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

CS251 Programming Languages
Spring 2018, Lyn Turbak
Department of Computer Science
Wellesley College

SOLUTIONS

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

Valex Dynamically Checks the Number and Types of Operands to Primitive Applications and Reports Dynamic Type Errors.

valex> (< 3)
Error: Expected two arguments but got: (3)
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

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.
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 0))
Error: Non-boolean test value -1 in if expression
racket> (if (- 1 2) (+ 3 4) (* 5 6))
7
```

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)))
```

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"
```

Characters

```
valex> (char= 'a' 'b')
#f
valex> (char< 'a' 'b')
#t
valex> (char->int 'a')
97
valex> (int->char (- (char->int 'a') 32))
'A'
```

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

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
```

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
```

Explode and implode

```
valex> (explode "foobar")  (list 'f 'o 'o 'b 'a 'r)
valex> (implode (list 'C 'S '2 '5 '1))  "CS251"
```

Valex 9

Valex 10

Valex 11

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? (sym a)) #t
Valex> (list? #e) #t
Valex> (list? (list 7 #f "foobar")) #t
Valex> (list? "foo") #f
Valex> (list? (sym a)) #f
Valex> (list? (string? 'a')) #f
Valex> (list? (string? "a")) #f

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

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

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!)))
**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 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)

---

**Implementation Strategy**

---

**Valex datatypes**

```plaintext
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
```
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-true value is treated as true)?

SOLUTION

```
| eval (If(tst,thn,els)) env = 
  (case eval tst env of 
    Bool false => eval els env 
    | _ => eval thn env) (* any non-false value is truthy *)
```

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<=), 
  _ 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`, `checkAny`, 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 
  | checkInt v _ = raise (EvalError ("Expected one argument 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: " ^ (valueToString 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: " ^ (valueToString 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

  \[
  \text{Primop}("\text{max}", \text{arithop} (\text{fn}(i1, i2) => \text{if } i1 \geq i2 \text{ then } i1 \text{ else } i2)), (* Or could use: \text{Primop}("\text{max}", \text{arithop} \text{Int.max}), *)
  \]

- **(getChar string index)**
  Returns the character at the given index (1-based) in the string. Raises an error for an out-of-bounds index.

  \[
  \text{Primop}("\text{getChar}", \text{checkTwoArgs (checkString,checkInt)} (\text{fn}(s,i) => \text{Char(String.sub(s,i-1)))}),
  \]

Desugaring Rules for **bindseq** and **bindpar**

\[
\begin{align*}
\text{(bindseq () E_body)} & \rightarrow \text{E_body} \\
\text{(bindseq ((Id E_defn) ... ) E_body)} & \rightarrow \text{(bind Id E_defn (bindseq (... ) E_body))} \\
\text{(bindpar ((Id_1 E_defn_1) ... (Id_n E_defn_n)) E_body)} & \rightarrow \text{(bind Id_list (* fresh variable name *))} \\
& \quad \text{(list E_defn_1 ... E_defn_n)} \\
& \quad (* \text{eval defns in parallel } *) \\
& \quad \text{(bindseq ((Id_1 (nth 1 Id_list))} \\
& \quad \quad ... \\
& \quad \quad \text{(Id_n (nth n Id_list)))} \\
& \quad \text{E_body})
\end{align*}
\]

Desugaring Examples in Valex REPL

\[
\begin{align*}
valex> \#\text{desugar } (\text{&& } (< a b) \text{ (< b c) }) & \rightarrow \text{(if } (< a b) \text{ (< b c) } \#f) \\
& \rightarrow \text{(if } (< a b) \text{ (< b c) } \#f) \\
\end{align*}
\]

\[
\begin{align*}
valex> \#\text{desugar } (\text{cond } ((> a 10) (* a a)) & \rightarrow \text{(if } (> a 10) \text{ (* a a) } \rightarrow \text{(if } (< b 5) (+ 1 b)) \\
& \rightarrow \text{(if } (< b 5) (+ 1 b) (+ a b)) \\
\end{align*}
\]

\[
\begin{align*}
valex> \#\text{desugar } (\text{bindseq } ((a (+ a b)) & \rightarrow \text{(list a b))} \\
& \rightarrow \text{(bind a (+ a b) (bind b (* a b) (prep a (prep b #e))))} \\
\end{align*}
\]

\[
\begin{align*}
valex> \#\text{desugar } (\text{bindpar } ((a (+ a b)) & \rightarrow \text{(list a b))} \\
& \rightarrow \text{(bind vals.0 (prep (+ a b) (prep (* a b) #e)))} \\
& \rightarrow \text{(bind a (nth 1 vals.0) (bind b (nth 2 vals.0) (prep a (prep b #e))))} \\
\end{align*}
\]
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

Desugaring Implementation, Part 2

and desugarRules sexp =
  case sexp of
    (* Note: the following desugaring for && and || allow non-boolean expressions for second argument! *)
    Seq [Sym "&&", x, y] => Seq [Sym "if", x, y, Sym "#t"]
    | 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]]
    ...
    | _ => sexp (* doesn't match a rule, so unchanged *)

Fresh Id in bindpar desugaring

(* Desugar (bindpar ((Id l) ... (Id n)) Ebody) to (bind vals (list El ... En) (* vals a "fresh" name *))
   (bindseq ((Id l nth 1 vals) ... (Id n nth n vals))) Ebody)
*)
| Seq [Sym "bindpar", Seq bindingxs, bodyx] =>
  let val listVar = Utils.fresh "vals"
  val (names, defnx) = parseBindings bindingxs
  in Seq [Sym "bind", Sym listVar, Seq [Sym "list" :: defnx],
         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

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 #f)) signals error ifpos nonint test: #t

(* clause in desugarRules function *)
| Seq [Sym "ifpos", testx, posx, elsex] =>
  let val testVar = Utils.fresh "test"
  in Seq [Sym "bind", Sym testVar, testx,
              Seq [Sym "if", Seq [Sym "int?", Sym testVar],
                   Seq [Sym "if", [Sym ""], Sym testVar, Lit (Int 0)],
                   posx, elsex],
              Seq [Sym "error", Lit (String "ifpos: nonint test"),
                   Sym testVar]]
end