CS251 Programming Languages **Handout # 19**
Prof. Lyn Turbak **March 6, 2001**
Wellesley College

Desugaring

Syntactic sugar causes cancer of the semicolon.
— Alan Perlis

1 Introduction

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.

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.

2 Simple Desugaring Rules

Let's begin by considering IBEX's `scand` and `scor` constructs. The expression `(scand` E_1 E_2`)` first evaluates E_1, which is expected to result in the boolean value V_1. If V_1 is `true`, `scand` returns the result of evaluating E_2. But if V_1 is `false`, `scand` immediately returns `false` without evaluating E_2. Because it does not evaluate E_2 when E_1 is false, `scand` is known as a **short-circuit** logical conjunction operator – hence the name `scand`[1]. The `scor` construct is the short-circuit logical disjunction construct that is the dual of `scand`. In `(scor` E_1 E_2`)`, E_1 is evaluated first to value V_1. If V_1 is `true`, then `scor` returns `true` immediately without evaluating E_2; otherwise, it returns the value of E_2.

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?` function we studied earlier this semeseter:

```
(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.

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`:

`(scand` E_1 E_2`)` $\Rightarrow$ `(if` E_1 E_2 `false)`
`(scor` E_1 E_2`)` $\Rightarrow$ `(if` E_1 `true` E_2`)`

[1] The names `scand` and `scor` are non-standard and are just made up for this course to distinguish them from the non-short-circuit logical operators `band` and `bor` (also made-up names). But the terminology "short-circuit" is standard in the community, as are "syntactic sugar" and "desugaring".

These notations are called **desugaring rules**. Here, the arrow $\Rightarrow$, 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**[2] 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:

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

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

$$\texttt{(bindseq2 ((}I_1\ E_1\texttt{) (}I_2\ E_2\texttt{))}\ E_{\text{body}}\texttt{)} \quad \Rightarrow \quad \texttt{(bind}\ I_1\ E_1\ \texttt{(bind}\ I_2\ E_2\ E_{\text{body}}\texttt{))}$$

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
}
```

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]

[2] 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.

[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?

3 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 1 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[4] (2) all subexpressions are recursively desugared.

In Figure 1, 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).

[4]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))
                    (desugar (scand-rand2 exp))
                    (scheme-bool-to-ibex-bool #f)))

          ((scor? exp)
           (make-if (desugar (scor-rand1 exp))
                    (scheme-bool-to-ibex-bool #t)
                    (desugar (scor-rand2 exp))))
          ((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 1: IBEX desugaring functions. A new syntactic sugar construct can be added to IBEX by adding a clause to `desugar`.

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.[5] During evaluation time, the desugared IBEX expression is evaluated.

4 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.

4.1 `bindseq`

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

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

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`:

```
(bindseq () Ebody)          ⇒   Ebody
(bindseq ((I E) ... ) Ebody)  ⇒   (bind I E (bindseq ( ... ) Ebody))
```

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:

[5] 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.

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.

4.2 `cond`

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

(cond (E_{test_1} E_{body_1})
⋮
(E_{test_n} E_{body_n})
(else E_{default}))

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

(cond (else E_{default})) ⇒ E_{default}
(cond (E_{test} E_{body}) ...) ⇒ (if E_{test} E_{body} (cond ...))

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)
                (< b c)
                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)))
```

5 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 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!

A IBEX Kernel Abstract Syntax

The abstract syntax for IBEX programs and kernel expressions nodes is manipulated by the Scheme functions in Figures 2–4.

(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 2: 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 3: 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 4: Functions for manipulating IBEX expressions, Part 2.