **semantically equivalent,** written $c_1 \equiv c_2$, if

1) For any two states s and s_f,
$\langle c_1,s\rangle \rightarrow \langle$**skip**,$s_f\rangle$ if and only if
$\langle c_2,s\rangle \rightarrow \langle$**skip**,$s_f\rangle$,
and

2) For any state s,
$\langle c_1,s\rangle \rightarrow \infty$ if and only if $\langle c_2,s\rangle \rightarrow \infty$.

Examples of Semantic Equivalence

For any c_1,c_2 : cmd and be : bexp,
if be **then** c_1 **else** $c_2 \equiv$
if not(be) **then** c_2 **else** c_1.

For any c:cmd, c **; skip** ≡ c.

For any c:cmd, **skip ;** c ≡ c.

For any be:bexp and c:cmd,
if be **then** c **else** c ≡ c.
assuming be does not become stuck.

An Equivalent Definition

Definition: Commands c_1 and c_2 are **semantically equivalent,** written $c_1 \equiv c_2$, if these three conditions hold:

1) For any two states s and s_f,
if $\langle c_1, s\rangle \rightarrow \langle$**skip**, $s_f\rangle$,
then $\langle c_2, s\rangle \rightarrow \langle$**skip**, $s_f\rangle$.

2) For any state s,
if $\langle c_1, s\rangle \rightarrow \infty$, then $\langle c_2, s\rangle \rightarrow \infty$.

3) For any state s,
if derivation of $\langle c_1, s\rangle$ becomes stuck,
then derivation of $\langle c_2, s\rangle$ becomes stuck.

The equivalence of the two definitions can be verified with symbolic logic.

For any be:bexp and c_1,c_2,c_3:cmd,
(**if** be **then** c_1 **else** c_2) **;** c_3 ≡
if be **then** (c_1 **;** c_3) **else** (c_2 **;** c_3).

Proof of equivalence: c **; skip** ≡ c.

Let s be an arbitrary state.

Case 1: <c,s> → <**skip**,s'>.
Then by (8), <c;**skip**,s> → <**skip**;**skip**,s'>
and by (9), <**skip**;**skip**,s'> → <**skip**,s'>

Case 2: <c,s> → <c',s'> is stuck.
Since rule (8) is the only one that can be applied,
<c;**skip**,s> → <c';**skip**,s'> is also stuck.

Case 3: <c,s> → ∞.

So the derivation
<c,s> → <c_1,s_1> → <c_2,s_2> → ….
continues forever.

Then by (8), the derivation
<c;**skip**,s> → <c_1;**skip**,s_1>
→ <c_2;**skip**,s_2> → ….
also continues forever.

Implementing Operational Semantics

Transition Function for Commands

```
transform(cfg(C,st(In,Out,Sto)),
                cfg(C1,st(In1,Out1,Sto1))).
```

Driver

```
interpret(cfg(skip,FinalState),FinalState).
interpret(Config,FinalState) :-
        transform(Config,NewConfig),
        interpret(NewConfig,FinalState).
```

If Commands (note order of clauses)

```
transform(cfg(if(bool(true),C1,C2),State),
        cfg(C1,State)).                          % 4

transform(cfg(if(bool(false),C1,C2),State),
        cfg(C2,State)).                          % 5

transform(cfg(if(Be,C1,C2),st(In,Out,Sto)),      % 3
        cfg(if(Be1,C1,C2),st(In,Out,Sto))) :-
          transform(cfg(Be,Sto),cfg(Be1,Sto)).
```


If-Then Command

```
transform(cfg(if(Be,C),State),                   % 6
                cfg(if(Be,C,skip),State)).
```

While Command

```
transform(cfg(while(Be,C),State),                % 7
        cfg(if(Be,[C,while(Be,C)],skip),State)).
```

Build an abstract syntax tree (structure) for the if command.

Note list of commands representing a sequence.

Command Sequencing

```
transform(cfg([skip|Cs],State), cfg(Cs,State)).  % 9

transform(cfg([ ],State), cfg(skip,State)).

transform(cfg([C|Cs],State), cfg([C1|Cs],State1))
    :- transform(cfg(C,State),cfg(C1,State1)).   % 8
```


Modeling the Store

A Prolog structure sto(var, val, restofSto).

Example

```
sto(m, int(13),
        sto(b, bool(false),
                sto(z, int(-34), nil)))
```

represents the store

{ m ↦*int*(13), b ↦*bool*(false), z ↦*int*(-34) }.

The empty store: the Prolog atom “nil”.

Auxiliary Functions

```
updateSto(sto(Ide,V,Sto),Ide,Val,
                        sto(Ide,Val,Sto)).
updateSto(sto(I,V,Sto),Ide,Val,sto(I,V,NewSto))
        :- updateSto(Sto,Ide,Val,NewSto).
updateSto(nil,Ide,Val,sto(Ide,Val,nil)).
```

```
applySto(sto(Ide,Val,Sto),Ide,Val).
applySto(sto(I,V,Sto),Ide,Val) :-
                        applySto(Sto,Ide,Val).
applySto(nil,Ide,undefined) :-
        write('Undefined variable'), nl, abort.
```

Assignment Command

```
transform(cfg(assign(Ide,int(N)),st(In,Out,Sto)),
        cfg(skip,st(In,Out,Sto1))) :-          % 2a
    updateSto(Sto,Ide,int(N),Sto1).

transform(cfg(assign(Ide,bool(B)),st(In,Out,Sto)),
        cfg(skip,st(In,Out,Sto1))) :-          % 2b
    updateSto(Sto,Ide,bool(B),Sto1).

transform(cfg(assign(Ide,E),st(In,Out,Sto)),   % 1
        cfg(assign(Ide,E1),st(In,Out,Sto))) :-
    transform(cfg(E,Sto),cfg(E1,Sto)).
```


Read Command

```
transform(cfg(read(Ide),st([ ],Out,Sto)),
        cfg(skip,st([ ],Out,Sto))) :-          % 10
    write('Attempted read of empty file'), nl, abort.

transform(cfg(read(Ide),st([NlT],Out,Sto)),
        cfg(skip,st(T,Out,Sto1))) :-           % 10
    updateSto(Sto,Ide,int(N),Sto1).
```

Write Command

```
transform(cfg(write(int(N)),st(In,Out,Sto)),
        cfg(skip,st(In,Out1,Sto))) :-
    concat(Out,[N],Out1).                      % 12

transform(cfg(write(E),st(In,Out,Sto)),
        cfg(write(E1),st(In,Out,Sto))) :-
    transform(cfg(E,Sto),cfg(E1,Sto)).         % 11
```


Transition Function for Expressions

```
transform(cfg(E,Sto), cfg(E1,Sto)).
```

Driver (Not used in Wren interpreter.)

```
evaluate(cfg(int(N),Sto), int(N)).
evaluate(cfg(bool(B),Sto), bool(B)).
evaluate(Config, FinalValue) :-
            transform(Config, NewConfig),
            evaluate(NewConfig, FinalValue).
```

Rules (7), (8), and (9)

```
transform(cfg(exp(Opr,int(N1),int(N2)),Sto),
        cfg(Val,Sto)) :-                       % 7+8
    compute(Opr,int(N1),int(N2),Val).
%  Opr can represent both lop and Rop.

transform(cfg(bexp(Opr,bool(B1),bool(B2)),Sto),
        cfg(Val,Sto)) :-                       % 9
    compute(Opr,bool(B1),bool(B2),Val).
```


Rules (4), (5), and (6)

```
transform(cfg(exp(Opr,int(N),E2),Sto),         % 4+5
        cfg(exp(Opr,int(N),E2p),Sto)) :-
    transform(cfg(E2,Sto),cfg(E2p,Sto)).

transform(cfg(bexp(Opr,bool(B),E2),Sto),       % 6
        cfg(bexp(Opr,bool(B),E2p),Sto)) :-
    transform(cfg(E2,Sto),cfg(E2p,Sto)).
```

Rules (1), (2), and (3)

```
transform(cfg(exp(Opr,E1,E2),Sto),             % 1+2
        cfg(exp(Opr,E1p,E2),Sto)) :-
    transform(cfg(E1,Sto),cfg(E1p,Sto)).

transform(cfg(bexp(Opr,E1,E2),Sto),            % 3
        cfg(bexp(Opr,E1p,E2),Sto)) :-
     transform(cfg(E1,Sto),cfg(E1p,Sto)).
```

Compute

```
compute(plus,int(M),int(N),int(R)) :- R is M+N.

compute(divides,int(M),int(0),int(0)) :-
                write('Division by zero'), nl, abort.
compute(divides,int(M),int(N),int(R)) :-
                                        R is M//N.

compute(equal,int(M),int(N),bool(true)) :-
                                        M =:= N.
compute(equal,int(M),int(N),bool(false)).

compute(neq,int(M),int(N),bool(false)) :-
                                        M =:= N.
compute(neq,int(M),int(N),bool(true)).

compute(less,int(M),int(N),bool(true)) :- M < N.
compute(less,int(M),int(N),bool(false)).

compute(and,bool(true),bool(true),bool(true)).
compute(and,bool(P),bool(Q),bool(false)).
```


Top Level Driver

```
go :-
    nl, write('>>> Interpreting Wren via
                Operational Semantics <<<'), nl, nl,
    write('Enter name of source file: '), nl,
                readfilename(File), nl,
    see(File), scan(Tokens), seen,
                write('Scan successful'), nl, !,
    program(prog(Dec,Cmd),Tokens,[eop]),
                write('Parse successful'), nl, !,
    write('Enter input list followed by a period: '),
                nl, read(In), nl,
    interpret(cfg(Cmd,st(In,[ ],nil)),
                st(FinalIn,Out,Sto)), nl,
    write('Output = '), write(Out), nl, nl,
    write('Final Store:'), nl, printSto(Sto), nl.
```

Try It

```
cp ~slonnegr/public/plf/sos .
```