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.

Dynamic Type Checking of Primapps

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

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))
#f
valex> (or (> 3 4) (> 5 5))
#f
valex> (or (> 3 4) (> 5 5))
#f
valex> (and (< 3 4) (> 5 5))
#t
valex> (+ 1 #t)
Error: Expected an integer but got: #t
valex> (&& (< 4 3)
Error: Expected two arguments but got: (5)
valex> (or (> 4 3)
Error: Expected two arguments but got: (5 6 7)
valex> (> 3 4)
Error: Expected two arguments but got: (5 6 7)
valex> (< 5)
Error: Expected two arguments but got: (5)

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"
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.

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
Symbols

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

```scheme
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

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

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

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

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

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

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

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

More Lists

```scheme
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

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

<table>
<thead>
<tr>
<th>Valex</th>
<th>Predicate</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(int? 3)</td>
<td>#t</td>
<td></td>
</tr>
<tr>
<td>(int? #t)</td>
<td>#f</td>
<td></td>
</tr>
<tr>
<td>(bool? #t)</td>
<td>#t</td>
<td></td>
</tr>
<tr>
<td>(bool? 3)</td>
<td>#f</td>
<td></td>
</tr>
<tr>
<td>(char? 'a')</td>
<td>#t</td>
<td></td>
</tr>
<tr>
<td>(char? &quot;a&quot;)</td>
<td>#f</td>
<td></td>
</tr>
<tr>
<td>(char? (sym a))</td>
<td>#f</td>
<td></td>
</tr>
<tr>
<td>(string? 'a')</td>
<td>#f</td>
<td></td>
</tr>
<tr>
<td>(string? (sym a))</td>
<td>#f</td>
<td></td>
</tr>
<tr>
<td>(list? #e)</td>
<td>#t</td>
<td></td>
</tr>
<tr>
<td>(list? (list 7 #f &quot;foobar&quot;))</td>
<td>#t</td>
<td></td>
</tr>
<tr>
<td>(list? &quot;foo&quot;)</td>
<td>#f</td>
<td></td>
</tr>
<tr>
<td>(list? (sym a))</td>
<td>#t</td>
<td></td>
</tr>
</tbody>
</table>

**General Equality**

<table>
<thead>
<tr>
<th>Valex</th>
<th>General Equality</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(equal? 3 3)</td>
<td>#t</td>
<td></td>
</tr>
<tr>
<td>(equal? 3 (+ 1 2))</td>
<td>#t</td>
<td></td>
</tr>
<tr>
<td>(equal? (&gt; 2 3) (&lt; 6 5))</td>
<td>#t</td>
<td></td>
</tr>
<tr>
<td>(equal? (&gt; 2 3) (&lt; 5 6))</td>
<td>#f</td>
<td></td>
</tr>
<tr>
<td>(equal? (+ 1 2) (&lt; 1 2))</td>
<td>#f</td>
<td></td>
</tr>
<tr>
<td>(equal? (list 5 6) (list (+ 2 3) (* 2 3)))</td>
<td>#t</td>
<td></td>
</tr>
<tr>
<td>(equal? (list #t) (list (&lt; 1 2) (&gt; 1 2)))</td>
<td>#f</td>
<td></td>
</tr>
</tbody>
</table>

**User-signaled errors**

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

<table>
<thead>
<tr>
<th>Valex</th>
<th>User-signal error</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(bind x 3 (if (&lt; x 0) (error &quot;negative!&quot; x) (* x x)))</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>(bind x -3 (if (&lt; x 0) (error &quot;negative!&quot; x) (* x x)))</td>
<td>EvalError: Valex Error -- negative!: -3</td>
<td></td>
</tr>
</tbody>
</table>

**Racket-like quote**

<table>
<thead>
<tr>
<th>Valex</th>
<th>Racket-like quote</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(quote CS251)</td>
<td>(sym CS251)</td>
<td></td>
</tr>
<tr>
<td>(quote 42)</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>(quote #t)</td>
<td>#t</td>
<td></td>
</tr>
<tr>
<td>(quote &quot;bunny&quot;)</td>
<td>&quot;bunny&quot;</td>
<td></td>
</tr>
<tr>
<td>(quote 'c')</td>
<td>'c'</td>
<td></td>
</tr>
<tr>
<td>(quote (CS251 42 #t &quot;bunny&quot; 'c' (just like Racket!)))</td>
<td>(list (sym CS251) 42 #t &quot;bunny&quot; 'c' (list (sym just) (sym like) (sym Racket!)))</td>
<td></td>
</tr>
</tbody>
</table>
**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
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)

---

**Valex datatypes**

```sml
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 defin 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

```haskell
| 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)?

Table of primitive operators

<table>
<thead>
<tr>
<th>primop =</th>
</tr>
</thead>
</table>
| val | [/* 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 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. |

Primitives & Dynamic Type Checking

```haskell
| 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. (+ 1) might be represented as:

```haskell
| 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))) |
| _ => raise EvalError ("Expected two arguments but got: " 
| ^ (valueToString rands)))
|)
```

Some dynamic type checking helper functions

```haskell
| 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 f v = 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 _ _ _ = 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:

- \( (\text{max } \text{int1 } \text{int2}) \)
  
  Returns the maximum of two integers

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

Desugaring Rules for \textit{bindseq} and \textit{bindpar}

\[
\begin{align*}
\text{(bindseq ( ) E\_body)} & \rightarrow \text{E\_body} \\
\text{(bindseq ((Id E\_defn) ... ) E\_body)} & \rightarrow \text{(bind Id E\_defn (bindseq ( ) E\_body))} \\
\text{(bindpar ((Id\_1 E\_defn\_1) ... (Id\_n E\_defn\_n) ) E\_body)} & \rightarrow \text{(bind Id\_list (* fresh variable name *) )} \\
& \quad \text{(list E\_defn\_1 ... E\_defn\_n)} \\
& \quad (* \text{eval defns in parallel } *) \\
& \quad (\text{bindseq ((Id\_1 (nth 1 Id\_list))} \\
& \quad \quad ...) \\
& \quad (\text{Id\_n (nth n Id\_list) )}} \\
& \quad \text{E\_body})
\end{align*}
\]

Desugaring Examples in Valex REPL

\[
\begin{align*}
\text{valex} & \rightarrow \text{(desugar (&& (< a b) (< b c)))} \\
& \rightarrow \text{(if (< a b) (< b c) #f)} \\
& \rightarrow \text{(if (< a b) (< b c) #f)} \\
& \rightarrow \text{(bind Id\_list (* fresh variable name *) )} \\
& \quad \text{(list E\_defn\_1 ... E\_defn\_n)} \\
& \quad (* \text{eval defns in parallel } *) \\
& \quad (\text{bindseq ((Id\_1 (nth 1 Id\_list))} \\
& \quad \quad ...) \\
& \quad (\text{Id\_n (nth n Id\_list) )}} \\
& \quad \text{E\_body})
\end{align*}
\]

Incremental Desugaring Rules

\[
\begin{align*}
(\&\& E\_rand1 E\_rand2) & \rightarrow (\text{if } E\_rand1 E\_rand2 \#f) \\
(|| E\_rand1 E\_rand2) & \rightarrow (\text{if } E\_rand1 \#t E\_rand2) \\
(\text{cond (else E\_default)}) & \rightarrow E\_default \\
(\text{cond (E\_test E\_then) ...}) & \rightarrow (\text{if } E\_test E\_then (\text{cond ...)}) \\
(list) & \rightarrow \#e \\
(list E\_head ... ) & \rightarrow \text{(prep E\_head (list ...))} \\
(quote \text{int}) & \rightarrow \text{int} \\
(quote \text{string}) & \rightarrow \text{string} \\
(quote \text{char}) & \rightarrow \text{char} \\
(quote \text{#t}) & \rightarrow \#t \\
(quote \text{#f}) & \rightarrow \#f \\
(quote \text{#e}) & \rightarrow \#e \\
(quote \text{symbol}) & \rightarrow (\text{sym symbol}) \\
(quote (sexp\_1 ... Sexp\_n)) & \rightarrow (\text{list (quote sexp\_1) ... (quote sexp\_n)})
\end{align*}
\]
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 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]]
    ...
  | _ => sexp (* doesn't match a rule, so unchanged *)
end

Fresh Id in bindpar desugaring

/* Desugar (bindpar ((Id1 E1) ... (Idn En)) Ebady)
  to (bind vals (list El ... En) (* vals a "fresh" name *)
  (bindseq ((Id1 nth 1 vals) ... (Idn nth n vals)))
  Ebady)
  */
| 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 #t) (* 5 6)) signals error ifpos nonint test: #t