CS251 Programming Languages **Handout # 18**
Prof. Lyn Turbak **February 26, 2002**
Wellesley College

IBEX

IBEX is a language that extends BINDEX with two new primitive datatypes (booleans and symbols) and some constructs controlled by booleans. We study IBEX for two reasons:

1. To show how multiple primitive datatypes are handled by the interpreter. In particular, the IBEX interpreter performs **dynamic type checking** to guarantee that operators are called only on the right types of operands.
2. To show that a language implementation can be significantly simplified by decomposing it into three parts:
   (a) a kernel,
   (b) synactic sugar, and
   (c) a standard library.

   We have already discussed how the Scheme language has this structure. In the context of IBEX, we see the implications of this structure for the interpreter.

1 The IBEX Language

IBEX is a language that extends BINDEX with booleans, symbols, and some boolean-controlled constructs.

1.1 Booleans

Whereas all values in INTEX and BINDEX are integers, IBEX also includes the two values `true` and `false`. 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 to booleans (`not`, `band`, `bor`, `bool=`). The `=` operator tests two integers for equality, while `!=` tests two integers for inequality. The `bool=` operator tests two booleans for equality. The `band` operator returns the logical conjunction ("and") of two boolean operands, while `bor` returns the logical disjunction ("or") of two boolean operands. For example:

```
ibex> (< 3 4)
true

ibex> (= 3 4)
false

ibex> (!= 3 4)
true
```

```
ibex> (not (= 3 4))
true

ibex> (band (< 3 4) (>= 5 5))
true

ibex> (band (< 3 4) (> 5 5))
false

ibex> (bor (< 3 4) (> 5 5))
true

ibex> (bor (> 3 4) (> 5 5))
false

ibex> (bool= false false)
true

ibex> (bool= true false)
false
```

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

```
ibex> (< 5)
Error! < expects 2 arguments, but was given 1.

ibex> (= 5 6 7)
Error! = expects 2 arguments, but was given 3.

ibex> (+ 1 true)
Error! + expects two integers, but was given an integer and a boolean.

ibex> (band true 3)
Error! band expects two booleans, but was given a boolean and an integer.

ibex> (= true false)
Error! = expects two integers, but was given two booleans.

ibex> (bool= 7 8)
Error! bool= expects two booleans, but was given two booleans.
```

1.2 Symbols

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

The only operation on symbols is equality, which is tested via the operator `sym=`. For example:

```
ibex> (sym= (symbol foo) (symbol foo))
true

ibex> (sym= (symbol foo) (symbol bar))
false
```

1.3 Boolean-Controlled Constructs

The key purpose of booleans is to direct the flow of control in a program with a branching control structure.

The fundamental control construct in IBEX is a conditional construct with the same syntax as Scheme's `if` construct: `(if` E_{test} E_{then} E_{else}`)`. Unlike Scheme, which treats any non-false value as true in the context of an `if`, IBEX requires that the test expression evaluate to a boolean. A non-boolean test expression is an error in IBEX.

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

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

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

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

ibex> (if (< 1 2) (+ 3 4) (div 5 0))
7

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

The last two test expressions 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 a Scheme definition of factorial:

```
(define fact
  (lambda (n)
    (if (= n 0)
        1
        (* 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.

IBEX also has a multi-clause conditional construct with the same syntax as Scheme's `cond` construct:

```
(program (x y)
  (cond ((< x y) (symbol less))
        ((= x y) (symbol equal))
        (else (symbol greater))))
```

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

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

(scand E_{rand1} E_{rand2})
(scor E_{rand1} E_{rand2})

These are similar to the binary operators `band` and `bor`, except that E_{rand2} is never evaluated if the result is determined by the value of E_{rand1}. For instance, in `scand`, E_{rand1} is first evaluated to the value V_{rand1}. If V_{rand1} is `true`, then E_{rand2} is evaluated, and its value (which must be a boolean) is returned as the value of the `scand` expression. But if V_{rand1} is `false`, then `false` is immediately returned as the value of the `scand` and E_{rand2} is never evaluated. Similarly, in `scor`, if V_{rand1} is `true`, a value of `true` is returned for the `scor` expression without E_{rand2} being evaluated; otherwise the value of E_{rand2} (which must be a boolean) is returned. In contrast, both operand expressions of `band` and `bor` are always evaluated.

```
ibex> (band (= 1 2) (> 3 4 5))
Error! > expects 2 arguments, but was given 3.

ibex> (scand (= 1 2) (> 3 4 5))
false

ibex> (bor (< 1 2) (+ 3 4))
Error! bor expects two boolean arguments,
but was given a boolean and an integer.

ibex> (scor (< 1 2) (+ 3 4))
true
```

In many cases `scand`/`scor` behave indistinguishably from the boolean operators `band`/`bor`, 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 IBEX, such a situation occurs when evaluating E_2 would otherwise give an error. For instance, consider the following IBEX program:

```
(program (x)
  (if (scor (= x 0)
            (> (div 100 x) 7))
      (+ x 1)
      (* x 2)))
```

This program returns 1 when applied to 0. But if the `scor` were changed to `bor`, the program would encounter a divide-by-zero error when applied to 0 because the `div` application would be evaluated 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 `forall?` Scheme function we studied earlier this semester:

```
(define forall?
  (lambda (pred lst)
    (if (null? lst)
        #t
        (and (pred (car lst))
             (forall? pred (cdr lst))))))
```

In Scheme, `and` is the short-circuit conjunction operator. It is important to use a short-circuit operator in `forall?` because it causes the recursion to stop as soon as an element is found for which the predicate is false. If `and` were not a short-circuit operator, then `forall?` 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:

```
public void insertion_sort (int[] A) {
  for (int i = A.length - 1; i >= 0; i--) {
    int x = A[i];
    int j = i-1;
    // Insertion loop
    while ((j >= 0) && (A[j] > x)) {
      A[j+1] = A[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.

2 The IBEX Kernel

The IBEX 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`).

We shall see that these five expression types are sufficient for representing *all* IBEX expressions. The abstract syntax for the IBEX kernel can be found in Figs. 9–10 in Appendix A.

The complete environment model evaluator for IBEX is shown in figref 1. The complete substitution model evaluator for IBEX is shown in figref 2.

3 Desugaring

Syntactic sugar causes cancer of the semicolon.
— Alan Perlis

It is hard work to add a new construct to a language like BINDEX or IBEX. For each construct, we have to extend a number of functions in the implementations of these languages:

- The `free-vars` function needs a new clause that specifies how to find the free variables of the construct.
- The `rename` function needs a new clause that specifies how to rename a free variable within the construct.
- The `subst` function needs a new clause that specifies how to substitute expressions for free variables within the construct.
- The `subst-eval` function needs a new clause that specifies how to evaluate the construct in the substitution model.
- The `env-eval` function needs a new clause that specifies how to evaluate the construct relative to an environment in the environment model.

So, all in all, five functions need to be updated whenever we add a new construct. And this is only for call-by-value evaluation. If there are other dimensions (such as call-by-name vs. call-by-value evaluation, or, as we'll soon see, lexical vs. dynamic scope), even more functions may need to be updated.

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.

```
;; Invoke a program on any number of integer arguments
(define env-run
  (lambda (pgm ints)
    (env-eval (desugar (program-body pgm))
              (env-make (program-formals pgm) ints))))

;; Evaluate EXP relative to the environment ENV.
;; The environment ENV is a collection of delayed substitutions.
(define env-eval
  (lambda (exp env)
    (cond
     ((literal? exp)
      (literal-value exp))
     ((varref? exp)
      (let ((probe (env-lookup (varref-name exp) env)))
        (if (unbound? probe)
            (throw 'unbound-variable (varref-name exp))
            probe)))
     ((bind? exp)
      (env-eval (bind-body exp)
                (env-bind (bind-name exp)
                          (env-eval (bind-defn exp) env)
                          env)))

     ;; -----------------------------------------------------
     ;; New in IBEX

     ;; PRIMAPP generalizes BINAPP
     ((primapp? exp)
      (primapply (primapp-rator exp)
                 (map (lambda (rand) (env-eval rand env))
                      (primapp-rands exp))))

     ((if? exp)
      (let ((test (env-eval (if-test exp) env)))
        (if (not (mini-boolean? test))
            (throw 'if:non-boolean-test test)
            (if (mini-bool-to-scheme-bool test)
                (env-eval (if-then exp) env)
                (env-eval (if-else exp) env)))))
     ;; -----------------------------------------------------

     (else (throw 'env-eval:unknown-expression exp))
     )))
```

Figure 1: The environment model evaluator for the IBEX kernel.

```
;; Invoke a program on any number of input values
(define subst-run
  (lambda (pgm vals)
    (literal-value
     (subst-eval (subst* (map make-literal vals)
                         (program-formals pgm)
                         (desugar (program-body pgm)))))))

;; SUBST-EVAL takes an expression and returns another expression
;; (e.g., an integer literal node, *not* an integer) that is the
;; result of evaluting the expression.
(define subst-eval
  (lambda (exp)
    (cond
     ((literal? exp) exp)
     ((varref? exp)
      (throw 'unbound-variable (varref-name exp)))
     ((bind? exp)
      (subst-eval
       (subst1 (subst-eval (bind-defn exp))
               (bind-name exp)
               (bind-body exp))))

     ;; -----------------------------------------------------
     ;; New in IBEX

     ;; PRIMAPP generalizes BINAPP
     ((primapp? exp)
      (make-literal
       (primapply (primapp-rator exp)
                  (map (compose literal-value subst-eval)
                       (primapp-rands exp)))))

     ((if? exp)
      (let ((test (literal-value (subst-eval (if-test exp)))))
        (if (not (mini-boolean? test))
            (throw 'if:non-boolean-test test)
            (if (mini-bool-to-scheme-bool test)
                (subst-eval (if-then exp))
                (subst-eval (if-else exp))))))
     ;; -----------------------------------------------------

     (else (error 'subst-eval:unknown-expression exp))
     )))
```

Figure 2: The substitution model evaluator for the IBEX kernel.

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 `free-vars`, `rename`, etc.

We will study desugaring via several examples, starting with simple ones and working our way up to more complex ones.

3.1 Simple Desugaring Rules

Let's begin by considering IBEX's `scand` and `scor` constructs. We *could* extend IBEX with `scand` and `scor` by extending `free-vars`, `rename`, etc. However, observe that `scand` and `scor` are really just particular patterns for using `if`:

$$\begin{array}{rcl} \texttt{(scand } E_1\ E_2\texttt{)} & \leadsto & \texttt{(if } E_1 \texttt{ (if } E_2 \texttt{ true false) false)} \\ \texttt{(scor } E_1\ E_2\texttt{)} & \leadsto & \texttt{(if } E_1 \texttt{ true (if } E_2 \texttt{ true false))} \end{array}$$

These notations are called **desugaring rules**. Here, the arrow $\leadsto$, which can be pronounced as "rewrites to" or "desugars to", specifies a rule for rewriting the IBEX expression to the left of the arrow to the IBEX expression to the right of the arrow without changing its meaning. The E_1 and E_2 are **meta-variables**[1] that stand for any IBEX expressions. Whatever IBEX expressions are matched by E_1 and E_2 on the left-hand side of the rule are substituted in for E_1 and E_2 on the right-hand side of the rule. For instance, the `scor` desugaring rule specifies that the IBEX expression `(scor (= x 0) (> (div 100 x) 7))` can be rewritten to the IBEX expression `(if (= x 0) true (> (div 100 x) 7))`.

As another example of a desugaring rule, consider `bindseq2`, a specialized version of the `bindseq` construct that requires exactly two bindings. It has the form:

$$\texttt{(bindseq2 ((}I_1\ E_1\texttt{) (}I_2\ E_2\texttt{)) } E_{body}\texttt{)}$$

The `bindseq2` construct can be rewritten into two occurrences of the `bind` construct via the following desugaring rule:

$$\texttt{(bindseq2 ((}I_1\ E_1\texttt{) (}I_2\ E_2\texttt{)) } E_{body}\texttt{)} \leadsto \texttt{(bind } I_1\ E_1 \texttt{ (bind } I_2\ E_2\ E_{body}\texttt{))}$$

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
}
```

[1]These are called meta-variables because they are variables of the meta-language that is used to talk about IBEX expressions. A meta-variable can be bound to any IBEX expression, including a reference to an IBEX (non-meta-)variable. In general, the prefix **meta-** indicates something at a higher level, usually one that "reifies" entities at the next level down. For instance, "meta-humor" is humor that is about humor itself. The notion of "meta" is crucial not only in computer science and mathematics, but also in artistic fields, where it is an important ingredient of postmodernism.

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`.[2]

3.2 Implementing Simple Desugaring Rules

If constructs like `scand`, `scor`, and `bindseq2` could be automatically removed by rewriting them according to their desugaring rules, then the resulting program could be executed in a regular IBEX interpreter without any need to extend each of the numerous functions implementing the interpreter. In this section, we present one approach for desugaring IBEX programs. The approach is based on a pair of Scheme functions with the following specifications:

(**desugar-program** *pgm*)
Returns the IBEX program that results from *pgm* by removing all syntactic sugar constructs according to their desugaring rules.

(**desugar** *exp*)
Returns the IBEX expression that results from *exp* by removing all syntactic sugar constructs according to their desugaring rules.

In this approach, only the `desugar` function needs to be extended when new syntactic sugar constructs are added to IBEX.

Figure 3 presents an implementation of `desugar-program` and `desugar`. The `desugar` function performs a case analysis on the node type of the expression. There are two categories of node types: (1) **kernel nodes** – those core node types that are not syntactic sugar; and (2) **sugar nodes** – those node types that can be desugared into kernel nodes.

For the kernel nodes, `desugar` simply makes a copy of each kernel node after recursively desugaring the components of the node. This means that if the given expression did not contain any syntactic sugar at all, `desugar` would just return a copy of the expression. For sugar nodes, `desugar` constructs new expressions according to the desugaring rules. The new expressions are formed by using kernel nodes to glue together the results of recursively desugaring the subexpressions of the sugar nodes. A simple inductive argument shows that `desugar` removes all sugar nodes. The inductive argument depends critically on the following facts: (1) the result of `desugar` is formed using only the constructors for kernel nodes[3] (2) all subexpressions are recursively desugared.

[2]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?

[3]It is also possible to use constructors for sugar nodes, as long as the result of every such constructor is itself desugared via `desugar`. Of course, to avoid an infinite regress, it is imperative to assure in such cases that the new sugar nodes are somehow "smaller" than the original ones. See the `bindseq` and `cond` desugarings in the next section for an example of this.

```
(define desugar-program
  (lambda (pgm)
    (make-program (program-formals pgm)
                  (desugar (program-body pgm)))))

(define desugar
  (lambda (exp)
    (cond
         ;; -----------------------------------------------------------
         ;; KERNEL EXPRESSIONS

          ((literal? exp) exp)
          ((varref? exp) exp)
          ((primapp? exp)
           (make-primapp (primapp-rator exp)
                         (map desugar (primapp-rands exp))))
          ((bind? exp)
           (make-bind (bind-name exp)
                      (desugar (bind-defn exp))
                      (desugar (bind-body exp))))
          ((if? exp)
           (make-if (desugar (if-test exp))
                    (desugar (if-then exp))
                    (desugar (if-else exp))))
          ;; -----------------------------------------------------------
          ;; SYNTACTIC SUGAR

          ((scand? exp)
           (make-if (desugar (scand-rand1 exp))
                    (make-if (desugar (scand-rand2 exp))
                             (scheme-bool-to-mini-bool #t)
                             (scheme-bool-to-mini-bool #f))
                        (scheme-bool-to-mini-bool #f)))

          ((scor? exp)
           (make-if (desugar (scor-rand1 exp))
                    (scheme-bool-to-mini-bool #t)
                    (make-if (desugar (scor-rand2 exp))
                             (scheme-bool-to-mini-bool #t)
                             (scheme-bool-to-mini-bool #f))))

          ((bindseq2? exp)
           (make-bind (bindseq2-name1 exp)
                      (bindseq2-defn1 exp)
                      (make-bind (bindseq2-name2 exp)
                                 (bindseq2-defn2 exp)
                                 (bindseq2-body exp))))
          ;; -----------------------------------------------------------
          (else (error "DESUGAR: unrecognized expression -- " exp))
          )))
```

Figure 3: IBEX desugaring functions. A new syntactic sugar construct can be added to IBEX by adding a clause to `desugar`.

In Figure 3, programs and expressions are pulled apart and put together using Scheme functions for manipulating the abstract syntax of IBEX. The operations on kernel syntax summarized in Appendix A. We do not present formal specifications for the operations on sugar syntax, but hope that they are clear from context.

Note that all desugaring does is to decompose each tree rooted at a sugar node into its subtrees and reassemble the (desugared) subtrees to form a tree that does not use any sugar nodes. In particular, desugaring does *not* perform any evaluation. Rather, it constructs expression trees that can be evaluated (or otherwise manipulated) at a later time. This means that the only Scheme functions that should be called in `desugar` are (1) the IBEX abstract syntax functions; (2) various list functions (for manipulating lists of operands, clauses, etc.); (3) user-defined auxiliary functions helpful for the desugaring; and (4) a small number of renaming and substitution functions (to be discussed later).

Evaluation occurs at some time *after* desugaring. For instance, consider the entry point to the environment model evaluator for IBEX:

```
(define env-run
  (lambda (pgm ints)
    (env-eval (desugar (program-body pgm))
              (env-make (program-params pgm) ints))))
```

When given a program, `env-eval` first uses `desugar` to remove syntactic sugar from the program body, and only then does it use `env-eval` to evaluate the desugared body.

Thus, when evaluating IBEX programs, there are really two distinct times: **desugaring time**, when the program body is desugared and **evaluation time**, when the desugared body is evaluated. During desugaring time, IBEX expressions are transformed to other IBEX expressions, but no evaluation is performed.[4] During evaluation time, the desugared IBEX expression is evaluated.

3.3 More Complex Desugarings

The desugarings considered thus far are atypically simple. More commonly, desugarings involve more complex manipulations of expressions. Here we consider desugaring two less trivial constructs.

3.3.1 `bindseq`

The general `bindseq` expression allows naming any number of values using the form:

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

Desugaring rules for constructs with arbitrary numbers of subforms are often expressed recursively. Here are a pair of rules that specify the desugaring of `bindseq`:

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

[4]It is possible to imagine desugarers that perform some evaluation in addition to rearranging the subexpressions of an expression. However, for simplicity, we will assume that no evaluation takes place during desugaring time.

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.

How can the desugaring rules for `bindseq` be expressed as a clause in `desugar`? There are two basic approaches:

1. *Smaller-sugar-node approach*: The most straightforward approach is to directly encode the recursive desugaring from the rules in the `desugar` clause by explicitly constructing a `bindseq` clause with a smaller number of bindings and calling `desugar` on it:

   ```
   ((bindseq? exp)
    (let ((names (bindseq-names exp))
          (defns (bindseq-defns exp))
          (body (bindseq-body exp)))
      (if (null? names)
          (desugar body)
          (make-bind (car names)
                     (desugar (car defns))
                     (desugar (make-bindseq (cdr names)
                                            (cdr defns)
                                            body))))))
   ```

   Note that the result of `make-bindseq` *must* itself be desugared. If it weren't, then desugaring wouldn't satisfy its contract to remove all sugar nodes.

2. *All-at-once approach*: Another approach is to have the clause for `bindseq` create the nested `bind` expressions all at once without constructing smaller `bindseq` expressions. There are many ways to do this. A particularly concise way is to use the higher-order `foldr2` function:

   ```
   ((bindseq? exp)
    (foldr2 make-bind
            (desugar (bindseq-body exp))
            (bindseq-names exp)
            (map desugar (bindseq-defns exp))))
   ```

   An alternative is to use an auxiliary recursive function to perform the unwinding of `bindseq` into nested `bind`s:

   ```
   ((bindseq? exp)
    (desugar-bindseq (bindseq-names exp)
                     (bindseq-defns exp)
                     (bindseq-body exp)))
   ```

   Here, `desugar-bindseq` is an auxiliary function defined as follows:

```
(define desugar-bindseq
  (lambda (names defns body)
    (if (null names)
        (desugar body)
        (make-bind (car names)
                   (desugar (car defns))
                   (desugar-bindseq (cdr names)
                                    (cdr defns)
                                    body)))))
```

Is one approach better than the other? In the case of `bindseq`, not really. The smaller-sugar-node approach may be easier to understand because it corresponds more directly to the desugaring rules. However, in this case the all-at-once approach (at least the `foldr` version) has the advantage of being more concise.

3.3.2 `cond`

As a second example of a non-trivial desugaring, we consider extending IBEX with a Scheme-like `cond` construct with the following syntax:

$$
\begin{array}{l}
\texttt{(cond (}E_{test_1}\ \ E_{body_1}\texttt{)} \\
\qquad\qquad\quad \vdots \\
\qquad\quad\ \texttt{(}E_{test_n}\ \ E_{body_n}\texttt{)} \\
\qquad\quad\ \texttt{(else }E_{default}\texttt{))}
\end{array}
$$

The meaning of `cond` is specified by the following desugaring rules:

$$
\begin{array}{rcc}
\texttt{(cond (else }E_{default}\texttt{))} & \rightsquigarrow & E_{default} \\
\texttt{(cond (}E_{test}\ \ E_{body}\texttt{) ...)} & \rightsquigarrow & \texttt{(if }E_{test}\ \ E_{body}\ \texttt{(cond ...))}
\end{array}
$$

For example, the leftmost expression below desugars to the rightmost one:

```
(cond ((> b 0) b)
      ((< a b) (* a b))
      ((scand (= a b)
              (< b c))
       (+ b c))
      (else (bindseq ((d (+ a b))
                      (e (* c d)))
              (+ d e))))
```

```
(if (> b 0)
    b
    (if (< a b)
        (* a b)
        (if (if (= a b)
                (if (< b c)
                    true
                    false)
                false)
            (+ b c)
            (bind d (+ a b)
              (bind e (* c d)
                (+ d e))))))
```

To implement the `cond` desugaring, we can again either use the smaller-sugar-node approach or the all-at-once approach:

1. *Smaller-sugar-node approach:*

```
((cond? exp)
 (let ((clauses (cond-clauses exp))
       (default (cond-default exp)))
   (if (null? (cond-clauses exp))
       (desugar default)
       (make-if (desugar (cond-clause-test (first clauses)))
                (desugar (cond-clause-body (first clauses)))
                (desugar (make-cond (cdr clauses) default))))))
```

2. *All-at-once approach:*

```
((cond? exp)
 (foldr (lambda (clause ifs)
          (make-if (desugar (cond-clause-test clause))
                   (desugar (cond-clause-body clause))
                   ifs))
        (desugar (cond-default exp))
        (cond-clauses exp)))
```

3.4 Naming Subtleties

Perhaps the trickiest aspect of desugaring is dealing with naming issues. There are two naming issues that are particularly common:

- Using source language names to avoid recomputing expressions.
- Avoiding accidental name capture.

We will explore these issues in the context of two constructs: `choose` and `bindpar`.

3.4.1 `choose`

Consider the following construct that we might wish to add to IBEX:

> (**choose** E_{int} E_{pos} E_{zero} E_{neg})
> Evaluates E_{int} to the value V_{int}, which should be an integer. If V_{int} is positive, returns the value of E_{pos}; if V_{int} is zero, returns the value of E_{zero}; and if V_{int} is negative, returns the value of E_{neg}. In each case, exactly one of E_{pos}, E_{zero}, and E_{neg} is evaluated.

It is easy to add `choose` to IBEX as syntactic sugar. A straightforward desugaring would be:

```
;; First desugaring
(if (> E_int 0)
    E_pos
    (if (= E_int 0)
        E_zero
        E_neg))
```

A drawback of this desugaring is that if the value of E_{int} is not positive, the expression E_{int} will be evaluated twice. This is undesirable, since E_{int} might be expensive to compute.

To avoid recomputation, we can use a binding construct in the source language to name the result of E_{int} so that it is only computed once. In our case, the source language is IBEX, and the appropriate binding construct is `bind`. Here is a stab at this approach:

```
;; Second desugaring
(bind x $E_{int}$
  (if (> x 0)
      $E_{pos}$
      (if (= x 0)
        $E_{zero}$
        $E_{neg}$)))
```

We have use the fixed IBEX variable name `x` to name the result of evaluating E_{int}. We emphasize that the desugarer is *not* evaluating E_{int} – it doesn't evaluate *any* IBEX expressions. Rather, it is inserting an IBEX `bind` expression so that when the desugared expression *is* finally evaluated, the result of evaluated E_{int} will be named `x`. Keeping straight the difference between desugaring time and evaluation time can be tricky, but it is very important.

A problem with the second desugaring is that the name `x` might accidentally capture a free variable named `x` and change the meaning of the program. For example, consider the following program:

```
(program (x)
  (choose (- x 10) (* x x) (* x 2) (+ x 1)))
```

If we desugar the program via the second desugaring, we obtain:

```
(program (x)
  (bind x (- x 10)
    (if (> x 0)
        (* x x)
        (if (= x 0)
            (* x 2)
            (+ x 1)))))
```

But this is incorrect! The name `x` introduced by the `bind` has "captured" the free variable `x` in the three branch expressions, thereby changing the meaning of the program. For example, the program should return $15 \times 15 = 225$ when run on 15, but the desugared program will return $5 \times 5 = 25$.

There is no fixed name for the `bind` variable that we can choose that will avoid this name capturing problem. Instead, we must choose a **fresh** name that does not conflict with any of the free variable names appearing in the subexpressions. In a desugaring rule, this is usually expressed by a side condition indicated which variables are fresh. So a correct desugaring for `choose` is:

```
;; Correct desugaring
(bind $I$ $E_{int}$ ; where $I$ is fresh
  (if (> $I$ 0)
      $E_{pos}$
      (if (= $I$ 0)
        $E_{zero}$
        $E_{neg}$)))
```

How do we implement "freshness"? The renaming module provides a handy function for this purpose:

> **(name-not-in** *name names*)
> Returns the first "subscripted" version of *name* that is not an element of name list *names*. For example, `(name-not-in 'a '(b c d))` returns `a_1` and `(name-not-in 'a '(a_2 a_4 a_1))` returns `a_3`. If *name* is already subscripted, the existing subscript is removed before computing the new one. For instance `(name-not-in 'a_7 '(a_2 a_4 a_1))` returns `a_3`.

Using `name-not-in`, we can express the desugaring for `choose` as the following `desugar` clause:

```
((choose? exp)
 (let ((dint (desugar (choose-int exp)))
       (dpos (desugar (choose-pos exp)))
       (dzero (desugar (choose-zero exp)))
       (dneg (desugar (choose-pos exp))))
   (let ((new-name (name-not-in 'x (free-vars-list (list dpos dzero dneg)))))
     (make-bind new-name
                dint
                (make-if (make-primapp '>
                           (list (make-varref new-name)
                                 (make-literal 0)))
                         dpos
                         (make-if (make-primapp '=
                                    (list (make-varref new-name)
                                          (make-literal 0)))
                                  dzero
                                  dneg))))))
```

There are several things to notice about this clause:

- `name-not-in` is used to pick a name that is guaranteed not to conflict with any free variables in the three branch subexpressions. In the counter-example above, it would choose `x_1`, which would not conflict with the free variable `x`.
- The free variables are calculated relative to the *desugared* subexpressions, not relative to the undesugared ones. This is important. The `free-vars` function is written a case dispatch on the kernel node types of IBEX, and it will not behave correctly if it is called on expressions that contain any sugar nodes.
- Because the newly chosen name will be used twice in the desugaring, it is helpful to use *Scheme's* `let` binding construct to name it (so it doesn't have to be calculated twice at desugaring time).
- The `make-bind` invocation constructs the IBEX `bind` construct that will later name the result of evaluating E_{int} at evaluation time. Similarly `make-if` and `make-primapp` construct IBEX nodes that will later be evaluated at evaluation time. *No part of the* `choose` *expression is evaluated at desugaring time!*

Students new to desugaring often confuse the Scheme expression that is constructing the IBEX expression with the resulting IBEX expression. So they might use `if` instead of `make-if`, or `>` instead of `(make-primapp '> ...)`. But these are wrong because they would imply that part of the IBEX expressions is being evaluated at desugaring time, which is not true. Furthermore, they would often introduce type errors; you can't apply Scheme's `>` to two IBEX expressions, only to two numbers!

3.4.2 `bindpar`

As a final example of a desugaring involving renaming, we consider IBEX desugaring of `bindpar` into `bind`. A single `bindpar` with multiple bindings can be desugared into a sequence of nested `bind` constructs *as long as* care is taken to appropriately rename the variables declared by the `bind` in order to avoid accidental name capture. For example, consider the following IBEX program:

```
(program (x)
  (bind a (+ x 1)
    (bindpar ((a (* x a))
              (b (+ x a)))
      (bindpar ((a (- a b))
                (b (* a a)))
        (+ a b)))))
```

This should desugar to:

```
(program (x)
  (bind a (+ x 1)
    (bind a_1 (* x a)
      (bind b (+ x a)
        (bind a_2 (- a_1 b)
          (bind b (* a_1 a_1)
            (+ a_2 b)))))))
```

Note that if all occurrences of `a_1` and `a_2` were replaced by `a`, the desugaring would effectively give the `bindpar` construct the same semantics as `bindseq`, which would clearly be wrong. Any time a definition expression references a free variable whose name happens to be bound in an earlier binding of the same `bindpar`, it is necessary to α-rename the bound variable of the earlier binding to avoid capture of the free variable.

The details of how this renaming is accomplished in the IBEX desugarer is show in figure 4. The clause for desugaring `bindpar` uses a pair of helper functions, `triple` and `untriple`, that combine and decompose three-element lists of values. For instance, `(triple 1 (+ 2 3) (* 4 5))` returns the three-element list `(1 5 20)`. If we give this triple the name `tpl`, then evaluating the expression `(untriple tpl (lambda (a b c) (* (+ a b) c)))` returns 120. Using a `lambda` abstraction in this fashion to deconstruct a list into named components is a common Scheme idiom.

The `bindpar` desugaring works from the bottom up, first recursively desugaring the definition and body expressions of the `bindpar`. Using `foldr2`, it calculates for each binding in the `bindpar` (bottom up, from last to first) a triple of:

- A list of the (possibly renamed) bound variables of the current and subsequent bindings;

```
(define desugar
  (lambda (exp)
    (cond
      ⋮
      ((bindpar? exp)
       (let ((names (bindpar-names exp))
             (desugared-defns (map desugar (bindpar-defns exp)))
             (desugared-body (desugar (bindpar-body exp))))
         (untriple (foldr2
                    ;; This FOLDR returns a triple (3-element list) containing:
                    ;; (1) A list of potentially renamed names.
                    ;; (2) A set of the free variables the defns.
                    ;; (3) A body expression renamed consistently with any
                    ;;     renamings in (1).
                    (lambda (name defn trip)
                      (untriple trip
                        (lambda (new-names defn-fvs new-body)
                          (if (set-member? name defn-fvs)
                              ;; Case where necessary to rename bound variable
                              (let ((new-name (name-not-in
                                                name
                                                set-union defn-fvs
                                                          (free-vars new-body)))))
                                (triple (cons new-name new-names)
                                        (set-union (free-vars defn) defn-fvs)
                                        (rename1 name new-name new-body)))
                              ;; Case where bound variable may stay unchanged
                              (triple (cons name new-names)
                                      (set-union (free-vars defn) defn-fvs)
                                      new-body)))))
                    (triple '()            ;; new names
                            (set-empty)    ;; FVs of defns
                            desugared-body) ;; new body
                    names
                    desugared-defns)
           (lambda (new-names defn-fvs new-body)
             (foldr2 make-bind new-body new-names desugared-defns)))))
      ⋮
      )))

(define triple
  (lambda (a b c)
    (list a b c)))

(define untriple
  (lambda (trip deconstructor)
    (deconstructor (first trip)
                   (second trip)
                   (third trip))))
```

Figure 4: The IBEX desugaring of `bindpar` into `bind`.

- The free variables appearing in the definition of the current binding and the definitions of subsequent bindings in the `bindpar`;
- A version of the `bindpar` body that reflects any renamings of the bound variable of the current binding.

In a final `foldr2`, it constructs a nested sequence of `bind`s from (1) the final renamed body, (2) the list of (possibly renamed) bound variables, and (3) the desugared definitions.

Let's study how this approach is used to desugar the `bindpar`s in the sample program from above:

```
(program (x)
  (bind a (+ x 1)
    (bindpar ((a (* x a))
              (b (+ x a)))
      (bindpar ((a (- a b))
                (b (* a a)))
        (+ a b)))))
```

To desugar the outer `bindpar`, we first desugar its two definitions, `(* x a)` and `(+ x a)`, which trivially desugar to themselves. We also desugar the body, which is itself a `bindpar`. The body of the inner `bindpar` and all of its definitions trivially desugar to themselves. The following diagram shows the triples calculated by the first `foldr2` in the `bindpar` clause of the IBEX `desugar` function:

```
                             Bound var   FVs     Renamed
                              names               body

(bindpar ((a (- a b))        (a_1, b)  a, b  (+ a_1 b)
                                 ^        ^         ^
                                 |        |         |
          (b (* a a)))          (b)      a     (+ a b)
                                 ^        ^         ^
                                 |        |         |
  (+ a b))                      ( )            (+ a b)
```

So the desugaring of the inner `bindpar` is:

```
(bindpar ((a_1 (- a b))
          (b (* a a)))
  (+ a_1 b))
```

Note that the occurrences of `a` in `(* a a)` are *not* renamed to `a_1` because they refer to the `a` declared by the outer `bindpar`, not the inner one. Indeed, the bound variable `a` declared by the inner `bindpar` had to be renamed to avoid capturing the two references to `a` in `(* a a)`.

Having desugared the body and definitions of the outer `bindpar`, we are ready to apply `foldr2` to accumulate a triple, as shown below:

```
                                Bound var  FVs          Renamed
                                 names                    body

     (bindpar ((a (* x a))      (a_1, b)  x, a  (bindpar ((a_2 (- a_1 b))
                                                            (b (* a_1 a_1)))
                                                    (+ a_2 b))
                                    ^        ^              ^
                                    |        |              |
               (b (+ x a)))        (b)    x, a  (bindpar ((a_1 (- a b))
                                    ^        ^              (b (* a a)))
                                    |        |      (+ a_1 b))
                                    |        |              ^
                                    |        |              |
       (bindpar ((a_1 (- a b))     ( )          (bindpar ((a_1 (- a b))
                 (b (* a a)))                               (b (* a a)))
         (+ a_1 b))                                 (+ a_1 b))
```

Because `a` appears free in the definition `(+ x a)`, it is necesary to rename the `a` declared by the outer binding. The `name-not-in` function returns `a_1` as a name that does not conflict with any free variables. Note that the `a_1` declared by the `bind` in the body of the outer `bindpar` is *not* a free variable, and so poses no problem yet. In the process of renaming free occurrences of `a` to `a_1`, however, it is necessary to rename the existing `a_1` to a new named (`name-not-in` chooses `a_2`) to avoid variable capture in the definition `(* a_1 a_1)`.

4 The IBEX Standard Library

Primitive applications in the IBEX interpreter are handled by the code in Figs. 5–7.

A IBEX Kernel Abstract Syntax

The abstract syntax for IBEX programs and kernel expressions nodes is manipulated by the Scheme functions in Figures 8–10.

```
(define make-pdesc
  (lambda (name arg-types result-type scheme-function)
    (list name arg-types result-type scheme-function)))

(define pdesc-name first)
(define pdesc-arg-types second)
(define pdesc-result-type third)
(define pdesc-function fourth)
(define pdesc-num-args (lambda (descriptor) (length (second descriptor))))

;; Handy abstractions:

(define make-binary-arithop
  (lambda (name scheme-fcn)
    (make-pdesc name '(int int) 'int scheme-fcn)))

(define make-binary-relop
  (lambda (name scheme-fcn)
    (make-pdesc name
                '(int int)
                'bool
                (lambda (n1 n2)
                  (scheme-bool-to-mini-bool (scheme-fcn n1 n2))))))

(define make-unary-logop
  (lambda (name scheme-fcn)
    (make-pdesc name
                '(bool)
                'bool
                (lambda (mini-bool)
                  (scheme-bool-to-mini-bool
                   (scheme-fcn
                    (mini-bool-to-scheme-bool mini-bool)))))))

(define make-binary-logop
  (lambda (name scheme-fcn)
    (make-pdesc name
                '(bool bool)
                'bool
                (lambda (mini-bool1 mini-bool2)
                  (scheme-bool-to-mini-bool
                   (scheme-fcn
                    (mini-bool-to-scheme-bool mini-bool1)
                    (mini-bool-to-scheme-bool mini-bool2)))))))
```

Figure 5: The `pdesc` primop descriptor ADT and associated functions.

```
(define make-binding
  (lambda (name value)
    (list name value)))

(define primop-env
  (bindings->env
   (list
    ;; BINDEX primops
    (make-binding '+ (make-binary-arithop '+ +))
    (make-binding '- (make-binary-arithop '- -))
    (make-binding '* (make-binary-arithop '* *))
    (make-binding 'div (make-binary-arithop
                        'div
                         (lambda (a b)
                          (if (= b 0)
                            (throw 'division-by-zero (list 'div a b))
                            (quotient a b)))))
    (make-binding 'mod (make-binary-arithop
                        'mod
                        (lambda (a b)
                          (if (= b 0)
                            (throw 'division-by-zero (list 'mod a b))
                            (remainder a b)))))

    ;; IBEX primops
    (make-binding '< (make-binary-relop '< <))
    (make-binding '<= (make-binary-relop '<= <=))
    (make-binding '= (make-binary-relop '= =))
    (make-binding '!= (make-binary-relop '!= (lambda (a b) (not (= a b)))))
    (make-binding '>= (make-binary-relop '>= >=))
    (make-binding '> (make-binary-relop '> >))
    (make-binding 'not (make-unary-logop 'not not))
    (make-binding 'band (make-binary-logop 'band (lambda (a b) (and a b))))
    (make-binding 'bor (make-binary-logop 'bor (lambda (a b) (or a b))))
    (make-binding 'bool= (make-binary-logop 'bor (lambda (a b) (eqv? a b))))
    (make-binding 'sym= (make-pdesc 'sym=
                                    '(sym sym)
                                    'bool
                                    (lambda (mini-sym1 mini-sym2)
                                      (scheme-bool-to-mini-bool
                                       (eq? (mini-sym-to-scheme-sym mini-sym1)
                                            (mini-sym-to-scheme-sym mini-sym2))))))
    )))

(define primop-name?
  (lambda (name)
    (not (unbound? (env-lookup name primop-env)))))
```

Figure 6: The primitive operator environment.

```
(define primapply
  (lambda (primop-name args)
    (let ((pdesc (env-lookup primop-name primop-env)))
      (if (unbound? pdesc)
        (throw 'unknown-primop primop-name)
        (if (not (= (length args) (pdesc-num-args pdesc)))
            (throw 'primapply:wrong-num-args
                   (list 'expected (pdesc-num-args pdesc)
                         'got (length args)
                         'in (make-primapp primop-name args)))
            (let ((type-mismatch (some2 type-error? (pdesc-arg-types pdesc) args)))
              (if (none? type-mismatch)
                  (apply (pdesc-function pdesc) args)
                  (throw 'primapply:wrong-arg-type
                         (list 'expected (first type-mismatch)
                               'got (second type-mismatch)
                               'in (make-primapp primop-name args)))))
            )))))

(define type-of
  (lambda (val)
    (cond ((mini-integer? val) 'int)
          ((mini-boolean? val) 'bool)
          ((mini-symbol? val) 'sym)
          (else (throw 'type-of-unknown-value val))
          )))

(define type-error?
  (lambda (type val)
    (not (eq? type (type-of val)))))
```

Figure 7: The `primapply` function.

(make-program *formals body***)**
Assume that *formals* is a Scheme list of symbols and *body* is an IBEX expression. Returns an IBEX program node whose formal parameters are *formals* and whose body is *body*.

(program-formals *pgm***)**
Returns the Scheme list of symbols that is formal parameter list of the given IBEX program *pgm*.

(program-body *pgm***)**
Returns the IBEXexpression that is the body of the IBEX program *pgm*.

Figure 8: Functions for manipulating IBEX programs

Literals

(make-literal *value***)**
Assume that *value* is a Scheme datum representing an IBEX literal value. Returns an IBEX literal node containing the value *value*.

(literal-value *literal-node***)**
Returns the Scheme datum that is the value of *literal-node*.

(varref? *node***)**
Returns $t if *node* is a literal node, and #f otherwise.

Variable References

(make-varref *name***)**
Assume that *name* is a Scheme symbol. Returns an IBEX variable reference node that references the variable named *name*.

(varref-name *varref-node***)**
Returns the Scheme symbol that is the name of *varref-node*.

(varref? *node***)**
Returns $t if *node* is a variable reference node, and #f otherwise.

Figure 9: Functions for manipulating IBEX expressions, Part 1.

Primitive Applications

(make-primapp *rator rands*)
Assume that *rator* is a Scheme symbol and that *rands* is a Scheme list of IBEX expressions. Returns an IBEX primitive application node whose operator is *rator* and whose operand list is *rands*.

(primapp-rator *primapp-node*)
Returns the Scheme symbol that is the operator of *primapp-node*.

(primapp-rands *primapp-node*)
Returns the Scheme list of IBEX expressions that is the operand list of *bind-node*.

(primapp? *node*)
Returns $t if *node* is a primitive application node, and #f otherwise.

Local Bindings

(make-bind *name defn body*)
Returns an IBEX local binding node whose declared variable name is *name*, whose definition expression is *defn*, and whose body is *body*.

(bind-name *bind-node*)
Returns the declared name of *bind-node*.

(bind-defn *bind-node*)
Returns the definition expression of *bind-node*.

(bind-body *bind-node*)
Returns the body expression of *bind-node*.

(bind? *node*)
Returns $t if *node* is a local binding node, and #f otherwise.

Conditionals

(make-if *test then else*)
Returns an IBEX conditional node whose test is *test*, whose then expression is *then*, and whose else expression is *else*.

(if-test *if-node*)
Returns the test expression of *if-node*.

(if-then *if-node*)
Returns the then expression of *if-node*.

(if-else *if-node*)
Returns the else expression of *if-node*.

(if? *node*)
Returns $t if *node* is a conditional node, and #f otherwise.

Figure 10: Functions for manipulating IBEX expressions, Part 2.