PL: Lecture #25

Tuesday, November 26th

Improving Picky

The following version does that, there are no types mentioned except for the input type for a function. Note that we can do that at this point because our language is so simple that many pieces of code have a specific type. (For example, if we add polymorphism things get more complicated.)

```
;; The Picky interpreter, almost no explicit types

#lang pl

#|
The grammar:
  <PICKY> ::= <num>
            | <id>
            | { + <PICKY> <PICKY> }
            | { - <PICKY> <PICKY> }
            | { = <PICKY> <PICKY> }
            | { < <PICKY> <PICKY> }
            | { fun { <id> : <TYPE> } <PICKY> }
            | { call <PICKY> <PICKY> }
            | { with { <id> <PICKY> } <PICKY> }
            | { if <PICKY> <PICKY> <PICKY> }
  <TYPE>  ::= Num | Number
            | Bool | Boolean
            | { <TYPE> -> <TYPE> }

Evaluation rules:
  eval(N,env)                 = N
  eval(x,env)                 = lookup(x,env)
  eval({+ E1 E2},env)         = eval(E1,env) + eval(E2,env)
  eval({- E1 E2},env)         = eval(E1,env) - eval(E2,env)
  eval({= E1 E2},env)         = eval(E1,env) = eval(E2,env)
  eval({< E1 E2},env)         = eval(E1,env) < eval(E2,env)
  eval({fun {x} E},env)       = <{fun {x} E}, env>
  eval({call E1 E2},env1)     = eval(B,extend(x,eval(E2,env1),env2))
                                 if eval(E1,env1) = <{fun {x} B}, env2>
                              = error!  otherwise <-- never happens
  eval({with {x E1} E2},env) = eval(E2,extend(x,eval(E1,env),env))
  eval({if E1 E2 E3},env)     = eval(E2,env)  if eval(E1,env) is true
                              = eval(E3,env)  otherwise

Type checking rules (note how implicit types are made):

  Γ ⊢ n : Number

  Γ ⊢ x : Γ(x)

  Γ ⊢ A : Number   Γ ⊢ B : Number
  ------------------------------
       Γ ⊢ {+ A B} : Number
```

```
  Γ ⊢ A : Number   Γ ⊢ B : Number
  ------------------------------
       Γ ⊢ {< A B} : Boolean

          Γ[x:=τ₁] ⊢ E : τ₂
  ----------------------------------
  Γ ⊢ {fun {x : τ₁} E} : (τ₁ -> τ₂)

  Γ ⊢ F : (τ₁ -> τ₂)  Γ ⊢ V : τ₁
  ------------------------------
       Γ ⊢ {call F V} : τ₂

  Γ ⊢ V : τ₁   Γ[x:=τ₁] ⊢ E : τ₂
  ------------------------------
     Γ ⊢ {with {x V} E} : τ₂

  Γ ⊢ C : Boolean   Γ ⊢ T : τ   Γ ⊢ E : τ
  ---------------------------------------
            Γ ⊢ {if C T E} : τ
```

```
|#

(define-type PICKY
  [Num   Number]
  [Id    Symbol]
  [Add   PICKY PICKY]
  [Sub   PICKY PICKY]
  [Equal PICKY PICKY]
  [Less  PICKY PICKY]
  [Fun   Symbol TYPE PICKY]  ; no output type
  [Call  PICKY PICKY]
  [With  Symbol PICKY PICKY] ; no types here
  [If    PICKY PICKY PICKY])

(define-type TYPE
  [NumT]
  [BoolT]
  [FunT TYPE TYPE])

(: parse-sexpr : Sexpr -> PICKY)
;; parses s-expressions into PICKYs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n)    (Num n)]
    [(symbol: name) (Id name)]
    [(list '+ lhs rhs) (Add   (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '- lhs rhs) (Sub   (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '= lhs rhs) (Equal (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '< lhs rhs) (Less  (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list 'call fun arg)
                       (Call  (parse-sexpr fun) (parse-sexpr arg))]
    [(list 'if c t e)
     (If (parse-sexpr c) (parse-sexpr t) (parse-sexpr e))]
    [(cons 'fun more)
     (match sexpr
       [(list 'fun (list (symbol: name) ': itype) body)
```

```
         (Fun name (parse-type-sexpr itype) (parse-sexpr body))]
        [else (error 'parse-sexpr "bad `fun' syntax in ~s" sexpr)])]
    [(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)])]
    [else (error 'parse-sexpr "bad expression syntax: ~s" sexpr)]))

(: parse-type-sexpr : Sexpr -> TYPE)
;; parses s-expressions into TYPEs
(define (parse-type-sexpr sexpr)
  (match sexpr
    ['Number  (NumT)]
    ['Boolean (BoolT)]
    ;; allow shorter names too
    ['Num  (NumT)]
    ['Bool (BoolT)]
    [(list itype '-> otype)
     (FunT (parse-type-sexpr itype) (parse-type-sexpr otype))]
    [else (error 'parse-type-sexpr "bad type syntax in ~s" sexpr)]))

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

;; Typechecker and related types and helpers

;; this is similar to ENV, but it holds type information for the
;; identifiers during typechecking; it is essentially "Γ"
(define-type TYPEENV
  [EmptyTypeEnv]
  [ExtendTypeEnv Symbol TYPE TYPEENV])

(: type-lookup : Symbol TYPEENV -> TYPE)
;; similar to `lookup' for type environments; note that the
;; error is phrased as a typecheck error, since this indicates
;; a failure at the type checking stage
(define (type-lookup name typeenv)
  (cases typeenv
    [(EmptyTypeEnv) (error 'typecheck "no binding for ~s" name)]
    [(ExtendTypeEnv id type rest-env)
     (if (eq? id name) type (type-lookup name rest-env))]))

(: typecheck : PICKY TYPE TYPEENV -> Void)
;; Checks that the given expression has the specified type.
;; Used only for side-effects (to throw a type error), so return
;; a void value.
(define (typecheck expr type type-env)
  (unless (equal? type (typecheck* expr type-env))
    (error 'typecheck "type error for ~s: expecting a ~s"
           expr type)))

(: typecheck* : PICKY TYPEENV -> TYPE)
;; Returns the type of the given expression (which also means that
;; it checks it). This is a helper for the real typechecker that
;; also checks a specific return type.
```

```
(define (typecheck* expr type-env)
  (: two-nums : PICKY PICKY -> Void)
  (define (two-nums e1 e2)
    (typecheck e1 (NumT) type-env)
    (typecheck e2 (NumT) type-env))
  (cases expr
    [(Num n) (NumT)]
    [(Id name) (type-lookup name type-env)]
    [(Add   l r) (two-nums l r) (NumT)]
    [(Sub   l r) (two-nums l r) (NumT)]
    [(Equal l r) (two-nums l r) (BoolT)]
    [(Less  l r) (two-nums l r) (BoolT)]
    [(Fun bound-id in-type bound-body)
     (FunT in-type
           (typecheck* bound-body
                       (ExtendTypeEnv bound-id in-type type-env)))]
    [(Call fun arg)
     (cases (typecheck* fun type-env)
       [(FunT in-type out-type)
        (typecheck arg in-type type-env)
        out-type]
       [else (error 'typecheck "type error for ~s: expecting a fun"
                    expr)])]
    [(With bound-id named-expr bound-body)
     (typecheck* bound-body
                 (ExtendTypeEnv bound-id
                                (typecheck* named-expr type-env)
                                type-env))]
    [(If cond-expr then-expr else-expr)
     (typecheck cond-expr (BoolT) type-env)
     (let ([type (typecheck* then-expr type-env)])
       (typecheck else-expr type type-env) ; enforce same type
       type)]))

;; Evaluator and related types and helpers

(define-type ENV
  [EmptyEnv]
  [Extend Symbol VAL ENV])

(define-type VAL
  [NumV  Number]
  [BoolV Boolean]
  [FunV  Symbol PICKY ENV])

(: lookup : Symbol ENV -> VAL)
;; lookup a symbol in an environment, return its value or throw an
;; error if it isn't bound
(define (lookup name env)
  (cases env
    [(EmptyEnv) (error 'lookup "no binding for ~s" name)]
    [(Extend id val rest-env)
     (if (eq? id name) val (lookup name rest-env))]))

(: strip-numv : Symbol VAL -> Number)
;; converts a VAL to a Racket number if possible, throws an error if
;; not using the given name for the error message
(define (strip-numv name val)
```

```
  (cases val
    [(NumV n) n]
    ;; this error will never be reached, see below for more
    [else (error name "expected a number, got: ~s" val)]))

(: arith-op : (Number Number -> Number) VAL VAL -> VAL)
;; gets a Racket numeric binary operator, and uses it within a NumV
;; wrapper
(define (arith-op op val1 val2)
  (NumV (op (strip-numv 'arith-op val1)
            (strip-numv 'arith-op val2))))

(: bool-op : (Number Number -> Boolean) VAL VAL -> VAL)
;; gets a Racket numeric binary predicate, and uses it
;; within a BoolV wrapper
(define (bool-op op val1 val2)
  (BoolV (op (strip-numv 'bool-op val1)
             (strip-numv 'bool-op val2))))

(: eval : PICKY ENV -> VAL)
;; evaluates PICKY expressions by reducing them to values
(define (eval expr env)
  (cases expr
    [(Num n) (NumV n)]
    [(Id name) (lookup name env)]
    [(Add   l r) (arith-op + (eval l env) (eval r env))]
    [(Sub   l r) (arith-op - (eval l env) (eval r env))]
    [(Equal l r) (bool-op  = (eval l env) (eval r env))]
    [(Less  l r) (bool-op  < (eval l env) (eval r env))]
    [(Fun bound-id in-type bound-body)
     ;; note that types are not used at runtime,
     ;; so they're not stored in the closure
     (FunV bound-id bound-body env)]
    [(Call fun-expr arg-expr)
     (define fval (eval fun-expr env))
     (cases fval
       [(FunV bound-id bound-body f-env)
        (eval bound-body
              (Extend bound-id (eval arg-expr env) f-env))]
       ;; `cases' requires complete coverage of all variants, but
       ;; this `else' is never used since we typecheck programs
       [else (error 'eval "`call' expects a function, got: ~s"
                    fval)])]
    [(With bound-id named-expr bound-body)
     (eval bound-body (Extend bound-id (eval named-expr env) env))]
    [(If cond-expr then-expr else-expr)
     (let ([bval (eval cond-expr env)])
       (if (cases bval
             [(BoolV b) b]
             ;; same as above: this case is never reached
             [else (error 'eval "`if' expects a boolean, got: ~s"
                          bval)])
         (eval then-expr env)
         (eval else-expr env)))]))

(: run : String -> Number)
;; evaluate a PICKY program contained in a string
(define (run str)
```

```
  (let ([prog (parse str)])
    (typecheck prog (NumT) (EmptyTypeEnv))
    (let ([result (eval prog (EmptyEnv))])
      (cases result
        [(NumV n) n]
        ;; this error is never reached, since we make sure
        ;; that the program always evaluates to a number above
        [else (error 'run "evaluation returned a non-number: ~s"
                     result)]))))

;; tests -- including translations of the FLANG tests
(test (run "5") => 5)
(test (run "{fun {x : Num} {+ x 1}}") =error> "type error")
(test (run "{call {fun {x : Num} {+ x 1}} 4}") => 5)
(test (run "{with {x 3} {+ x 1}}") => 4)
(test (run "{with {identity {fun {x : Num} x}} {call identity 1}}")
      => 1)
(test (run "{with {add3 {fun {x : Num} {+ x 3}}}
              {call add3 1}}")
      => 4)
(test (run "{with {add3 {fun {x : Num} {+ x 3}}}
              {with {add1 {fun {x : Num} {+ x 1}}}
                {with {x 3}
                  {call add1 {call add3 x}}}}}")
      => 7)
(test (run "{with {identity {fun {x : {Num -> Num}} x}}
              {with {foo {fun {x : Num} {+ x 1}}}
                {call {call identity foo} 123}}}")
      => 124)
(test (run "{with {x 3}
              {with {f {fun {y : Num} {+ x y}}}
                {with {x 5} {call f 4}}}}")
      => 7)
(test (run "{call {with {x 3} {fun {y : Num} {+ x y}}} 4}")
      => 7)
(test (run "{with {f {with {x 3} {fun {y : Num} {+ x y}}}}
              {with {x 100}
                {call f 4}}}")
      => 7)
(test (run "{call {call {fun {x : {Num -> {Num -> Num}}} {call x 1}}
                        {fun {x : Num} {fun {y : Num} {+ x y}}}}
                  123}")
      => 124)
(test (run "{call {fun {x : Num} {if {< x 2} {+ x 5} {+ x 6}}} 1}")
      => 6)
(test (run "{call {fun {x : Num} {if {< x 2} {+ x 5} {+ x 6}}} 2}")
      => 8)
```

Finally, an obvious question is whether we can get rid of *all* of the type declarations. The main point here is that we need to somehow be able to typecheck expressions and assign “temporary types” to them that will later on change — for example, when we typecheck this:

```
{with {identity {fun {x} x}}
  {call identity 1}}
```

we need to somehow decide that the named expression has a general function type, with no commitment on the actual input and output types — and then change them after we typecheck the body. (We could try to resolve that somehow by typechecking the body first, but that will not work, since the body must be checked with *some* type assigned to the identifier, or it will fail.)

Even better...

This can be done using *type variables* — things that contain boxes that can be used to change types as typecheck progresses. The following version does that. (Also, it gets rid of the `typecheck*` thing, since it can be achieved by using a type-variable and a call to `typecheck`.) Note the interesting tests at the end.

```
;; The Picky interpreter, no explicit types

#lang pl

#|
The grammar:
  <PICKY> ::= <num>
            | <id>
            | { + <PICKY> <PICKY> }
            | { - <PICKY> <PICKY> }
            | { = <PICKY> <PICKY> }
            | { < <PICKY> <PICKY> }
            | { fun { <id> } <PICKY> }
            | { call <PICKY> <PICKY> }
            | { with { <id> <PICKY> } <PICKY> }
            | { if <PICKY> <PICKY> <PICKY> }

The types are no longer part of the input syntax.

Evaluation rules:
  eval(N,env)                = N
  eval(x,env)                = lookup(x,env)
  eval({+ E1 E2},env)        = eval(E1,env) + eval(E2,env)
  eval({- E1 E2},env)        = eval(E1,env) - eval(E2,env)
  eval({= E1 E2},env)        = eval(E1,env) = eval(E2,env)
  eval({< E1 E2},env)        = eval(E1,env) < eval(E2,env)
  eval({fun {x} E},env)      = <{fun {x} E}, env>
  eval({call E1 E2},env1)    = eval(B,extend(x,eval(E2,env1),env2))
                               if eval(E1,env1) = <{fun {x} B}, env2>
                             = error!  otherwise <-- never happens
  eval({with {x E1} E2},env) = eval(E2,extend(x,eval(E1,env),env))
  eval({if E1 E2 E3},env)    = eval(E2,env)  if eval(E1,env) is true
                             = eval(E3,env)  otherwise

Type checking rules (note the extra complexity in the `fun' rule):

  Γ ⊢ n : Number

  Γ ⊢ x : Γ(x)

  Γ ⊢ A : Number   Γ ⊢ B : Number
  ------------------------------
       Γ ⊢ {+ A B} : Number

  Γ ⊢ A : Number   Γ ⊢ B : Number
  ------------------------------
       Γ ⊢ {< A B} : Boolean

       Γ[x:=τ₁] ⊢ E : τ₂
  ----------------------------
```

```
  Γ ⊢ {fun {x} E} : (τ₁ -> τ₂)

  Γ ⊢ F : (τ₁ -> τ₂)  Γ ⊢ V : τ₁
  ------------------------------
       Γ ⊢ {call F V} : τ₂

  Γ ⊢ C : Boolean   Γ ⊢ T : τ   Γ ⊢ E : τ
  ---------------------------------------
            Γ ⊢ {if C T E} : τ

  Γ ⊢ V : τ₁   Γ[x:=τ₁] ⊢ E : τ₂
  ------------------------------
     Γ ⊢ {with {x V} E} : τ₂

|#

(define-type PICKY
  [Num   Number]
  [Id    Symbol]
  [Add   PICKY PICKY]
  [Sub   PICKY PICKY]
  [Equal PICKY PICKY]
  [Less  PICKY PICKY]
  [Fun   Symbol PICKY] ; no types even here
  [Call  PICKY PICKY]
  [With  Symbol PICKY PICKY]
  [If    PICKY PICKY PICKY])

(: parse-sexpr : Sexpr -> PICKY)
;; parses s-expressions into PICKYs
(define (parse-sexpr sexpr)
  (match sexpr
    [(number: n)    (Num n)]
    [(symbol: name) (Id name)]
    [(list '+ lhs rhs) (Add   (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '- lhs rhs) (Sub   (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '= lhs rhs) (Equal (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list '< lhs rhs) (Less  (parse-sexpr lhs) (parse-sexpr rhs))]
    [(list 'call fun arg)
                       (Call  (parse-sexpr fun) (parse-sexpr arg))]
    [(list 'if c t e)
     (If (parse-sexpr c) (parse-sexpr t) (parse-sexpr e))]
    [(cons 'fun more)
     (match sexpr
       [(list 'fun (list (symbol: name)) body)
        (Fun name (parse-sexpr body))]
       [else (error 'parse-sexpr "bad `fun' syntax in ~s" sexpr)])]
    [(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)])]
    [else (error 'parse-sexpr "bad expression syntax: ~s" sexpr)]))

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

```
  (parse-sexpr (string->sexpr str)))

;; Typechecker and related types and helpers

;; this is not a part of the AST now, and it also has a new variant
;; for type variables (see `same-type' for how it's used)
(define-type TYPE
  [NumT]
  [BoolT]
  [FunT TYPE TYPE]
  [?T (Boxof (U TYPE #f))])

;; this is similar to ENV, but it holds type information for the
;; identifiers during typechecking; it is essentially "Γ"
(define-type TYPEENV
  [EmptyTypeEnv]
  [ExtendTypeEnv Symbol TYPE TYPEENV])

(: type-lookup : Symbol TYPEENV -> TYPE)
;; similar to `lookup' for type environments; note that the
;; error is phrased as a typecheck error, since this indicates
;; a failure at the type checking stage
(define (type-lookup name typeenv)
  (cases typeenv
    [(EmptyTypeEnv) (error 'typecheck "no binding for ~s" name)]
    [(ExtendTypeEnv id type rest-env)
     (if (eq? id name) type (type-lookup name rest-env))]))

(: typecheck : PICKY TYPE TYPEENV -> Void)
;; Checks that the given expression has the specified type. Used
;; only for side-effects, so return a void value. There are two
;; side-effects that it can do: throw an error if the input
;; expression doesn't typecheck, and type variables can be mutated
;; once their values are known -- this is done by the `types='
;; utility function that follows.
(define (typecheck expr type type-env)
  ;; convenient helpers
  (: type= : TYPE -> Void)
  (define (type= type2) (types= type type2 expr))
  (: two-nums : PICKY PICKY -> Void)
  (define (two-nums e1 e2)
    (typecheck e1 (NumT) type-env)
    (typecheck e2 (NumT) type-env))
  (cases expr
    [(Num n)     (type= (NumT))]
    [(Id name)   (type= (type-lookup name type-env))]
    [(Add   l r) (type= (NumT))  (two-nums l r)] ; note that the
    [(Sub   l r) (type= (NumT))  (two-nums l r)] ; order in these
    [(Equal l r) (type= (BoolT)) (two-nums l r)] ; things can be
    [(Less  l r) (type= (BoolT)) (two-nums l r)] ; swapped...
    [(Fun bound-id bound-body)
     (let (;; the identity of these type variables is important!
           [itype (?T (box #f))]
           [otype (?T (box #f))])
       (type= (FunT itype otype))
       (typecheck bound-body otype
                  (ExtendTypeEnv bound-id itype type-env)))]
    [(Call fun arg)
```

```
     (let ([type2 (?T (box #f))]) ; same here
       (typecheck arg type2 type-env)
       (typecheck fun (FunT type2 type) type-env))]
    [(With bound-id named-expr bound-body)
     (let ([type2 (?T (box #f))]) ; and here
       (typecheck named-expr type2 type-env)
       (typecheck bound-body type
                  (ExtendTypeEnv bound-id type2 type-env)))]
    [(If cond-expr then-expr else-expr)
     (typecheck cond-expr (BoolT) type-env)
     (typecheck then-expr type type-env)
     (typecheck else-expr type type-env)]))

(: types= : TYPE TYPE PICKY -> Void)
;; Compares the two input types, and throw an error if they don't
;; match. This function is the core of `typecheck', and it is used
;; only for its side-effect. Another side effect in addition to
;; throwing an error is when type variables are present -- they will
;; be mutated in an attempt to make the typecheck succeed. Note that
;; the two type arguments are not symmetric: the first type is the
;; expected one, and the second is the one that the code implies
;; -- but this matters only for the error messages. Also, the
;; expression input is used only for these errors. As the code
;; clearly shows, the main work is done by `same-type' below.
(define (types= type1 type2 expr)
  (unless (same-type type1 type2)
    (error 'typecheck "type error for ~s: expecting ~a, got ~a"
           expr (type->string type1) (type->string type2))))

(: type->string : TYPE -> String)
;; Convert a TYPE to a human readable string,
;; used for error messages
(define (type->string type)
  (format "~s" type)
  ;; The code below would be useful, but unfortunately it doesn't
  ;; work in some cases. To see the problem, try to run the example
  ;; below that applies identity on itself. It's left here so you
  ;; can try it out when you're not running into this problem.
  #|
  (cases type
    [(NumT)  "Num"]
    [(BoolT) "Bool"]
    [(FunT i o)
     (string-append (type->string i) " -> " (type->string o))]
    [(?T box)
     (let ([t (unbox box)])
       (if t (type->string t) "?"))])
  |#)

;; Convenience type to make it possible to have a single `cases'
;; dispatch on two types instead of nesting `cases' in each branch
(define-type 2TYPES [PairT TYPE TYPE])

(: same-type : TYPE TYPE -> Boolean)
;; Compares the two input types, return true or false whether
;; they're the same. The process might involve mutating ?T type
;; variables.
(define (same-type type1 type2)
```

```
  ;; the `PairT' type is only used to conveniently match on both
  ;; types in a single `cases', it's not used in any other way
  (cases (PairT type1 type2)
    ;; flatten the first type, or set it to the second if it's unset
    [(PairT (?T box) type2)
     (let ([t1 (unbox box)])
       (if t1
         (same-type t1 type2)
         (begin (set-box! box type2) #t)))]
    ;; do the same for the second (reuse the above case)
    [(PairT type1 (?T box)) (same-type type2 type1)]
    ;; the rest are obvious
    [(PairT (NumT) (NumT)) #t]
    [(PairT (BoolT) (BoolT)) #t]
    [(PairT (FunT i1 o1) (FunT i2 o2))
     (and (same-type i1 i2) (same-type o1 o2))]
    [else #f]))

;; Evaluator and related types and helpers

(define-type ENV
  [EmptyEnv]
  [Extend Symbol VAL ENV])

(define-type VAL
  [NumV  Number]
  [BoolV Boolean]
  [FunV  Symbol PICKY ENV])

(: lookup : Symbol ENV -> VAL)
;; lookup a symbol in an environment, return its value or throw an
;; error if it isn't bound
(define (lookup name env)
  (cases env
    [(EmptyEnv) (error 'lookup "no binding for ~s" name)]
    [(Extend id val rest-env)
     (if (eq? id name) val (lookup name rest-env))]))

(: strip-numv : Symbol VAL -> Number)
;; converts a VAL to a Racket number if possible, throws an error if
;; not using the given name for the error message
(define (strip-numv name val)
  (cases val
    [(NumV n) n]
    ;; this error will never be reached, see below for more
    [else (error name "expected a number, got: ~s" val)]))

(: arith-op : (Number Number -> Number) VAL VAL -> VAL)
;; gets a Racket numeric binary operator, and uses it within a NumV
;; wrapper
(define (arith-op op val1 val2)
  (NumV (op (strip-numv 'arith-op val1)
            (strip-numv 'arith-op val2))))

(: bool-op : (Number Number -> Boolean) VAL VAL -> VAL)
;; gets a Racket numeric binary predicate, and uses it
;; within a BoolV wrapper
(define (bool-op op val1 val2)
```

```
  (BoolV (op (strip-numv 'bool-op val1)
             (strip-numv 'bool-op val2))))

(: eval : PICKY ENV -> VAL)
;; evaluates PICKY expressions by reducing them to values
(define (eval expr env)
  (cases expr
    [(Num n) (NumV n)]
    [(Id name) (lookup name env)]
    [(Add   l r) (arith-op + (eval l env) (eval r env))]
    [(Sub   l r) (arith-op - (eval l env) (eval r env))]
    [(Equal l r) (bool-op  = (eval l env) (eval r env))]
    [(Less  l r) (bool-op  < (eval l env) (eval r env))]
    [(Fun bound-id bound-body) (FunV bound-id bound-body env)]
    [(Call fun-expr arg-expr)
     (define fval (eval fun-expr env))
     (cases fval
       [(FunV bound-id bound-body f-env)
        (eval bound-body
              (Extend bound-id (eval arg-expr env) f-env))]
       ;; `cases' requires complete coverage of all variants, but
       ;; this `else' is never used since we typecheck programs
       [else (error 'eval "`call' expects a function, got: ~s"
                    fval)])]
    [(With bound-id named-expr bound-body)
     (eval bound-body (Extend bound-id (eval named-expr env) env))]
    [(If cond-expr then-expr else-expr)
     (let ([bval (eval cond-expr env)])
       (if (cases bval
             [(BoolV b) b]
             ;; same as above: this case is never reached
             [else (error 'eval "`if' expects a boolean, got: ~s"
                          bval)])
         (eval then-expr env)
         (eval else-expr env)))]))

(: run : String -> Number)
;; evaluate a PICKY program contained in a string
(define (run str)
  (let ([prog (parse str)])
    (typecheck prog (NumT) (EmptyTypeEnv))
    (let ([result (eval prog (EmptyEnv))])
      (cases result
        [(NumV n) n]
        ;; this error is never reached, since we make sure
        ;; that the program always evaluates to a number above
        [else (error 'run "evaluation returned a non-number: ~s"
                     result)]))))

;; tests -- including translations of the FLANG tests
(test (run "5") => 5)
(test (run "{fun {x} {+ x 1}}") =error> "type error")
(test (run "{call {fun {x} {+ x 1}} 4}") => 5)
(test (run "{with {x 3} {+ x 1}}") => 4)
(test (run "{with {identity {fun {x} x}} {call identity 1}}") => 1)
(test (run "{with {add3 {fun {x} {+ x 3}}} {call add3 1}}") => 4)
(test (run "{with {add3 {fun {x} {+ x 3}}}
              {with {add1 {fun {x} {+ x 1}}}
```

```
                {with {x 3}
                  {call add1 {call add3 x}}}}}")
      => 7)
(test (run "{with {identity {fun {x} x}}
              {with {foo {fun {x} {+ x 1}}}
                {call {call identity foo} 123}}}")
      => 124)
(test (run "{with {x 3}
              {with {f {fun {y} {+ x y}}}
                {with {x 5} {call f 4}}}}")
      => 7)
(test (run "{call {with {x 3} {fun {y} {+ x y}}} 4}")
      => 7)
(test (run "{with {f {with {x 3} {fun {y} {+ x y}}}}
              {with {x 100}
                {call f 4}}}")
      => 7)
(test (run "{call {call {fun {x} {call x 1}}
                        {fun {x} {fun {y} {+ x y}}}}
                  123}")
      => 124)
(test (run "{call {fun {x} {if {< x 2} {+ x 5} {+ x 6}}} 1}") => 6)
(test (run "{call {fun {x} {if {< x 2} {+ x 5} {+ x 6}}} 2}") => 8)

;; Note that we still have a language with the same type system,
;; even though it looks like it could be more flexible -- for
;; example, the following two examples work:
(test (run "{with {identity {fun {x} x}}
              {call identity 1}}")
      => 1)
(test (run "{with {identity {fun {x} x}}
              {if {call identity {< 1 2}} 1 2}}")
      => 1)
;; but this doesn't, since identity can not be used with different
;; types:
(test (run "{with {identity {fun {x} x}}
              {if {call identity {< 1 2}}
                {call identity 1}
                2}}")
      =error> "type error")
;; this doesn't work either -- with an interesting error message:
(test (run "{with {identity {fun {x} x}}
              {call {call identity identity} 1}}")
      =error> "type error")
;; ... but these two work fine:
(test (run "{with {identity1 {fun {x} x}}
              {with {identity2 {fun {x} x}}
                {+ {call identity1 1}
                   {if {call identity2 {< 1 2}} 1 2}}}}")
      => 2)
(test (run "{with {identity1 {fun {x} x}}
              {with {identity2 {fun {x} x}}
                {call {call identity1 identity2} 1}}}")
      => 1)
```

Here are two other interesting things to try out — in particular, the type that is shown in the error message is interesting:

```
(run "{fun {x} x}")
(run "{call {fun {x} {call x x}} {fun {x} {call x x}}}")
```

More specifically, it is interesting to try the following to see explicitly what our typechecker infers for `{fun {x} {call x x}}`:

```
> (define b (?T (box #f)))
> (typecheck (parse "{fun {x} {call x x}}") b (EmptyTypeEnv))
> (cases b [(?T b) (unbox b)] [else #f])
- : TYPE
(?T #&(FunT #0=(?T #&(FunT (?T #&#0#) #1=(?T #&#f))) #1#))
```

To see it clearly, we can replace each `(?T #&...)` with the `...` that it contains:

```
(FunT #0=(FunT #0# #1=#f) #1#)
```

and to clarify further, convert the `FunT` to an infix `->` and the `#f` to a `<?>` and use α for the unknown "type variable" that is represented by the `#1` (which is used twice):

```
(#0=(#0# -> α) -> α)
```

This shows us that the type is recursive.

> ***Sidenote#1**: You can now go back to the code and look at `type->string`, which is an attempt to implement a nice string representation for types. Can you see now why it cannot work (at least not without more complex code)?*
>
> ***Sidenote#2**: Compare the above with OCaml, which can infer such types when started with a `-rectypes` flag:*
>
> ```
> # let foo = fun x -> x x ;;
> val foo : ('a -> 'b as 'a) -> 'b = <fun>
> ```
>
> *The type here is identical to our type: `'a` and `'b` should be read as α and β resp., and `as` is used in the same way that Racket shows a cyclic structure using `#0#`. As for the question of why OCaml doesn't always behave as if the `-rectypes` flag is given, the answer is that its type checker might fall into the same trap that ours does — it gets stuck with:*
>
> ```
> # let foo = (fun x -> x x) (fun x -> x x) ;;
> ```

The α that we see here is "kind of" in a direction of something that resembles a polymorphic type, but we really don't have polymorphism in our language: each box can be filled just one time with one type, and from then on that type is used in all further uses of the same box type. For example, note the type error we get with:

```
{with {f {fun {x} x}}
  {call f {< {call f 1} {call f 2}}}}
```

Typing Recursion

We already know that without recursion life can be very boring… So we obviously want to be able to have recursive functions — but the question is how will they interact with our type system. One thing that we have seen is that by just having functions we get recursion. This was achieved by the Y combinator function. It seems like the same should apply to our simple typed language. The core of the Y combinator was using an expression similar to Omega that generates the infinite loop that is needed. In our language:

```
{call {fun {x} {call x x}} {fun {x} {call x x}}}
```

This expression was impossible to evaluate completely since it never terminates, but it served as a basis for the Y combinator so we need to be able to perform this kind of infinite loop. Now, consider the type of the first `x` — it's used in a `call` expression as a function, so its type must be a function type, say τ_1->τ_2. In addition, its argument is `x` itself so its type is also τ_1 — this means that we have:

```
τ₁ -> τ₂ = τ₁
```

and from this we get:

```
=> τ₁ = τ₁ -> τ₂
      = (τ₁ -> τ₂) -> τ₂
      = ((τ₁ -> τ₂) -> τ₂) -> τ₂
      = ...
```

And this is a type that does not exist in our type system, since we can only have finite types. Therefore, we have a proof by contradiction that this expression cannot be typed in our system.

This is closely related to the fact that the typed language we have described so far is “strongly normalizing”: no matter what program you write, it will always terminate! To see this, very informally, consider this language without functions — this is clearly a language where all programs terminate, since the only way to create a loop is through function applications. Now add functions and function application — in the typing rules for the resulting language, each `fun` creates a function type (creates an arrow), and each function application consumes a function type (deletes one arrow) — since types are finite, the number of arrows is finite, which means that the number of possible applications is finite, so all programs must run in finite time.

> *Note that when we discussed how to type the Y combinator we needed to use a `Rec` constructor — something that the current type system has. Using that, we could have easily solved the `τ₁ = τ₁ -> τ₂` equation with `(Rec τ₁ (τ₁ -> τ₂))`.*

In the our language, therefore, the halting problem doesn’t even exist, since all programs (that are properly typed) are guaranteed to halt. This property is useful in many real-life situations (consider firewall rules, configuration files, devices with embedded code). But the language that we get is very limited as a result — we really want the power to shoot our feet…

Extending Picky with recursion

As we have seen, our language is strongly normalizing, which means that to get general recursion, we must introduce a new construct (unlike previously, when we didn’t really need one). We can do this as we previously did — by adding a new construct to the language, or we can somehow extend the (sub) language of type descriptions to allow a new kind of type that can be used to solve the `τ₁ = τ₁ -> τ₂` equation. An example of this solution would be similar to the `Rec` type constructor in Typed Racket: a new type constructor that allows a type to refer to itself — and using `(Rec τ₁ (τ₁ -> τ₂))` as the solution. However, this complicates things: type descriptions are no longer unique, since we have `Num`, `(Rec this Num)`, and `(Rec this (Rec that Num))` that are all equal.

For simplicity we will now take the first route and add `rec` — an explicit recursive binder form to the language (as with `with`, we’re going back to `rec` rather than `bindrec` to keep things simple).

First, the new BNF:

```
<PICKY> ::= <num>
          | <id>
          | { + <PICKY> <PICKY> }
          | { < <PICKY> <PICKY> }
          | { fun { <id> : <TYPE> } : <TYPE> <PICKY> }
          | { call <PICKY> <PICKY> }
          | { with { <id> : <TYPE> <PICKY> } <PICKY> }
          | { rec { <id> : <TYPE> <PICKY> } <PICKY> }
          | { if <PICKY> <PICKY> <PICKY> }

<TYPE>  ::= Number
          | Boolean
          | ( <TYPE> -> <TYPE> )
```

We now need to add a typing judgment for `rec` expressions. What should it look like?

```
            ???
--------------------------
Γ ⊢ {rec {x : τ₁ V} E} : τ₂
```

`rec` is similar to all the other local binding forms, like `with`, it can be seen as a combination of a function and an application. So we need to check the two things that those rules checked — first, check that the body expression has the right type assuming that the type annotation given to `x` is valid:

```
  Γ[x:=τ₁] ⊢ E : τ₂   ???
--------------------------
Γ ⊢ {rec {x : τ₁ V} E} : τ₂
```

Now, we also want to add the other side — making sure that the τ_1 type annotation is valid:

```
Γ[x:=τ₁] ⊢ E : τ₂   Γ ⊢ V : τ₁
------------------------------
 Γ ⊢ {rec {x : τ₁ V} E} : τ₂
```

But that will not be possible in general — `V` is an expression that usually includes `x` itself — that's the whole point. The conclusion is that we should use a similar trick to the one that we used to specify evaluation of recursive binders — the same environment is used for both the named expression and for the body expression:

```
Γ[x:=τ₁] ⊢ E : τ₂   Γ[x:=τ₁] ⊢ V : τ₁
-------------------------------------
     Γ ⊢ {rec {x : τ₁ V} E} : τ₂
```

You can also see now that if this rule adds an arrow type to the Γ type environment (i.e., τ_1 is an arrow), then it is doing so in a way that makes it possible to use it over and over, making it possible to run infinite loops in this language.

Our complete language specification is below.

```
<PICKY> ::= <num>
          | <id>
          | { + <PICKY> <PICKY> }
          | { < <PICKY> <PICKY> }
          | { fun { <id> : <TYPE> } : <TYPE> <PICKY> }
          | { call <PICKY> <PICKY> }
          | { with { <id> : <TYPE> <PICKY> } <PICKY> }
          | { rec  { <id> : <TYPE> <PICKY> } <PICKY> }
          | { if <PICKY> <PICKY> <PICKY> }

<TYPE>  ::= Number
          | Boolean
          | ( <TYPE> -> <TYPE> )

Γ ⊢ n : Number

Γ ⊢ x : Γ(x)

Γ ⊢ A : Number   Γ ⊢ B : Number
--------------------------------
     Γ ⊢ {+ A B} : Number

Γ ⊢ A : Number   Γ ⊢ B : Number
--------------------------------
     Γ ⊢ {< A B} : Boolean
```

$$\frac{\Gamma[x:=\tau_1] \vdash E : \tau_2}{\Gamma \vdash \{\text{fun } \{x : \tau_1\} : \tau_2\ E\} : (\tau_1 \to \tau_2)}$$

$$\frac{\Gamma \vdash F : (\tau_1 \to \tau_2) \quad \Gamma \vdash V : \tau_1}{\Gamma \vdash \{\text{call } F\ V\} : \tau_2}$$

$$\frac{\Gamma \vdash C : \text{Boolean} \quad \Gamma \vdash T : \tau \quad \Gamma \vdash E : \tau}{\Gamma \vdash \{\text{if } C\ T\ E\} : \tau}$$

$$\frac{\Gamma \vdash V : \tau_1 \quad \Gamma[x:=\tau_1] \vdash E : \tau_2}{\Gamma \vdash \{\text{with } \{x : \tau_1\ V\}\ E\} : \tau_2}$$

$$\frac{\Gamma[x:=\tau_1] \vdash V : \tau_1 \quad \Gamma[x:=\tau_1] \vdash E : \tau_2}{\Gamma \vdash \{\text{rec } \{x : \tau_1\ V\}\ E\} : \tau_2}$$