Procedural Relection in Programming Languages

3c Lambda Abstraction and Procedural Intension†

Atoms, as I said in [dissertation] section 4a, are used in 2Lisp as context dependent names. I also made clear, both in that section and in [dissertation] chapter 3, that they are taken to designate the referent of the expression to which they are bound. Finally, I have said that they will be statically scoped. It is appropriate to look at all of these issues with a little more care. A1 A2

The semantical equation governing atoms was given in [dissertation] section 4.a.iii, repeated here: A3

$$\forall e \in \text{ENVS}, f \in \text{FIELDS}, c \in \text{CONTS}, a \in \text{ATOMS} \quad [1] \quad \text{A4}$$
$$[\Sigma(a, e, f, c) = c(e(a), \varphi e f(e(a)), e, f)]$$

If we discharge the use of the abbreviatory φ, this becomes:

$$\forall e \in \text{ENVS}, f \in \text{FIELDS}, c \in \text{CONTS}, a \in \text{ATOMS} \quad [2]$$
$$[\Sigma(a, e, f, c) = c(e(a), \Sigma(e(a), e, f, [\lambda{<}s, d, e_1, f_1{>}\ .\ d]), e, f)]$$

Because all *bindings* are in normal-form, the above equation can be proved equivalent to the following:

$$\forall e \in \text{ENVS}, f \in \text{FIELDS}, c \in \text{CONTS}, a \in \text{ATOMS} \quad [3]$$
$$[\Sigma(a, e, f, c) = \Sigma(e(a), e, f, c)]$$

This is true because, if $e(a)$ is normal, then it will not affect the e and f that are passed to it. Nonetheless, [2] must stand as the definition; [3] as a consequence.

What I did not explain, however, is how environments are

†Dissertation section 4·c·i, pp. 377–392, of Procedural Reflection in Programming Languages. A link to an internet version of the dissertation is on p. ·.

constructed. The answer, of course, has first and foremost to do with λ-binding. A full account of the significance of atoms and variables, therefore, must rest on the account of the significance of λ-terms. In brief, a λ-term is a complex expression A5 that designates a function. Structurally, in 2Lisp it is any reduction (pair) formed in terms of a designator of the primitive lambda closure and three arguments: a *procedure type*, a *parameter list*, and a *body expression*. The primitive lambda closure is the binding, in the initial environment, of the atom LAMBDA, although there is nothing inviolate about this association. The procedure type argument is typically either EXPR or IMPR (for *extensional procedure* and *intensional* procedure, respectively; I will discuss these terms more below). The parameter list is a pattern against which arguments are matched, and the body expression is an expression that, typically, contains occurrences of the variables named in the parameter pattern. Thus I am assuming LAMBDA-terms of the following form:

(lambda *procedure-type parameters body*)
[4]

I have of course used λ-terms throughout the dissertation, both in Lisp and in the meta-language. It is important, however, not to be misled by this familiarity into thinking we either understand or have yet encountered the full set of issues having to do with λ-abstraction. For this reason the following discussion is framed as if LAMBDA were being introduced for the first time. In this spirit, it is helpful to start by reviewing some simple examples of the use of LAMBDA-terms embedded in larger composite expressions—without any of the complexities of global variables, top-level definitions, recursion or the like. These examples are similar in structure to the kind of term that can be expressed in the λ-calculus (using '⇓', as always, to mean 'normalises to'):

```
((lambda expr [x] (+ x 1)) 3)          ⇒ 4
[5]
((lambda expr [f]                                    [6]
    (f (f 3 4) (f 5 6))
 +)                                    ⇒  18
((lambda expr [g1 g2]                                [7]
    (g1  (=   (nth 1 '[$t])
              (nth 1 ['$t]))
         (g2 [10 20 30])
         (g2 '[10 20 30])))
 if
 (lambda expr [r] (tail 2 r)))         ⇒  [30]
```

[5] is a standard example, of the sort 1Lisp would support: the expression (LAMBDA EXPR [X] (+ X 1)) designates the increment function. [6] illustrates the use of a function designator as an argument, making evident the fact that 2Lisp is higher order. Finally, [7] shows that procedurally intensional designators or IMPRs (IF) can be passed as arguments as readily as EXPRs.

There is nothing distinguished or special about these LAMBDA terms, other than the fact that LAMBDA designates a primitive closure. Unlike standard Lisps and the original λ-calculus, in other words, in 2Lisp the label LAMBDA is not treated as a syntactic mark to distinguish one kind of expression from general function applications. Like all pairs, LAMBDA terms are reductions, in which the procedure to which LAMBDA is bound is reduced with a standard set of arguments. I will show below that LAMBDA is initially bound to an intensional procedure, but, as the following example demonstrates, this fact docs not prevent that closure from itself being passed as an argument, or bound to a different atom:

```
(((lambda expr [f]                                   [8]
      (f expr [y] (+ y y)))
  lambda)
  5)                                 ⇒  10
```

It happens that EXPR also names a function; thus it is even possible to have such expressions as:

```
(((lambda expr [funs]                                    [9]
      ((nth 2 funs) (nth 1 funs) [y] (+ y y)))
  [expr lambda])
 5)                                  ⇒  10
```

Finally, as usual it is the normal-form closures, rather than their names in the standard environment, that are primitively recognized: A6

```
> (define beta lambda)
[10]
> beta
> (define standard expr)
> standard                                               A7
> ((beta standard (f) (f f)) type)
> 'function
```

LAMBDA, in other words, is a **functional**: a function whose range is the set of functions:

```
(type lambda)                       ⇒  'function          [11]
(type (lambda expr [x] (+ x 1)))    ⇒ 'function
```

Similarly, EXPR is a function, although I will show how it can be used in function position only later:

```
(type expr)                         ⇒  'function          [12]
```

Though the examples just given illustrate only a fraction of the behavior of LAMBDA that I will ultimately need to characterize, some of the most important features are clear.

First, LAMBDA is first and foremost a *naming* operator: moreover, the *procedural* import of LAMBDA terms in this or any other Lisp arises not from LAMBDA alone, but from general principles that permeate structures of all sort, and from the type argument I have here made explicit as LAMBDA's first argument. A8

In what follows I will explore the procedural significance of LAMBDA terms at length, but it is important to enter into that discussion fully recognizing that it is the body expression that establishes that procedural import, not LAMBDA itself.

Second, LAMBDA is itself an *intensional* procedure; neither the parameter pattern nor the body expression is processed when the LAMBDA reduction is itself processed. This is clear in all of the foregoing examples: the parameters—the atoms that will be bound when the pattern is matched against the arguments, as discussed below—are unbound when the LAMBDA term itself is normalised; but the LAMBDA term does not generate an error when processed. This is because neither the pattern nor the body is treated extensionally—i.e., as being in what is called an "extensional context." (Less clear, although hinted by [9], is the fact that the ***PROCEDURE-TYPE*** argument to LAMBDA *is* processed at reduction time.)

Further evidence of LAMBDA's procedural intensionality with respect to its second and third argument position is provided in this example:

```
> ((lambda expr [fun]
[13]
        (block (print 'last) (fun 1 2)))
    (block (print 'shoe) +)) shoe last
> 3
```

In other words processing of the argument to the LAMBDA term occurred *before* processing of the body internal to that term. The body of a LAMBDA term is then processed each time the function it designates is applied. This fundamental fact about these expressions will motivate the semantical account. A9

In spite of LAMBDA's intensionality, however, there is nevertheless an important sense in which the context in which the LAMBDA term is itself reduced affects, or at least is relevant to, the behavior of the resultant procedure when it is used. In particular, we have the following:

```
((lambda expr [fun]                                          [14]
    ((lambda expr [y]
        (fun y))
     2))                                                  (cont'd)
 ((lambda expr [y]
    (lambda expr [x] (+ x y)))
  1))                                          ⇒ 3
```

In this example, the atom FUN is bound to a closure designating a function that adds 1 to its argument. This is because the Y in the body of the lexically last λ-term in the example (the second last line) receives its meaning from the context *in which it was reduced* (a context in which Y is bound to 1), not from the context in which *the function it designates is applied* (a context in which Y is bound to 2). In a dynamically scoped system, [14] would of course reduce to 4.

The expression in [14] is undeniably difficult to read. I will adopt a 2Lisp LET macro, similar to the 1Lisp macro of the same name, to abbreviate the use of embedded LAMBDA terms of this form (this LET will be defined in [dissertation] section 4.d.vii). In particular, expressions of the form

```
(let [[param1 arg1]
[15]
      [param2 arg2]
      ...
      [paramk argk]]
   body)
```

will expand into the corresponding expressions

```
((lambda expr [param1 param2 ... paramk]
 [16]
    body)
 arg1 arg2 ... arg2)
```

Similarly, I will define a "sequential LET", called LET*, so that expressions of the form

```
(let* [[param1 arg1]
[17]
       [param2 arg2]
          ...
       [paramk argk]]
   body)
```

will expand into the corresponding expression

```
((lambda expr param1
[18]
    ((lambda expr param2
        ...
           ((lambda expr paramk body)
            argk)
        ...)
     arg2))
 arg1)
```

Thus in a use of LET* each arg_i may depend on the bindings of the parameters before it. The difference between these two is illustrated in:

```
(let [[x 1]]                                          [19]
   (let [[x (+ x 1)]
         [y (- x 1)]]
     y))                          ⇒  0

(let [(x 1]]                                          [20]
   (let* [[x (+ x 1)]
          [y (- x 1)]]
     y))                          ⇒  1
```

Although some of the generality of LAMBDA is lost by using this abbreviation (all LETs and LET*s, for example, are assumed to be EXPRs—i.e., extensional LAMBDAs), I will employ LET and LET* forms widely in subsequent examples. The expression in [14], for example, can be recast using LET, generating an expression much easier to understand, as follows:

```
(let [[fun                                            [21]
```

```
(let [[y 1]]
   (lambda expr [x] (+ x y)))]]
(let [[y 2]] (fun y)))                    ⍰  3
```

The behavior demonstrated in [14] and again in [21] is of course evidence of what is called *static* or *lexical* scoping; if [14] or [21] reduced to the numeral 4 we would say that *dynamic* or *fluid* scoping was in effect.

The concepts of dynamic and static scoping, however, are by and large described in the literature in terms of *mechanisms* and/or behavior: one protocol is treated this way; the other that. It is not my policy, in this entire exercise, to accept behavioral accounts as explanations. Throughout, I am committed to being able to answer such questions as "*Why* do these scoping regimens behave the way that they do?" and "*Why* was static scoping used in 2Lisp and 3Lisp?"

Fortunately, the way we have come at these issues leads to a much deeper characterization of what is going on. In particular, I said that LAMBDA was *intensional*, but example [21] makes it clear that it is not *hyper-intensional*, in the sense of treating its main argument—the body expression—purely as a structural or textual object. It is not the case, in other words, that the reductions involving the function bound to FUN in the third line of [21] consist in the replacing, as a substitute for the word term 'FUN', the textual object '(+ X Y)'. To treat it so would yield an answer of 4—i.e., would imply that 2Lisp has adopted dynamic scoping. Rather, the behavior demonstrated in [21] shows that what is bound to FUN is neither the body itself, as a textual entity, nor the result of *processing* the body, but rather something intermediate. In ways that we need to understand, what is bound to FUN is an object that in some sense is closer to, or anyway can be associated with, the *intension of the body at the point of the original reduction*.

If we had an adequate theory of intensionality, it might be tempting to say something like the following: that LAMBDA is

an [intensional] function from *textual objects* (the body expression and so forth) onto the *intension* of those textual objects in the context in force at the time of reduction. The subsequent use of such a procedure would then "reduce" (or "apply", or whatever intermediate term was chosen as proper to use for A10 combining functions-in-intension with arguments) this intension with the appropriate arguments. There is something right about this, though for two reasons we cannot let it stand as is. Sadly, first, we have no such theory of functions-in-intension to express it in terms of. Second, it is not quite right, anyway. LAMBDA is of course a function from textual objects *onto functions*, A11 as was made clear earlier; what I need to show, rather, is that (and how) the functions onto which LAMBDA maps its textual arguments somehow preserve, in a context-*independent* way, the potentially context-*dependent* intension of the textual argument in the original context.

Moreover, we can also see that a statically scoped LAMBDA, of the sort constitutive of 2Lisp and 3Lisp, is a *coarser-grained* intensional procedure than is a dynamically scoped LAMBDA. That is:

T1 *Static scoping corresponds to an intensional abstraction operator; dynamic scoping, to a hyper-intensional abstraction operator.*

In order to understand T1 in depth, we need to retreat a little from the rather behavioral view of LAMBDA that I have been presenting, and look more closely at what λ-abstraction consists in from the original perspective of its being a naming op- A12 erator. It is all very well to show how LAMBDA terms *behave*, in other words; but we have not yet adequately answered the question "What do LAMBDA terms *mean*?"

Speaking extensionally, LAMBDA terms designate functions; that much is clear. We also know that functions are sets of ordered pairs, such that no two pairs coincide in their first ele-

ment. We know, too, what application is: a function applied to an argument is the second clement of that ordered pair in the set whose first element is the argument. However none of this elementary knowledge suggests any relationship between a function and a function *designator*. And until we understand that relation, we will not be in a position to understand, intensionally, that designator's *meaning*.

Informally, we have a consensual intuition about λ—that it is an operator over a list of variables and expressions, designating the function that is signified by the λ-abstraction of the given variables in the expression that is its "body" argument. However this intuition—including its telling use of the phrase 'λ-abstraction'—must arise independently of any of the extensional points made in the preceding paragraph. To understand the meaning of a λ-term, therefore, requires an analysis of it *as a term*.

The fundamental intuition underlying λ-terms and λ-abstraction in general can be traced at least as far back as Frege's study of predicates and sentences in natural language. In particular, I believe that it is best to understand a λ-term is as a *designator with a hole in it*, just as Frege understood a predicate term as a *sentence with a hole in it*. If, for example, we take (and assume to be true) the sentence "*Mordecai was Esther's cousin*," and delete the first designating term, then we obtain the expression "_____ *was Esther's cousin*." It is easy to imagine constructing an infinite set of other derivative sentences from this fragment, by filling in the blank with a variety of other designating terms. Thus for example we might construct "*Aaron was Esther's cousin*" and "*the person who lives across the fjord was Esther's cousin*" and so forth. In general, some of these A13 constructed sentences will be true, and some will be false. In the simplest case, also, the truth or falsity hinges not on the actual form of the designator we insert into the blank (whether we say 'the person who lives across the fjord' or 'the person who

was here for tea yesterday'), but on the *referent* of that designator. Thus our example sentence will be true if the supplied designator refers to Mordecai; any term codesignative with the proper name "Mordecai" would serve equally well.

Predicates arise naturally from consideration of *sentences* containing blanks; that was Frege's insight. The situation regarding *designators* containing blanks—and the resultant functions—is entirely parallel. Thus if we take a complex noun phrase such as "*the country immediately to the south of Ethiopia*," and remove the final constituent noun phrase, we get the open phrase "the country immediately to the south of _____." Once again, by filling in the blank with any of an infinite set of possible terms (designating noun phrases), the resultant composite noun phrase will (perhaps) designate another object. In those cases where the resultant phrase succeeds in picking out a unique referent, we say: (i) that the object so selected is in the *range* of what is designated by the phrase that contained the blank; and (ii) that the object designated by the phrase we insert into the blank is in that entity's *domain*. In this way we erect the entire notion of *function* with which we are so familiar. A14

Once this basic approach is adopted, a raft of more specific questions arise. What happens, for example, if we construct a phrase with *two* blanks? The answer, of course, is that we are led to a function of more than one argument. What if the noun phrase we wish to delete occurs more than once (as for example the term 'Ichabod' in "The first person to like Ichabod and Ichabod's horse")? The power of the λ-calculus can be seen as a formal device to answer all of these various questions. In particular, we can understand the formal *parameters* as a *method of labeling the holes*: if one parameter occurs in more than one position within the body of the lambda expression, then tokens of the formal parameters stand in place of a single designator that had more than one occurrence. If there

is more than one formal parameter, then more than a single noun phrase position has been made "blank." And so on and so forth—all of this is familiar.

It is instructive to review this history, for it leads to a particular stance on some otherwise difficult questions. Note for one thing how it clarifies a point we started with: that the function of LAMBDA as a first and foremost a *naming* operator. In addition, it is important to recognize how *syntactic* a characterization this has been: I have talked almost completely about signs and expressions (terms, phrases, etc.), even though we realized that the semantical import of the resultant sentences or completed noun phrases depended (in the simple extensional case) only on the referents of the noun phrases that were inserted into the blank(s). It was Frege's technique to motivate the abstract ontological notions of *predicates*, *relations*, *functions*, etc. as derivative on such syntactic manoeuvring. The technique is important in the present case because it gives us a stance from which to ask essentially syntactic or structural questions in order to get at the ontological intuitions behind λ-abstraction (indeed, it is because I want the structural answers to these questions that I am pursuing this whole line of thought). A15

Suppose, then, to stay with the case of defining predicates, that we wish to define (i.e., name) a predicate by inserting a blank into some otherwise complete sentence—i.e., by deleting a noun phrase from it. What context, we may ask, determines the meaning of the resulting open expression? The only plausible answer that honours the referential character of naming is *the context in which the definition was originally introduced*. Suppose, for example, that while writing this paragraph I utter the sentence "*Bob is going to vote for the President's eldest daughter*." Again staying with the simplest case, it is natural to assume that I refer to the (current) President's eldest daughter, known by the name "Maureen Reagan." If I excise the noun "Bob" and construct the open sentence "_____ *is going to vote* A16

for the President's eldest daughter," then I have constructed a predicate true of people who will vote for Maureen Reagan. That is, the interpretation of "the President's eldest daughter" is determined by the context where the predicate was introduced. This, at least, is the simplest and most straightforward reading. It would undeniably be more complex, even if one could nonetheless argue that it would be logically coherent, to suggest that what is designated hyper-intensionally involves the whole open sentence *qua sentence*—so that when we asked whether the resultant predicate is true of some person we would determine the referent of the phrase *"the President"* only at that point. The ground intuition is unarguably extensional. A17 A18

What does this suggest regarding Lisp? Simply this—that the natural way to view λ-terms is as:

1. Expressions that designate functions, derived from
2. Composite referring terms, in which one or more ingredient terms have been replaced by blanks, where
3. The *parameters* are a formal mechanism to label the blanks, so as to facilitate a subsequent process of filling the blanks in with other terms, and
4. Where the function designated is determined *with respect to the context of use where the* **LAMBDA** *term is stated or introduced* (as opposed to where the designated function is subsequently applied). A19

Notably, the foregoing four points *lead us* to an adoption of statically scoped free variables, because we can show how that procedural mechanism correctly captures the original (declarative) naming intuition. In other words, I am claiming that:

T2 *Static scoping is the truest formal reconstruction of the (ultimately referential) linguistic intuitions and practice upon which the notion of λ-abstraction is based.*

In general, in order to remain true to Church's λ-calculus, we

must be true to the understanding that his calculus embodies, rather than slavishly mimic its operations. This mandate has further-reaching consequences than those articulated in T2. In A20 particular, to propose a full *substitutional* procedural regimen for 2Lisp would be crazy—it would be to *mimic his mechanism*, rather than *accomplish what his mechanism was for*. Since 2Lisp is a formalism with procedural side-effects, such a regime would imply that every occurrence of a formal parameter within a procedure body would engender another instance of any side-effects implied by an argument expression. This was not a problem for Church because the λ-calculus of course has no side-effects.

In sum, I will insist that the term

```
(let [[y 1]]                                    [22]
   (lambda expr [x] (+ x y)))
```

designate the increment function, rather than designating that function that adds to its argument the referent of the sign "Y" in the context of use of the designating procedure.

As far as it goes, this is straightfoward. I showed in [dissertation] chapter 2 how the static reading leads naturally to a higher-order dialect, to uniform processing of the expression in "function position" in a redex, and so forth, though in that chapter I did not do what we have done here: examine the underlying semantical motivation for this particular choice. Nor, in that context, did I explicitly examine another subject to which we must now turn: *the intensional significance of a* ***LAMBDA*** *term*. That this further question remains open is seen when one realizes that the preceding discussion argues only that the *extension* of the LAMBDA term be determined by the context of use in force at the point where the LAMBDA term it introduced. However I have not yet examined the full computational significance of the term in "body" position—i.e., to

use the reconstruction I am recommending, the full computational significance of the open designator containing demarcated blanks.

For pointers, it is again instructive to look at the reduction regimen that Church adopted for the λ-calculus. As I have said, the λ-calculus is a statically-scoped higher order formalism. By the discussion just advanced, the λ-calculus should depend on an intensional LAMBDA, but of course no theory of "functions in-intension" accompanies theoretical treatments of the λ-calculus. This is related to the fact that, in the λ-calculus, the item "λ" is not itself considered to be in a function-designating position. This is because the λ-calculus is strictly an *extensional* system; there is no way in which an appropriately *intensional* function could be defined within its boundaries. It is thus effectively a *necessary* rather than contingent fact that λ-terms in the λ-calculus are demarcated *notationally*, as they were in the first version of 1Lisp that I presented in [dissertation] chapter 2. (In the λ-calculus, that is, the "λ" is a syncategorematic uninterpreted mark, on a par with parentheses and dots). A21

In order to understand the λ-calculus and λ-abstraction more generally, it is essential to recognize that its substitutional reduction regime is defined within this set of constraints. Superficially, after all, substitution is a hyper-intensional kind of practice. During β-reductions, *actual textual expressions* are substituted, one within another, during the reduction of a composite λ-calculus term. This would appear to conflict with the claim made above in T1: that hyper-intensional abstraction corresponds to dynamic scoping, and intensional abstraction to static or lexical. How then can I defend my claim of intensional abstraction in a statically scoped formalism, and yet use the λ-calculus as a motivating example? A22

The answer is that the λ-calculus has been crafted in such a way as to enable its hyper-intensional substitution protocols

to *mimic* or *implement* the more abstract intensional behavior that ideally, if we had a adequate theory of functions-in-intension, we would be able to define more directly. In particular, three properties of the λ-calculus contribute to this ability. First, as already mentioned: the λ-calculus is *extensional*, the mark 'λ' is not used as a term (i.e., not in function position), and no facility is provided for the user to construct intensional functions. Second, there is no primitive quotation operator in the λ-calculus (and of course no corresponding mechanism of disquotation), so that it is not possible in general and unpredictable ways to capture an expression from one context and to slip it into the course of the reduction in some other place ("behind the back of the reduction rules," so to speak)—a practice that genuinely would engender something like dynamic scoping. Third, as well as superficially involving hyper-intensional β-reduction, the λ-calculus also depends on a seemingly pesky but in fact critically important additional rule, having to do with variable capture, called α-reduction. It is a constraint on β-reduction—the main reductive rule in the λ-calculus—that terms may not be substituted into positions *in such a way that any open (unbound) variables would be "captured" by an encompassing λ-abstraction*. If such a capture would arise, one is obligated first, using α-reduction, to rename the parameters involved in such a fashion that the capture is avoided. The following, for example, is an incorrect series of β-reductions:

```
(λf . ((λg . (λf . fg)) f))      ; an illegal derivation      [23]
(λf . (λf . ff))
```

Rather, one must use an instance of α-reduction to rename the inner f so that the substitution, for g, of the binding of g will not inadvertently lead that substitution to "become" an instance of the inner binding. Thus the following is correct:

```
(λf . ((λg . (λf . fg)) f))      ; a legal derivation         [24]
(λf . ((λg . (λh . hg)) f))      ; first an α-reduction
```

(λf . (λh . hf)) ; then a valid β-reduction.

In other words α-reduction is expressly designed, from a procedural point of view, to ensure that, in those cases where the context of *use* of a λ-term might conflict with the context of *introduction*, the λ-term is adjusted so that the function it designates remains uninfluenced. That is: the role of α-reduction in the λ-calculus is precisely to rearrange textual objects so as to *avoid the dynamic scoping that would be implied if α-reduction did not exist*.

Together, in sum, these three conditions ensure that, in spite of β-reduction's hyper-intensional character, the λ-calculus' overall procedural regimen honours the conditions of static or lexical scoping.

We are still not done. We need to ask *why* the reduction in [23] is ruled out—*why* dynamic scoping is so carefully avoided. The answer cannot be that the resulting system is incoherent, since, modulo issues of side effects, β-reductions with no α-reductions is one way to view Lisp 1.5 and all its descendants. Sure enough the Church-Rosser theorem would not hold, but, as our experience with these Lisps has shown, one can simply discard that theorem and decide rather arbitrarily on one reduction order. But we now have an answer: dynamic scoping violates the condition we adopted above: that the "meaning" or intension of a λ-term be determined by the context in force in the place where that term is introduced. Variable capture is bad because it *alters* that intension—thereby violating intention.

Thus we have reached the following important conclusion:

T3 *The reduction of* **LAMBDA** *terms must preserve, in a context-independent way, the (potentially) context-dependent intension of the body expression.*

This of course is a much stronger result than the overarching mandate that in every case ψ preserve *designation*. In general, reduction (ψ) of composite terms does *not* preserve intension, according to a commonsense notion of intension. This is difficult to say formally, for two reasons. The most serious is the standard one: that we do not have a theory of intension with respect to which to formulate it. If one takes the intension of an expression to be the function from possible worlds onto extensions of that expression in each possible world—the approach taken in possible world semantics and by such theorists as Montague[1]—then it emerges (if one believes that arithmetic is not contingent) that all designators of the same number are intensionally as well as extensionally equivalent. Thus (+ 1 1) and (SQRT 4) would be considered intensionally equivalent to 2 (providing of course we are in a context in which SQRT designates the square-root function). I would argue, however, that this conclusion violates lay intuition—that a more adequate treatment of (even mathematical) intensionality should be finer grained (perhaps of a sort suggested by Lewis[2]). Second, A23
without specifying the intensions of the primitive nominals in a Lisp system, it is difficult to know whether intension is preserved in a reduction. Suppose, for example, that the atom PLANETS designates the sun's planets, and is bound to the rail A24
[MERCURY VENUS EARTH … PLUTO]. Then (CARDINALITY PLANETS) might reduce to the numeral 9 if CARDINALITY was procedurally defined in terms of LENGTH. It is argued, however, that the phrases "*the number of planets*" and "*nine*" are intensionally distinct because "*the number of planets*" might have designated some other number, if there were a different number of planets, whereas, in this language, "*nine*" necessarily designates the number nine. On such an account the reduction of (CARDINALITY PLANETS) to 9 is not intension preserving.

1. Cf. Montague (1970, 1973). [Note: this footnote was numbered 4 in the original version of the disseratation.]
2. Lewis (1972).

Making precise our intuitions about the nature of intensionality in general is not my present subject matter, however. Furthermore, and fortunately, if all we ask of the reduction of LAMBDA terms to normal form is that intension be *preserved*, we do not have to *reify* intensions at all—we do not even have to take a position on whether intensions are *things*. All that we are bound to ensure is the substance of T2: that the intensional character of the expression over which the LAMBDA term abstracts must be preserved, in a context-independent way, in the normal-form function designator to which the LAMBDA term reduces.

At the declarative level this suffices—it will be my guiding mandate in defining the procedural treatment of LAMBDA terms. A further set of questions needs to be answered, however, having to do with the relationship between the intensional content of a Lisp expression and its full computational significance, including its procedural consequence. The issue is best introduced with an example that I will make use of later. It is a widely appreciated fact that, if an expression X should not be processed at a given time, but should be processed at another time, it is standard computational technique to wrap it in a procedure definition, and then to *reduce* it subsequently, rather than simply *using* it. A simple example is illustrated in the following two cases: in the first the (print 'there) happens *before* the call to (print 'in); in the second it happens *after*:

```
> (let [[x (print 'there)]]
[25]
      (block (print 'in) x)) there in
> $T
> (let [[x (lambda expr [] (print 'there))]]
[26]
      (block (print 'in) (x))) in there
```

```
> $T
```

Because of 2Lisp's static scoping, which corresponds to this intensional reading of LAMBDA, this approach can be used even if variables are involved:

```
> (let* [[x 'there]                                          [27]
        [y (print x)]]
    (block (print 'in) y)) there in
> $T
> (let* [[x 'there]                                          [28]
        [y (lambda expr [] (print x))]]
    (block (print 'in) (y))) in there
> $T
```

What this example illustrates is that the side-effects engendered by a term (input/output behavior is the form of side-effect illustrated here, but of course control and field effects are similar) take place *only when the term is processed in an extensional position*. In other words if the reduction of a LAMBDA-term takes (and preserves) an *intensional* reading of the body expression, it does not thereby engender the full computational significance of that expression. Such significance arises only when some other function or context requires an *extensional* reading. Side-effects, that is, can be considered to be parts of the **procedural extension** of a 2Lisp expression. A25

The QUOTE function in 2Lisp that I defined in §4-132, and handles in general, are *hyper-intensional* operators; it was clear in their situation that the significance of the mentioned term was not engendered by the reduction of the hyper-intensional operator over the term. I have not, however, previously been forced to ask the question of what happens with respect to *intensional* operators, but the examples just adduced yield an answer: their processing, too, does not release the potential significance of the term. Or to put it another way: A26

T4 *The full computational significance of both hyper-inten-*

sional and intensional computational expressions does not release the full computational significance latent in their ingredients.

It is for this reason that the "deferring" technique alluded to above works in the way that it does. (Note, again, that no suggestion is afforded by the λ-calculus with respect to this concern, since that calculus contains no side-effects at all.) Thus we might say that 'intensional' and 'hyper-intensional' are defined not just declaratively, but more generally in terms of full computational significance.

In sum, we have reach the following constraint: *intension-preserving* term transformations do not engender the procedural consequences latent in an expression; those consequences emerge only during the normalisation of an extensional redex, in which case *intension is not (in general) preserved*. Recall that although (+ 2 3) reduces to *co-extensional* 6, it is on my view *not* the case that (+ 2 3) and 6 are intensionally equivalent. A27

One more question needs to be examined, before I am ready to characterize the full significance of LAMBDA. As noted above, in spite of my claim that LAMBDA is an intensional operator, it cannot be the case that LAMBDA is a function from expressions onto intensions, nor is it the case that LAMBDA terms reduce to intensions. If x is a term (LAMBDA …), in other words, neither φ(x) nor ψ(x) is an intension; both possibilities are rejected by protocols long since accepted. In particular, note that in any 2Lisp form (***F*** . ***A***), the significance of the whole arises from the application of the function designated by ***F*** to the arguments signified by ***A***. Thus in (+ 2 3), which is in reality (+ . [2 3]), I said that the whole designated five because the atom "+" designated the extensionalised addition function, which when applied to a syntactic designator of a sequence of two numbers, yielded their sum.

Similarly, in any expression

((lambda *type params body*) . *args*)
[29]

it follows that the term (LAMBDA …) must designate a function. Similarly, in a construct like

(let [[f (lambda ...)]] [30]
(f . *args*)]

F must also designate a function. This is all consistent with the requirement that variable binding be designation-preserving. In [30], F and (LAMBDA …) must be co-designative.

It follows, then, that F cannot *designate* the intension of the (LAMBDA …) term. Hence (LAMBDA …) cannot normalise to a designator of that function's intension. For we do not know what intensions are, but they are presumably not syntactic, structural entities. They are not, in other words, elements of the set of structural field elements S, and ψ has S as its range. I said earlier, however, that F must be intensionally similar to the LAMBDA term—what this brings out is that F must be **co-intensional** with the LAMBDA term, as well as *co-extensional*. The normalisation of LAMBDA redexes, in other words, must *preserve intension as well as extension*. That is, to put it all together:

T5 *Structures of the form* (lambda *pattern body*) *are context-dependent function designators. The normalisation of such forms must yield structures that preserve, in a context-independent way, the* full computational significance *of those designators—both intensional and extensional, both declarative and procedural.*

* * *

This is as much as I will say regarding LAMBDA in its simple uses. As usual, in accord with my general approach, I have attempted to characterize LAMBDA terms primarily in terms of

what they mean (declaratively, as names); from this I have attempted to justify an account of *how they are to behave*. As always, that is, ψ is subservient to φ.

Finally, in terms of this analysis of LAMBDA, I need to say how reduction *works*. The answer is of course quickly stated, and familiar. When a LAMBDA term is reduced, a closure is constructed and returned as the result. When a pair whose CAR normalises to a non-primitive closure is encountered, the closure is *reduced* with the arguments. If that closure is an EXPR, then this reduction begins with the reduction of the CDR of the pair, followed by a process of binding the variables in the parameter pattern to the resultant normal-form argument designator. If the closure is an IMPR, no argument normalisation is performed; instead a handle designating the CDR of the pair is matched against the parameter pattern. In either case the body of the closure (the body of the original reduction with LAMBDA) is processed in a context that, as usual, consists of a field and an environment The field is the field that results from the processing of the arguments—as usual there is no structure to the use of fields: a single field is merely passed along throughout the computation. The environment, however—this is the mechanism that allows the intension to be that of the point of introduction—is the environment that was in force at the point when the closure was constructed, but augmented to include the bindings generated by the pattern match of arguments against variables.

If we were equipped with a theory of functions in intension, and could therefore avail ourselves of an intensional operator in the meta-language, called INTENSION-OF, that mapped terms and lists of formal parameters into *intensions*—whatever they might be—then we could specify this entire desired semantical import of LAMBDA in its terms. Lacking such a theory, I will instead look at LAMBDA from the point of view of designation and reduction, armed with the mandate that it is the intensional properties of the resultant structures that are of