The T Manual
Fifth Edition
— Pre-Beta Draft —

Jonathan A. Rees Norman I. Adams James R. Meehan

October 2, 1990

Chapter 1

Objects

T is an *object-oriented* language. **T** programs are concerned for the most part with creating and manipulating *objects*, which represent all data, and form the currency of computation in **T**. Particular objects are defined not by bit patterns or by addresses within a computer but rather by their behavior when called or when passed to procedures which manipulate them.

Objects are obtained most primitively through the use of `QUOTE` (page 3), `LAMBDA` (page 4), and `OBJECT` (page **??**) special forms, and less primitively by calling procedures defined in the standard environment, e.g. `CONS` and `COMPOSE`, which create new objects or return existing ones. (An implementation of **T** would presumably define many such procedures using more primitive object constructors; for example, `COMPOSE` might be defined by a **T** program which employs `LAMBDA` to construct the composed procedure. Thus the question of what kinds of objects are truly primitive and which are not is left unanswered by this manual.)

1.1 Literals

As described in section **??**, some expressions evaluate "to themselves" or to copies of themselves. These numbers, strings, and characters, and are called *self-evaluating literals.* However, some expressions, lists and symbols in particular, do not evaluate to themselves. When an expression yielding a particular constant value is required, it is necessary to use `QUOTE`. Self-evaluating literals and quoted constants are called *literals.*

Although not all objects have external representations, some objects which do have external representations have undefined evaluation semantics. These vectors (section **??**) and the empty list. `QUOTE` must also be used to obtain these values as constants.

`(QUOTE` *object*`)` $\longrightarrow$ *object* Special form

Yields *object*. The *object* is not evaluated; thus `QUOTE` provides a literal notation for constants.

```
(QUOTE A)           ⟹  A
(QUOTE (A B))       ⟹  (A B)
(QUOTE (+ X 8))     ⟹  (+ X 8)
(QUOTE (QUOTE A))   ⟹  (QUOTE A)
```

Since `QUOTE` is used so frequently, an abbreviated external syntax is provided for `QUOTE` forms: '*object* is an alternative external representation for the list `(QUOTE` *object*`)`.

```
'A          ⟹  A
'(A B)      ⟹  (A B)
'(QUOTE A)  ⟹  (QUOTE A)
''A         ⟹  (QUOTE A)
```

Objects returned by literal expressions are read-only; they should not be altered using `SET` or any other side-effecting form.

1.2 Procedures

A *procedure* (or *routine*) is any object which may be called. Ordinarily, a procedure is called as a result of the evaluation of a *call* (page **??**). The most primitive mechanism for creating procedures is the LAMBDA special form.

(LAMBDA *variables* . *body*) ⟶ *procedure* Special form

A LAMBDA-expression evaluates to a procedure. When the procedure is called, the *variables* are bound to the arguments to the call, and the *body*, an implicit block, is evaluated. The call yields the value of the last form in the *body*.

```
((LAMBDA (X Y) (LIST Y X)) 7 'FOO) ⟹ (FOO 7)
```

If *variables* is not a proper list, but ends in a symbol x, then the variable x will be bound to a list of the arguments beginning at the position corresponding to x.

```
((LAMBDA (X . Y) (LIST Y X)) 7 'FOO 'BAZ)  ⟹  ((FOO BAZ) 7)
((LAMBDA Y Y) 7 'FOO 'BAZ)                 ⟹  (7 FOO BAZ)
```

If *body* in the LAMBDA-expression is null, a syntax error will be signalled.

If any *variable* is () instead of a symbol, then the corresponding argument in a call is ignored. Also see, IGNORE, on page **??**.

Scoping: The values of the bound variables are apparent only in code lexically contained in the body of the LAMBDA-expression, and not to routines *called* from the body. That is, like SCHEME and ALGOL and unlike most Lisp dialects, **T** is a lexically scoped language, not a dynamically scoped language.

Closure: The *procedure* is said to be a *closure* of the LAMBDA-expression in the current lexical environment. That is, when *procedure* is called, the *body* is evaluated in an environment which is the same as that in which the LAMBDA-expression was evaluated, augmented by the bindings of the *variables*. For example, if Z is mentioned in the *body*, and it is not one of the *variables*, then it refers to whatever Z was in scope when the LAMBDA-expression was evaluated, *not* (necessarily) to the the variable Z that is in scope when *procedure* is called.

LAMBDA-bindings do not shadow syntax table entries in the standard compiler. The `standard-compiler` and ORBIT, the new optimizing compiler, now have the same evaluation semantics. This is consistent with the **T** manual. In **T2**, TC, the old compiler, complied with the manual but the standard compiler did not. Thus,

```
(LET ((SET LIST) (X 5)) (SET X 8)) ⟹ 8 not (5 8)
```

However, this doesn't mean that the LAMBDA-binding has no effect, but rather that the binding is not recognized as such when the name appears in the CAR of a form. Thus,

```
(LET ((SET LIST) (X 5)) ((BLOCK SET) X 8)) ⟹ (5 8)
```

This is not a final decision. This was the easiest semantics to implement, and it is consistent with the documentation. In the future lambda bindings may shadow syntax.

1.3 Object identity

Every object has *identity* in the sense that one may determine whether two given values are the same object.

In the following:

```
(DEFINE A (LIST 'P 'Q))
(DEFINE B (LIST 'P 'Q))
```

there is no way (other than EQ?) to distinguish the two objects which are the values of A and B. Nonetheless, since LIST is defined to return a new object each time it is called, the two objects are distinct; and indeed, a side-effect to one object will not affect the value of the other:

```
(SET (CAR A) 'R)
A ⟹ (R Q)
B ⟹ (P Q)
```

Some system procedures and special forms create new objects, others return objects which already exist. In some cases, it is not defined which of the two happens; it is only guaranteed that some object with appropriate characteristics is returned. For example, `CONS` creates new objects; `QUOTE` expressions yield pre-existing constant objects; and numerical routines such as `+` may or may not create new numbers.

The `EQ?` predicate primitively determines object identity.

`(EQ?` *object1 object2*`)` ⟶ *boolean*

> Returns true if *object1* and *object2* are identically the same object. `EQ?` is the *finest* comparison predicate for objects, in the sense that if `EQ?` cannot distinguish two objects, then neither can any other equality predicate.

`(NEQ?` *object1 object2*`)` ⟶ *boolean*

> `NEQ?` is the logical complement of `EQ?`.
>
> `(NEQ?` *object1 object2*`)` ≡ `(NOT (EQ?` *object1 object2*`))`

Uniqueness of literals (other than symbols and characters) isn't defined in general. For example,

```
(EQ? '(A B C) '(A B C))
```

may yield either true or false, depending on the implementation. Two similar-looking literal expressions in different places may or may not yield distinct objects.

However, a given literal expression will always yield the same object each time it is evaluated.

```
(LET ((F (LAMBDA () '(A B C)))) (EQ? (F) (F))) ⟹ true
```

1.4 Symbols

Symbols are named objects which have wide applicability as tokens and identifiers of various sorts. Symbols are uniquely determined by their names, and have a convenient external syntax by which they may be obtained.

Uniqueness of symbols *is* defined:

```
(EQ? 'FOO 'FOO) ⟹ true
```

`(SYMBOL?` *object*`)` ⟶ *boolean* Type predicate

> Returns true if *object* is a symbol.

See also `SYMBOL->STRING` and `STRING->SYMBOL`, page **??**.

1.5 Predicates and truth values

Conditional expressions in **T** (see section **??**) usually involve the evaluation of a *test* expression; the object yielded by the test is then used to make a control decision, depending on whether the value is *false* or *true*. There is one distinguished false value called `#F`. Any other value is considered to be true. A value intended to be used in this way is called a *boolean* or *truth value*.

A *predicate* is any procedure which yields truth values. Predicates are typically given names which end in `?`, e.g. `NULL?` and `EQ?`. Calls to predicates are naturally used as test expressions.

An *equality predicate* is a two-argument predicate which is side-effectless and unaffected by side-effects, and acts like an equivalence relation in the mathematical sense.

The canonical boolean objects have convenient read syntax associated with them.

`#T` *reads as true* Read syntax

Read syntax for canonical *true* object.

`#F` *reads as false* Read syntax

Read syntax for canonical *false* object.

```
(true?  '#T)              ⟹  true
(false?  (car '(#F #T)))  ⟹  true
```

`()` *reads as empty-list* Read syntax

Read syntax for *empty-list* object.

`NIL` ⟶ `#F`

This system variable has as its value the canonical false object #F. In **T2** it was instead bound to *null*.

`T` ⟶ `#T`

The system variable `T` has as its value the canonical true value. Later versions of **T** may leave the system variable `T` unbound, so users should not depend on these semantics. Any non-false value is considered to be true for the purposes of tests.

(), #T, and #F are elements of read syntax, not evaluatable symbols.

```
(car '(#T foo))   ⟹  #T
(list #T foo)     ⟹  undefined, and is probably a syntax error.
In T2:  ()        ⟹  ()
In T3.1:  ()      ⟹  () with warning.
In future:  ()    ⟹  error
```

In **T3**, the canonical false value '#F is not necessarily the same object as the empty list, (). For compatibility with previous versions, in **T3.1** *false* and the empty list will continue to be the same object, but this will change in a future release.

`(eq?  '#F '())` ⟶ *undefined*

In **T3.1** `(eq?  #F ()` ⟹ *true*').

It is an error to use () in an evaluated position. This error currently generates a warning and treats () as '(), i.e. as if the empty list were self evaluating. An error will be signalled in the future. Use '() for empty lists, and `NULL?` to check for the empty list. `NIL` or '#F can be used for false values, and `NOT` to check for false values.

1.6 Types

A *type* may be seen either as a collection of objects with similar behavior, or as a characteristic function for such a collection, or as the behavior characteristic of similar objects; these views are equivalent.

A *type predicate* is a predicate which is defined on all objects and whose value is not affected by any side-effects (that is, calling the type predicate on a particular object will return the same value regardless of the point at which it is called with respect to arbitrary other computations). Type predicates are usually used to determine a given object's membership in a type.

1.7 Continuations

`(CALL-WITH-CURRENT-CONTINUATION` *proc*`)` ⟶ *value-of-proc* proc

The procedure `CALL-WITH-CURRENT-CONTINUATION` packages up the current continuation as an "escape procedure" and passes it as an argument to *proc*. *proc* must be a procedure of one argument. The escape procedure is an n-ary procedure, which if later invoked with zero or more arguments, will ignore whatever continuation is in effect at that later time and will instead pass the arguments to whatever continuation was in effect at the time the escape procedure was created.

The escape procedure created by `CALL-WITH-CURRENT-CONTINUATION` has unlimited extent just like any other procedure. It may be stored in variables or data structures and may be called as many times as desired. For a more thorough explanation consult the Revised3 Report on Scheme.