Valex: Dynamic Type Checking and Desugaring

VALEX is a language that extends BINDEX with several new primitive data types and some constructs that express branching control flow. We study VALEX for two reasons:

1. To show how multiple primitive data types are handled by the interpreter. In particular, the VALEX interpreter performs **dynamic type checking** to guarantee that operators are called only on the right number and types of operands and that conditionals use only booleans to direct control flow.
2. To show that a language implementation can be significantly simplified by decomposing it into three parts:
   (a) a small **kernel language** with only a few kinds of expressions;
   (b) **synactic sugar** for expressing other constructs in terms of kernel expressions;
   (c) an easily extensible **library of primitives**.

1 The VALEX Language

Whereas all values in INTEX and BINDEX are integers, VALEX supports several additional types of values: booleans, strings, characters, symbols, and lists. It also supports branching control flow constructs controlled by booleans.

1.1 Booleans

VALEX includes the two values `#t` (stands for truth) and `#f` (stands for falsity). These values are called **booleans** in honor of George Boole, the nineteenth century mathematician who invented boolean algebra.

The two boolean values can be written directly as literals, but can also be returned as the result of applying **relational operators** (`<=`, `<`, `>`, `>=`, `=` `!=`) to integers and **logical operators** (`not`, `and`, `or`, `bool=`) to booleans. The `=` operator tests two integers for equality, while `!=` tests two integers for inequality. The `and` operator returns the logical conjunction ("and") of two boolean operands, while `or` returns the logical disjunction ("or") of two boolean operands. The `bool=` operator tests two booleans for equality. For example:

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

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

If a VALEX operator is supplied with the wrong number or wrong types of operands, a **dynamic type checking** error is reported.

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

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

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

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

valex> (bool= 7 8)
EvalError: Expected a boolean but got: 7

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

The final example illustrates the necessity of the `bool=` operator; The `=` operator tests only integer equality in VALEX, so each non-integer value type needs its own operator to test equality for that type.

In contrast, many languages support **overloaded** operators that may be used on different types of operands (and whose meaning may depend on the types of those operands). For example:

- JAVA's `==` operator tests equality for any primitive type (e.g., `int`, `boolean`, `char`, etc.) and every reference type (i.e., object type).
- OCAML's relational functions (`<`, `<=`, `!=`, `=`, `>=`, `>`) are defined at every type, as are its `min` and `max` functions.
- JAVA's `+` operator can be used with operands of any numeric type; it's behavior depends on the types of the operands. For example, if both operands are integers it performs integer addtion; if both operands are floating point numbers, it performs floating point addition; and if one operand is an integer and the other is a float, it converts the integer to a float before performing floating point addition. Additionally, the `+` operator denotes string concatenation when applied to two strings; and when applied to one string, causes the non-string operand to be converted to a string before concatenation is performed.

1

[1] The relational operators in VALEX are not ¿ overloaded, as they are in other languages, such as OCAML.

1.2 Branching Control Constructs

The purpose of booleans is to direct the flow of control in a program with a branching control structure. The fundamental control construct in VALEX is the conditional construct (if E_{test} E_{then} E_{else}), which first evaluates E_{test} to a value V_{test}, and then returns the value of E_{then} if V_{test} is true and returns the value of E_{else} if V_{test} is false.

```
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

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

The last two examples highlight the fact that exactly *one* of E_{then} and E_{else} is evaluated. The expression in the branch not taken is never evaluated, and so the fact that such branches might contain an error is never detected.

Evaluating only one of the two branches is more than a matter of efficiency. In languages with recursion, it is essential to the correctness of recursive definitions. For example, consider an OCAML definition of factorial:

```
let fact n =
  if n = 0 then
    1
  else
    n * (fact(n-1))
```

If *both* branches of the `if` were evaluated, then an application of `fact`, such as `fact 3`, would never terminate! This is why `if` must be a "special form" in call-by-value languages and not just an application of a primitive operator; in applications of primitive operators in a call-by-value language, *all* operand expressions must be evaluated.

The VALEX `if` construct has the same syntax as SCHEME's `if` construct, but its semantics differs. Unlike Scheme, which treats any non-false value as true VALEX requires that the test expression evaluate to a boolean. A non-boolean test expression is an error in VALEX:

```
valex> (if (- 1 2) (+ 3 4) (* 5 6))
Error! Non-boolean test in an if expression.

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

VALEX also has a multi-clause conditional construct with the same syntax as SCHEME's `cond` construct. For example, the VALEX program

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

is equivalent to the following program using nested conditionals:

```
(valex (x y)
  (if (< x y)
      -1
      (if (= x y)
          0
          1)))
```

The only difference in meaning between the VALEX `cond` and a SCHEME`cond` is the same as that for `if`: each test expression evaluated in the VALEX `cond` must be a boolean.

Like many languages, VALEX provides "short-circuit" logical conjunction and disjunction constructs, respectively `&&` (cf. OCAML/JAVA/C's `&&` and Scheme's `and`) and `||` (cf. OCAML/JAVA/C's `||` and Scheme's `or`):

(`&&` E_{rand1} E_{rand2})
(`||` E_{rand1} E_{rand2})

These are similar to VALEX's binary operators `and` and `or`, except that E_{rand2} is never evaluated if the result is determined by the value of E_{rand1}. For instance, with `&&`, E_{rand1} is first evaluated to the value V_{rand1}. If V_{rand1} is `#t`, then E_{rand2} is evaluated, and its value is returned as the value of the `&&` expression. But if V_{rand1} is `#f`, then `#f` is immediately returned as the value of the `&&` and E_{rand2} is never evaluated. Similarly, with `||`, if V_{rand1} is `#t`, a value of `#t` is returned for the `||` expression without E_{rand2} being evaluated; otherwise the value of E_{rand2} is returned. In contrast, both operand expressions of `and` and `or` are always evaluated.

```
valex> (and (= 1 2) (> 3 4 5))
EvalError: Expected two arguments but got: (3 4 5)

valex> (&& (= 1 2) (> 3 4 5))
#f

valex> (or (< 1 2) (+ 3 4))
EvalError: Expected a boolean but got: 7

valex> (|| (< 1 2) (+ 3 4))
#t

valex> (and (< 1 2) (+ 3 4))
EvalError: Expected a boolean but got: 7

valex> (&& (< 1 2) (+ 3 4))
7

valex> (|| (> 2 3) (* 4 5))
20
```

The final two examples shows that when the first operand does not determine the value of an `&&` or `||` construct, the value of its second operand is returned, regardless of whether or not it is a boolean.

In many cases, `&&`/`||` behave indistinguishably from the boolean operators `and`/`or`, which evaluate *both* of their operands. To see the difference, it is necessary to consider cases where not evaluating E_2 makes a difference. In VALEX, such a situation occurs when evaluating E_2 would otherwise give an error. For instance, consider the following VALEX program:

```
(valex (x)
  (if (|| (= x 0)
          (> (/ 100 x) 7))
      (+ x 1)
      (* x 2)))
```

This program returns 1 when applied to 0. But if the `||` were changed to `or`, the program would encounter a divide-by-zero error when applied to 0 because the division would be performed even though `(= x 0)` is true.

This example is somewhat contrived, but real applications of short-circuit operators abound in practice. For example, consider the higher-order OCAML `for_all` function we studied earlier this semester:

```
let rec for_all p xs =
  match xs with
    [] -> true
  | x::xs' -> (p x) && for_all p xs'
```

In OCAML, `&&` is the short-circuit conjunction operator. It is important to use a short-circuit operator in `for_all` because it causes the recursion to stop as soon as an element is found for which the predicate is false. If `&&` were not a short-circuit operator, then `for_all` of a very long list would explore the whole list even in the case where the very first element is found to be false.

As another example, consider the following Java `insertion_sort` method for an array:

```
public void insertion_sort (int[] a) {
  for (int i = 0; i < a.length; i++) {
    int x = a[i];
    int j = i-1;
    // Insertion loop
    while ((j >= 0) && (a[j] > x)) { // Critical that && is short-circuit!
      a[j+1] = a[j];
      j--;
    }
    a[j+1] = x;
  }
}
```

The use of the short-circuit `&&` operator in the test of the `while` loop is essential. In the case where `j` is `-1`, the test `((j >= 0) && (a[j] > x))` is false. But if both operands of the `&&` were evaluated, the evaluation of `a[-1]` would raise an array out-of-bounds exception.

1.3 Strings

VALEX supports string values. As usual, string literals are delimited by double quotes.

1.4 Characters

VALEX supports character values. As usual, string literals are delimited by single quotes.

1.5 Symbols

VALEX supports a Scheme-like symbol data type. A symbolic literal, written `(sym` *`symbolname`*`)`, denotes the name *symbolname*. So `sym` is a kind of "quotation mark", similar to `quote` in Scheme, that distinguishes symbols (such as `(sym x)`) from variable references (such as `x`).

The only operation on symbols is the test for equality via the `sym=` operator. For example:

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

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

1.6 Lists

VALEX supports list values. The empty list is written `#e`. The prepending function `prep` adds an element to the front of a list. The `head` function returns the head of a list while `tail` returns the tail. A list is tested for emptiness via `empty?`. The notation:

(list E_1 ... E_n)

is a shorthand for creating a list of n elements.

2 The VALEX Kernel

The VALEX kernel language has only five kinds of expressions:

1. literals (which include boolean and symbolic literals as well as integers),
2. variable references,
3. single-variable local variable declarations (i.e., `bind`),
4. primitive applications (can have any number of operands of any type), and
5. conditional expressions (i.e., `if`).

In Sec. 4, we shall see that these five expression types are sufficient for representing *all* VALEX expressions.

The abstract syntax for the VALEX kernel is presented in Fig. 1. The `exp` type expresses the five different kinds of VALEX expressions. The `valu`[2] type expresses the six different types of VALEX values.

Primitive operators are represented via the `primop` type, whose single constructor `Primop` combines the name of the operator with an OCAML function of type `valu list -> valu` that specifies the behavior of the operator. The two components of a `primop` can be extracted via the functions `primopName` and `primopFunction`. We will study the specification of primitives in Sec. 3. We will see that the key benefit of the VALEX approach to specifying primitives is that the VALEX abstract syntax need not be extended every time a new primitive operator is added to the language. In contrast, INTEX and BINDEX were implemented with a `binop` type that did need to be extended:

```
and binop = | Add | Sub | Mul | Div | Rem
```

Unparsing in VALEX is straightforward (Fig. 2). The only feature worth noting is that there is a `valuToSexp` function that handles the unparsing of the boolean true value to `#t`, the boolean false value to `#f`, the empty list `#e`, and non-empty lists to the form (list V_1 ... V_n).

VALEX parsing is more complicated. We delay presenting this until we discuss desugaring in Sec. 4.

In VALEX, the free variables are calculated as in BINDEX, except there are two new clauses: one for general primitive applications and one for conditionals:

```
and freeVarsExp e =
  match e with
    ⋮
  | PrimApp(_,rands) -> freeVarsExps rands
  | If(tst,thn,els) -> freeVarsExps [tst;thn;els]
```

Similarly, the VALEX `subst` function has two new clauses:

[2]The name `valu` was chosen because the names `val` and `value` are already reserved keywords in OCAML that cannot be used as type names.

```
type var = string

type pgm = Pgm of var list * exp (* param names, body *)

and exp =
    Lit of valu (* integer, boolean, character, string, and symbol literals *)
  | Var of var (* variable reference *)
  | PrimApp of primop * exp list (* primitive application with rator, rands *)
  | Bind of var * exp * exp (* bind name to value of defn in body *)
  | If of exp * exp * exp (* conditional with test, then, else *)

and valu =
    Int of int
  | Bool of bool
  | Char of char
  | String of string
  | Symbol of string
  | List of valu list

and primop = Primop of var * (valu list -> valu) (* primop name, function *)

let primopName (Primop(name,_)) = name

let primopFunction (Primop(_,fcn)) = fcn
```

Figure 1: Data types for Valex abstract syntax.

```
let rec subst exp env =
  match exp with
    ⋮
  | PrimApp(op,rands) -> PrimApp(op, map (flip subst env) rands)
  | If(tst,thn,els) -> If(subst tst env, subst thn env, subst els env)
```

The complete environment model evaluator for Valex is shown in Fig. 3. It is very similar to the Bindex environment model evaluator except:

- In the top-level call to `eval` from `run`, it is necessary to inject each integer argument into the `valu` type using the `Int` constructor. (For simplicity, we still assume that all program arguments are integers even though our language supports a richer collection of values.)
- Valex environments hold arbitrary values rather than just integers, so the type of `eval` is:

  ```
  val eval : Valex.exp -> valu Env.env -> valu
  ```

- Since each `primop` holds the Ocaml function specifying its behavior, all the primitive application clause has to do is apply this function to the evaluated operands. There is no need for the analog of the auxiliary `binApply` function used in the Intex and Bindex interpreters.
- It has a clause for evaluating conditionals. Note that:
  - Ocaml's `if` is used to implement Valex's `if`;
  - at most one of the two conditional branches (`thn`, `els`) is evaluated;
  - because Valex has many different kinds of values, dynamic type checking must be performed on the test expression `tst` to ensure that it is a boolean. If not, a dynamic type error is reported.

```
(* val pgmToSexp : pgm -> Sexp.sexp *)
let rec pgmToSexp p =
  match p with
    Pgm (fmls, e) ->
      Seq [Sym "valex"; Seq(map (fun s -> Sym s) fmls); expToSexp e]

(* val expToSexp : exp -> Sexp.sexp *)
and expToSexp e =
  match e with
    Lit v -> valuToSexp v
  | Var s -> Sym s
  | PrimApp (rator, rands) ->
      Seq (Sym (primopName rator) :: map expToSexp rands)
  | Bind(n,d,b) -> Seq [Sym "bind"; Sym n; expToSexp d; expToSexp b]
  | If(tst,thn,els) -> Seq [Sym "if"; expToSexp tst; expToSexp thn; expToSexp els]

(* val valuToSexp : valu -> sexp *)
let rec valuToSexp valu =
  match valu with
    Int i -> Sexp.Int i
  | Bool b -> Sym (if b then "#t" else "#f")
  | Char c -> Sexp.Chr c
  | String s -> Sexp.Str s
  | Symbol s -> Seq [Sym "sym"; Sym s]
  | List [] -> Sym "#e" (* special case *)
  | List xs -> Seq (Sym "list" :: (map valuToSexp xs))

(* val valuToString : valu -> string *)
let valuToString valu = sexpToString (valuToSexp valu)

(* val valusToString : valu list -> string *)
and valusToString valus = sexpToString (Seq (map valuToSexp valus))

(* val expToString : exp -> string *)
and expToString s = sexpToString (expToSexp s)

(* val pgmToString : pgm -> string *)
and pgmToString s = sexpToString (pgmToSexp s)
```

Figure 2: Unparsing functions for the Valex abstract syntax.

The complete substitution model evaluator for Valex is shown in Fig. 4. It is similar to the Bindex substitution model evaluator except for differences analagous to the ones discussed for the environment model evaluator.

This completes the presentation of the implementation of the Valex kernel. Even though Valex has many more features than Bindex, its kernel differs from the Bindex kernel in only relatively minor ways. And in some ways, such as the evaluation of primitive applications, it is even simpler.

We will now discuss in more detail the specification of primitive operators and syntactic sugar, features that are key in simplifying the Valex implementation.

```
(* val run : Valex.pgm -> int list -> valu *)
let rec run (Pgm(fmls,body)) ints =
  let flen = length fmls
  and ilen = length ints
  in
    if flen = ilen then
      eval body (Env.make fmls (map (fun i -> Int i) ints))
    else
      raise (EvalError ("Program expected " ^ (string_of_int flen)
                        ^ " arguments but got " ^ (string_of_int ilen)))

(* val eval : Valex.exp -> valu Env.env -> valu *)
and eval exp env =
  match exp with
    Lit v -> v
  | Var name ->
      (match Env.lookup name env with
         Some(v) -> v
        | None -> raise (EvalError("Unbound variable: " ^ name)))
  | PrimApp(op, rands) -> (primopFunction op) (map (flip eval env) rands)
  | Bind(name,defn,body) -> eval body (Env.bind name (eval defn env) env)
  | If(tst,thn,els) ->
     (match eval tst env with
       Bool b -> if b eval thn env else eval els env
     | v -> raise (EvalError ("Non-boolean test value "
                              ^ (valuToString v)
                             ^ " in if expression"))
     )
```

Figure 3: The environment model evaluator for the Valex kernel.

3 Primitive Operators and Dynamic Type Checking

In the implementation architecture exemplified by Bindex, adding a new primitive is more tedious than it should be. To show this, we will consider the four steps required to add an exponentiation operator ^ to Bindex:

1. Extend the `binop` type with a nullary `Expt` constructor:

   ```
   and binop = ... | Expt
   ```

2. Extend the `stringToBinop` function with a clause for `Expt`:

   ```
   and stringToBinop s =
     match s with
       ⋮
     | "^" -> Expt
     |  _ -> raise (SyntaxError ("invalid Bindex primop: " ^ s))
   ```

3. Extend the `binopToString` function with a clause for `Expt`:

   ```
   and binopToString p =
     match p with
       ⋮
     | Expt -> "^"
   ```

4. Extend the `binApply` function with a clause for `Expt`:

```
(* val run : Valex.pgm -> int list -> int *)
let rec run (Pgm(fmls,body)) ints =
  let flen = length fmls
  and ilen = length ints
  in
    if flen = ilen then
      eval (subst body (Env.make fmls (map (fun i -> Lit (Int i)) ints)))
    else
      raise (EvalError ("Program expected " ^ (string_of_int flen)
                        ^ " arguments but got " ^ (string_of_int ilen)))

(* val eval : Valex.exp -> valu *)
and eval exp =
  match exp with
    Lit v -> v
  | Var name -> raise (EvalError("Unbound variable: " ^ name))
  | PrimApp(op, rands) -> (primopFunction op) (map eval rands)
  | Bind(name,defn,body) -> eval (subst1 (Lit (eval defn)) name body)
  | If(tst,thn,els) ->
     (match eval tst with
       Bool true -> eval thn
     | Bool false -> eval els
     | v -> raise (EvalError ("Non-boolean test value "
                              ^ (valuToString v)
                              ^ " in if expression"))
     )
```

Figure 4: The substitution model evaluator for the VALEX kernel.

```
(* val binApply : Bindex.binop -> int -> int -> int *)
and binApply op x y =
  match op with
    ⋮
  | Expt -> if y < 0 then
              raise (EvalError ("Exponentiation by negative base: "
                                ^ (string_of_int y)))
            else
              let rec loop n ans = if n = 0 then ans else loop (n-1) (y*ans)
              in loop x ans
```

These four extensions are spread across two modules in two files of the BINDEX implementation. So adding a primitive requires touching many parts of the code and ensuring that they are consistent.

It would be preferable to have a means of specifying primitives that only requires changing *one* part of the code instead of four. The VALEX implementation has this feature. The collection of primitives handled by the language are specified in a single list `primops` of type `primop list`. Recall that `primop` is defined as:

```
and primop = Primop of var * (valu list -> valu) (* primop name, function *),
```

so each primitive is specified by providing its name and behavior. To facilitate the manipulation of primitive operators by their names, names are associated with the primitive operators in the environment `primopEnv`:

```
let primopEnv = Env.make (map (fun (Primop(name,_)) -> name) primops) primops

let isPrimop s = match Env.lookup s primopEnv with Some _ -> true | None -> false

let findPrimop s = Env.lookup s primopEnv
```

We now consider the specification of individual primitives. Here is one way we could specify the addition, less-than, and boolean negation primitives:

```
(* Addition primitive *)
Primop("+", fun vs -> match vs with
                        [Int i1, Int i2] -> Int (i1+i2)
                      | _ -> raise (EvalError "invalid args to +"))

(* Relational primitive *)
Primop("<", fun vs -> match vs with
                        [Int i1, Int i2] -> Bool (i1<i2)
                      | _ -> raise (EvalError "invalid args to <"))

(* Logical primitive *)
Primop("not", fun vs -> match vs with
                          [Bool b] -> Bool (not b)
                      | _ -> raise (EvalError "invalid args to not"))
```

Note that each Ocaml function must test the number of argument values and the types of these values to check that they are correct (or raise an exception if they aren't). This **dynamic type checking** process is required whenever a language has multiple value types and the types are not checked statically (i.e., before the program is run). We will study how to perform static type checking later in the semester.

To simplify checking the number of arguments and their types, we employ the auxiliary functions in Fig. 5. The **checker** functions `checkInt`, `checkBool`, and friends abstract over checking the type of an individual argument. The `checkZeroArgs`, `checkOneArgs`, and `checkTwoArgs` functions abstract over the checking for 0, 1, and 2 arguments, respectively. Each of these takes a number of checkers equal to the number of arguments for checking the individual arguments.

Abstracting over the dynamic type checking, particularly the details of generating helpful error messages, considerably simplifies the specification of our three sample primitives:

```
Primop("+", checkTwoArgs (checkInt, checkInt) (fun i1 i2 -> Int(i1+i2)))
Primop("<", checkTwoArgs (checkInt, checkInt) (fun i1 i2 -> Bool(i1<i2)))
Primop("not", checkOneArg checkBool (fun b -> Bool(not b)))
```

We can abstract even more over common patterns like arithmetic an relational operators:

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

With these further abstractions, our first two become:

```
Primop("+", arithop (+))
Primop("<", relop (<))
```

Figs. 6 and 7 present the complete specification of all Valex primitives.

```
let checkInt v f =
  match v with
    Int i -> f i
  | _  -> raise (EvalError ("Expected an integer but got: " ^ (valuToString v)))

let checkBool v f =
  match v with
    Bool b -> f b
  | _  -> raise (EvalError ("Expected a boolean but got: " ^ (valuToString v)))

let checkChar v f =
  match v with
    Char c -> f c
  | _  -> raise (EvalError ("Expected a char but got: " ^ (valuToString v)))

let checkString v f =
  match v with
    String s -> f s
  | _  -> raise (EvalError ("Expected a string but got: " ^ (valuToString v)))

let checkSymbol v f =
  match v with
    Symbol s -> f s
  | _  -> raise (EvalError ("Expected a symbol but got: " ^ (valuToString v)))

let checkList v f =
  match v with
    List vs -> f vs
  | _  -> raise (EvalError ("Expected a list but got: " ^ (valuToString v)))

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

let checkZeroArgs f =
  fun vs ->
    match vs with
      [] -> f ()
    | _ -> raise (EvalError ("Expected zero arguments but got: " ^ (valusToString vs)))

let checkOneArg check f =
  fun vs ->
    match vs with
      [v] -> check v f
    | _ -> raise (EvalError ("Expected one argument but got: " ^ (valusToString vs)))

let checkTwoArgs (check1,check2) f =
  fun vs ->
    match vs with
      [v1;v2] -> check1 v1 (fun x1 -> check2 v2 (fun x2 -> f x1 x2))
    | _ -> raise (EvalError ("Expected two arguments but got: " ^ (valusToString vs)))
```

Figure 5: Auxiliary functions for dynamic type checking of primitive operators.

```
let primops = [
  (* Arithmetic ops *)
  Primop("+", arithop (+));
  Primop("-", arithop (-));
  Primop("*", arithop ( * ));
  Primop("/", arithop (fun x y ->
                        if (y = 0) then
                          raise (EvalError ("Division by 0: "
                                            ^ (string_of_int x)))
                        else x/y));
  Primop("%", arithop (fun x y ->
                        if (y = 0) then
                          raise (EvalError ("Remainder by 0: "
                                            ^ (string_of_int x)))
                        else x mod y));

  (* Relational ops *)
  Primop("<", relop (<));
  Primop("<=", relop (<=));
  Primop("=", relop (=));
  Primop("!=", relop (<>));
  Primop(">=", relop (>=));
  Primop(">", relop (>));

  (* Logical ops *)
  Primop("not", checkOneArg checkBool (fun b -> Bool(not b)));
  Primop("and", logop (&&)); (* *not* short-circuit! *)
  Primop("or", logop (||)); (* *not* short-circuit! *)
  Primop("bool=", logop (=));

  (* Char ops *)
  Primop("char=", checkTwoArgs (checkChar, checkChar) (fun c1 c2 -> Bool(c1=c2)));
  Primop("char<", checkTwoArgs (checkChar, checkChar) (fun c1 c2 -> Bool(c1<c2)));
  Primop("int->char", checkOneArg checkInt (fun i -> Char(char_of_int i)));
  Primop("char->int", checkOneArg checkChar (fun c -> Int(int_of_char c)));
  Primop("explode", checkOneArg checkString
           (fun s -> List (let rec loop i chars =
                             if i < 0 then chars
                             else loop (i-1) ((Char (String.get s i)) :: chars)
                            in loop ((String.length s)-1) [])));
  Primop("implode", checkOneArg checkList
           (fun chars -> String (let rec recur cs =
                                   match cs with
                                     [] -> ""
                                   | ((Char c)::cs') -> (String.make 1 c) ^ (recur cs')
                                   | _ -> raise (EvalError "Non-char in implode")
                                 in recur chars)));
```

Figure 6: VALEX primitive operators, Part 1.

```
  (* String ops *)
  Primop("str=", checkTwoArgs (checkString,checkString) (fun s1 s2 -> Bool(s1=s2)));
  Primop("str<", checkTwoArgs (checkString,checkString) (fun s1 s2 -> Bool(s1<s2)));
  Primop("strlen", checkOneArg checkString (fun s -> Int(String.length s)));
  Primop("str+", checkTwoArgs (checkString,checkString) (fun s1 s2 -> String(s1^s2)));
  Primop("toString", checkOneArg checkAny (fun v -> String(valuToString v)));

  (* Symbol op *)
  Primop("sym=", checkTwoArgs (checkSymbol,checkSymbol) (fun s1 s2 -> Bool(s1=s2)));

  (* List ops *)
  Primop("prep", checkTwoArgs (checkAny,checkList) (fun v vs -> List (v::vs)));
  Primop("head", checkOneArg checkList
                (fun vs ->
                   match vs with
                     [] -> raise (EvalError "Head of an empty list")
                   | (v::_) -> v));
  Primop("tail", checkOneArg checkList
                (fun vs ->
                  match vs with
                    [] -> raise (EvalError "Tail of an empty list")
                  | (_::vs') -> List vs'));
  Primop("empty?", checkOneArg checkList (fun vs -> Bool(vs = [])));
  Primop("empty", checkZeroArgs (fun () -> List []));

  (* Predicates *)
  Primop("int?", pred (fun v -> match v with Int _ -> true | _ -> false));
  Primop("bool?", pred (fun v -> match v with Bool _ -> true | _ -> false));
  Primop("char?", pred (fun v -> match v with Char _ -> true | _ -> false));
  Primop("sym?", pred (fun v -> match v with Symbol _ -> true | _ -> false));
  Primop("string?", pred (fun v -> match v with String _ -> true | _ -> false));
  Primop("list?", pred (fun v -> match v with List _ -> true | _ -> false));

]
```

Figure 7: VALEX primitive operators, Part 2.

4 Desugaring

Syntactic sugar causes cancer of the semicolon.
— Alan Perlis

4.1 Motivation

It is hard work to add a new construct to a language like BINDEX or VALEX by extending the abstract syntax. For each construct, we have to perform the following steps:

1. Extend the `exp` data type with a constructor for the new construct.
2. Extend the `sexpToExp` function to parse the new construct.
3. Extend the `expToSexp` function to unparse the new construct.
4. Extend the `freeVarsExp` function to determine the free variables of the new construct.

5. Extend the `subst` function to perform substitution on the new construct.
6. Extend the environment model `eval` function handle the new construct.
7. Extend the substitution model `eval` function handle the new construct.

In sum, at least seven steps must be taken whenever we add a new construct. And this does not include other functions, like `uniquify` (for uniquely renaming expressions) that we might want. Nor does it consider other variants with which we might want to experiment, such as call-by-name evaluation. So even more functions might need to be updated in practice.

In some cases the functions are straightforward but tedious to extend. In other cases (especially constructs involving variable declarations), the clauses for the new construct can be rather tricky. In any of these cases, the work involved is an impediment to experimenting with new language constructs. This is sad, because ideally interpreters should encourage designing and tinkering with programming language constructs.

Fortunately, for many language constructs there is a way to have our cake and eat it too! Rather than extending lots of functions with a new clause for the construct, we can instead write a single clause that transforms the new construct into a pattern of existing constructs that has the same meaning. When this is possible, we say that the new construct is **syntactic sugar** for the existing constructs, suggesting that it makes the language more palatable without changing its fundamental structure. The process of remove syntactic sugar by rewriting a construct into other constructs of the language is known is **desugaring**. After a construct has been desugared, it will not appear in any expressions, and thus must not be explicitly handled by functions like `freeVarsExp`, `subst`, etc.

4.2 Simple Examples

Many constructs can be understood by translating them into other constructs of a language. For instance, the short-circuit conjunction construct

```
(&& E1 E2)
```

is equivalent to

```
(if E1 E2 #f)
```

and the short-circuit disjunction construct

```
(|| E1 E2)
```

is equivalent to

```
(if E1 #t E2)
```

As a more complex example, consider the `bindseq` expression:

```
(bindseq ((I1 E1)
          (I2 E2)
            ⋮
          (In En))
  Ebody)
```

This can be desugared into a nested sequence of `bind` expressions:

```
(bind I1 E1
  (bind I2 E2
      ⋮
    (bind In En
      Ebody) ... ))
```

Even `bindpar` can be desugared in a similar fashion as long as we rename all the bound variables. That is,

$$
\begin{array}{l}
\texttt{(bindpar ((}I_1\ \ E_1\texttt{)} \\
\qquad\qquad\quad \texttt{(}I_2\ \ E_2\texttt{)} \\
\qquad\qquad\qquad \vdots \\
\qquad\qquad\quad \texttt{(}I_n\ \ E_n\texttt{))} \\
\quad E_{body}\texttt{)}
\end{array}
$$

can be desugared to

$$
\begin{array}{l}
\texttt{(bind } I_1{}'\ E_1 \\
\quad \texttt{(bind } I_2{}'\ E_2 \\
\qquad\quad \vdots \\
\qquad \texttt{(bind } I_n{}'\ E_n \\
\qquad\quad E_{body}{}'\texttt{) ...))}
\end{array}
$$

where $I_1{}' \ldots I_n{}'$ are fresh variables and $E_{body}{}'$ is the result of renaming $I_1 \ldots I_n$ to $I_1{}' \ldots I_n{}'$ in E_{body}.

As a final VALEX example, consider the `cond` construct:

$$
\begin{array}{l}
\texttt{(cond (}E_{test_1}\ \ E_{result_1}\texttt{)} \\
\qquad\qquad\quad \vdots \\
\qquad\quad \texttt{(}E_{test_n}\ \ E_{result_n}\texttt{)} \\
\qquad\quad \texttt{(else } E_{default}\texttt{))}
\end{array}
$$

This desugars to:

$$
\begin{array}{l}
\texttt{(if } E_{test_1} \\
\qquad E_{result_1} \\
\qquad\qquad \ddots \\
\qquad\quad \texttt{(if } E_{test_n} \\
\qquad\qquad\quad E_{result_n} \\
\qquad\qquad\quad E_{default}\texttt{) ...)}
\end{array}
$$

It turns out that many programming language constructs can be expressed as synactic sugar for other other constructs. For instance, C and Java's `for` loop

```
for (init; test; update) {
  body
}
```

can be understood as just syntactic sugar for the `while` loop

```
{
  init;
  while (test) do {
    body;
    update;
  }
}
```

Other looping constructs, like C/Java's `do/while` and Pascal's `repeat/until` can likewise be viewed as desugarings. As another example, the C array subscripting expression `a[i]` is actually just syntactic sugar for `*(a + i)`, an expression that dereferences the memory cell at offset `i` from the base of the array pointer a.[3]

4.3 A First Cut at Desugaring: The All-at-once Approach

We can implement the kinds of desugaring examples given above by including a clause for each one in the `sexpToExp` function that parses s-expressions into instances of the Valex `exp` type. For example, the clause to handle `&&` would be:

```
| Seq [Sym "&&"; rand1x; rand2x] ->
    If(sexpToExp rand1x, sexpToExp rand2x, Lit (Bool false))
```

Here's a clause to handle `cond`:

```
| Seq (Sym "cond" :: clausexs) -> desugarCond clausexs
```

In this case, we need an auxiliary recursive function to transform the clauses into a nested sequence of `if` expressions:

```
and desugarCond clausexs =   (* clausesx is a list of sexp clauses *)
  match clausexs with
    [Seq[Sym "else"; defaultx]] -> sexpToExp defaultx
  | (Seq[testx; resultx])::restx ->
      If(sexpToExp testx, sexpToExp resultx, desugarCond restx)
  | _ -> raise (SyntaxError ("invalid cond clauses: " ^ (sexpToString (Seq clausexs))))
```

We call this approach to desugaring the **all-at-once** approach because it performs the complete desugaring in a single pass over the s-expression. Figs. 9 and 9 present the complete all-at-once desugarings for Valex.

```
(* val sexpToExp : Sexp.sexp -> exp *)
and sexpToExp sexp =
  match sexp with
    ⋮
  (* "All-at-once" desugarings *)
  | Seq [Sym "&&"; rand1x; rand2x] ->
      If(sexpToExp rand1x, sexpToExp rand2x, Lit (Bool false))
  | Seq [Sym "||"; rand1x; rand2x] ->
      If(sexpToExp rand1x, Lit (Bool true), sexpToExp rand2x)
  | Seq (Sym "cond" :: clausexs) -> desugarCond clausexs
  | Seq [Sym "bindseq"; Seq bindingxs; bodyx] ->
      let (names, defns) = parseBindings bindingxs in
      desugarBindseq names defns (sexpToExp bodyx)
  | Seq [Sym "bindpar"; Seq bindingxs; bodyx] ->
      let (names, defns) = parseBindings bindingxs in
      let names' = map StringUtils.fresh names in
      desugarBindseq names' defns (renameAll names names' (sexpToExp bodyx))
  | Seq (Sym "list" :: eltxs) -> desugarList eltxs
  | Seq [Sym "quote"; sexp] -> Lit (desugarQuote sexp)
    ⋮
```

Figure 8: Valex all-at-once desugarings, Part 1.

[3] An interesting consequence of this desugaring is that the commutativity of addition implies `a[i]` = `*(a + i)` = `*(i + a)` = `i[a]`. So in fact you can swap the arrays and subscripts in a C program without changing its meaning! Isn't C a fun language?

```
(* parse bindings of the form ((<namex_1> <defnx_1>) ... (<namex_n> <defnx_n>))
   into ([name_1;...;name_n], [defn_1; ...; defn_n]) *)
and parseBindings bindingxs =
  unzip (map (fun bindingx ->
               (match bindingx with
                 Seq[Sym name; defn] -> (name, sexpToExp defn)
               | _ -> raise (SyntaxError ("ill-formed bindpar binding"
                                           ^ (sexpToString bindingx)))))
           bindingxs)

and desugarCond clausexs =   (* clausesx is a list of sexp clauses *)
  match clausexs with
    [Seq[Sym "else"; defaultx]] -> sexpToExp defaultx
  | (Seq[testx; resultx])::restx ->
      If(sexpToExp testx, sexpToExp resultx, desugarCond restx)
  | _ -> raise (SyntaxError ("invalid cond clauses: "
                             ^ (sexpToString (Seq clausexs))))

(* defns and body have already been parsed *)
and desugarBindseq names defns body =
  foldr2 (fun name defn rest -> Bind(name, defn, rest)) body names defns

and desugarList eltxs =
  match eltxs with
    [] -> Lit(List[])
  | eltx::eltxs' -> PrimApp(valOf(findPrimop "prep"),
                            [sexpToExp eltx; desugarList eltxs'])

(* turns an sexp directly into a literal value *)
and desugarQuote sexp =
  match sexp with
    Sexp.Int i -> Int i
  | Sexp.Chr s -> Char s
  | Sexp.Str s -> String s
  | Sexp.Sym "#t" -> Bool true
  | Sexp.Sym "#f" -> Bool false
  | Sexp.Sym "#e" -> List []
  | Sexp.Sym s -> Symbol s
  | Seq eltxs -> List (map desugarQuote eltxs)
  | _ -> raise (SyntaxError ("invalid quoted expression" ^ (sexpToString sexp)))
```

Figure 9: VALEX all-at-once desugarings, Part 2.

4.4 A Better Approach: Incremental Desugaring Rules

Rather than desugaring constructs like `bindseq` all at once, we can desugar them incrementally, one step at a time, by applying rules like the following:

$$\begin{array}{rcl} \texttt{(bindseq ()}\ E_{body}\texttt{)} & \leadsto & E_{body} \\ \texttt{(bindseq ((}I\ E\texttt{) ...)}\ E_{body}\texttt{)} & \leadsto & \texttt{(bind}\ I\ E\ \texttt{(bindseq (...)}\ E_{body}\texttt{))} \end{array}$$

The first rule says that that a `bindseq` with an empty binding list is equivalent to its body. The second rule says that a `bindseq` with n bindings can be rewritten into a `bind` whose body is a `bindseq` with $n-1$ bindings. Here the ellipses notation "..." should be viewed as a kind of meta-variable that matches the "rest of the bindings" on the left-hand side of the rule, and means the same set of bindings on the right-hand side of the rule. Because the rule decreases the number of bindings in the `bindseq` with each rewriting step, it specifies the well-defined unwinding of a given `bindseq` into a finite number of nested `bind` expressions.

Fig. 10 shows a complete list of incremental desugaring rules for VALEX. There are no incremental rules for `bindpar` because the required renaming is challenging to implemented as a transformation on s-expressions. (Recall that the `rename` function works on instances of `exp`, not instances of `sexp`.)

We can implement the desugaring rules by changing the `sexpToExp` function to perform these rules. For instance, we can use the following clauses to implement `bindseq`:

```
| Seq [Sym "bindseq"; Seq []; bodyx] -> sexpToExp bodyx
| Seq [Sym "bindseq"; Seq ((Seq[Sym name; defnx])::bindingxs); body] ->
    sexpToExp (Seq[Sym "bind"; Sym name; defnx;
                          Seq[Sym "bindseq"; Seq bindingxs; body]])
```

Note that it is necessary to recursively invoke `sexpToSexp` on the result of transforming the `bindseq` s-expression into a `bind` expression with a `bindseq` body.

(&& E_{rand1} E_{rand2})	$\leadsto$	(if E_{rand1} E_{rand2} #f)
(|| E_{rand1} E_{rand2})	$\leadsto$	(if E_{rand1} #t E_{rand2})
(bindseq () E_{body})	$\leadsto$	E_{body}
(bindseq ((I E) ...) E_{body})	$\leadsto$	(bind I E (bindseq (...) E_{body}))
(cond (else $E_{default}$))	$\leadsto$	$E_{default}$
(cond (E_{test} $E_{default}$) ...)	$\leadsto$	(if E_{test} $E_{default}$ (cond ...))
(list)	$\leadsto$	#e
(list E_{hd} ...)	$\leadsto$	(prep E_{hd} (list ...))
(quote *int*))	$\leadsto$	*int*
(quote *char*))	$\leadsto$	*char*
(quote *string*))	$\leadsto$	*string*
(quote #t)	$\leadsto$	#t
(quote #f)	$\leadsto$	#f
(quote #e)	$\leadsto$	#e
(quote *sym*)	$\leadsto$	(sym *sym*)
(quote (*sexp1* ... *sexpn*))	$\leadsto$	(list (quote *sexp1*) ... (quote *sexpn*))

Figure 10: Desugaring rules for VALEX.

We can implement all the desugaring rules in Fig. 10 in a simlar fashion by directly extending `sexpToExp`. However, if we are not careful, it is easy to forget to call `sexpToExp` recursively on the results of our desugarings. It would be preferable to have an approach in which we could express the desugaring rules more directly and they were executed in a separate pass rather than being interleaved with the "regular" parsing of `sexpToExp`. Fig. 11 presents such an approach. It shows how to encode incremental desugaring rules into an OCAML `desugarRules` construct. The `desugar` function repeatedly applies these rules on an expression and all its subexpressions until no more of them match.

Fig. 12 shows how to integrate the `desugar` function with the `sexpToExp` function. We rename the existing `sexpToExp` to `sexpToExp'`. Then `sexpToSexp` is simply the result of invoking `sexpToExp'` on the result of desugaring a given s-expression. So parsing now occurs in two distinct phases: the desugaring phase (implemented by `desugar`) and the parsing phase (implemented by `sexpToExp'`).

```
let rec desugar sexp =
  let sexp' = desugarRules sexp in
    if sexp' = sexp then (* efficient in OCAML if they're pointer equivalent *)
      match sexp with
        Seq sexps -> Seq (map desugar sexps)
      | _ -> sexp
    else desugar sexp'

and desugarRules sexp =
  match sexp with

  (* Handle Intex arg refs as var refs *)
    Seq [Sym "$"; Sexp.Int i] -> Sym ("$" ^ (string_of_int i))

  (* 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]

  (* Scheme-style cond *)
  | Seq [Sym "cond"; Seq [Sym "else"; default]] -> default
  | Seq (Sym "cond" :: Seq [test; body] :: clauses) ->
      Seq [Sym "if"; test; body; Seq(Sym "cond" :: clauses)]

  | Seq [Sym "bindseq"; Seq[]; body] -> body
  | Seq [Sym "bindseq"; Seq ((Seq[Sym name;defn])::bindings); body]
      -> Seq[Sym "bind"; Sym name; defn; Seq[Sym "bindseq"; Seq bindings; body]]
  (* Note: can't handle bindpar here, because it requires renaming *)
  (* See sexpToExp' below for handling bindpar *)

  (* list desugarings *)
  | Seq [Sym "list"] -> Sym "#e"
  | Seq (Sym "list" :: headx :: tailsx) ->
      Seq [Sym "prep"; headx; Seq (Sym "list" :: tailsx)]

  (* Scheme-like quotation *)
  | Seq [Sym "quote"; Sexp.Int i] -> Sexp.Int i (* These are sexps, not Valex valus! *)
  | Seq [Sym "quote"; Chr i] -> Chr i
  | Seq [Sym "quote"; Str i] -> Str i
  (* Quoted special symbols denote themselves *)
  | Seq [Sym "quote"; Sym "#t"] -> Sym "#t"
  | Seq [Sym "quote"; Sym "#f"] -> Sym "#f"
  | Seq [Sym "quote"; Sym "#e"] -> Sym "#e"
  (* Other quoted symbols s denote (sym s) *)
  | Seq [Sym "quote"; Sym s] -> Seq [Sym "sym"; Sym s]
  (* (quote (x1 ... xn)) -> (list (quote x1) ... (quote xn)) *)
  | Seq [Sym "quote"; Seq xs] ->
      Seq (Sym "list" :: (map (fun x -> Seq[Sym "quote"; x]) xs))

  | _ -> sexp

(* For testing *)
let desugarString str =
  StringUtils.println (sexpToString (desugar (stringToSexp str)))
```

Figure 11: VALEX desugaring expressed via incremental desugaring rules.

```
and sexpToExp sexp = sexpToExp' (desugar sexp)

(* val sexpToExp' : Sexp.sexp -> exp *)
and sexpToExp' sexp =
  match sexp with
    Sexp.Int i -> Lit (Int i)
  | Sexp.Chr c -> Lit (Char c)
  | Sexp.Str s -> Lit (String s)
  (* Symbols beginning with # denote special values (not variables!) *)
  | Sym s when String.get s 0 = '#' -> Lit (stringToSpecialValue s)
  | Sym s -> Var s
  | Seq [Sym "sym"; Sym s] -> Lit (Symbol s)
  | Seq [Sym "bind"; Sym name; defnx; bodyx] ->
      Bind (name, sexpToExp' defnx, sexpToExp' bodyx)
  | Seq [Sym "if"; testx; thenx; elsex] ->
      If(sexpToExp' testx, sexpToExp' thenx, sexpToExp' elsex)
   (* Implement BINDPAR desugaring directly here.
      Can't handle desugarings with renamings in desugar function *)
  | Seq [Sym "bindpar"; Seq bindingxs; bodyx] ->
      let (names, defnxs) = parseBindings bindingxs
      in desugarBindpar names (map sexpToExp' defnxs) (sexpToExp' bodyx)
  (* This clause must be last! *)
  | Seq (Sym p :: randsx) when isPrimop p ->
      PrimApp(valOf (findPrimop p), map sexpToExp' randsx)
  | _ ->  raise (SyntaxError ("invalid Valex expression: " ^ (sexpToString sexp)))

(* Strings beginning with # denote special values *)
and stringToSpecialValue s =
  match s with
  | "#t" -> Bool true   (* true and false are keywords *)
  | "#f" -> Bool false  (* for literals, not variables *)
  | "#e" -> List []     (* empty list literal *)
  | _ -> raise (SyntaxError ("Unrecognized special value: " ^ s))

(* parse bindings of the form ((<name_1> <defnx_1>) ... (<name_n> <defnx_n>))
   into ([name_1;...;name_n], [defnx_1; ...; defnx_n]) *)
and parseBindings bindingxs =
  unzip (map (fun bindingx ->
               (match bindingx with
                 Seq[Sym name; defnx] -> (name, defnx)
               | _ -> raise (SyntaxError ("ill-formed bindpar binding"
                                          ^ (sexpToString bindingx)))))
          bindingxs)

(* desugars BINDPAR by renaming all BINDPAR-bound variables and
   then effectively treating as a BINDSEQ *)
and desugarBindpar names defns body =
  let freshNames = map StringUtils.fresh names in
    foldr2 (fun n d b -> Bind(n,d,b)) (renameAll names freshNames body) freshNames defns

(* val stringToExp : string -> exp *)
and stringToExp s = sexpToExp (stringToSexp s) (* Desugar when possible *)
```

Figure 12: A version of sexpToExp that incorporates desugaring.

```
(* val sexpToPgm : Sexp.sexp -> pgm *)
let rec sexpToPgm sexp =
  match sexp with
    Seq [Sym "valex"; Seq formals; bodyx] ->
          Pgm(map symToString formals, sexpToExp bodyx)
  (* Handle Bindex programs as well *)
  | Seq [Sym "bindex"; Seq formals; bodyx] ->
          Pgm(map symToString formals, sexpToExp bodyx)
  (* Handle Intex programs as well *)
  | Seq [Sym "intex"; Sexp.Int n; bodyx] ->
          Pgm(map
                (fun i -> "$" ^ (string_of_int i))
                (ListUtils.range 1 n),
              sexpToExp bodyx)
  | _ -> raise (SyntaxError ("invalid Valex program: " ^ (sexpToString sexp)))

(* val symToString : Sexp.sexp -> string *)
and symToString sexp =
  match sexp with
    Sym s -> s
  | _ -> raise (SyntaxError ("symToString: not a string -- " ^ (sexpToString sexp)))

(* val stringToPgm : string -> pgm *)
and stringToPgm s = sexpToPgm (stringToSexp s)
```

Figure 13: The VALEX `sexpToPgm` function. Note how it treats INTEX and BINDEX programs as VALEX programs.