```
(: subst : WAE Symbol WAE -> WAE)
;; substitutes the second argument with the third argument in the
;; first argument, as per the rules of substitution; the resulting
;; expression contains no free instances of the second argument
(define (subst expr from to) ; returns expr[to/from]
  (cases expr
    [(Num n) expr]
    [(Add l r) (Add (subst l from to) (subst r from to))]
    [(Sub l r) (Sub (subst l from to) (subst r from to))]
    [(Mul l r) (Mul (subst l from to) (subst r from to))]
    [(Div l r) (Div (subst l from to) (subst r from to))]
    [(Id name) (if (eq? name from) to expr)]
    [(With bound-id named-expr bound-body)
     (if (eq? bound-id from)
       expr           ; <-- don't go in!
       (With bound-id
             named-expr
             (subst bound-body from to)))]))
```

... and this is just the same as writing a formal "paper version" of the substitution rule.

... but we still have bugs!

Before we find the bugs, let us see the bigger context…

When and How substitution is used in the evaluation process.

Modifying our evaluator:

We will need rules to deal with the new syntax pieces -- `with' expressions and identifiers.

- Evaluating `with' expressions – for an expression of the following form:

  {with {x E1} E2}

  We need to:
  1. *evaluate* `E1' to get a **value** `V1',
  2. we then substitute the identifier `x' with the expression `V1' in `E2', and
  3. finally, we evaluate this resulting new expression.

  In other words, we have the following evaluation rule:

  eval({with {x E1} E2}) = eval(E2[eval(E1)/x])

  So we know what to do with `with' expressions.

- Evaluating **identifiers** –

  The main feature of `subst', as said in the purpose statement, is that **it leaves no free instances of the substituted variable**

around. This means that if the initial expression is valid (did not contain any free variables), then when we go from

```
{with {x E1} E2}
```

to

```
E2[E1/x]
```

Then, the result is an expression that has ***no* free instances of** `x'. So we don't need to handle identifiers in the evaluator -- substitutions make them all go away.

We can now extend the formal definition of **AE** to that of **WAE**:

```
eval(...) = ... same as the AE rules ...
eval({with {x E1} E2}) = eval(E2[eval(E1)/x])
eval(id) = error!
```

Important to note: (*** what should we feed to `subst')
If you're paying close attention, you might catch a potential problem in this definition. We're substituting `eval(E1)' for `x' in `E2'

- `subst' **requires a WAE expression**. However,
- **what we are really getting** is `eval(E1)' which **is a number**.

(Look at the type of the `eval' definition we had for **AE**, then look at the above definition of `subst'.) This seems like being overly pedantic, but it will require some resolution when we get to the code.

The above rules are easily coded as follows:

```
(: eval : WAE -> Number)
;; evaluates WAE expressions by reducing them to numbers
(define (eval expr)
  (cases expr
    [(Num n) n]                                ;; same as before
    [(Add l r) (+ (eval l) (eval r))]          ;; same as before
    [(Sub l r) (- (eval l) (eval r))]          ;; same as before
    [(Mul l r) (* (eval l) (eval r))]          ;; same as before
    [(Div l r) (/ (eval l) (eval r))]          ;; same as before
    [(With bound-id named-expr bound-body)
     (eval (subst bound-body
                  bound-id
                  (Num (eval named-expr))))] ; <-*** (see note)
    [(Id name) (error 'eval "free identifier: ~s" name)]))
```

Note the `Num' expression in the marked line: evaluating the named expression gives us back a number -- we need to convert this number into an abstract syntax (**WAE**) to be able to use it with `subst'.

The solution is to use the constructor `**Num**' to convert the resulting number into a numeral (the syntax of a number). It's not an elegant solution, but it will do for now.

- -

A few test cases:
We use a new `**test**' special form which is part of the course plugin. The way to use ` **test**' is with two expressions and an `=>' arrow -- DrRacket evaluates both, and nothing will happen if the results are equal. If the results are different, you will get a warning line, but evaluation will continue so you can try additional tests. You can also use an `=error>' arrow to test an error message -- use it with some text from the expected error, `?' stands for any single character, and `*' is a sequence of zero or more characters.
(When you use `test' in your homework, the handin server will abort when tests fail.) We expect these tests to succeed (make sure that you understand *why* they should succeed).

```
;; tests
(test (run "5") => 5)
(test (run "{+ 5 5}") => 10)
(test (run "{with {x {+ 5 5}} {+ x x}}") => 20)
(test (run "{with {x 5} {+ x x}}") => 10)
(test (run "{with {x {+ 5 5}} {with {y {- x 3}} {+ y y}}}") => 14)
(test (run "{with {x 5} {with {y {- x 3}} {+ y y}}}") => 4)
(test (run "{with {x 5} {+ x {with {x 3} 10}}}") => 15)
(test (run "{with {x 5} {+ x {with {x 3} x}}}") => 8)
(test (run "{with {x 5} {+ x {with {y 3} x}}}") => 10)
(test (run "{with {x 5} {with {y x} y}}") => 5)
(test (run "{with {x 5} {with {x x} x}}") => 5)
(test (run "{with {x 1} y}") =error> "free identifier")
```

Putting this all together, we get the following code; trying to run this code will raise an unexpected error...

```
------------------------------------------------------------------------

#lang pl

#| BNF for the WAE language:
     <WAE> ::= <num>
             | { + <WAE> <WAE> }
             | { - <WAE> <WAE> }
             | { * <WAE> <WAE> }
             | { / <WAE> <WAE> }
             | { with { <id> <WAE> } <WAE> }
             | <id>
|#

;; WAE abstract syntax trees
(define-type WAE
  [Num  Number]
  [Add  WAE WAE]
  [Sub  WAE WAE]
  [Mul  WAE WAE]
  [Div  WAE WAE]
  [Id   Symbol]
  [With Symbol WAE WAE])

(: parse-sexpr : Sexpr -> WAE)
;; to convert s-expressions into WAEs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n)    (Num n)]
    [(symbol: name) (Id name)]
    [(cons 'with more)
     (match sexpr
       [(list 'with (list (symbol: name) named) body)
        (With name (parse-sexpr named) (parse-sexpr body))]
       [else (error 'parse-sexpr "bad `with' syntax in ~s" sexpr)])]
    [(list '+ lhs rhs) (Add (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '- lhs rhs) (Sub (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '* lhs rhs) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '/ lhs rhs) (Div (parse-sexpr lhs) (parse-sexpr rhs))]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))

(: parse : String -> WAE)
;; parses a string containing a WAE expression to a WAE AST
(define (parse str)
  (parse-sexpr (string->sexpr str)))

(: subst : WAE Symbol WAE -> WAE)
;; substitutes the second argument with the third argument in the
;; first argument, as per the rules of substitution; the resulting
;; expression contains no free instances of the second argument
(define (subst expr from to)
  (cases expr
    [(Num n) expr]
    [(Add l r) (Add (subst l from to) (subst r from to))]
```

```
      [(Sub l r) (Sub (subst l from to) (subst r from to))]
      [(Mul l r) (Mul (subst l from to) (subst r from to))]
      [(Div l r) (Div (subst l from to) (subst r from to))]
      [(Id name) (if (eq? name from) to expr)]
      [(With bound-id named-expr bound-body)
       (if (eq? bound-id from)
         expr
         (With bound-id
               named-expr
               (subst bound-body from to)))]))

  (: eval : WAE -> Number)
  ;; evaluates WAE expressions by reducing them to numbers
  (define (eval expr)
    (cases expr
      [(Num n) n]
      [(Add l r) (+ (eval l) (eval r))]
      [(Sub l r) (- (eval l) (eval r))]
      [(Mul l r) (* (eval l) (eval r))]
      [(Div l r) (/ (eval l) (eval r))]
      [(With bound-id named-expr bound-body)
       (eval (subst bound-body
                    bound-id
                    (Num (eval named-expr))))]
      [(Id name) (error 'eval "free identifier: ~s" name)]))

  (: run : String -> Number)
  ;; evaluate a WAE program contained in a string
  (define (run str)
    (eval (parse str)))

  ;; tests
  (test (run "5") => 5)
  (test (run "{+ 5 5}") => 10)
  (test (run "{with {x {+ 5 5}} {+ x x}}") => 20)
  (test (run "{with {x 5} {+ x x}}") => 10)
  (test (run "{with {x {+ 5 5}} {with {y {- x 3}} {+ y y}}}") => 14)
;; in reality returns -- eval: free identifier: x
  (test (run "{with {x 5} {with {y {- x 3}} {+ y y}}}") => 4)
  (test (run "{with {x 5} {+ x {with {x 3} 10}}}") => 15)
  (test (run "{with {x 5} {+ x {with {x 3} x}}}") => 8)
  (test (run "{with {x 5} {+ x {with {y 3} x}}}") => 10)
  (test (run "{with {x 5} {with {y x} y}}") => 5)
  (test (run "{with {x 5} {with {x x} x}}") => 5)
  (test (run "{with {x 1} y}") =error> "free identifier")
  ----------------------------------------------------------------------
```

Oops, this program **still has problems** that were caught by the tests -- we encounter unexpected free identifier errors. What's the problem now?
In expressions like:

```
  {with {x 5}
    {with {y x}
      y}}
```

The problem:
We forgot to substitute `x' in the expression that `y' is bound to. We need to perform the recursive substitution in two places:

1. **The "with"'s body expression** (which we already have), and
2. **The "with"'s named expression.**

The new **'subst'** code will be:

```
(: subst : WAE Symbol WAE -> WAE)
;; substitutes the second argument with the third argument in the
;; first argument, as per the rules of substitution; the resulting
;; expression contains no free instances of the second argument
(define (subst expr from to)
  (cases expr
    [(Num n) expr]
    [(Add l r) (Add (subst l from to) (subst r from to))]
    [(Sub l r) (Sub (subst l from to) (subst r from to))]
    [(Mul l r) (Mul (subst l from to) (subst r from to))]
    [(Div l r) (Div (subst l from to) (subst r from to))]
    [(Id name) (if (eq? name from) to expr)]
    [(With bound-id named-expr bound-body)
     (if (eq? bound-id from)
       expr
       (With bound-id
             (subst named-expr from to)        ; <-- new
             (subst bound-body from to)))]))
```

However, we *still* have a problem...

Look at the expression:

```
{with {x 5}
  {with {x x}
    x}}
```

Halts with an error -- **eval: free identifier: x**
It should, however, evaluate to **5**. Carefully trying out our substitution code reveals **the problem**:
We do not go inside the inner `with'** when we substitute `5' for the outer `x'. This is because it has the **same name 'x' – but we *do* need to go into its named expression.
We need to substitute in the named expression even if the identifier has the *same* name as the one we're substituting:

```
(: subst : WAE Symbol WAE -> WAE)
;; substitutes the second argument with the third argument in the
;; first argument, as per the rules of substitution; the resulting
;; expression contains no free instances of the second argument
(define (subst expr from to)
  (cases expr
    [(Num n) expr]
    [(Add l r) (Add (subst l from to) (subst r from to))]
    [(Sub l r) (Sub (subst l from to) (subst r from to))]
    [(Mul l r) (Mul (subst l from to) (subst r from to))]
    [(Div l r) (Div (subst l from to) (subst r from to))]
    [(Id name) (if (eq? name from) to expr)]
```

```
     [(With bound-id named-expr bound-body)
      (With bound-id
            (subst named-expr from to)
            (if (eq? bound-id from) ;; new - only ask on body
              bound-body
              (subst bound-body from to)))]))
```

The complete (and, finally, correct) version of the code is now:

```
---<<<WAE>>>-----------------------------------------------------------

#lang pl

#| BNF for the WAE language:
     <WAE> ::= <num>
             | { + <WAE> <WAE> }
             | { - <WAE> <WAE> }
             | { * <WAE> <WAE> }
             | { / <WAE> <WAE> }
             | { with { <id> <WAE> } <WAE> }
             | <id>
|#

;; WAE abstract syntax trees
(define-type WAE
  [Num  Number]
  [Add  WAE WAE]
  [Sub  WAE WAE]
  [Mul  WAE WAE]
  [Div  WAE WAE]
  [Id   Symbol]
  [With Symbol WAE WAE])

(: parse-sexpr : Sexpr -> WAE)
;; to convert s-expressions into WAEs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n)    (Num n)]
    [(symbol: name) (Id name)]
    [(cons 'with more)
     (match sexpr
       [(list 'with (list (symbol: name) named) body)
        (With name (parse-sexpr named) (parse-sexpr body))]
       [else (error 'parse-sexpr "bad `with' syntax in ~s" sexpr)])]
    [(list '+ lhs rhs) (Add (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '- lhs rhs) (Sub (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '* lhs rhs) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '/ lhs rhs) (Div (parse-sexpr lhs) (parse-sexpr rhs))]
    [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))

(: parse : String -> WAE)
;; parses a string containing a WAE expression to a WAE AST
(define (parse str)
  (parse-sexpr (string->sexpr str)))

#| Formal specs for `subst':
   (`N' is a <num>, `E1', `E2' are <WAE>s, `x' is some <id>, `y' is a
```

```
   *different* <id>)
      N[v/x]                 = N
      {+ E1 E2}[v/x]         = {+ E1[v/x] E2[v/x]}
      {- E1 E2}[v/x]         = {- E1[v/x] E2[v/x]}
      {* E1 E2}[v/x]         = {* E1[v/x] E2[v/x]}
      {/ E1 E2}[v/x]         = {/ E1[v/x] E2[v/x]}
      y[v/x]                 = y
      x[v/x]                 = v
      {with {y E1} E2}[v/x] = {with {y E1[v/x]} E2[v/x]}
      {with {x E1} E2}[v/x] = {with {x E1[v/x]} E2}
|#

(: subst : WAE Symbol WAE -> WAE)
;; substitutes the second argument with the third argument in the
;; first argument, as per the rules of substitution; the resulting
;; expression contains no free instances of the second argument
(define (subst expr from to)
  (cases expr
    [(Num n) expr]
    [(Add l r) (Add (subst l from to) (subst r from to))]
    [(Sub l r) (Sub (subst l from to) (subst r from to))]
    [(Mul l r) (Mul (subst l from to) (subst r from to))]
    [(Div l r) (Div (subst l from to) (subst r from to))]
    [(Id name) (if (eq? name from) to expr)]
    [(With bound-id named-expr bound-body)
     (With bound-id
           (subst named-expr from to)
           (if (eq? bound-id from)
             bound-body
             (subst bound-body from to)))]))

#| Formal specs for `eval':
     eval(N)          = N
     eval({+ E1 E2}) = eval(E1) + eval(E2)
     eval({- E1 E2}) = eval(E1) - eval(E2)
     eval({* E1 E2}) = eval(E1) * eval(E2)
     eval({/ E1 E2}) = eval(E1) / eval(E2)
     eval(id)         = error!
     eval({with {x E1} E2}) = eval(E2[eval(E1)/x])
|#

(: eval : WAE -> Number)
;; evaluates WAE expressions by reducing them to numbers
(define (eval expr)
  (cases expr
    [(Num n) n]
    [(Add l r) (+ (eval l) (eval r))]
    [(Sub l r) (- (eval l) (eval r))]
    [(Mul l r) (* (eval l) (eval r))]
    [(Div l r) (/ (eval l) (eval r))]
    [(With bound-id named-expr bound-body)
     (eval (subst bound-body
                  bound-id
                  (Num (eval named-expr))))]
    [(Id name) (error 'eval "free identifier: ~s" name)]))

(: run : String -> Number)
```

```
  ;; evaluate a WAE program contained in a string
  (define (run str)
    (eval (parse str)))

  ;; tests
  (test (run "5") => 5)
  (test (run "{+ 5 5}") => 10)
  (test (run "{with {x {+ 5 5}} {+ x x}}") => 20)
  (test (run "{with {x 5} {+ x x}}") => 10)
  (test (run "{with {x {+ 5 5}} {with {y {- x 3}} {+ y y}}}") => 14)
  (test (run "{with {x 5} {with {y {- x 3}} {+ y y}}}") => 4)
  (test (run "{with {x 5} {+ x {with {x 3} 10}}}") => 15)
  (test (run "{with {x 5} {+ x {with {x 3} x}}}") => 8)
  (test (run "{with {x 5} {+ x {with {y 3} x}}}") => 10)
  (test (run "{with {x 5} {with {y x} y}}") => 5)
  (test (run "{with {x 5} {with {x x} x}}") => 5)
  (test (run "{with {x 1} y}") =error> "free identifier")
  ----------------------------------------------------------------------
-
```

Reminder:

* We started doing substitution, with a `let'-like form: `with'.

* Reasons for using bindings:
  - Avoid writing expressions twice.
    -> More expressive language (can express identity).
    -> Duplicating is bad! (=> DRY, Don't Repeat Yourself)
    --> Static redundancy.
  - Avoid redundant computations.
    --> Dynamic redundancy.

* BNF:

```
  <WAE> ::= <num>
          | { + <WAE> <WAE> }
          | { - <WAE> <WAE> }
          | { * <WAE> <WAE> }
          | { / <WAE> <WAE> }
          | { with { <id> <WAE> } <WAE> }
          | <id>
```

  Note that we had to introduce two new rules: one for introducing an identifier, and one for using it.

* Type definition:

```
  (define-type WAE
    [Num  Number]
    [Add  WAE WAE]
    [Sub  WAE WAE]
    [Mul  WAE WAE]
    [Div  WAE WAE]
    [Id   Symbol]
    [With Symbol WAE WAE])
```

* Parser:

```
  (: parse-sexpr : Sexpr -> WAE)
  ;; to convert s-expressions into WAEs
  (define (parse-sexpr sexpr)
    (match sexpr
      [(number: n)    (Num n)]
      [(symbol: name) (Id name)]
      [(cons 'with more)
       (match sexpr
         [(list 'with (list (symbol: name) named) body)
          (With name (parse-sexpr named) (parse-sexpr body))]
         [else (error 'parse-sexpr "bad `with' syntax in ~s" sexpr)])]
      [(list '+ lhs rhs) (Add (parse-sexpr lhs) (parse-sexpr rhs))]
      [(list '- lhs rhs) (Sub (parse-sexpr lhs) (parse-sexpr rhs))]
      [(list '* lhs rhs) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]
      [(list '/ lhs rhs) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]
      [else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))
```

* We need to define substitution.
  Terms:
  1. Binding Instance.
  2. Scope.
  3. Bound Instance.
  4. Free Instance.

* After lots of attempts:

```
  e[v/i] -- To substitute an identifier `i' in an expression `e' with
an
  expression `v', replace all instances of `i' that are free in `e'
with
  the expression `v'.
```

* Implemented the code, and again, needed to fix a few bugs:

```
  (: subst : WAE Symbol WAE -> WAE)
  ;; substitutes the second argument with the third argument in the
  ;; first argument, as per the rules of substitution; the resulting
  ;; expression contains no free instances of the second argument
  (define (subst expr from to)
    (cases expr
      [(Num n) expr]
      [(Add l r) (Add (subst l from to) (subst r from to))]
      [(Sub l r) (Sub (subst l from to) (subst r from to))]
      [(Mul l r) (Mul (subst l from to) (subst r from to))]
      [(Div l r) (Div (subst l from to) (subst r from to))]
      [(Id name) (if (eq? name from) to expr)]
      [(With bound-id named-expr bound-body)
       (With bound-id
             (subst named-expr from to)
             (if (eq? bound-id from)
               bound-body
               (subst bound-body from to)))]))
```

(Note that the bugs that we fixed clarify the exact way that our scopes work: in `{with {x 2} {with {x {+ x 2}} x}}', the scope of the first `x' is: ^^^^^^^)

* We then extended the AE evaluation rules:

```
eval(...) = ... same as the AE rules ...
eval({with {x E1} E2}) = eval(E2[eval(E1)/x])
eval(id) = error!
```

and noted the possible type problem.

* The above translated into a Racket definition for an `eval' function (with a hack to avoid the type issue):

```
(: eval : WAE -> Number)
;; evaluates WAE expressions by reducing them to numbers
(define (eval expr)
  (cases expr
    [(Num n) n]
    [(Add l r) (+ (eval l) (eval r))]
    [(Sub l r) (- (eval l) (eval r))]
    [(Mul l r) (* (eval l) (eval r))]
    [(Div l r) (/ (eval l) (eval r))]
    [(With bound-id named-expr bound-body)
     (eval (subst bound-body
                  bound-id
                  (Num (eval named-expr))))]
    [(Id name) (error 'eval "free identifier: ~s" name)]))
```