

# Valex: Multiple Value Types, Conditionals, Dynamic Type Checking and Desugaring



**CS251 Programming Languages**  
Fall 2018, Lyn Turbak

Department of Computer Science  
Wellesley College

## A New Mini-Language: Valex

Valex extends Bindex in the following ways:

- In addition to integer values, Valex also has boolean, character, string, symbol, and list values.
  - A Valex program still takes a list of integers as arguments, but the result and intermediate values may be of any type.
- Valex has an easy-to-extend library of primitive operators for manipulating values of different types
- Valex has a generalized primitive operator application mechanism that performs dynamic type checking on the operands of primitive operators
- Valex has a conditional (`if`) expression.
- Valex desugars numerous special forms into a small set of five kernel constructs: literals, variable references, primitive applications, bind expressions, conditional expressions.

Valex 2

## Valex Booleans

```
valex> (< 3 4)
#t

valex> (= 3 4)
#f

valex> (!= 3 4)
#t

valex> (not (= 3 4))
#t

valex> (and (< 3 4) (>= 5 5))
#t

valex> (and (< 3 4) (> 5 5))
#f

valex> (or (< 3 4) (> 5 5))
#t
```

```
valex> (or (> 3 4) (> 5 5))
#f

valex> (and (< 4 3)
            (< 5 (/ 1 0)))
Error: Division by 0: 1

valex> (&& (< 4 3)
          (< 5 (/ 1 0)))
#f ; && is short-circuit and

valex> (or (> 4 3)
          (< 5 (/ 1 0)))
Error: Division by 0: 1

valex> (|| (> 4 3)
         (< 5 (/ 1 0)))
#t ; || is short-circuit or
```

Valex 3

## Dynamic Type Checking of Primapps

Valex dynamically checks the number and types of operands to primitive applications and reports dynamic type errors.

```
valex> (< 5)
Error: Expected two arguments but got: (5)

valex> (= 5 6 7)
Error: Expected two arguments but got: (5 6 7)

valex> (+ 1 #t)
Error: Expected an integer but got: #t

valex> (and #t 3)
Error: Expected a boolean but got: 3

valex> (= #t #f)
Error: Expected an integer but got: #t

valex> (bool= #t #f)
#f
```

Valex 4

## Conditional (`if`) expressions

```
valex> (if (< 1 2) (+ 3 4) (* 5 6))
7

valex> (if (> 1 2) (+ 3 4) (* 5 6))
30

valex> (if (< 1 2) (+ 3 4) (/ 5 0))
7 ; only evaluates then branch

valex> (if (> 1 2) (+ 3 4 5) (* 5 6))
30 ; only evaluates else branch

valex> (if (- 1 2) (+ 3 4) (* 5 6))
Error: Non-boolean test value -1 in if expression

racket> (if (- 1 2) (+ 3 4) (* 5 6))
7
```

Valex 5

## Multibranch conditionals (`cond`)

Valex includes a multibranch `cond` conditional like Racket's `cond`:

```
(valex (x y)
      (cond ((< x y) -1)
            ((= x y) 0)
            (else 1)))
```

Valex 6

## Strings

```
valex> (str= "foo" "bar")
#f

valex> (str< "bar" "foo")
#t

valex> (str< "foo" "bar")
#f

valex> (strlen "foo")
3

valex> (strlen "")
0

valex> (str+ "foo" "bar")
"foobar"
```

```
valex> (toString (* 3 4))
"12"

valex> (toString (= 3 4))
"#f"
```

### Notes:

- The only string comparison ops are `str=` and `str<`, though it would be easy to add others
- `toString` turns any Valex value into a string.

Valex 7

## Characters

```
valex> (char= 'a' 'b')
#f

valex> (char< 'a' 'b')
#t

valex> (char->int 'a')
97

valex> (int->char (- (char->int 'a') 32))
'A'
```

The only character comparison ops are `char=` and `char<`, though it would be easy to add others

Valex 8

## Symbols

Valex has Racket-like symbols that can only be  
(1) tested for equality and  
(2) converted to/from strings.

```
valex> (sym= (sym foo) (sym foo))
#t

valex> (sym= (sym foo) (sym bar))
#f

valex> (sym->string (sym baz))
"baz"

valex> (string->sym "quux")
(sym quux)
```

Valex 9

## Lists

```
valex> (prep 1 (prep 2 (prep 3 #e)))
(list 1 2 3)

valex> (prep (+ 3 4)
         (prep (= 3 4) (prep (str+ "foo" "bar") #e)))
(list 7 #t "foo")

valex> (list (+ 3 4) (= 3 4) (str+ "foo" "bar"))
(list 7 #f "foobar")

valex> (head (list 7 #t "foo"))
7

valex> (tail (list 7 #t "foo"))
(list #t "foo")

valex> (head (tail (list 7 #t "foo")))
#t

valex> (head #e)
EvalError: Head of an empty list
```

Valex 10

## More Lists

```
valex> (empty? #e)
#t

valex> (empty? (list 7 #t "foo"))
#f

valex> (nth 1 (list 7 #t "foo"))
7

valex> (nth 2 (list 7 #t "foo"))
#t

valex> (nth 3 (list 7 #t "foo"))
"foo"

valex> (nth 0 (list 7 #t "foo"))
EvalError: nth -- out-of-bounds index 0

valex> (nth 4 (list 7 #t "foo"))
EvalError: nth -- out-of-bounds index 4
```

Valex 11

## Explode and implode

```
valex> (explode "foobar")
(list 'f' 'o' 'o' 'b' 'a' 'r')

valex> (implode (list 'C' 'S' '2' '5' '1'))
"CS251"
```

Valex 12

## Type Predicates

```
valex> (int? 3)
#t

valex> (int? #t)
#f

valex> (bool? #t)
#t

valex> (bool? 3)
#f

valex> (char? 'a')
#t

valex> (char? "a")
#f

valex> (char? (sym a))
#f

valex> (string? 'a')
#f
```

```
valex> (string? 'a')
#f

valex> (string? "a")
#t

valex> (string? (sym a))
#f

valex> (sym? 'a')
#f

valex> (sym? "a")
#f

valex> (sym? (sym a))
#t

valex> (list? #e)
#t

valex> (list? (list 7 #f "foobar"))
#t

valex> (list? "foo")
#f
```

Valex 13

## General Equality

```
valex> (equal? 3 3)
#t

valex> (equal? 3 (+ 1 2))
#t

valex> (equal? (> 2 3) (< 6 5))
#t

valex> (equal? (> 2 3) (< 5 6))
#f

valex> (equal? (+ 1 2) (< 1 2))
#f

valex> (equal? (list 5 6) (list (+ 2 3) (* 2 3)))
#t

valex> (equal? (list #t) (list (< 1 2) (> 1 2)))
#f
```

Valex 14

## User-signaled errors

The `Valex error` operator takes a string message and any value and halts computation with an error message including this value:

```
valex> (bind x 3 (if (< x 0)
                    (error "negative!" x)
                    (* x x)))
9

valex> (bind x -3 (if (< x 0)
                     (error "negative!" x)
                     (* x x)))
EvalError: Valex Error -- negative!: -3
```

Valex 15

## Racket-like quote

```
valex> (quote CS251)
(sym CS251)

valex> (quote 42)
42

valex> (quote #t)
#t

valex> (quote "bunny")
"bunny"

Valex> (quote 'c')
'c'

valex> (quote (CS251 42 #t "bunny" 'c' (just like Racket!)))
(list (sym CS251) 42 #t "bunny" 'c'
      (list (sym just) (sym like) (sym Racket!)))
```

Valex 16

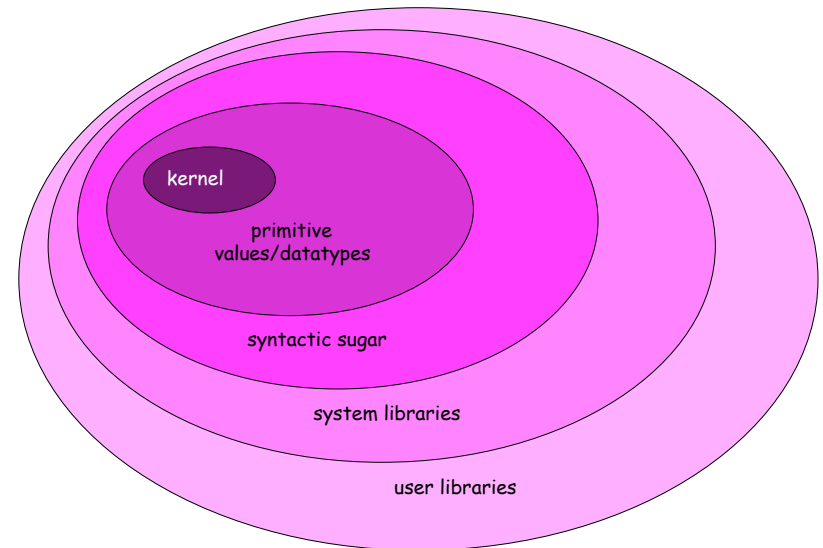
## bind vs. bindpar vs. bindseq

In addition to `bind`, Valex also has a `bindpar` construct similar to Racket's `let` and a `bindseq` construct similar to Racket's `let*`.

```
valex> (#args (a 2) (b 3))  
  
valex> (bindpar ((a (+ a b)) (b (* a b))) (list a b))  
(list 5 6)  
  
valex> (bindseq ((a (+ a b)) (b (* a b))) (list a b))  
(list 5 15)
```

Valex 17

## Implementation Strategy



Valex 18

## Valex has a Small Kernel

Kernel has only 5 kinds of expressions!

1. Literals: integers, booleans, strings, characters, symbols
2. Variable references,
3. Primitive applications (unlike in Bindex these can have any number of operands of any type),
4. Single-variable local variable declarations (i.e., `bind`),
5. Conditional expressions (i.e., `if`).

Unlike Bindex, where the only expression values are integers, Valex has 6 kinds of expression values:

1. Integers
2. Booleans
3. Strings
4. Characters
5. Symbols
6. Lists of values (recursively defined)

Valex 19

## Valex datatypes

```
type ident = string
```

```
datatype pgm = Valex of ident list * exp (* param names, body *)
```

```
and exp =  
  Lit of value  
  | Var of ident (* variable reference *)  
  | PrimApp of primop * exp list (* prim application with rator, rands *)  
  | Bind of ident * exp * exp (* bind name to value of defn in body *)  
  | If of exp * exp * exp (* conditional with test, then, else *)
```

```
and value = (* use value rather than val because val is an SML keyword *)  
  Int of int  
  | Bool of bool  
  | Char of char  
  | String of string  
  | Symbol of string  
  | List of value list (* recursively defined value *)
```

```
and primop = Primop of ident * (value list -> value)  
  (* Valex bakes the primop meaning function into the syntax! *)
```

```
fun primopName (Primop(name,_)) = name  
fun primopFunction (Primop(_,fcn)) = fcn
```

Valex 20



# Evaluating if

```
| eval (If(tst,thn,els)) env =
  (case eval tst env of
    Bool b => if b then eval thn env else eval els env
  | v => raise (EvalError ("Non-boolean test value "
    ^ (valueToString v)
    ^ " in if expression")))
```

- Use SML's if to implement Valex's if
- Choose to require that test expression have a boolean value.
- But we could make a different choice. How would we change the above clause to implement Racket semantics (i.e., any non-false value is treated as true)?

# Primitive Applications & Dynamic Type Checking

```
| eval (PrimApp(primop, rands)) env =
  (primopFunction primop) (map (Utils.flip2 eval env) rands)
```

This clause is deceptively simple. Almost all the details are handled by the primitive function baked into the syntax. E.g. (+ x 1) might be represented as:

```
PrimApp(Primop("+",
  (fn [v1, v2] =>
    (case v1 of
      Int i1 =>
        (case v2 of
          Int i2 => Int (i1 + i2)
          | _ => raise EvalError
            ("Expected an integer but got: "
             ^ (valueToString v2)))
          | _ => raise EvalError
            ("Expected and integer but got: "
             ^ (valueToString v1)))
        | args => raise EvalError
            ("Expected two arguments but got: "
             ^ (valuesToString args)))
    )
  [Var "x", Lit (Int 1)])
```

Annotations: "dynamic type checking" with arrows pointing to the nested case expressions; "result if all checks pass" with a bracket over the Int (i1 + i2) result.

# Table of primitive operators

```
val primops = [
  (* Arithmetic ops *)
  Primop("+", arithop op+),
  ... other arithmetic ops omitted ...
  Primop("/", arithop (fn(x,y) =>
    if (y = 0) then
      raise (EvalError ("Division by 0: "
        ^ (Int.toString x)))
    else x div y)),
  ... other arithmetic ops omitted ...
  (* Relational ops *)
  Primop("<", relop op<),
  Primop("<=", relop op<=),
  ... other relational ops omitted ...
  (* Logical ops *)
  Primop("not", checkOneArg checkBool (fn b => Bool(not b))),
  Primop("and", logop (fn(a,b) => a andalso b)), (* not short-circuit! *)
  Primop("or", logop (fn(a,b) => a orelse b)), (* not short-circuit! *)
  Primop("bool=", logop op=),

  (* Char ops *)
  Primop("char=", checkOneArg (checkChar, checkChar)
    (fn(c1,c2) => Bool(c1=c2))),
  ... many other primops omitted ...]
```

Most of the details of dynamic type checking are "hidden" in the helper functions **arithop**, **relop**, **logop**, **checkOneArg**, **checkOneArg**, **checkBool**, **checkChar**, etc. These helper functions form a mini-language for expressing dynamic type checking. See the next slide for details.

# Some dynamic type checking helper functions

```
fun checkInt (Int i) f = f i
  | checkInt v _ = raise (EvalError ("Expected an integer but got: "
    ^ (valueToString v)))

fun checkBool (Bool b) f = f b
  | checkBool v _ = raise (EvalError ("Expected a boolean but got: "
    ^ (valueToString v)))

(* Other checkers like checkInt and checkBool omitted *)

fun checkAny v f = f v (* always succeeds *)

fun checkOneArg check f [v] = check v f
  | checkOneArg _ f vs = raise (EvalError ("Expected one argument but got: "
    ^ (valuesToString vs)))

fun checkTwoArgs (check1,check2) f [v1,v2] =
  check1 v1 (fn x1 => check2 v2 (fn x2 => f(x1,x2)))
  | checkTwoArgs _ _ vs = raise (EvalError ("Expected two arguments but got: "
    ^ (valuesToString vs)))

fun arithop f = checkTwoArgs (checkInt,checkInt) (fn(i1,i2) => Int(f(i1, i2)))
fun relop f = checkTwoArgs (checkInt,checkInt) (fn(i1,i2) => Bool(f(i1, i2)))
fun logop f = checkTwoArgs (checkBool,checkBool) (fn(b1,b2) => Bool(f(b1, b2)))
fun pred f = checkOneArg checkAny (fn v => Bool(f v))
```

## Exercise: Add new primops to Valex



Extend Valex with these primitive operators:

- `(max int1 int2)`  
Returns the maximum of two integers
- `(getChar string index)`  
Returns the character at the given index (1-based) in the string.  
Raises an error for an out-of-bounds index.

Valex 25

## Incremental Desugaring Rules

```
(&& E_rand1 E_rand2) ↪ (if E_rand1 E_rand2 #f)
(|| E_rand1 E_rand2) ↪ (if E_rand1 #t E_rand2)

(cond (else E_default)) ↪ E_default
(cond (E_test E_then) ...) ↪ (if E_test E_then (cond ...))

(list) ↪ #e
(list E_head ...) ↪ (prep E_head (list ...))

(quote int) ↪ int
(quote string) ↪ string
(quote char) ↪ char
(quote #t) ↪ #t
(quote #f) ↪ #f
(quote #e) ↪ #e
(quote symbol) ↪ (sym symbol)
(quote (sexp_1 ... Sexp_n))
  ↪ (list (quote sexp_1) ... (quote sexp_n))
```

Valex 26

## Desugaring Rules for `bindseq` and `bindpar`

```
(bindseq () E_body) ↪ E_body
(bindseq ((Id E_defn) ...) E_body)
  ↪ (bind Id E_defn (bindseq (...) E_body))

(bindpar ((Id_1 E_defn_1) ... (Id_n E_defn_n)) E_body)
  ↪ (bind Id_list (* fresh variable name *)
      (list E_defn_1 ... E_defn_n)
      (* eval defns in parallel *)
      (bindseq ((Id_1 (nth 1 Id_list))
                ...
                (Id_n (nth n Id_list)))
                E_body))
```

Valex 27

## Desugaring Examples in Valex REPL

```
valex> (#desugar (&& (< a b) (< b c)))
(if (< a b) (< b c) #f)

valex> (#desugar (cond ((> a 10) (* a a))
                      ((< b 5) (+ 1 b))
                      (else (+ a b))))
(if (> a 10) (* a a) (if (< b 5) (+ 1 b) (+ a b)))

valex> (#desugar (bindseq ((a (+ a b))
                          (b (* a b)))
                          (list a b)))
(bind a (+ a b) (bind b (* a b) (prep a (prep b #e))))

valex> (#desugar (bindpar ((a (+ a b))
                          (b (* a b)))
                          (list a b)))
(bind vals.0 (prep (+ a b) (prep (* a b) #e))
 (bind a (nth 1 vals.0)
 (bind b (nth 2 vals.0)
 (prep a (prep b #e)))))
```

Valex 28

## Desugaring Implementation, Part 1

```
(* Incremental rule-based desugaring *)
fun desugar sexp =
  let val sexp' = desugarRules sexp in
    if Sexp.isEqual(sexp',sexp)
      then case sexp of
        Seq sexps => Seq (map desugar sexps)
        | _ => sexp
      else desugar sexp'
    end
```

Valex 29

## Desugaring Implementation, Part 2

```
and desugarRules sexp =
  case sexp of

    (* Note: the following desugarings for && and || allow
       non-boolean expressions for second argument! *)
    Seq [Sym "&&", x, y] => Seq [Sym "if", x, y, Sym "#f"]
  | Seq [Sym "||", x, y] => Seq [Sym "if", x, Sym "#t", y]

    (* Racket-style cond *)
  | Seq [Sym "cond", Seq [Sym "else", defaultx]] => defaultx
  | Seq (Sym "cond" :: Seq [testx, bodyx] :: clausexs) =>
    Seq [Sym "if", testx, bodyx, Seq(Sym "cond" :: clausexs)]

  | Seq [Sym "bindseq", Seq[], bodyx] => bodyx
  | Seq [Sym "bindseq", Seq ((Seq [Sym name, defnx])::bindingxs),
        bodyx]
    => Seq [Sym "bind", Sym name, defnx,
            Seq [Sym "bindseq", Seq bindingxs, bodyx]]

    ... many other rules omitted ...

  | _ => sexp (* doesn't match a rule, so unchanged *)
```

Valex 30

## Fresh Id in bindpar desugaring

```
(* Desugar (bindpar ((Id1 E1) ... (Idn En)) Ebody)
   to (bind vals (list E1 ... En) (* vals a "fresh" name *)
       (bindseq ((Id1 (nth 1 vals)) ... (Idn (nth n vals)))
           Ebody))
*)
| Seq [Sym "bindpar", Seq bindingxs, bodyx] =>
  let val listVar = Utils.fresh "vals"
      val (names, defnxs) = parseBindings bindingxs
      in Seq [Sym "bind", Sym listVar, Seq (Sym "list" :: defnxs),
            Seq [Sym "bindseq",
                  Seq (map (fn (name, index) =>
                        Seq [Sym name,
                              Seq [Sym "nth", Sexp.Int index,
                                    Sym listVar]])
                          (ListPair.zip(names,
                                        Utils.range 1
                                        (1 + (length names))))),
                  bodyx]]
      end
```

Valex 31

## Desugaring exercise



Extend Valex with this syntactic sugar construct:

(ifpos  $E\_test$   $E\_pos$   $E\_else$ )  
Evaluates  $E\_test$  to a value  $v\_test$ . If  $v\_test$  is a positive integer, returns the value of  $E\_pos$  without evaluating  $E\_else$ . If  $v\_test$  is a nonpositive integer, returns the value of  $E\_else$  without evaluating  $E\_pos$ . Otherwise signals an ifpos nonint test error.

For example:

(ifpos (+ 1 2) (\* 3 4) (/ 5 0)) evaluates to 12  
(ifpos (- 1 2) (+ 3 #t) (\* 5 6)) evaluates to 30  
(ifpos (< 1 2) (+ 3 4) (\* 5 6)) signals error ifpos nonint test: #t

Valex 32