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
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
```
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
```

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') #t
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.

```racket
valex> (sym= (sym foo) (sym foo)) #t
valex> (sym= (sym foo) (sym bar)) #f
valex> (sym->string (sym baz)) "baz"
```

### Lists

```racket
valex> (prep 1 (prep 2 (prep 3 #e)))
(list 1 2 3)
valex> (prep (+ 3 4)
(prepare (- 3 4) (prepare (+ 3 4) (prepare "foo" "bar") #e)))
(list 7 #t "foo")
valex> (list (+ 3 4) (- 3 4) (str "foo" "bar") #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

```racket
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"
```

### Explode and implode

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

### Type Predicates

```racket
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? (sym a)) #f
valex> (list? #e) #t
valex> (list? (list 7 #f "foobar")) #t
valex> (list? "foo") #f
```
**General Equality**

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

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

**Racket-like quote**

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

<table>
<thead>
<tr>
<th>Valex&gt; (#args (a 2) (b 3))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valex&gt; (bindpar ((a (+ a b)) (b (* a b))) (list a b)) (list 5 6)</td>
</tr>
<tr>
<td>Valex&gt; (bindseq ((a (+ a b)) (b (* a b))) (list a b)) (list 5 15)</td>
</tr>
</tbody>
</table>
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

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 *)
| 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 prim meaning function into the syntax! *)

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

Core of Valex Interpreter

(* val run : Valex.pgm -> int list -> value *)
fun run (Valex(fmls, body)) ints =
  let val flen = List.length fmls
      val ilen = List.length ints
  in if flen = ilen then
      eval body (Env.make fmls (List.map (fn i => Int i) ints))
    else raise (EvalError ("Program expected " ^ (Int.toString flen) ^ " arguments but got " ^ (Int.toString ilen)))
  end

(* val eval : Valex.exp -> value Env.env -> value *)
and eval (Lit v) env = v
| eval (Var name) env =
  (case Env.lookup name env of
   SOME(v) => v
   NONE => raise (EvalError("Unbound variable: " ^ name)))
| eval (PrimApp (primop, rands)) env = (* see slides 22-24 *)
  (primopFunction primop) (List.map (Utils.flip2 eval env) rands)
| eval (Bind (name, defn, body)) env =
  eval body (Env.bind name (eval defn env) env)
| eval (If (tst, thn, els)) env = (* see slide 21 *)
  (case eval tst env of
   Bool b => if b then eval thn env else eval els env
   | _ => v => raise (EvalError ...))
Evaluating if

```
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 opc),
  Primop("<=", relop opc<=),
  ... 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.

| 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)?

```
val primops = [
  (* Arithmetic ops *)
  Primop("+", arithop op+),
  ... other arithmetic ops omitted ... |
```

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. (+ 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 an integer but got: " ^ (valueToString v1))) |
        _ => |
        raise EvalError |
            ("Expected two arguments but got: " ^ (valuesToString args))) |
      args => |
      raise EvalError |
        ("Expected two arguments but got: " ^ (valuesToString args))) |
    _ => |
    raise EvalError |
      ("Expected one argument but got: ") ^ (valuesToString v2))) |
  _ => |
  raise EvalError |
    ("Expected two arguments but got: " ^ (valuesToString args))) |
| [Var "x", Lit (Int 1)] |
```

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 _ 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))
```

Valex 21

Valex 22

Valex 23

Valex 24
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.

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))

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))))

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))))))
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 "bindseq", Seq[], bodyx] => bodyx
    | Seq [Sym "bindseq", Seq [(Seq [Sym name, defnx])::bindingxs), bodyx] =>
      Seq [Sym "if", Sym name, defnx, Seq [Sym "bindseq", Seq bindingxs, bodyx]]
    ...
    | _ => sexp (* doesn't match a rule, so unchanged *)

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]]

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