Evaluation by Rewriting

One very nice feature of pure functional languages is that we can explain their behavior (semantics) by means of simple **rewriting** rules based on **substitution**.

We view evaluation as a sequence of transformations on the program **text**. We begin with the original source program. At each evaluation step, the program is rewritten to a new form, hopefully smaller and simpler than before.

Execution is completed when the program reaches a particularly simple **normal** form, such as constants, constructed data values, and functions. We call these **values** (v).

Here are some suitable rules for pure, non-recursive Standard ML expressions (e). Remember, SML uses **eager** (or "**call-by-value**") semantics.

e_1 `[` e_2 `/` x `]` means e_1 with e_2 substituted for variable x

Examples:

```
(p + q) [ 10 / q ] ≡ (p + 10)
(p + q) [ q / p ]  ≡ (q + q)
(p + q) [ (r + 32) / q ]  ≡ (p + (r + 32))
```

Evaluation by Rewriting - Example

```
let fun f x = if x < 0 then 0 else x
in let val p = f 2
   in f (p - 3)
   end
end

≡
let val f = fn x => if x < 0 then 0 else x
in let val p = f 2
   in f (p - 3)
   end
end

⇝[1]
let val p = (fn x => if x < 0 then 0 else x) 2
in (fn x => if x < 0 then 0 else x) (p - 3)
end

⇝[2,6]
let val p = if 2 < 0 then 0 else 2
in (fn x => if x < 0 then 0 else x) (p - 3)
end

⇝[2,5,9]
let val p = if false then 0 else 2
in (fn x => if x < 0 then 0 else x) (p - 3)
end
```

Rules

(1) `let val` x `=` v `in` e `end` ⇝ e `[` v `/` x `]`

(2) *If* $e_1 \rightsquigarrow e_1'$ *then*
`let val` x `=` e_1 `in` e_2 `end` ⇝ `let val` x `=` e_1' `in` e_2 `end`

(3) `if true then` e_1 `else` $e_2 \rightsquigarrow e_1$

(4) `if false then` e_1 `else` $e_2 \rightsquigarrow e_2$

(5) *If* $e_0 \rightsquigarrow e_0'$ *then*
`if` e_0 `then` e_1 `else` $e_2 \rightsquigarrow$ `if` e_0' `then` e_1 `else` e_2

(6) `(fn` x `=>` e`)` $v \rightsquigarrow e$ `[` v `/` x `]`

(7) *If* $e_1 \rightsquigarrow e_1'$ *then*
$e_1\ e_2 \rightsquigarrow e_1'\ e_2$

(8) *If* $e_2 \rightsquigarrow e_2'$ *then*
$v_1\ e_2 \rightsquigarrow v_1\ e_2'$

(9) v_1 *primop* $v_2 \rightsquigarrow$ result of performing *primop* on v_1, v_2

(10) *If* $e_1 \rightsquigarrow e_1'$ *then*
e_1 *primop* $e_2 \rightsquigarrow e_1'$ *primop* e_2

(11) *If* $e_2 \rightsquigarrow e_2'$ *then*
v_1 *primop* $e_2 \rightsquigarrow v_1$ *primop* e_2'

Example - continued

```
⇝[2,4]
let val p = 2
in (fn x => if x < 0 then 0 else x) (p - 3)
end

⇝[1] (fn x => if x < 0 then 0 else x) (2 - 3)

⇝[8,9] (fn x => if x < 0 then 0 else x) (~1)

⇝[6] if ~1 < 0 then 0 else ~1

⇝[5,9] if true then 0 else ~1

⇝[3] 0
```

Substitition Revisited

The correct definition of substitution is a little tricky.

Basic idea: only substitute for **free** variables, i.e., those not redefined by a `let` or `fn` binding.

```
let val a = 10
in let val b = a+10
   in let val a = b+20
      in a+30
   end
end
```

$\rightsquigarrow^1$

```
let val b = 10+10
in let val a = b+20
   in a+30
   end
end
```

$\rightsquigarrow \rightsquigarrow$ 70

NOT $\rightsquigarrow^1$

```
let val b = 10+10
in let val a = b+20
   in 10+30
   end
end
```

$\rightsquigarrow \rightsquigarrow$ 40