The T Manual
Fifth Edition
— Pre-Beta Draft —

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

October 2, 1990

Copyright ©1988
Computer Science DepartmentYale University
New Haven CT

All rights reserved.
Permission is granted to anyone to make or distribute verbatim copies of this document provided that the copyright notice and this permission are preserved.

Contents

Preface

This manual describes **T**, a programming language being developed at the Yale University Computer Science Department [RA82], and its current implementations. It is a reference manual, intended to define the language and to describe the general direction of its development. It is *not* intended as an introduction or a tutorial.

The bulk of this document describes **T** as a programming language. However, chapters 17, 18, and 19 describe pragmatic features which are specific to the current implementations.

The reader is expected to have some programming experience; the reader may also find a knowledge of Lisp helpful, although **T** is different enough from Lisp that prior experience with Lisp might actually be a hindrance.

Chapter F describes ways in which the language is deficient and directions in which it is likely to grow. Also, the implementations are not faithful to the language definition presented in the manual; section F.3 catalogs known deviations.

Please send questions and comments about the language, this manual, or the various implementations of the system, via Internet mail to T-Project@Yale.Edu, or via U.S. Mail to

T Project
Department of Computer Science
Yale University
P.O. Box 2158 Yale Station
New Haven, Connecticut 06520

Announcements of interest to **T** users are regularly broadcast via Arpanet and Usenet electronic mail to the mailing list T-Discussion@MC.LCS.MIT.EDU; send mail to T-Discussion-Request@MC.LCS.MIT.EDU if you want to be added to (or removed from) this list. Send reports about errors in the manual or implementations to `T3-Bugs@YALE.EDU`.

To be added to the Scheme mailing list send mail to Scheme-Request@MC.LCS.MIT.EDU, or subscribe to the comp.lang.scheme Usenet newsgroup.

The **T** project has been funded exclusively by the Computing Facility of the Yale Computer Science Department.

The Fifth Edition of the **T** Manual is the result of converting the 1984 **T2.7** manual to LaTeX and then slapping in the release notes from **T3.0** and **T3.1**. The **T3.0** release notes were written by James Philbin. The **T3.1** release notes are by David Kranz. Most of the work in converting the original Scribe sources to LaTeX was done by Mitchell Charity. Final cleaning up and some amount of integration was done by Dorab Patel. Thanks to Jonathan Rees, David Kranz, (and others to be mentioned), who helped with the cleanup.

[0]VMS, and VAX are trademarks of Digital Equipment Corporation. Domain and Aegis are trademarks of Apollo Computer, Inc. UNIX is a trademark of Bell Laboratories.

Acknowledgements

The authors wish to thank Kent Pitman for his continuing assistance in making **T** true.

Gerry Sussman has provided essential guidance and inspiration.

Guy Steele wrote most of the compiler we're using, and has otherwise been a strong influence, originating much of the project's design philosophy and shaping and inspiring many of its features.

This document borrows from the *Lisp Machine Manual* [WM81] and the *Common Lisp Reference Manual* [Ste84]. We are grateful to their authors.

We wish to acknowledge the influence and valuable advice provided during the design process by the following people at Yale, MIT, and elsewhere: Alan Bawden, Richard Bryan, David Byrne, George Carrette, William Clinger, Peter Deutsch, John Ellis, William Ferguson, Christopher Hanson, Carl Hoffman, David Kranz, David Littleboy, Drew McDermott, Nathaniel Mishkin, Robert Nix, Jim Philbin, John Ramsdell, Christopher Riesbeck, John Ruttenberg, Olin Shivers, and Steve Wood.

Thanks to Judy Martel for her patient proofreading.

The NIL project at MIT was the source of many of **T**'s good ideas. NIL is the work of Richard Bryan, Glenn Burke, George Carrette, Michael Genereseth, Robert Kerns, Jim Purtilo, John White, and one of the present authors (JR).

Jim Purtilo wrote the integer arithmetic package used in the current implementation while he was working on the NIL project.

We appreciate the patience of our user community, at Yale and elsewhere, who have had to put up with an incompatible, incomplete, and untuned new system.

Finally, we wish to thank John O'Donnell for helping to shelter us from the real world, and for having the foresight, or perhaps folly, to initiate the project in the first place.

Chapter 1

Introduction

T is a dialect of Lisp derived from Scheme [SS78]. It is comparable in power and expressivness to other recent Lisp dialects such as Lisp Machine Lisp [WM81], Common Lisp [Ste84], and NIL [Whi79], but fundamentally more similar in spirit to Scheme than these other Lisp dialects.

T is an attempt to create a powerful and expressive language which is portable, efficient, and suitable for a broad variety of applications, including symbolic, numerical, and systems programming.

T draws from many sources in its attempt to provide a complete yet simple and unified system. For the most part, the language's designers are collators, not originators. Common Lisp, NIL, and Scheme have been the strongest influences.

1.1 Notation

Throughout this document the following conventions are used:

{ ... } Curly braces group together whatever they enclose.

[...] Square brackets indicate that what they enclose is optional. See also *Dot-notation* below.

* Braces or brackets followed by a star, e.g. $\{ \ldots \}^*$, indicate zero or more occurences of the enclosed item. See also *Dot-notation* below.

$^+$ Braces or brackets followed by a plus sign, e.g. $[\ldots]^+$, indicate one or more occurences of the enclosed item.

| A vertical bar is used to separate alternatives in a braced or bracketed group, e.g. { thing1 | ... | thingN }.

$\Longrightarrow$ is used to indicate *evaluation*. (Evaluation is described in section 2.2.) It is read as "evaluates to". For example, `(+ 3 5)` $\Longrightarrow$ `8` means that evaluating the expression `(+ 3 5)` yields `8`.

$\equiv$ indicates *code equivalence*. It is read as "is equivalent to". For example, `(CAR (CDR` *x*`))` $\equiv$ `(CADR` *x*`)` means that for any *x*, the value and effects of `(CAR (CDR` *x*`))` are always the same as the value and effects of `(CADR` *x*`)`.

$\longrightarrow$ is used in definitions to indicate the *type* of the result. For example, `(NUMBER?` *object*`)` $\longrightarrow$ *boolean* means that the procedure named `NUMBER?` yields a boolean result.

`(PRINT` *object port*`)` $\longrightarrow$ *undefined*

means that `PRINT` returns a value of no particular interest; that is, one should not depend on it to return anything in particular.

Italicized names in the code examples are meta-variables that stand for pieces of code to be filled in. Restrictions on the particular types or values of a meta-variable are often suggested by its name or are given in the text associated with the example. The term *object* denotes data of unrestricted type.

Dot-notation , as in (ADD . *numbers*), is used where any number of sub-forms are permitted. In this example, there may be zero or more *numbers*; the description subsumes cases such as (ADD), (ADD 3), and (ADD 3 5 2). See also * and ([. . .]) above.

Settable Some routines described in this manual, such as VREF (page 89), serve as *access routines* in the sense that they are appropriate for use in SET and related forms to designate locations. These routines have their descriptions flagged with the notation *Settable*.

Type Predicate Functions that take an object of any type as an argument and return a boolean are called type predicates. They return true if the argument was of the required type and false otherwise.

Operation Functions that dispatch on their first argument are labelled as *operations*. See Chapter 7.

1.2 Naming Conventions

Below are listed some of the naming conventions used for standard **T** procedures, variables, and reserved words.

. . . ? The suffix ? indicates a predicate. For example, ZERO? is a predicate which returns true if its argument is numerically equal to zero.

. . . ! The suffix ! indicates a side-effecting variant of a non-side-effecting procedure. For example, REVERSE! is the same as REVERSE, but it alters its argument, recycling the storage to form its result.

. . . ->. . . Infix -> is used in names of *coercion procedures*, which are routines which perform conversions between objects of different types. For example, given a string, STRING->LIST forms a list of the characters in the string.

*. . . * The names of mutable variables whose scope is not visually apparent to someone reading the program is usually bounded on either side with *'s.

DEFINE-. . . Relates to the assignment of something in a permanent way. DEFINE is typically used for defining procedures, DEFINE-OPERATION to define generic operations, and so forth.

MAKE-. . . Indicates a routine which creates an object of some type, given size (and sometimes datatype) information. For example, (MAKE-STRING 5) makes a string of length 5.

NOT-. . . The prefix "NOT-" means that the truth value of some predicate is complemented. For example, (NOT-LESS? *x y*) is the same as (NOT (LESS? *x y*)).

. . . CDR Relates to the subtail or successive subtails of a list. For example, while the procedure named NTH returns the nth *element* of a list, there is another procedure named NTHCDR that returns the nth *subtail*.

. . . Q The presence of the trailing character "Q" on a name typically means that the predicate "EQ?" is involved. For example,

(MEMQ? *x l*) ≡ (MEM? EQ? *x l*)

1.3 Language principles and conventions

The design of **T** employs a number of conventions. These conventions make the language more regular, predictable, and easy to learn.

Anonymity: objects are not tightly coupled to their names.

Locality: effects may be achieved locally.

Argument order: procedures which extract components from aggregate objects generally take the aggregate object as the first argument, and selection information (if any) as subsequent arguments. Similarly, assignment procedures take the aggregate object or location specification early, and the value to be stored last.

Indexing: where numeric indices are involved, the indices begin with zero and range up to a given limit. Ranges are specified as *half-open intervals*, inclusive at the low end and exclusive at the high end.

Abbreviations: are avoided in names of **T** entities. Words are spelled out and separated by hyphens, e.g. `DEFINE-OPERATION`, `STRING-LENGTH`. Where abbreviations are used, they are used consistently: e.g. `CHAR` abbreviates `CHARACTER`, as in `CHAR?`, and `ELT` abbreviates `ELEMENT`, as in `STRING-ELT`.

Chapter 2

Syntax and semantics

This chapter gives an overview of **T** as a programming language.

The **T** language logically comprises three distinct components:

an *external representation* for objects as character sequences,

a *core language* - syntax and semantics for expressions, and

a *standard environment* - the behavior of the objects which are values of system variables.

2.1 External representation

Like other Lisp dialects, and in contrast with most other programming languages, **T** has two different kinds of syntax: the *external representation* by which objects (data) are represented as a linear sequence of characters, and an *expression syntax* by which these objects can be understood as programs. That is, the meaning of a **T** program represented as characters in a file or other external storage medium must be determined in two stages: first, by mapping the characters to objects, and then by interpreting these objects as executable programs.

The external syntax of **T** is very similar to that of other Lisp dialects, and is discussed in detail at appropriate places in this manual. The following gives only the most cursory description. Characters may be classified according to their lexical properties; the two most important distinctions are between *constituent* and *delimiter* characters, and between *read-macro* characters and non-read-macro characters. Alphabetic, numeric, and some special characters (e.g. `-` and `$`) are constituent characters; whitespace and some special characters (e.g. left and right parenthesis) are delimiter characters. Some special characters such as `(` and ' are read-macro characters; these introduce special syntactic constructs which are idiosyncratic to the particular character introducing them.

A delimited sequence of consecutive constituent characters represents either a *number* or a *symbol*. For example, `255` represents the integer 255, and `APPEND` and `append` both represent the symbol `APPEND`. Here, and throughout the manual, the term *symbol* is being used in a technical manner to refer a particular kind of named object (see section 3.4).

Balanced parentheses with a sequence of (representations of) objects between them represent a list of the (represented) objects. For example, the characters `(A B (C 12) D)` represent a list of four objects, three of which are symbols, and one of which (the third one) is a list of two objects, the symbol `C` and the integer 12. Parentheses which enclose no objects represent an empty list.

There are also external representations for strings and characters, as well as for some kinds of objects which are syntactically illegal as expressions (for example, vectors). Not all objects have external representations, however.

2.2 Core language

The core language is described in terms of a hypothetical machine (called an *evaluator*) which executes **T** code directly. In practice, the existing **T** implementations have (at least) two evaluators, both of which perform evaluation as a two-stage process consisting of compilation (syntactic and semantic analysis) followed by interpretation.

Evaluation is a process whereby an object called an *expression* (the term *form* is used synonymously with *expression*) is mapped to another object, called its *value*. Evaluation occurs in the context of a particular *variable environment*; see below. The expression is said to *yield* its value.

The evaluation mapping is not a purely mathematical mapping, since in some cases the evaluation of an expression may depend not only on the variable environment but on the *state* of the running **T** system, or on the state of the world outside it; and evaluation may cause changes in the state of system or the world, which may, in turn, affect future evaluations. These state changes are called *side-effects*.

The rules by which an object is evaluated are as follows:

Self-evaluating literals: All numbers, strings, and characters are syntactically valid expressions which, when evaluated, yield themselves. For example,

```
-2102          ⟹  -2102
#\M            ⟹  #\M
"A string."    ⟹  "A string."
```

The notation "*expression* $\Longrightarrow$ *value*" means that *expression*, when evaluated, yields *value*. (Note also that we are making use of the external object syntax itself as a notational device: "the object `-2102`" could be said more precisely as "an object externally represented by the characters "`-2102`"." Just as a program should never be confused with an object which represents it, an object should never be confused with a sequence of characters which notates it.)

Symbols: As evaluable expressions, symbols are interpreted as variable references. A symbol evaluates to its value according to the current variable environment. For example, in an environment in which the variable `DELTA` has the value `15`, evaluating the expression `DELTA` will yield `15`.

Symbols have many uses other than as names for variables. It is important not to confuse the use of a symbol as a datum manipulated by a program with the occurrence of a symbol as a variable reference in a program. Symbols do not have values *a priori*; variables only have values by virtue of the context in which they occur.

Lists: Non-empty lists are classified either as *calls* or as *special forms*. If the first element of the list is a symbol, and the symbol is a *reserved word*, then the list is a special form; otherwise it is considered to be a *call*. **T** reserved words , for example, the symbols `QUOTE`, `IF`, and `LAMBDA`. (However, see section 14.4.)

Special forms: The syntax and semantics of a special form are idiosyncratic to the reserved word which introduces it; descriptions of the meaning of expressions introduced by the various reserved words are therefore distributed throughout the manual.

Calls: Calls are evaluated as follows: the elements of the list (including the first) are evaluated, **in an undefined order**. The first must evaluate to a procedure; this procedure is *applied* to the rest of the values, which are called *arguments*. (The verbs *call* and *invoke* mean the same as *apply*.)

The evaluation order of arguments in a combination is undefined. ORBIT produces code in which the evaluation order of arguments is undefined and not necessarily left to right.

Particularly insidious bugs have resulted from `LET`, see page 27, forms whose clauses contain order dependent side effects. `LET*` should be used to ensure sequential evaluation order.

2.3 The standard environment

The standard environment corresponds to what is usually known as the "run-time library" in other language environments such as C or Pascal.

New variable environments may be introduced in various ways (see chapter 4), but a **T** system is obliged to supply one standard environment in which system variables are bound to system procedures and constants, as defined by this manual. Program execution typically occurs in an environment inferior to this standard environment (see section 17.1), so that these objects are easily accessible as values of lexically apparent variables. For example, in this standard environment, the variable `CONS` has a certain procedure as its value.

For the most part, the values of system variables are procedures, and therefore the only behavior of interest is what they do when called. (See the discussion of calls, above.) In other cases, values are objects such as numbers or symbols.

A representation of the standard environment is available as the value of `STANDARD-ENV` (page 116).

2.4 Undefined

The term *undefined* is used in two different ways in this manual. An expression may *yield an undefined value*, in which case it yields some value, the particular value not being defined by this manual. In such cases it is unwise to depend on this value having any particular characteristics, for example, it being null, or not a number, or whatever. The evaluation of the expression and the creation of the undefined value are not in error; it is the use to which this value is put that may lead to problems. Expressions yielding undefined values are generally useful only for any side-effects they cause.

On the other hand, the evaluation of an expression may *have an undefined effect*, in which case an implementation will endeavor to signal an error condition, permitting a user to take appropriate action. For example, a call to a non-procedure, or adding two symbols together, have undefined effects. An implementation is not *obliged* however, to signal an error, and in fact it may be in the interest of efficiency to avoid the overhead of detecting circumstances under which undefined effects will happen.

The two procedures `UNDEFINED-VALUE` and `UNDEFINED-EFFECT` are used for expository purposes in examples throughout this manual; they are described in section 13.2.

2.5 Multiple values

T now supports multiple return values. This makes procedure call and return uniform, in the sense that a procedure can be invoked with zero or more values and can return zero or more values. See page 38.

Like procedures, continuations have certain expectations about the number of arguments (values) that will be delivered to them. It is an error if more or fewer values are delivered than expected.

The idiom `(RETURN)` is useful for procedures that return an undefined value and many of the system procedures whose value(s) is undefined now return no value. However, the procedure `undefined-value` may provide a more informative error message.

Other forms, such as `CATCH` and `RET`, have been extended to allow multiple return values.

Chapter 3

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 21), `LAMBDA` (page 22), and `OBJECT` (page 47) 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.)

3.1 Literals

As described in section 2.2, 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 13.6) 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.

3.2 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 18). 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 85.

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. 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.

3.3 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*

3.4 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 84.

3.5 Predicates and truth values

Conditional expressions in **T** (see section 5.1) 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)`	$\Longrightarrow$	*true*
`(false? (car '(#F #T)))`	$\Longrightarrow$	*true*

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

Read syntax for *empty-list* object.

`NIL` $\longrightarrow$ `#F`

This system variable has as its value the canonical false object `#F`.

`T` $\longrightarrow$ `#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))`	$\Longrightarrow$	`#T`
`(list #T foo)`	$\Longrightarrow$	*undefined*, and is probably a syntax error.
In **T3.1**: `()`	$\Longrightarrow$	`()` with *warning*.
In future: `()`	$\Longrightarrow$	*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 '())` $\longrightarrow$ *undefined*

In **T3.1** `(eq? #F ()` $\Longrightarrow$ *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.

3.6 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.

3.7 Continuations

(`CALL-WITH-CURRENT-CONTINUATION` *proc*) $\longrightarrow$ *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.

Chapter 4

Environments

4.1 Environments and contours

Environments are associations between *identifiers* (variable names) and *values*. One such association between an identifier and a value is called a *binding*. In general, environments are created implicitly, for example, on entering LAMBDA-bodies.

Environments are organized hierarchically into *contours*. Each contour augments an "outer" environment, providing bindings for a few additional identifiers. These bindings are in effect in the body of the expression which introduces the contour. The portion of a program in which a binding is in effect is known as the variable's *scope*.

For example:

```
(BLOCK
       (LIST 'A 'B)
       ...
       (LAMBDA (A B C)
               ... ; [1]
               (LAMBDA (D E)
                       ... ; [2]
                       ())
               ... ; [3]
               (LAMBDA (A F G)
                       ... ; [4]
                       )
               ... )
       ... )
```

Each LAMBDA introduces a new contour. This code presumably occurs in an outermost environment in which the variable LIST is bound. (The values of BLOCK and LAMBDA are not in question here; they are reserved words, and their bindings as variables are irrelevant.) At point [1] in the program, a new environment is in effect which now has bindings for A, B, and C, as well as for any variables known in the outer environment. At [2], D and E also have values. [3] is in the scope of A, B, and C but not in the scope of D and E. Finally, the outer binding of any given identifier is *shadowed* by inner bindings of that identifier; an occurrence of the identifier A appearing at [4] refers not to the outer A but to the inner one, because the inner one shadows the outer.

There are two kinds of contours, known as *lambda-contours* and *locales*. Lambda-contours are introduced by many special forms, such as LAMBDA, LET, and LABELS. Locales are introduced by LOCALE special forms.

4.2 Local variables

(LET *specs* . *body*) ⟶ *value-of-body* Special form

LET provides a convenient syntax for introducing local bindings. Each *spec* should be a two-element list of the form (*variable value*). The *value* expressions are all evaluated (in no particular order), then the *body* (an implicit block) is evaluated in an environment where the *variables* are bound to those values. The value of *body* is yielded. The result of the expression is undefined if more than one *spec* has its first element to be the same *variable*.

```
(LET ((X 2)) X) ⟹ 2
(LET ((var1 val1) (var2 val2) ... (varn valn)) . body)
≡ ((LAMBDA (var1 var2 ... varn) . body) val1 val2 ... valn)
```

See also DESTRUCTURE (page 70).

(LET* *specs* . *body*) ⟶ *value-of-body* Special form

LET* is like LET, except that the binding of variables to values is done sequentially, so that the second binding is done in an environment in which the first binding has already occurred, etc. In a LET-expression, all the bindings are done in one environment. For example, suppose the variable A is already bound to 10 when the expression (LET ((A 20) (B A)) B) is evaluated. The result is 10, not 20, because B is bound to A's value in the outer environment. In contrast, (LET* ((A 20) (B A)) B) evaluates to 20.

```
(LET* ((var1 val1) (var2 val2) ... (varn valn)) . body)
     ≡
(LET ((var1 val1))
     (LET ((var2 val2))
          ...
          (LET ((varn valn)) . body)... ))
≡
((LAMBDA (var1)
         ((LAMBDA (var2)
                  ...
                  ((LAMBDA (varn) . body)
                  valn) ... )
         val2))
val1)
```

(LABELS *specs* . *body*) ⟶ *value-of-body* Special form

LABELS is useful for defining local procedures that may be mutually recursive. Each *spec* should be of the form (*variable value*). The *value*-expressions are evaluated (in no particular order), and the *body* is evaluated in an environment where all the *variables* are bound to those values. However, LABELS differs from LET in that the bindings of the *variables* are already in effect (scope) before the *value*-expressions are evaluated, so that the *value*-expressions can refer to any of the *variables*, including the ones to which they themselves will be bound (thus permitting local recursive procedures).

Each *spec* may alternatively have the form

```
((variable . argument-vars) . body)
```

This is equivalent to

```
(variable (LAMBDA argument-vars . body))
```

This is intended to parallel the syntax for DEFINE (see page 29).

In the following example, note that REV-1 is bound to a value that refers to REV-1.

```
(DEFINE (REVERSE L)
 (LABELS (((REV-1 L RESULT-SO-FAR)
          (COND ((NULL? L) RESULT-SO-FAR)
```

```
            (ELSE
             (REV-1 (CDR L)
                    (CONS (CAR L) RESULT-SO-FAR))))))
      (REV-1 L '()))))
```

As an extension to the dialect of SCHEME described in [SS78], **T** does not restrict the *value* expressions to be LAMBDA-expressions. However, the values of the *variables* are not defined until *body* is evaluated; i.e., the variables should not be used in the *value*-expressions. For example, (LABELS ((A B) (B 1)) . *body*) is ill-formed; at the point when the expressions B and 1 are being evaluated, the variable B is bound, but will have an undefined value. Consequently, A will have an undefined value. By contrast, in the well-formed expression

    (LABELS ((A (LAMBDA (X) (+ X B))) (B 1)) . body)

which can also be written

    (LABELS (((A X) (+ X B)) (B 1)) . body)

the value of B is not used before *body* is evaluated. The LAMBDA-expression evaluates to a *closure* that includes B. By the time A (the closure) is called, B will have a value.

LABELS cannot easily be described in terms of LAMBDA. However, it can be described in terms of LAMBDA and SET. (SET performs side-effects on bindings; see page 41.) This is misleading because LABELS should not be thought of as a side-effecting operator.

```
(LABELS ((var1 val1)
         (var2 val2)
         ...
         (varn valn))
        .  body)
        ≡
(LET ((var1 (UNDEFINED-VALUE))
      (var2 (UNDEFINED-VALUE))
      ...
      (varn (UNDEFINED-VALUE)))
     (SET var1 val1)
     (SET var2 val2)
     ...
     (SET varn valn)
     .  body)
```

This equivalence is overly concrete in that the order of evaluation of the val_i expressions, and the sequence in which the variables are given their values, is not defined.

4.3 Locales

T3: We are working on a module system which will eventually subsume the functionality of locale.

Locales serve two purposes. First, they provide an incremental or declarative syntax for creating variable bindings in a contour. Second, they provide access to environments as objects which can be dynamically manipulated. The term *locale* is used to mean both the kind of contour introduced by a LOCALE-expression, and the object which represents the environment.

There are no global variables in **T**; all variables are lexically bound in some contour. Locales play the role of what is known in other Lisp and Scheme dialects as the global environment.

(DEFINE *variable value*) ⟶ *undefined* Special form
(DEFINE (*variable* . *arguments*) . *body*) ⟶ *undefined*

DEFINE creates a binding for *variable* in the lexically innermost locale in which the DEFINE expression occurs. The variable is given *value* as its value.

The second DEFINE syntax is for defining procedures.

(DEFINE (*variable* . *arguments*) . *body*)

is equivalent to

(DEFINE *variable* (LAMBDA *arguments* . *body*))

For example:

```
(DEFINE (F X) (LIST X 2))    ≡  (DEFINE F (LAMBDA (X) (LIST X 2)))
(DEFINE (F . X) (LIST X 2))  ≡  (DEFINE F (LAMBDA X (LIST X 2)))
```

(LSET *variable value*) ⟶ *undefined* Special form
(LSET *variable value*) ⟶ *value*

The value of the LSET special form is undefined, and it is an error to use an LSET form in a value requiring position. In **T3.0**, LSET will continue to return a value, but an error will be signalled in the future.

LSET is identical to DEFINE except that the procedure definition syntax is not permitted, and the variable is declared to be alterable. Variables bound by DEFINE may not be altered with SET; variables bound by LSET may be. See SET, page 41.

(MAKE-LOCALE *superior-locale identification*) ⟶ *locale*

Creates a locale inferior to *superior-locale*. *Identification* may be any object, and is used only for debugging purposes.

```
(DEFINE *FOO-ENV* (MAKE-LOCALE STANDARD-ENV '*FOO-ENV*))
```

(MAKE-EMPTY-LOCALE *identification*) ⟶ *locale*

Creates a locale containing no bindings whatsoever.

(LOCALE? *object*) ⟶ *boolean* Type predicate

Returns true if *object* is a locale.

4.4 Non-local reference

(*VALUE *locale identifier*) ⟶ *object* Settable

Accesses the value of *identifier* in *locale*. If *identifier* has no value in *locale*, then the effect of the call to *VALUE is undefined.

```
(*VALUE STANDARD-ENV 'T)                    ⟹  true
(LSET MY-LOCALE (MAKE-EMPTY-LOCALE 'MYSTUFF))
(*DEFINE MY-LOCALE 'A 10)
(*VALUE MY-LOCALE 'A)                       ⟹  10
```

(*DEFINE *locale identifier value*) ⟶ *undefined*

Defines the value of *identifier* in *locale* to be *value*.

(*LSET *locale identifier value*) $\longrightarrow$ *value*

Creates a binding for *identifier* in *locale* with initial value *value*.

(IMPORT *locale* . *variables*) $\longrightarrow$ *undefined*

Locally defines *variables* to have the same values as their values in *locale*.

```
(IMPORT locale var1 var2 ... varn)
         ≡
(BLOCK (DEFINE var1 (*VALUE locale 'var1))
       (DEFINE var2 (*VALUE locale 'var2))
       ...
       (DEFINE varn (*VALUE locale 'varn)))
```

Chapter 5

Control

5.1 Conditionals

(COND . *clauses*) $\longrightarrow$ *value-of-clause* Special form

COND is a general conditional construct. Each *clause* should have one of the following forms:

(*test* . *body*)

(*test*)

(*test* => *procedure*)

The *test* expressions are evaluated in sequence until one yields true, at which point:

If it is part of a clause of the form (*test* . *body*), then *body*, an implicit block, is evaluated, and its value is what the COND-expression yields.

If the clause has the form (*test*), then the COND-expression yields the value of *test* (which, of course, is non-null).

If the clause has the form (*test* => *procedure*), then the *procedure* expression is evaluated, and its value, which should be a procedure, is called with the value of the *test* as its one and only argument. The COND yields the value of that call.

Example:

```
(COND ((PAIR? X) (GAUR (CAR X)))
      ((FIXNUM? X) (GAUR X)))
```

If all the *tests* yield false, then the COND yields an undefined value. See section 2.4 for a discussion of undefined values.

The system variable ELSE has a non-null value. This provides a convenient way to specify a branch to take when all other *tests* fail.

```
(COND ((PAIR? X) (GAUR (CAR X)))
      ((ASSQ X *SAMPLE-ALIST*) => CDR)
      ((GAUR X))
      (ELSE NIL))
```

(XCOND . *clauses*) $\longrightarrow$ *value-of-clause* Special form

Similar to `COND`, but its effect is undefined (and presumably an error is signalled) if no *clause* is selected. `X` is mnemonic for "exhaustive".

`ELSE` ⟶ *true*

`ELSE` has as its value some true value. It exists mainly for use in `COND`.

`(IF` *test consequent alternate*`)` ⟶ *value-of-arm* Special form
`(IF` *test consequent*`)` ⟶ *undefined*

`IF` is a primitive two-way conditional. The *test* expression is evaluated. If it yields true, then the *consequent* expression is evaluated, and the `IF` yields its value. Otherwise, the *alternate* expression, if any, is evaluated, and the `IF` yields its value. If *test* yields false and there is no *alternate*, then the value of the `IF` is undefined.

```
(IF T 1 2)                 ⟹  1
(IF (PAIR? 'X) 'FOO 'BAR)  ⟹  BAR
(IF test consequent)       ≡   (IF test consequent (UNDEFINED-VALUE))
```

`(CASE` *key* `.` *clauses*`)` ⟶ *value-of-clause* Special form

`CASE` performs a multi-way dispatch based on the value of the *key* expression. Each *clause* should be of the form `(`*keys* `.` *body*`)`, where *keys* is a list of values (*not* expressions!) against which the value of the *key* expression is compared (using `EQ?`), and *body* is an implicit block. The body of the clause which matches the *key* value is evaluated, and the `CASE`-expression yields the value of the body. The last *clause* may be of the form `(ELSE .` *body*`)`; this designates a default action to be taken if no other clause is selected. If the *key* never matches and there is no default clause, then the `CASE`-expression yields an undefined value.

```
(CASE 'B ((A) 'LOSE) ((B C) 'WIN) (ELSE NIL)) ⟹ WIN
```

`(XCASE` *key* `.` *clauses*`)` ⟶ *value-of-clause* Special form

Similar to `CASE`, but no `ELSE`-clause is permitted, and the effect is undefined (and presumably an error is signalled) if no *clause* is selected.

`(SELECT` *key* `.` *clauses*`)` ⟶ *value-of-clause* Special form

`SELECT` is like `CASE`, but the car of each *clause* is a list of expressions that are evaluated, instead of a list of constants.

```
(DEFINE *RED-COLOR* ... )
(DEFINE *GREEN-COLOR* ... )
(DEFINE *BLUE-COLOR* ... )
(SELECT COLOR
        ((*RED-COLOR* *BLUE-COLOR*) 'INORGANIC)
        ((*GREEN-COLOR*)            'ORGANIC))
```

`(XSELECT` *key* `.` *clauses*`)` ⟶ *value-of-clause* Special form

Similar to `SELECT`, but no `ELSE`-clause is permitted, and the effect is undefined (and presumably an error is signalled) if no *clause* is selected.

`(NOT` *object*`)` ⟶ *boolean* Type predicate

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

`NOT` returns true if *object* is false, and returns false otherwise.

```
(NOT NIL)   ⟹   true
(NOT T)     ⟹   false
(NOT 3)     ⟹   false
```

`(AND . tests)` ⟶ *value-of-test* or *false* — Special form

The *test* expressions are evaluated from left to right, until one evaluates to false, at which point the `AND` yields false. If no *test* evaluates to false then the value of the `AND` is the value of the last *test*.

```
(AND 3 4)     ⟹   4
(AND 3 NIL)   ⟹   false
(AND)         ⟹   true
```

`(OR . tests)` ⟶ *value-of-test or false* — Special form

The *test* expressions are evaluated from left to right, until one evaluates to true (i.e., not null); its value is yielded as the value of the `OR`. If no *test* evaluates to true, then the value of the `OR` is false.

```
(OR 3 4)     ⟹   3
(OR 3 NIL)   ⟹   3
(OR)         ⟹   false
```

`(*AND . objects)` ⟶ *object*

If any of the *objects* is false, `*AND` returns false; otherwise it returns the last *object*. This is superficially similar to `AND`, but it is a procedure, not a special form. That is, the value of `*AND` is a procedure, whereas `AND` is a reserved word which introduces a special syntactic form. This fact implies that (a) the variable `*AND` has a value suitable to be passed as an argument to some procedure, and (b) a call to `*AND` behaves differently from an `AND` special form in the case that the argument expressions have side-effects. In

```
(AND NIL (FOO))
```

`FOO` will not be called, whereas in

```
(*AND NIL (FOO))
```

`FOO` *will* be called.

`(*OR . objects)` ⟶ *object*

If any of the *objects* is true, `*OR` returns that object; otherwise it returns false. `*OR`'s relation to `OR` is analogous to `*AND`'s relation to `AND`.

`(*IF test consequent alternate)` ⟶ *object*

If *test* is true, `*IF` returns *consequent*; otherwise it returns *alternate*. `*IF`'s relation to `IF` is analogous to `*AND`'s relation to `AND`.

5.2 Iteration

Separate iteration constructs are not strictly necessary in **T**, since iteration may be written using recursive procedures defined by `LABELS` (or even `DEFINE`). There is no penalty for this in **T**, either in efficiency or practicality (for example, stack size), because tail-recursions are implemented without net growth of stack, and it is expected that compilers will implement local procedures efficiently.

However, the following two iteration constructs provide a somewhat more convenient notation for loops than does `LABELS`. (There are also some higher-order procedures which perform iteration; see for example `MAP`, page 65, and `WALK`, page 66.)

`(DO specs (end-test . exit-forms) . body)` ⟶ *value-of-exit* — Special form

Iterates until *end-test* yields true. Each *spec* should be of the form (*variable initial next*). The *variables* are initialized (bound) in parallel to the values of the *initial*-expressions, as with LET; then, the *end-test* and *body* are executed iteratively. If the *end-test* yields true on any iteration, then the loop terminates, at which point the *exit-forms* are evaluated. The value of the DO-expression is the value of the last *exit-form*, or the value of the *end-test* if there are no *exit-forms* (this is like a COND clause). At the end of each iteration, the *next*-expressions are all evaluated, and the *variables* are re-bound to the values of the *next*-expressions.

DO is useful for loops which have a single termination condition, such as numerical, ALGOL-style for-loops, loops where desired results are accumulated in variables, and loops that perform side-effects on every element in a sequence (see WALK, page 66).

```
(REVERSE L) ≡
 (DO ((L L (CDR L))
      (RESULT '() (CONS (CAR L) RESULT)))
     ((NULL? L) RESULT)))
(VECTOR-FILL VECTOR CONSTANT) ≡
 (DO ((I (-1+ (VECTOR-LENGTH VECTOR)) (-1+ I)))
     ((<0?  I) VECTOR)
     (VSET VECTOR I CONSTANT))
```

Any DO-expression may be rewritten using the more primitive LABELS construct. The following example has only one induction variable and one *body*-expression, and assumes that there are no name conflicts with the variable LOOP.

```
(DO ((variable initial next)) exit-clause body)
    ≡
(LABELS (((LOOP variable)
          (COND exit-clause
                (ELSE body
                      (LOOP next)))))
        (LOOP initial))
```

(ITERATE *variable specs* . *body*) ⟶ *value-of-body* Special form

Another iteration construct. *Specs* has the form

((arg_1 val_1) (arg_2 val_2) ... (arg_n val_n))

The *variable* is bound to a local procedure whose formal parameters are arg_1 arg_2 ... arg_n. The procedure is initially invoked with the *val*s bound to the corresponding *arg*s.

ITERATE is more flexible than DO. It is useful in loops where the terminating condition occurs in the middle of the body of the loop, or where there is more than one terminating condition, or where the iteration may occur in several places.

For example, consider the following example (from [SS78]), a procedure to sort a list of trees into atoms and lists:

```
(DEFINE (COLLATE X)
        (ITERATE COL ((Z X) (ATOMS '()) (LISTS '()))
                 (COND ((NULL? Z)
                        (LIST ATOMS LISTS))
                 ((ATOM? (CAR Z))
                        (COL (CDR Z) (CONS (CAR Z) ATOMS) LISTS))
                 (ELSE
                        (COL (CDR Z) ATOMS (CONS (CAR Z) LISTS))))))
```

A rough analogy ITERATE : LABELS :: LET : LAMBDA holds. ITERATE expands trivially into LABELS, but may be more convenient than LABELS for the same reason that a LET form may be preferable than a call to a LAMBDA-expression.

```
(ITERATE variable
         ((arg1 val1) (arg2 val2) ... (argn valn))
         .  body)
         ≡
(LABELS (((variable arg1 arg2 ... argn) .  body))
        (variable val1 val2 ... valn))
```

5.3 Procedure application

The only operation which all procedures support is invocation. All initial system routines, like `CAR`, `LIST`, and `EQ?`, are procedures. Procedures are usually created by evaluating `LAMBDA`- or `DEFINE`-forms.

The standard way to apply a procedure to arguments is with a *call* expression. Calls are described on page 18.

(`PROCEDURE?` *object*) $\longrightarrow$ *boolean* — Type predicate

Returns true if *object* is a procedure, i.e., if it may be called.

(`APPLY` *procedure* . *objects*) $\longrightarrow$ *object*

`APPLY` applies a procedure to a list of arguments.

The sequence of *object* arguments should have the form

arg_1 arg_2 ... arg_n arglist

where n may be 0. The *procedure* is called with arg_1, arg_2, etc., as the first n arguments, and the members of *arglist* as final arguments.

```
(APPLY CONS '(1 2))            ⟹  (1 . 2)
(APPLY CONS 1 '(2))            ⟹  (1 . 2)
(APPLY CONS 1 2 '())           ⟹  (1 . 2)
(APPLY LIST 1 2 3 '(4 5 6))    ⟹  (1 2 3 4 5 6)
```

5.4 Sequencing

(`BLOCK` . *body*) $\longrightarrow$ *value-of-body* — Special form

Body is a non-empty sequence of expressions. These expressions are evaluated in order from left to right. The `BLOCK`-expression yields the value of the last expression.

Because their values are ignored, expressions other than the last are useful only for side-effects they may cause.

Since the bodies of most special forms are implicitly blocks, `BLOCK` is useful only in places where syntactically a single expression is needed, but some sequencing is required.

(`BLOCK0` *first-form* . *body*) $\longrightarrow$ *value-of-first-form* — Special form

The *first-form* is evaluated, then the expressions in the *body* are evaluated. The `BLOCK0`-expression yields *first-form*'s value.

5.5 Explicit return

(`RET` $\{value\}^*$)

Returns zero or more values as the value of the current `read-eval-print` loop. *value* defaults to an undefined value if not supplied. See section 19.1 for examples.

5.5.1 Non-local exits

`(CATCH` *identifier* `.` *body*`)` $\longrightarrow$ *value-of-body* Special form

CATCH provides a structured, non-local exit facility similar to PROG and RETURN (without GO), or CATCH and THROW, in other Lisps. The *identifier* is bound to an "escape procedure" corresponding to the *continuation* of the CATCH form. The escape procedure is now an n-ary procedure. This means that CATCH forms can ruturn multiple values. The escape procedure can be invoked only during the dynamic extent of the CATCH form (see section 3.7). If the escape procedure is called as a result of evaluating *body*, then the argument to the *escape procedure* is returned as the value of the CATCH-expression. If the escape procedure is not called, then CATCH returns whatever *body* returns. A call to an escape procedure is called a *throw*.

In the following example, the call to LIST never happens, because one of its subforms performs a throw.

```
(CATCH X (LIST 1 (X 2) 3))     ==>  2
(CATCH X (LIST 1 (X 2 3) 4))   ==>  2 3
```

A throw may cause side-effects if the dynamic state in effect when the escape procedure is called differs from that in effect when evaluation of its corresponding CATCH-expression began. **T**'s CATCH is a restriction of SCHEME's. Continuations are only valid within the dynamic extent of the CATCH-expression; however, because of CALL-WITH-CURRENT-CONTINUATION (see page 25) the CATCH can yield a value at most once. In this case the throw undoes the effects of BIND forms, and must perform cleanup action requested by UNWIND-PROTECT forms, in the process of "unwinding" the evaluation context from that of the throw to that of the CATCH. See section 6.3 for descriptions of BIND and UNWIND-PROTECT.

5.5.2 Returning multiple values

T supports multiple return values. This makes procedure call and return uniform, in the sense that a procedure can be invoked with zero or more values and can return zero or more values.

`(RETURN` $\{$*value*$\}^*$ `)`

RETURN returns its arguments as the *values* of the current expression. In order to access the *values* of a RETURN expression the *values* must be bound to identifiers using either RECEIVE or RECEIVE-VALUES.

For example,

`(LAMBDA () (RETURN 1 2 3))` $\Longrightarrow$ `1 2 3`

where "$\Longrightarrow$ 1 2 3" denotes *evaluates to* the *three* values 1, 2, and 3.

Like procedures, continuations have certain expectations about the number of arguments (values) that will be delivered to them. It is an error if more or fewer values are delivered than expected. There are only a small number of ways to create continuations, thus one need only understand these cases:

Implicit continuations, e.g. those receiving an argument of a combination or the predicate of an IF, expect exactly one value, thus

```
(LIST (VALUES 1) 2) ==> (1 2)
```

but

```
(LIST (VALUES) 2)        is an error
(LIST (VALUES 1 2) 2)    is an error
```

In a BLOCK, a continuation which proceeds to execute subsequent commands (e.g. the continuation to the call to FOO in (BLOCK (FOO) 2)) accepts an arbitrary number of values, and discards all of them.

RECEIVE expressions (and the subprimitive RECEIVE-VALUES) creates a continuation which accepts whatever values are delivered to it, and passes them to a procedure; and of course it is an error if this procedure is passed the wrong number of values.

RETURN when *invoked* with no arguments returns to the calling procedure with no value. Thus (RETURN) will return to its caller with no value. It is an error to return no value to a value-requiring position. For example,

```
(LIST 'A (RETURN)) ⟹ error
```

The idiom (RETURN) is useful for procedures that return an undefined value. Many of the system procedures whose value is undefined now return no value. The procedure UNDEFINED-VALUE may provide a more informative error message.

(RECIEVE-VALUES *receiver sender*) ⟹ *value(s) of receiver*

RECEIVE-VALUES returns the value of applying *receiver*, a procedure of n arguments, to the values returned by *sender*. *sender* is a *thunk*, a procedure of no arguments, which returns n values.

For example,

```
(RECEIVE-VALUES (LAMBDA (X Y) (LIST X Y))
                (LAMBDA () (RETURN 1 2))) ⟹ (1 2)
```

(RECEIVE ({ident}*) *expression* {body}*) ⟹ *value-of-body* Syntax

In a RECEIVE form the *expression* is evaluated in the current environment and the values returned by the *expression* are bound to the corresponding identifiers. *body*, which should be a LAMBDA body, i.e. a sequence of one or more expressions, is evaluated in the extended environment and the *values* of the last expression in *body* is returned.

The expression

```
(RECEIVE (A B C) (RETURN 1 2 3)
         (LIST A B C))
         ⟹ (1 2 3)
```

is equivalent to

```
(RECEIVE-VALUES (LAMBDA (A B C) (LIST A B C))
                (LAMBDA () (RETURN 1 2 3)))
                ⟹ (1 2 3)
```

5.6 Lazy evaluation

Delays provide a general mechanism for delayed (lazy) evaluation, also known as "call-by-need".

(DELAY *expression*) ⟶ *delay* Special form

Returns a delay of the *expression*. The computation of the expression, which happens at most once, is delayed until its value is requested with FORCE.

```
(DEFINE (INF-LIST-OF-INTEGERS N)
        (CONS N (DELAY (INF-LIST-OF-INTEGERS (1+ N)))))
(HEAD (TAIL (TAIL (INF-LIST-OF-INTEGERS 1)))) ⟹ 3
(DEFINE HEAD CAR)
(DEFINE (TAIL OBJ) (FORCE (CDR OBJ)))
```

The behavior of DELAY and FORCE can be described by the following hypothetical implementation:

```
(DEFINE-SYNTAX (DELAY EXP)
               ‘ (MAKE-DELAYED-OBJECT (LAMBDA () ,EXP)))
(DEFINE (MAKE-DELAYED-OBJECT THUNK)
        (LET ((FORCED-YET? NIL)
              (FORCED-VALUE NIL))
```

```
                    (OBJECT NIL
                            ((FORCE SELF)
                             (COND ((NOT FORCED-YET?)
                                    (SET FORCED-VALUE (THUNK))
                                    (SET FORCED-YET? T)))
                             FORCED-VALUE))))
      (DEFINE-OPERATION (FORCE OBJ) OBJ)
```

(FORCE *delay*) $\longrightarrow$ *object*

Forces the computation of the *delay*'s value, If FORCE has not previously been applied to the delay, then the *expression* in the original DELAY-expression which created *delay* (see above) is evaluated, and its value is returned. If FORCE has previously been applied to the delay, then the value previously computed is returned.

If FORCE is applied to a non-delay, then it simply returns its argument.

Chapter 6

Side effects

A *side-effect* is a change in the state of a running **T** system, such as a change in the value stored in some location, or an effect on the outside world.

Many useful programs can be written which make no use of side-effects. In principle, side-effects other than those controlling the outside world are unnecessary. However, side effects, when used carefully, can be used to promote program modularity and readability. In addition, they may be necessary in a **T** implementation to obtain efficient program execution.

6.1 Assignment

T provides several special forms which perform assignment to *locations*. A location is a place in which a value may be stored, for example, a variable, or the car of a list, or a component of a structure. Locations are not objects *per se*, although references to them may be obtained through the use of *locatives*.

(SET *location new-value*) $\longrightarrow$ *undefined* — Special form

SET is a generalized assignment operator. It alters the location specified by *location* to hold *new-value*. The value of any SET-expression is the *new-value* thus stored.

If *location* is a symbol, then it names a variable which has been previously bound by a LAMBDA or an LSET (or indirectly by any form which does LAMBDA-binding, such as LET), whose value is to be altered.

```
(SET X (LIST 1 2 3))  ⟹  (1 2 3)
X                     ⟹  (1 2 3)
```

Location may also have the syntax of a call to an *access-type* procedure or operation such as CAR. (In this manual, these are marked as *Settable*.)

```
(SET (CAR X) 'A)  ⟹  A
X                 ⟹  (A 2 3)
```

(SET (*proc* arg_1 ... arg_n) *value*)
≡ ((SETTER *proc*) arg_1 ... arg_n value)

It is an error to use a SET form in a value requiring position. For example,

(SET (p x y ...) val)

is conceptually equivalent

```
(LAMBDA ()
       ((SETTER p) x y ... val)
       (RETURN))
```

where `(RETURN)` invokes the calling continuation with no arguments. For more information on `RETURN` see section 5.5.2.

In **T3.1** `SET` will continue to return the value being assigned to the location, but an error will be signalled in the future.

`(SETTER` *procedure*`)` $\longrightarrow$ *procedure* Operation

Given an access routine, `SETTER` returns a corresponding alteration routine. `(SETTER CAR)`, for example, evaluates to a primitive procedure that takes two arguments, a list whose car is to be changed, and a value to store in the car of the list.

See also `DEFINE-SETTABLE-OPERATION`, page 49.

`(SWAP` *location new-value*`)` $\longrightarrow$ *object* Special form

This behaves the same as `SET`, except that the value returned is the old value found in *location*, rather than the *new-value*. For example, the following expression exchanges the values of the variables `X` and `Y`:

```
(SET X (SWAP Y X))
```

`(EXCHANGE` *location1 location2*`)` $\longrightarrow$ *undefined* Special form

Exchanges the values in the two locations.

`(MODIFY` *location procedure*`)` $\longrightarrow$ *undefined* Special form

Stores a value in the *location* which is obtained by applying *procedure* to the old value in *location*.

`(MODIFY` *location* `1+)` $\equiv$ `(INCREMENT` *location*`)`

The value of the `MODIFY` special form is undefined, and it is an error to use a `MODIFY` form in a value requiring position. In **T3.1** `MODIFY` will continue to return a value.

`(MODIFY-LOCATION` *location continuation*`)` $\longrightarrow$ *object* Special form

Calls *continuation*, which should be a procedure of two arguments, passing it an access procedure of no arguments, which fetches the value in *location*; and an update procedure of one argument, which will store a new value in *location*.

`MODIFY-LOCATION`'s purpose is to ensure that any subforms of the form for *location* are evaluated only once. For example,

```
(INCREMENT (CAR (HACK))
           ≡
(MODIFY-LOCATION (CAR (HACK))
                 (LAMBDA (FETCH STORE) (STORE (+ (FETCH) 1))))
```

In this expression, `(HACK)` will be evaluated only once, whereas in the following naive expansion

```
(SET (CAR (HACK)) (+ (CAR (HACK)) 1))
```

it is evaluated twice, which is presumably neither necessary nor desirable. During the expansion of `MODIFY-LOCATION` forms extra temporary variables are introduced, as needed, to account for this.

`MODIFY-LOCATION` is useful in macro expansions, although awkward when written directly. Other "modification" forms such as `PUSH`, `MODIFY`, and `SWAP` are implementeded as macros in terms of the `MODIFY-LOCATION` special form.

For example, one might define a `DOUBLE` macro, which multiplies by two the value in a location, as follows:

```
(DEFINE-SYNTAX (DOUBLE FORM)
               ` (MODIFY-LOCATION
               ,FORM
               (LAMBDA (FETCH STORE) (STORE (* (FETCH) 2)))))
(DOUBLE (CAR (HACK)))
```

6.2 Locatives

Locatives in **T** are similar to the *pointers* or *references* of languages like C or Algol 68.

Locatives are obtained by evaluating `LOCATIVE`-expressions. The value in the location they refer to may be fetched using the `CONTENTS` operation, and stored using `SET`.

Objects which act as locatives may be created using `OBJECT`, as long as they handle the `CONTENTS` and `(SETTER CONTENTS)` operations appropriately, and handle `LOCATIVE?` by returning true.

`(LOCATIVE` *location*`)` ⟶ *locative* — Special form

Returns a locative which accesses the location specified by *location*. *Location* receives similar treatment to the *location* position in the `SET` special form; it may refer to a variable, or to a "field" within some mutable structure, for example the car of a pair.

The following equivalence approximates the behavior of `LOCATIVE`:

```
(LOCATIVE location)
          ≡
(OBJECT NIL
        ((CONTENTS SELF) location)
        (((SETTER CONTENTS) SELF VALUE) (SET location VALUE))
        ((LOCATIVE? SELF) T))
```

This is accurate if *location* is a variable, but if it is a call, the subforms of the call are evaluated when the `LOCATIVE`-expression is evaluated, not when the contents of the locative are required. For example,

```
(LOCATIVE (FOO (BAR BAZ)))
          ≡
(LET ((TEMP1 FOO)
      (TEMP2 (BAR BAZ)))
     (OBJECT NIL
             ((CONTENTS SELF) (TEMP1 TEMP2))
             (((SETTER CONTENTS) SELF VALUE) (SET (TEMP1 TEMP2) VALUE))
             ((LOCATIVE? SELF) T)))
```

`(CONTENTS` *locative*`)` ⟶ *object* — Settable operation

Dereferences (indirects through) the *locative*.

```
(LSET Z (LIST 'A 'B))                          ⟹  (A B)
(CONTENTS (LOCATIVE (CAR Z)))                  ⟹  A
(SET (CONTENTS (LOCATIVE (CAR Z))) 'C)         ⟹  C
Z                                              ⟹  (C B)
(CONTENTS (LOCATIVE location))                 ≡   location
(SET (CONTENTS (LOCATIVE location)) value      ≡   (SET location value)
```

`(LOCATIVE?` *object*`)` ⟶ *boolean* — Type predicate operation

Returns true if *object* is a locative.

6.3 Dynamic state

(BIND *specs* . *body*) ⟶ *value-of-body* Special form

BIND implements dynamic binding. Syntactically, it is similar to LET, but semantically it is completely different. BIND operates by doing temporary assignments to locations, for example, to bound variables. Unlike LET, it does not introduce a new lexical binding contour.

Note that the use of the term *bind* in this context is unrelated to its use elsewhere (for example, at the beginning of chapter 4.1).

Each *spec* should have the syntax

(*location value*)

The value in each *location* is obtained and saved. Each *location* is assigned the *value*, as with SET. The *body* expressions are evaluated and the value of the last is saved. The *locations* are then restored to their original saved values. The BIND-expression finally yields the saved value of the *body*.

(BIND ((*location value*)) . *body*)

is therefore similar to

```
(LET ((SAVED-VALUE location))
     (SET location value)
     (BLOCK0 (BLOCK . body)
             (SET location SAVED-VALUE)))
```

or

```
(LET ((SAVED-VALUE location))
     (SET location value)
     (UNWIND-PROTECT (BLOCK . body)
                     (SET location SAVED-VALUE)))
```

Examples:

```
(LSET A 1)
(DEFINE (F) (+ A 1))
(F) ⟹ 2
(LET ((A 10)) (F)) ⟹ 2
(BIND ((A 10)) (F)) ⟹ 11
(LSET X (LIST 1 2 3))
(BIND (((CAR X) 10))
(LSET Y (COPY-LIST X)))
X ⟹ (1 2 3)
Y ⟹ (10 2 3)
```

The values in the *locations* are restored, even if a throw out of *body* occurs (see CATCH, page 38). For example:

```
(LET ((A 10))
     (CATCH EP (BIND ((A 20)) (EP NIL)))
     A)
     ⟹ 10
```

Any *location* in a BIND-form which is a variable name should name a variable which has been previously bound using, for example, LSET or LET.

(UNWIND-PROTECT *form* . *unwind-forms*) ⟶ *value-of-form* Special form

UNWIND-PROTECT behaves nearly the same as BLOCK0: the *form* is evaluated, and the value is saved; the *unwind-forms* are evaluated; and the UNWIND-PROTECT-expression yields the (saved) value of *form*. However, unlike BLOCK0, UNWIND-PROTECT guarantees that the *unwind-forms* will be evaluated, even if a throw occurs during the evaluation of *form*.

In the following example, the motor will be stopped even if DRILL-HOLE attempts to throw out of the UNWIND-PROTECT.

```
(UNWIND-PROTECT (BLOCK (START-MOTOR)
                       (DRILL-HOLE))
                (STOP-MOTOR))
```

A throw (see page 38) out of the BLOCK - for example, as a result of calling DRILL-HOLE, will evaluate the exit form (STOP-MOTOR) before control finally returns to its corresponding CATCH.

It is an error to throw out of the unwind forms of an UNWIND-PROTECT.

Chapter 7

Operations

T provides a simple mechanism for programming in what has come to be known as the *object-oriented style* of programming. Object-oriented style may be contrasted to what will here be called *procedural style*. In object-oriented programming, programs are organized by giving the behavior of objects when various operations are applied to them, whereas in procedural style, programs are organized by giving the behavior of procedures when applied to various kinds of objects. These styles are equivalent in terms of computational power, but each offers different advantages in terms of readability and modularity.

T supports the object-oriented style with a special kind of procedure known as an *operation*, and with forms which permit one to create objects which exhibit certain behavior when a given operation is applied to them.

When an operation is called, the following sequence of events occurs:

> A *handler* is obtained for the object which was the operation's first argument. (Operations must always be called with at least one argument.)
>
> The operation asks the handler for a *method* which will handle the operation for the object.
>
> The handler either provides a method, or it indicates that the object is not prepared to handle the operation.
>
> If the handler provided a method, the method is invoked. If not, then the operation's *default method*, if any, is invoked. If the operation has no default method, then the effect of the call to the operation is undefined (and presumably an error condition is signalled).

In this way, an object's handler may determine how the operation is to be performed for the object - that is, which particular method is to be invoked as a result of invoking the operation.

Handlers map operations to methods. Many objects may have the same handler, or a handler may be idiosyncratic to a particular object. However, *every* object has a handler, so all objects participate in the generic operation dispatch protocol.

7.1 Fundamental forms

The basis of the generic operation system consists of the two special forms `OBJECT` and `OPERATION`. `OBJECT`-expressions create objects which respond appropriately to generic operations, and `OPERATION`-expressions evaluate to operations.

(`OBJECT` *procedure* . *method-clauses*) ⟶ *object* — Special form

> An `OBJECT`-expression yields an object which is prepared to handle generic operations according to the *method-clauses*. In the following description, "the object" refers to the value of a given `OBJECT`-expression.
>
> Each *method-clause* should be of the form
>
> ((*operation* . *variables*) . *body*)

Operation is an evaluated position, and is typically a variable which evaluates to an operation, although it may be any expression. When an operation is called with the object as its first argument, the *operation*-expressions are evaluated, and if one yields the operation being applied to the object, the corresponding *method-clause* is selected. The operation is then performed according to the selected *method-clause*: the clause's *variables* are bound to the arguments to the operation, and its *body*, an implicit block, is evaluated.

```
(DEFINE OP (OPERATION NIL))
(OP (OBJECT NIL ((OP SELF) 34)))      ⟹  34
(OP (OBJECT NIL ((OP SELF X) X)) 55)  ⟹  55
```

Procedure may be any expression, and is evaluated at the time the `OBJECT`-expression is evaluated. The object, when called, simply calls the value of the *procedure* expression, passing on any arguments. Typically *procedure* might be either a `LAMBDA`-expression, if the object is to be callable, or it is `NIL`, which by convention means that the object is not intended to be called, the value of `NIL` being an uncallable object.

In the degenerate case, where there are no method clauses, the value of

```
(OBJECT (LAMBDA args . body))
```

is indistinguishable from that of

```
(LAMBDA args . body).
```

The semantics of the `OBJECT` and `OPERATION` special forms can be described in terms of hypothetical primitive procedures `*OBJECT` and `GET-HANDLER`. These primitives do not actually exist in **T**, but are introduced here as expository aids. `*OBJECT` takes two arguments, and returns an object which, when called, calls the object which was `*OBJECT`'s first argument, and when given to `GET-HANDLER` returns the object which was `*OBJECT`'s second argument. That is, `*OBJECT` creates a two-component record (like a pair), `GET-HANDLER` extracts one component, and the other component is called when the record is called.

```
(GET-HANDLER (*OBJECT proc handler))  ≡  handler
((*OBJECT proc handler) arg ... )     ≡  (proc arg ... )
```

In addition, `GET-HANDLER` is defined on *all* objects to return some handler, even objects not created by `*OBJECT` (if indeed there are any such objects).

Given these primitives, the following rough equivalence holds:

```
(OBJECT proc
        ((op1 . args1) . body1)
        ((op2 . args2) . body2)
        ...
        ((opn . argsn) . bodyn))
        ≡
(*OBJECT proc
         (LAMBDA (OP) LNL (SELECT OP
                                  ((op1) (LAMBDA args1 . body1))
                                  ((op2) (LAMBDA args2 . body2))
                                  ...
                                  ((opn) (LAMBDA argsn . bodyn))
                                  (ELSE NIL))))
```

The outer `LAMBDA`-expression yields the object's handler; the inner `LAMBDA`-expressions yield the methods, and the mapping from operations to methods is accomplished by the `SELECT`-expression (see `SELECT`, page 34). Note that the syntactic positions op_1, op_2, ... op_n are evaluated positions, and the operation expressions are evaluated when an operation is applied to the object, not when the object is created.

`(OPERATION` *default* `.` *method-clauses*`)` ⟶ *operation* Special form

The syntax of OPERATION is the same as that of OBJECT, but its semantics and application are somewhat different. An OPERATION-expression evaluates to an operation. When called, the operation obtains a handler for its first argument, calls the handler to obtain a method, and then invokes the method. The default method for the operation is established as being *default*.

As the subject of another generic operation, an operation is an object like any other, and in this case the operation acts just as if it had been created by an OBJECT-expression with the same *method-clauses*. In this way one can establish the behavior of an operation when subject to other operations, for example SETTER.

The following rough equivalence describes the semantics of OPERATION. Some details have been omitted.

```
(OPERATION default . methods)
≡
(LABELS ((OP (OBJECT (LAMBDA (OBJ . ARGS)
                        (LET ((METHOD ((GET-HANDLER OBJ) OP)))
                             (COND (METHOD
                                    (APPLY METHOD OBJ ARGS))
                                   (ELSE
                                    (APPLY default OBJ ARGS)))))
                             . methods)))
                             OP)
```

For example:

```
(DEFINE OP (OPERATION (LAMBDA (OBJ) 'ZEBU)))
(OP (OBJECT NIL ((OP SELF) 'QUAGGA)))          ⟹   QUAGGA
(OP 'ELAND)                                    ⟹   ZEBU
```

An operation is created, and the variable OP is bound to it. The operation's default method always returns the symbol ZEBU. When the operation is applied to the value of the OBJECT-expression, the appropriate method is invoked, and the call to the operation yields the symbol QUAGGA. When the operation is applied to an object which doesn't handle it - the symbol ELAND - the operation's default method is invoked, so the call yields the symbol ZEBU.

(OPERATION? *object*) ⟶ *boolean* Type predicate

Returns true if *object* is an operation.

7.2 Defining operations

(DEFINE-OPERATION (*variable* . *argument-vars*) . *body*) ⟶ *undefined* Special form

Defines *variable* to be an operation. The syntax is intended to be analogous to that of DEFINE. The operation's default method is defined by *argument-vars* and *body*. If there is no *body*, then the operation's default method is undefined. In this case, the *argument-vars* appear only for documentary purposes.

```
(DEFINE-OPERATION (var . args) . body)
≡ (DEFINE var (OPERATION (LAMBDA args . body)))
(DEFINE-OPERATION (var . args))
≡ (DEFINE var (OPERATION UNDEFINED-EFFECT))
```

(DEFINE-SETTABLE-OPERATION (*variable* . *argument-vars*) . *body*) ⟶ *undefined* Special form

Defines *variable* to be an operation, as with DEFINE-OPERATION, but arranges for the operation's "setter" to be another operation, so that the operation is "settable" (see page 41).

(DEFINE-SETTABLE-OPERATION (*var* . *args*) . *body*)
≡
(DEFINE *var*
        (LET ((THE-SETTER (OPERATION UNDEFINED-EFFECT)))
             (OPERATION (LAMBDA *args* . *body*)
                        ((SETTER SELF) THE-SETTER))))

(DEFINE-PREDICATE *variable*) ⟶ *undefined* Special form

Defines *variable* to be an operation which, by default, returns false.

(DEFINE-PREDICATE *var*)
≡
(DEFINE-OPERATION (*var* OBJ) NIL)

The intent is that particular OBJECT-expressions contain clauses of the form ((*variable* SELF) T). This way the operation defined by DEFINE-PREDICATE may act as a type predicate that returns true only for those objects returned by such OBJECT-expressions.

7.3 Example

Hypothetical implementation of CONS:

```
(DEFINE-PREDICATE PAIR?)
(DEFINE-SETTABLE-OPERATION (CAR PAIR))
(DEFINE-SETTABLE-OPERATION (CDR PAIR))
(DEFINE (CONS THE-CAR THE-CDR)
        (OBJECT NIL
                ((PAIR? SELF) T)
                ((CAR SELF) THE-CAR)
                ((CDR SELF) THE-CDR)
                (((SETTER CAR) SELF NEW-CAR) (SET THE-CAR NEW-CAR))
                (((SETTER CDR) SELF NEW-CDR) (SET THE-CDR NEW-CDR))))
```

Chapter 8

Numbers

Numbers in **T** are objects which represent real numbers. Numbers come in three varieties: integers, ratios, and floating point numbers. Integers and ratios are exact models of the corresponding mathematical objects. Floating point numbers give discrete approximations to real numbers within a certain implementation-dependent range.

As expressions, numbers are self-evaluating literals (see section 3.3).

```
-377              ⟹  -377
72723460248141    ⟹  72723460248141
13/5              ⟹  13/5
34.55             ⟹  34.55
```

There may be many different objects which all represent the same number. The effect of this is that `EQ?` may behave in nonintuitive ways when applied to numbers. It is guaranteed that if two numbers have different types or different numerical values, then they are *not* `EQ?`, but two numbers that appear to be the same may or may not be `EQ?`, even though there is no other way to distinguish them. Use `EQUAL?` or `=` (page 56) for comparing numbers.

8.1 Predicates

(`NUMBER?` *object*) $\longrightarrow$ *boolean* — Type predicate

Returns true if *object* is a number.

(`INTEGER?` *object*) $\longrightarrow$ *boolean* — Type predicate

Returns true if *object* is an integer.

(`FLOAT?` *object*) $\longrightarrow$ *boolean* — Type predicate

Returns true if *object* is a floating point number.

(`RATIONAL? obj`) $\longrightarrow$ *boolean*

`RATIONAL?` returns true if *obj* is an integer or ratio; otherwise, it returns false.

(`RATIO?` *object*) $\longrightarrow$ *boolean* — Type predicate

Returns true if *object* is a ratio.

(ODD? *integer*) $\longrightarrow$ *boolean*

Returns true if *integer* is odd.

(EVEN? *integer*) $\longrightarrow$ *boolean*

Returns true if *integer* is even.

8.2 Arithmetic

(+ . *numbers*) $\longrightarrow$ *number*

(ADD . *numbers*) $\longrightarrow$ *number*

Returns the sum of the *numbers*.

```
(+ 10 27)      ⟹  37
(+ 10 27.8)    ⟹  37.8
(+)            ⟹  0
(+ -35)        ⟹  -35
(+ 1 2 3)      ⟹  6
```

(- *n1 n2*) $\longrightarrow$ *number*

(SUBTRACT *n1 n2*) $\longrightarrow$ *number*

Returns the difference of *n1* and *n2*.

(- *n*) $\longrightarrow$ *number*

(NEGATE *n*) $\longrightarrow$ *number*

Returns the additive inverse of *n*.

(* . *numbers*) $\longrightarrow$ *number*

(MULTIPLY . *numbers*) $\longrightarrow$ *number*

Returns the product of the *numbers*.

(/ *n1 n2*) $\longrightarrow$ *number*

(DIVIDE *n1 n2*) $\longrightarrow$ *number*

Returns the quotient of *n1* and *n2*.

```
(/ 20 5)     ⟹  4
(/ 5 20)     ⟹  1/4
(/ 5.0 20)   ⟹  0.25
```

(QUOTIENT *n1 n2*) $\longrightarrow$ *integer*

Integer division; truncates the quotient of *n1* and *n2* towards zero.

(REMAINDER *n1 n2*) $\longrightarrow$ *number*

Returns the remainder of the division of *n1* by *n2*, with the sign of *n1*.

`(QUOTIENT&REMAINDER` *n1 n2*`)` ⟶ *integer1 integer2*

Returns the quotient and remainder (respectively) as muiltiple values of dividing *n1* by *n2*.

`(FLOOR` *n1 n2*`)` ⟶ *number*

Returns the greatest multiple of *n2* which is less than or equal to *n1*; "round down". A particularly useful case is where *n2* is 1, in which case `FLOOR` returns *n1*'s integer part.

```
(FLOOR 21 8)     ⟹  16
(FLOOR -21 8)    ⟹  -24
(FLOOR 2.67 1)   ⟹  2
```

`(CEILING` *n1 n2*`)` ⟶ *number*

Returns the least multiple of *n2* which is greater than or equal to *n1*; "round up".

```
(CEILING 21 8)     ⟹  24
(CEILING -21 8)    ⟹  16
(CEILING 2.67 1)   ⟹  3
```

`(TRUNCATE` *n1 n2*`)` ⟶ *number*

Returns the greatest multiple of *n2* which is less than or equal to the absolute value of *n1*, with *n1*'s sign; "round towards zero".

```
(TRUNCATE 21 8)     ⟹  16
(TRUNCATE -21 8)    ⟹  -24
(TRUNCATE 2.67 1)   ⟹  1
```

`(ROUND` *n1 n2*`)` ⟶ *number*

Rounds *n1* toward the nearest multiple of *n2*.

`(MOD` *n1 n2*`)` ⟶ *number*

Returns a number k in the range from zero, inclusive, to *n2*, exclusive, such that k differs from *n1* by an integral multiple of *n2*. If *n1* is positive, this behaves like `REMAINDER`.

`(EXPT` *base exponent*`)` ⟶ *number*

Returns *base* raised to the *exponent* power. *Base* and *exponent* may be any numbers.

> **Bug:** In **T** 3.1, *exponent* must be a finxnum.

`(ABS` *n*`)` ⟶ *number*

Returns the absolute value of *n*.

`(GCD` *integer1 integer2*`)` ⟶ *integer*

Returns the greatest common divisor of *integer1* and *integer2*.

```
(GCD 36 60) ⟹ 12
```

(`truncate` *number*) $\longrightarrow$ *integer*

`truncate` returns the integer of maximal absolute value not larger than the absolute value of *number* with the same sign as *number*. `truncate` truncates its argument toward zero.

(`ADD1` *n*) $\longrightarrow$ *number*

(`1+` *n*) $\longrightarrow$ *number*

Returns n + 1. (See also `INCREMENT`, page 59.)

(`SUBTRACT1` *n*) $\longrightarrow$ *number*

Returns n - 1. (See also `DECREMENT`, page 59.)

(`MIN .` *numbers*) $\longrightarrow$ *number*

Returns the *number* with the least numerical magnitude. (There must be at least one *number*.)

(`MAX .` *numbers*) $\longrightarrow$ *number*

Returns the *number* with the greatest numerical magnitude. (There must be at least one *number*.)

8.3 Comparison

(`=` *n1 n2*) $\longrightarrow$ *boolean*

(`EQUAL?` *n1 n2*) $\longrightarrow$ *boolean*

Returns true if *n1* is numerically equal to *n2*.

(`<` *n1 n2*) $\longrightarrow$ *boolean*

(`LESS?` *n1 n2*) $\longrightarrow$ *boolean*

Returns true if *n1* is numerically less than *n2*.

(`>` *n1 n2*) $\longrightarrow$ *boolean*

(`GREATER?` *n1 n2*) $\longrightarrow$ *boolean*

Returns true if *n1* is numerically greater than *n2*.

(`N=` *n1 n2*) $\longrightarrow$ *boolean*

(`NOT-EQUAL?` *n1 n2*) $\longrightarrow$ *boolean*

Returns true if *n1* is not numerically equal to *n2*.

(`>=` *n1 n2*) $\longrightarrow$ *boolean*

(`NOT-LESS?` *n1 n2*) $\longrightarrow$ *boolean*

Returns true if *n1* is not numerically less than *n2*.

(`<=` *n1 n2*) $\longrightarrow$ *boolean*

Returns true if *n1* is not numerically greater than *n2*.

8.4 Sign predicates

(=0? *n*) ⟶ *boolean*

(ZERO? *n*) ⟶ *boolean*

Returns true if *n* is numerically equal to zero.

(<0? *n*) ⟶ *boolean*

(NEGATIVE? *n*) ⟶ *boolean*

Returns true if *n* is negative.

(>0? *n*) ⟶ *boolean*

(POSITIVE? *n*) ⟶ *boolean*

Returns true if *n* is positive.

(N=0? *n*) ⟶ *boolean*

(NOT-ZERO? *n*) ⟶ *boolean*

Returns true if *n* is not numerically equal to zero.

(>=0? *n*) ⟶ *boolean*

(NOT-NEGATIVE? *n*) ⟶ *boolean*

Returns true if *n* is non-negative.

(<=0? *n*) ⟶ *boolean*

(NOT-POSITIVE? *n*) ⟶ *boolean*

Returns true if *n* is non-positive.

8.5 Transcendental functions

(EXP *float*) ⟶ *float*

Exponential function (e^x).

(LOG *float*) ⟶ *float*

Natural logarithm ($\log_{\mathrm{e}} x$).

(SQRT *float*) ⟶ *float*

Returns the square root of its argument.

(COS *float*) ⟶ *float*

Returns the cosine of its argument.

(SIN *float*) $\longrightarrow$ *float*

Returns the sine of its argument.

(TAN *float*) $\longrightarrow$ *float*

Returns the tangent of its argument.

(ACOS *float1*) $\longrightarrow$ *float2*

Arccosine. Returns a number whose cosine is *float1*.

(ASIN *float1*) $\longrightarrow$ *float2*

Arcsine. Returns a number whose sine is *float1*.

(ATAN2 x y) $\longrightarrow$ *float*

Two-argument arctangent. This computes the angle between the positive x-axis and the point (x, y). For example, if (x, y) is in the first or third quadrant, (ATAN2 x y) returns the arctangent of y/x.

Bug: ATAN2 isn't implemented in **T** 3.1. **T** 3.1 does implement ATAN (arctangent), however.

8.6 Bitwise logical operators

Bug: In **T** 3.1, the routines in this section are restricted to take fixnums (see page 138), not arbitrary integers.

(LOGAND *integer1 integer2*) $\longrightarrow$ *integer*

Returns the bitwise logical *and* of its arguments.

(LOGAND 10 12) $\Longrightarrow$ 8

(LOGIOR *integer1 integer2*) $\longrightarrow$ *integer*

Returns the bitwise logical inclusive *or* of its arguments.

(LOGXOR *integer1 integer2*) $\longrightarrow$ *integer*

Returns the bitwise logical exclusive *or* of its arguments.

(LOGNOT *integer*) $\longrightarrow$ *integer*

Returns the bitwise logical *not* of its argument.

(ASH *integer amount*) $\longrightarrow$ *integer*

Shifts *integer* to the left *amount* bit positions. If *amount* is negative, shifts *integer* right by the absolute value of *amount* bit positions.

(BIT-FIELD *integer position size*) $\longrightarrow$ *integer*

Extracts a bit field out of *integer*. The field extracted begins at bit position *position* from the low-order bit of the integer and consists of *size* bits. The extracted field is returned as a positive integer.

```
(BIT-FIELD 27 1 3)  ⟹  5
(BIT-FIELD -1 4 5)  ⟹  31
```

(SET-BIT-FIELD *integer position size value*) ⟶ *integer*

Inserts a value into a bit field of *integer*. The field begins at bit position *position* from the low-order bit of the integer and consists of *size* bits. A new integer is returned which is the same as the argument *integer* except that this field has been altered to be *value*.

```
(SET-BIT-FIELD 32 1 3 5) ⟹ 42
```

8.7 Coercion

(->INTEGER *number*) ⟶ *integer*

Coerces *number* to an integer, truncating towards zero if necessary.

```
(->INTEGER 17)     ⟹  17
(->INTEGER 7/4)    ⟹  1
(->INTEGER 17.6)   ⟹  17
(->INTEGER -17.6)  ⟹  -17
```

(->FLOAT *number*) ⟶ *float*

Coerces *number* to a floating point number.

```
(->FLOAT 17)    ⟹  17.0
(->FLOAT 7/4)   ⟹  1.75
(->FLOAT 17.6)  ⟹  17.6
```

8.8 Assignment

INCREMENT and DECREMENT are assignment forms, like SET and PUSH. See section 6.1 for a general discussion.

(INCREMENT *location*) ⟶ *number*

Adds one to the value in *location*, stores the sum back into *location*, and yields the sum.

```
(LSET L 6)
(INCREMENT L)             ⟹  7
L                         ⟹  7
(LSET M (LIST 10 20 30))  ⟹  (10 20 30)
(INCREMENT (CAR M))       ⟹  11
M                         ⟹  (11 20 30)
(INCREMENT location)      ≡   (MODIFY location ADD1)
```

(DECREMENT *location*) ⟶ *number*

Subtracts one from the value in *location*, stores the difference back into *location*, and yields the difference.

```
(DECREMENT location) ≡ (MODIFY location SUBTRACT1)
```

Chapter 9

Lists

This section describes routines available for creating, examining, and otherwise manipulating list structure.

9.1 Predicates

(NULL? *object*) $\longrightarrow$ *boolean* Type predicate

Returns true if *object* is the empty list (null). Since the empty list is also used for the logical false value, and non-false objects are considered to be true, it turns out that NULL? is the same as NOT.

(PAIR? *object*) $\longrightarrow$ *boolean* Type predicate

Returns true if *object* is a pair.

```
(PAIR? '(A B C))    ⟹  true
(PAIR? '(A . 15))   ⟹  true
(PAIR? '())         ⟹  false
(PAIR? 'FOO)        ⟹  false
```

(ATOM? *object*) $\longrightarrow$ *boolean* Type predicate

Returns true if *object* is an atom. An atom is any object that isn't a pair.

(ATOM? *object*) $\equiv$ (NOT (PAIR? *object*))

(LIST? *object*) $\longrightarrow$ *boolean* Type predicate

Returns true if *object* is either a pair or the empty list.

```
(LIST? '(A B C))    ⟹  true
(LIST? '(A . 15))   ⟹  true
(LIST? '())         ⟹  true
(LIST? 'FOO)        ⟹  false
```

(PROPER-LIST? *object*) $\longrightarrow$ *boolean* Type predicate

Returns true if *list* is a properly formed list, i.e., if its last pair's cdr is null.

```
(PROPER-LIST? '(A B C))     ⟹  true
(PROPER-LIST? '(A B . C))   ⟹  false
(PROPER-LIST? '())          ⟹  true
```

(NULL-LIST? *list*) $\longrightarrow$ *boolean*

Like NULL?, but has an undefined effect (and may, for example, signal an error) if its argument is not either a pair or null. (NULL-LIST? *x*) is roughly the same as (NULL? (ENFORCE LIST? *x*)).

9.2 Constructors

(CONS *object1 object2*) $\longrightarrow$ *pair*

Returns a new pair whose car is *object1* and whose cdr is *object2*.

```
(CONS 'A '())               ⟹  (A)
(CONS 'A 'B)                ⟹  (A . B)
(CONS 'A '(B))              ⟹  (A B)
(CONS 'A (CONS 'B '()))     ⟹  (A B)
(CONS 'A (LIST 'B 'C))      ⟹  (A B C)
```

(LIST . *objects*) $\longrightarrow$ *list*

Returns a new list of its arguments.

```
(LIST)                      ⟹  ()
(LIST 'A)                   ⟹  (A)
(LIST 'A 'B)                ⟹  (A B)
(LIST 'A (LIST 'B 'C))      ⟹  (A (B C))
LIST                        ≡   (LAMBDA THINGS THINGS)
```

(CONS* . *objects*) $\longrightarrow$ *pair*

CONS* is a generalized CONS. It returns a new, possibly improper, list, by consing the initial *objects* onto the last *object*.

```
(CONS* 1 2 3)               ⟹  (1 2 . 3)
(CONS* 'A 'B '(C D))        ⟹  (A B C D)
(CONS* object)              ≡   object
(CONS* object1 object2)     ≡   (CONS object1 object2)
```

(COPY-LIST *list*) $\longrightarrow$ *list*

Makes a “top-level” copy of a list; that is, returns a new list whose elements are the same as the original.

```
(COPY-LIST list) ≡
           (COND ((NULL-LIST? list) '())
                 (ELSE (CONS (CAR list) (COPY-LIST (CDR list)))))
(COPY-LIST list) ≡ (MAP IDENTITY list)
```

9.3 List access

(CAR *pair*) $\longrightarrow$ *object* Settable

Returns the car of the *pair*. (CAR () $\longrightarrow$ *error*)', though, for compatibility, in **T3.1** it will evaluate to ().

(CDR *pair*) $\longrightarrow$ *object* Settable

Returns the cdr of the *pair*. `(CDR () ⟶` *error*`)`', though, for compatibility, in **T3.1** it will evaluate to `()`.

`(C...R` *pair*`)` ⟶ *object* Settable

Compositions of `CAR` and `CDR`, up to four deep, are defined.

```
(CADAR x)    ≡  (CAR (CDR (CAR x)))
(CADDDR x)   ≡  (CAR (CDR (CDR (CDR x))))
```

`(NTH` *list n*`)` ⟶ *object* Settable

Returns the *nth* element of *list*. The first element (the car) is `(NTH` *list* `0)`, the second element is `(NTH` *list* `1)`, and so on. In general, *list* must have at least n+1 elements.

```
(NTH '(A B) 0)  ⟹  A
(NTH '(A B) 1)  ⟹  B
(NTH '(A B) 2)      has an undefined effect
(NTH '() n)         has an undefined effect for any n
```

`(NTHCDR` *list n*`)` ⟶ *list* Settable

Returns the *nth* tail of *list*. In general, *list* must have at least n tails.

```
(NTHCDR '(A B) 0)  ⟹  (A B)
(NTHCDR '(A B) 1)  ⟹  (B)
(NTHCDR '(A B) 2)  ⟹  ()
(NTHCDR '(A B) 3)      has an undefined effect
(NTHCDR '() n)         has an undefined effect for any nonzero n
```

Bug: `NTHCDR` doesn't handle `SETTER` (i.e. is not settable) in **T** 3.1.

`(LAST` *list*`)` ⟶ *object* Settable

Returns the last element of *list*.

```
(LAST '(A B C))  ⟹  C
(LAST list)       ≡   (CAR (LASTCDR list))
```

`(LASTCDR` *list*`)` ⟶ *pair* Settable

Returns the last pair in *list*.

```
(LASTCDR '(A B C))    ⟹  (C)
(LASTCDR '(A B . C))  ⟹  (B . C)
```

Bug: `LASTCDR` doesn't handle `SETTER` (i.e. is not settable) in **T** 3.1.

9.4 Lists as sequences

`(LENGTH` *list*`)` ⟶ *integer*

Return the length of *list*.

```
(LENGTH '())       ⟹  0
(LENGTH '(A B C))  ⟹  3
```

`(APPEND . ` *lists*`)` ⟶ *list*

Constructs a list which is the concatenation of *lists*. A new list is constructed, except that the result has the last non-null argument as a tail.

```
(APPEND '(A B C) '(D E) '(F G)) ⟹ (A B C D E F G)
```

`(APPEND! . ` *lists*`)` ⟶ *list*

Destructive version of `APPEND`. Splices the lists together, setting the cdr of the last pair in the first list to the second list, and so on. Returns its first non-null argument.

```
(DEFINE L1 (LIST 'A 'B 'C))
(DEFINE L2 (LIST 'D 'E))
L1                          ⟹  (A B C)
(APPEND! L1 L2)             ⟹  (A B C D E)
L1                          ⟹  (A B C D E)
(APPEND! '() L2)            ⟹  (D E)
```

`(REVERSE ` *list*`)` ⟶ *list*

Returns a new list whose elements are those of *list*, in reverse order.

```
(REVERSE '(A B C)) ⟹ (C B A)
```

`(REVERSE! ` *list*`)` ⟶ *list*

This is a "destructive" version of `REVERSE`; it allocates no storage for the result, but rather recycles the pairs in the source *list*'s to form the result list.

`(SUBLIST ` *list start count*`)` ⟶ *list*

Returns a list of *count* elements of *list*, beginning with element *start*.

```
(SUBLIST '(A B C D E F) 2 3) ⟹ (C D E)
```

9.5 Lists as sets

`(MEMQ? ` *object list*`)` ⟶ *boolean*

Returns true if *object* is an element of *list*.

```
(MEMQ? 'B '(A B C))        ⟹  true
(MEMQ? 'B '(A (B C) D))    ⟹  false
(MEMQ? 'B '(B))            ⟹  true
(MEMQ? 'B '())             ⟹  false
(MEMQ? 17 '(10 17))        ⟹  undefined
(MEMQ? foo '(foo bar))     ⟹  undefined
(LET ((X foo))
(MEMQ? X (LIST 'B X)))     ⟹  true
```

`(MEM? ` *predicate object list*`)` ⟶ *boolean*

Returns true if *list* contains an object which is equal to *object* according to *predicate*, which should be an equality predicate.

```
(MEM? = 17 '(10 17))                  ⟹  true
(MEM? STRING-EQUAL? foo '(foo bar))   ⟹  true
```

(ANY? *predicate* . *lists*) ⟶ *boolean*

Returns true if any element of *list* answers true to *predicate*.

```
(ANY? NUMBER? '(FOO 15 BAR)) ⟹ true
```

(EVERY? *predicate* . *lists*) ⟶ *boolean*

Returns true if every element of *list* answers true to *predicate*.

```
(EVERY? SYMBOL? '(A B C)) ⟹ true
```

(DELQ *object list*) ⟶ *list*

Returns a list which is the same as the argument *list* with all occurrences of *object* removed.

(DEL *predicate object list*) ⟶ *list*

Returns a list which is the same as *list* except that all elements which, according to *predicate*, are equal to *object*, have been removed. *Predicate* should be an equality predicate.

(DELQ *object list*) ≡ (DEL EQ? *object list*)

(DELQ! *object list*) ⟶ *list*

Destructive version of DELQ. Removes all occurrences of *object* from *list*, possibly altering it, and returns the new and/or modified list.

(DEL! *predicate object list*) ⟶ *list*

Destructive version of DEL.

(DELQ! *object list*) ≡ (DEL! EQ? *object list*)

9.6 Mapping Procedures

(MAP *proc* . *lists*) ⟶ *list*

Returns a list of the results of applying *proc* to successive elements of the *lists*; the *nth* element of the result list is the result of calling *proc* on the *nth* elements of the *lists*. The length of the result list is the same as the length of the shortest of the *lists*.

```
(MAP (LAMBDA (X) (LIST 'A X)) '(CAT DOG))  ⟹  ((A CAT) (A DOG))
(MAP + '(10 20) '(5 6))                     ⟹  (15 26)
(MAP LIST '(A B) '(C D E) '(F))             ⟹  ((A C F))
```

(MAPCDR *proc* . *lists*) ⟶ *list*

MAPCDR is similar to MAP, except that *proc* is applied to successive *tails* of the *lists*.

(MAP! *proc list*) ⟶ *list*

For each pair in *list*, applies *proc* to the car of the pair, and then modifies the pair so that its car is the value of the application.

```
(DEFINE L (LIST 'CAT 'DOG))          ⟹  (CAT DOG)
(MAP! (LAMBDA (X) (LIST 'A X)) L)
L                                    ⟹  ((A CAT) (A DOG))
```

(WALK *proc* . *lists*) ⟶ *undefined*

This is similar to MAP but the result is a value of no particular interest. Thus WALK is useful only for the side-effects that *proc* performs.

(WALKCDR *proc* . *lists*) ⟶ *undefined*

This is similar to MAPCDR but the result is a value of no particular interest.

9.7 Lists as associations

(ASSQ *object list*) ⟶ *pair* or *false*

(ASSQ *object list*) ≡ (ASS EQ? *object list*)

(ASS *predicate object list*) ⟶ *pair* or *false*

An *association list* is a list of pairs that represents a "lookup table" where the car of each pair matches a lookup key. ASS and its related procedures search association lists by comparing (applying *predicate*) the key (*object*) against the car of each element of *list* (i.e., the caar of each tail). If it finds a match, it returns that element (the car of the tail). If it finds no match, it returns false.

```
(ASS EQ? 'B '((A 10 20) (B 30 40) (C 50 60)))  ⟹  (B 30 40)
(ASS = 9 '((1 Y) (4 Y) (9 Z) (12 P)))          ⟹  (9 Z)
(ASS = 10 '((1 Y) (4 Y) (9 Z) (12 P)))         ⟹  false
```

9.8 Lists as stacks

PUSH and POP permit a convenient syntax for the use of lists in implementing simple stacks. They are assignment forms, like SET and INCREMENT. See section 6.1 for a general discussion of assignment forms.

(PUSH *location object*) ⟶ *undefined* Special form

Cons a pair whose car is *object* and whose cdr is the value in *location* (usually a list), and store the extended list back into *location*.

```
(LSET L '(34 55))   ⟹  (34 55)
(PUSH L 21)
L                   ⟹  (21 34 55)
(PUSH L '(A B C))
(PUSH (CAR L) 'D)
L                   ⟹  ((D A B C) 21 34 55)
```

In general,

(PUSH *location object*)
  ≡ (SET *location* (CONS *object location*))
  ≡ (MODIFY *location* (LAMBDA (L) (CONS *object* L)))

`(POP` *location*`)` $\longrightarrow$ *object* Special form

The car of the value in *location* (this value should be a pair) is returned. As a side-effect, the cdr of the list is stored back in the location.

```
L        ⟹  (21 34 55)
(POP L)  ⟹  21
L        ⟹  (34 55)
```

In general,

```
(POP location) ≡
     (BLOCK0 (CAR location) (SET location (CDR location)))
                            ≡
     (BLOCK0 (CAR location) (MODIFY location CDR))
```

Chapter 10

Trees

This chapter describes procedures available for manipulating trees. The term *tree* is used in a technical sense to refer to non-circular list structure considered as tree data-structures, as contrasted with *lists*, which use chained pairs to represent a one-dimensional sequence of objects. Thus one might say either that `(A (B C) D)` is a list of three elements, or that it is a tree with four (non-null) leaves.

Trees are used to represent programs. The tree-manipulation procedures and special forms are designed with this in mind.

10.1 Comparison

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

> `EQUIV?` is an equality predicate, not for comparing trees but for comparing leaves.
>
> > If `(EQ?` *object1 object2*`)`, then `(EQUIV?` *object1 object2*`)`.
> >
> > If *object1* and *object2* are both numbers of the same type, then they are `EQUIV?` iff they have the same numeric value (see `=`, page 56).
> >
> > If *object1* and *object2* are both strings, then they are `EQUIV?` if they are `STRING-EQUAL?` (page 79).
>
> See also the descriptions of `EQ?` (page 23) and `=` (page 56).
>
> > **Bug:** In **T** 3.1, `EQUIV?` (and `ALIKEV?`) may return false for two numbers which are `=`. It will do the right thing, however, for FIXNUMs.

`(ALIKE?` *predicate tree1 tree2*`)` ⟶ *boolean*

> Returns true if *tree1* and *tree2* have the same shape and their corresponding leaves are equal according to *predicate*, which must be an equality predicate.
>
> > `(ALIKE? = '(1 (2 . 8) 3) '(1 (2 . 8) 3))` ⟹ *true*

`(ALIKEQ?` *tree1 tree2*`)` ⟶ *boolean*

> > `(ALIKEQ?` *tree1 tree2*`)` ≡ `(ALIKE? EQ?` *tree1 tree2*`)`

`(ALIKEV?` *tree1 tree2*`)` ⟶ *boolean*

> > `(ALIKEV?` *tree1 tree2*`)` ≡ `(ALIKE? EQUIV?` *tree1 tree2*`)`
>
> Roughly speaking, two trees are `ALIKEV?` if they print the same way.

10.2 Tree utilities

(SUBST *predicate new old tree*) ⟶ *tree*

Returns a result tree which is the same as the argument *tree*, except that leaves in the argument tree which, according to *predicate*, are equal to *old*, have been changed to be *new*. *Predicate* must be an equality predicate.

`(SUBST EQUIV? 17 13 '(10 (20 13) 30))` ⟹ `(10 (20 17) 30)`

The result returned by SUBST may or may not share structure with its tree argument.

(SUBSTQ *new old tree*) ⟶ *tree*

(SUBSTQ *new old tree*) ≡ (SUBST EQ? *new old tree*)

(SUBSTV *new old tree*) ⟶ *tree*

(SUBSTV *new old tree*) ≡ (SUBST EQUIV? *new old tree*)

(COPY-TREE *tree*) ⟶ *tree*

Recursively make a copy of the *tree*.

(COPY-TREE *tree*) ≡ (SUBSTQ NIL NIL *tree*)

(TREE-HASH *tree*) ⟶ *integer*

Compute a hash code for *tree*. The hash computed is a non-negative fixnum with the property that if *tree1* and *tree2* are ALIKEV?, then their hashes are the same.

10.3 Destructuring

(DESTRUCTURE *specs . body*) ⟶ *value-of-body* Special form

Specs has the form ((*pattern value*) (*pattern value*) ...) where each *pattern* is either an identifier (e.g., X) or a *tree* (see page 69), all of whose non-null leaves are identifiers (e.g., ((X Y . Z) P Q)).

DESTRUCTURE is similar to LET, and in the case that all the *patterns* are symbols, DESTRUCTURE and LET are equivalent. But DESTRUCTURE is especially useful when one wants to bind variables to the various nodes in a single tree structure. For example, suppose Z has the value (1 (2 3) ((4 5 . 6) 7)), and we want to bind A to 1, B to 2, C to 3, D to 6, and E to 7. We could write

```
(LET ((A (CAR Z))
      (B (CAADR Z))
      (C (CDADR Z))
      (D (CDDAR (CADDR Z)))
      (E (CADADR (CDR Z))))
     ... )
```

However, we could also write

```
(DESTRUCTURE (((A (B . C) ((() () . D) E)) Z))
... )
```

The ()'s notate ignored positions.

(DESTRUCTURE* *specs . body*) ⟶ *value-of-body* Special form

DESTRUCTURE* is the "serial" form of DESTRUCTURE, just as LET* is the "serial" form of LET.

```
(DESTRUCTURE* ((pattern1 value1)
               (pattern2 value2)
               ...
               (patternn valuen))
              code)
              ≡
(DESTRUCTURE ((pattern1 value1))
 (DESTRUCTURE ((pattern1 value1))
  ...
     (DESTRUCTURE ((patternn valuen))
                  code )... ))
```

10.4 Quasiquote

T's *quasiquote* facility, inherited from the Maclisp family of Lisps, is useful for building programs whose form is determined, but where some of the structure is constant and some is to be filled in. This is especially convenient in the definitions of macro expanders. It is so useful that a special external syntax is provided for notating such templates. It takes its name from the name of the character (quasiquote, ‘) which introduces this special syntax.

The character ("‘ ") is called *quasiquote* because it behaves like QUOTE except that one can specify that subforms of the quasiquote'd form should be evaluated. Inside a form that is preceded by a quasiquote, two additional pieces of external syntax are active: comma (,), and comma at-sign (,@); these are the ways to indicate that some subform should be evaluated (i.e., unquoted). A subform preceded by a comma is evaluated and replaced by the result of the evaluation. A subform preceded by a comma at-sign is evaluated and replaced by *splicing in* the result of the evaluation.

Here are some simple examples. Note that the first example shows code equivalence, not evaluation.

```
‘ (A B ,X Y)          ≡    (LIST 'A 'B X 'Y)
(DEFINE X '(3))
(CDR ‘ (A B ,X C))    ⟹   (B (3) C)
(CDR ‘ (A B ,@X C))   ⟹   (B 3 C)
```

```
(DEFINE-SYNTAX (REPEAT N . CODE) ;Execute CODE N times
              ‘ (LET ((COUNT ,N) (THUNK (LAMBDA () ,@CODE)))
                     (DO ((COUNT COUNT (-1+ COUNT)))
                         ((<=0?  COUNT) '())
                         (THUNK))))
```

It is possible to nest quasiquoted forms. This is useful when writing complex source-to-source transformations.

```
(DEFINE-SYNTAX (FOO X Y)
              ‘ (LET ((STUFF ,X))
                     ‘ ((STUFF-IS ,STUFF)
                     (Y-IS ,',Y))))
(FOO (+ 3 5) BAZ) ⟹ ((STUFF-IS 8) (Y-IS BAZ))
```

quasiquote now works on vectors. Thus,

```
‘ #(1 2 ,(+ 1 2)) ⟹ #(1 2 3)
```

Chapter 11

Structures

A structure corresponds to what is called a record in some other languages: a data structure with a fixed set of named fields or slots, known in **T** as *components*. For example, one might want an "employee" structure, with slots for the employee's name, age, and salary. A structure, then, is an organized set of *locations*, places to store things.

11.1 Terminology

Every structure has an associated an object called a *structure type* (sometimes abbreviated *stype*, pronounced "ess type"). Structure types are created by the procedure `MAKE-STYPE` (see below) and by the special form `DEFINE-STRUCTURE-TYPE`

Given a structure type S, one can access several things of interest:

Constructor a procedure that will create uninitialized structures of type S (e.g., the structure for a particular employee).

Predicator a procedure (a type predicate) that returns true if its argument is a structure of type S.

Component selectors procedures that access the various components of structures of type S (one selector per component).

Handler the mechanism for specifying how a set of generic operations will behave when applied to structures of type S.

11.2 Defining structure types

The `DEFINE-STRUCTURE-TYPE` macro is a convenient way to define a new structure type and its associated constructor, predicator, and component selectors. `DEFINE-STRUCTURE-TYPE` is a macro defined in terms of `MAKE-STYPE` and the other low-level routines described in the next section. One may want to define one's own interface to these routines, one that initializes the components when a new instance is constructed, for example.

`(DEFINE-STRUCTURE-TYPE` *typename* $\{$*components*$\}^+$ $\{$*methods*$\}^*$ `)` $\longrightarrow$ *stype* — Special form

Creates a structure type (using `MAKE-STYPE`), and defines a number of variables:

typename`-STYPE` is defined to be the structure type.

`MAKE-`*typename* is defined to be the constructor.

typename`?` is defined to be the predicator.

typename`-`*componentname* is defined to be a component selector, for each of the *componentnames*.

For example,

```
(DEFINE-STRUCTURE-TYPE EMPLOYEE NAME AGE SALARY)
```

defines

`EMPLOYEE-STYPE`, a variable whose value is the structure type whose name is `EMPLOYEE`.

`(MAKE-EMPLOYEE)`, a procedure that creates uninitialized `EMPLOYEE`-structures.

`(EMPLOYEE?` *object*`)`, a type predicate.

`(EMPLOYEE-NAME` *employee*`)`
`(EMPLOYEE-AGE` *employee*`)`
`(EMPLOYEE-SALARY` *employee*`)`
selectors that access the components of `EMPLOYEE`-structures.

methods is an optional list of method clauses. For example,

```
(DEFINE-STRUCTURE-TYPE EMPLOYEE
                       NAME
                       AGE
                       SALARY
                       (((HUMAN? SELF) '#T)
                       (PRINT SELF PORT)
                        (FORMAT PORT "#{Employee (~A) ~A }"
                                (OBJECT-HASH SELF)
                                (EMPLOYEE-NAME SELF)))))
```

The methods in the methods clauses cannot reference the components directly. They must use the standard structure accessors. For example, in the print method above the *name* component of the *employee* structure must be accessed as `(EMPLOYEE-NAME SELF)` **not** as `NAME`.

11.3 Manipulating structure types

`(MAKE-STYPE` *typename componentnames method-handler*`)` $\longrightarrow$ *stype*

Returns a new structure type. The *typename*, which should be a symbol, is used for printing and identification purposes only. *Componentnames* should be a list of symbols. The *componentnames* correspond to the components that instances of the structure will have. The *method-handler* should be an object that handles whatever operations are required, or '#`F`, if a handler is not required.

For example:

```
(MAKE-STYPE 'EMPLOYEE '(NAME AGE SALARY) '#F) ⟹ #{Stype EMPLOYEE}
```

This creates a structure type, identified as `EMPLOYEE`, whose instances have components `NAME`, `AGE`, and `SALARY`. The object returned, a structure type, is appropriate as an argument to other routines described in this section. However,

```
(MAKE-STYPE 'EMPLOYEE
            '(NAME AGE SALARY)
            (OBJECT NIL
                    ((EMP? SELF) '#T)))
            ⟹ #{Stype EMPLOYEE}
```

defines an *stype* that handles the predicate `EMP?`.

`(STYPE-ID` *stype*`)` $\longrightarrow$ *typename*

Returns the type identifier for *stype*.

(STYPE-CONSTRUCTOR *stype*) $\longrightarrow$ *procedure*

Given a structure type *stype*, returns the procedure which will instantiate (i.e., create instances of) the structure type. The constructor procedure takes no arguments and creates a structure with uninitialized components.

Note that the constructor procedure is created at the time the structure type is created, not at the time that STYPE-CONSTRUCTOR is called.

((STYPE-CONSTRUCTOR *stype*)) $\equiv$ (COPY-STRUCTURE (STYPE-MASTER *stype*))

(STYPE-MASTER *stype*) $\longrightarrow$ *structure*

Returns the "master copy" from which all structures created by the structure type's constructor-procedure are made. This can be convenient for establishing default values for components in newly created structures. The way to do this is by storing into the master copy the desired default values; these values will be copied into all new instances of the structure type. For example,

```
(SET (EMPLOYEE-SALARY (STYPE-MASTER EMPLOYEE-STYPE)) 12000)
(EMPLOYEE-SALARY (MAKE-EMPLOYEE)) ⟹ 12000
```

(STYPE-PREDICATOR *stype*) $\longrightarrow$ *procedure*

Given a structure type *stype*, returns the procedure (a type predicate) which will predicate membership in the structure type. This procedure takes one argument and returns true if that argument is an instance of the structure type.

Note that the predicator procedure is created at the time the structure type is created, not at the time that STYPE-PREDICATOR is called.

(STYPE-SELECTOR *stype componentname*) $\longrightarrow$ *procedure*

Given a structure type *stype*, returns the procedure which will select the component of the structure type's instances identified by *componentname*. The selector procedure takes one argument, which must be a structure whose structure type is *stype*, and returns the appropriate component of the instance.

Such selector procedures support SETTER operations, which is to say that one may alter components of structures by using SET or related special forms.

The effect of selecting a structure component that has not been previously set, or initialized from the master structure (see STYPE-MASTER, above), is undefined.

Note that the selector procedures are created at the time the structure type is created, not at the time that STYPE-SELECTOR is invoked.

(STYPE-SELECTORS *stype*) $\longrightarrow$ *list of procedures*

Given a structure type *stype*, returns a list of its selector procedures.

(SELECTOR-ID *selector*) $\longrightarrow$ *identifier*

Given a selector procedure, e.g., as returned by STYPE-SELECTOR, returns its corresponding componentname.

(STYPE-HANDLER *stype*) $\longrightarrow$ *handler* Settable

Given a structure type *stype*, accesses the handler to be used for generic operations applied to instances of the structure type. For a newly created structure type, this handler handles no operations, but one may set the handler to be any handler at all.

11.4 Manipulating structures

The primary means for manipulating structures is to call their selector procedures and their setters. For example, in the context of the examples above, if `EMPLOYEE-23` is an `EMPLOYEE`-structure, then `EMPLOYEE-23`'s `NAME` component may be retrieved by evaluating

```
(EMPLOYEE-NAME EMPLOYEE-23)
```

and this value may be set by evaluating

```
(SET (EMPLOYEE-NAME EMPLOYEE-23) 'FRED)
```

This works because the selector procedures for `EMPLOYEE-STYPE` handle the generic operation called `SETTER`. Structures may also be manipulated using other generic operations.

`(STRUCTURE?` *object*`)` $\longrightarrow$ *boolean* Type predicate

Returns true if *object* is a structure.

`(COPY-STRUCTURE` *structure*`)` $\longrightarrow$ *structure*

Makes a copy of *structure*. The value returned is a new object of the same structure type as *structure* whose components are the same (in the `EQ?` sense) as *structure*'s.

`(COPY-STRUCTURE!` *destination source*`)` $\longrightarrow$ *structure*

Copies the components of *source* into *destination*, which is returned. *Source* and *destination* must be structures of the same type.

Chapter 12

Characters and strings

T has a special data type for representing *characters*. Characters are objects which may be stored in strings and communicated between the **T** system and external media such as files and terminals. Most characters represent printed graphics such as letters, digits, and punctuation.

The external syntax #*x* is used for characters. *x* may either be a single character or the "name" of a character. Valid character names `SPACE`, `TAB`, `FORM`, and `NEWLINE`. For example:

#\b	The alphabetic character lower-case *b*
#\7	The digit 7
#\;	The special character *semicolon*
#\tab	The tab character
#\NEWLINE	The new-line character

Some graphic characters are also readable by name:

```
#\LEFT-PAREN     ≡  #\(
#\RIGHT-PAREN    ≡  #\)
#\LEFT-BRACKET   ≡  #\[
#\RIGHT-BRACKET  ≡  #\]
#\LEFT-BRACE     ≡  #\{
#\RIGHT-BRACE    ≡  #\}
#\BACKSLASH      ≡  #\\
#\QUOTE          ≡  #\'
#\QUASIQUOTE     ≡  #\`
#\DOUBLEQUOTE    ≡  #\"
#\COMMA          ≡  #\,
#\SEMICOLON      ≡  #\;
```

The syntax `#[Ascii` *n*`]` may also be used for characters, where *n* is the ASCII code for the character (see section 12.8). This is not preferred, however, since it is less readable and less abstract than the #\ syntax.

```
#[Ascii 65] ≡ #\A
```

Unlike numbers, characters *are* uniquely instantiated. There is only one object which represents a given graphic or other character.

```
(EQ? #\x #\x) ⟶ true
```

Characters and strings are self-evaluating. There is no need to quote them to use them as constants in programs.

Strings are sequences of characters. Strings actually consist of two distinct components, a *header* and a *text*, which may be manipulated independently of each other. If one doesn't use the routines in section 12.5even be aware of this fact, and can treat strings as if they are similar to lists of characters.

Strings are notated simply by enclosing the actual sequence of characters within double quote characters. For example,

```
"Horse"
```

notates a five-character string consisting of the characters #\H, #\o, #\r, #\s, and #\e. The escape character (also known as backslash: \) may be used to a double-quote or a backslash within a string:

```
"The \"second\" word in this string is enclosed in double-quotes."
"\\ This string begins with one backslash."
```

There is no standard way to notate a string which contains non-graphic characters (e.g. control characters).

Strings are not uniquely instantiated; e.g.

```
(EQ? "Eland" "Eland")
```

may or may not yield true, depending on the implementation.

12.1 Predicates

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

Returns true if *object* is a character.

`(CHAR? #\X)` ⟹ *true*

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

Returns true if *object* is a string.

`(STRING? "Tapir.")` ⟹ *true*

`(GRAPHIC?` *character*`)` ⟶ *boolean*

Returns true if *character* is either the space character (#\SPACE) or it corresponds to a printed graphic such as a letter, digit, or punctuation mark.

`(GRAPHIC? #\X)` ⟹ *true*
`(GRAPHIC? #\NEWLINE)` ⟹ *false*

`(WHITESPACE?` *character*`)` ⟶ *boolean*

Returns true if *character* is a whitespace character (blank, tab, newline, carriage return, line feed, or form feed).

`(WHITESPACE? #\X)` ⟹ *false*
`(WHITESPACE? #\NEWLINE)` ⟹ *true*

`(ALPHABETIC?` *character*`)` ⟶ *boolean*

Returns true if *character* is an alphabetic (upper or lower case) character.

`(ALPHABETIC? #\y)` ⟹ *true*
`(ALPHABETIC? #\7)` ⟹ *false*

`(UPPERCASE? character)` ⟶ *boolean*

Returns true if *character* is an upper-case letter.

```
(UPPERCASE? #\y)      ⟹ false
(UPPERCASE? #\Y)      ⟹ true
(UPPERCASE? #\COMMA)  ⟹ false
```

`(LOWERCASE? character)` ⟶ *boolean*

Returns true if *character* is a lower-case letter.

```
(LOWERCASE? #\y)      ⟹ true
(LOWERCASE? #\Y)      ⟹ false
(LOWERCASE? #\COMMA)  ⟹ false
```

`(DIGIT? character radix)` ⟶ *boolean*

Returns true if *character* is a digit with respect to the given *radix*.

```
(DIGIT? #\5 10)  ⟹ true
(DIGIT? #\a 10)  ⟹ false
(DIGIT? #\a 16)  ⟹ true
```

12.2 Comparison

`(CHAR= char1 char2)` ⟶ *boolean*

`(CHAR< char1 char2)` ⟶ *boolean*

`(CHAR> char1 char2)` ⟶ *boolean*

`(CHARN= char1 char2)` ⟶ *boolean*

`(CHAR>= char1 char2)` ⟶ *boolean*

`(CHAR<= char1 char2)` ⟶ *boolean*

Six comparison predicates are defined for characters. `CHAR=` and `CHARN=` are defined for all characters. The others are defined only when the arguments are both upper-case letters, or both lower-case letters, or both digits.

`(STRING-EQUAL? string1 string2)` ⟶ *boolean*

Returns true if the two strings have the same length and characters.

`(MAKE-STRING length)` ⟶ *string*

Makes a string of null characters whose length is *length*.

`(STRING-APPEND . strings)` ⟶ *string*

Returns a new string which is the concatenation of *strings*.

```
(STRING-APPEND "llama" "and " "alpaca") ⟹ "llama and alpaca"
```

`(COPY-STRING string)` ⟶ *string*

Returns a new string, with new text, whose characters and length are the same as those of *string*.

(CHAR->STRING *character*) ⟶ *string*

Creates a string of length one whose single element is *character*.

(CHAR->STRING #\B) ⟹ "B"

(LIST->STRING *list*) ⟶ *string*

Converts a list of characters to a string.

(LIST->STRING '(#\Z #\e #\b #\u)) ⟹ "Zebu"

(STRING->LIST *string*) ⟶ *list*

Converts a string to a list of characters.

(STRING->LIST "Zebu") ⟹ (#\Z #\e #\b #\u)

12.3 String access

(STRING-LENGTH *string*) ⟶ *integer* Settable

Returns *string*'s length. A string's length may be SET, but the new length must be less than or equal to the original length.

(STRING-EMPTY? *string*) ⟶ *boolean*

Returns true if *string* is an empty string.

(STRING-EMPTY? "")	⟹	*true*
(STRING-EMPTY? "Bharal")	⟹	*false*
(STRING-EMPTY? *string*)	≡	(=0? (STRING-LENGTH *string*))

(STRING-ELT *string n*) ⟶ *character* Settable

(NTHCHAR *string n*) ⟶ *character* Settable

Returns the *nth* character in *string* (zero-based).

(NTHCHAR "SAIGA" 2) ⟹ #\I

(STRING-HEAD *string*) ⟶ *character* Settable

(CHAR *string*) ⟶ *character* Settable

Returns first character in *string*.

(STRING-TAIL *string*) ⟶ *string*

(CHDR *string*) ⟶ *string*

Returns the "tail" of *string*.

(STRING-TAIL "Ibex.") ⟹ "bex."

(STRING-NTHTAIL *string n*) $\longrightarrow$ *string*

(NTHCHDR *string n*) $\longrightarrow$ *string*

Returns the *nth* tail of *string*.

(NTHCHDR "SAIGA" 2) $\Longrightarrow$ "IGA"

(SUBSTRING *string start count*) $\longrightarrow$ *string*

Returns a substring of *string*, beginning with the $start^{th}$ character, for a length of *count* characters.

(SUBSTRING "A small oryx" 2 5) $\Longrightarrow$ "small"

(STRING-SLICE *string start count*) $\longrightarrow$ *string*

Returns a substring (slice) of *string*, beginning with the $start^{th}$ character, for a length of *count* characters.

(STRING-SLICE "A small oryx" 2 5) $\Longrightarrow$ "small"

Unlike SUBSTRING, the characters returned by STRING-SLICE are shared with the original string; that is, any changes to characters in the original string which have been selected in the substring are reflected in the substring, and vice versa.

12.4 String manipulation

(STRING-POSQ *character string*) $\longrightarrow$ *integer* or *false*

Returns the index of the first occurrence of *character* in *string*, if it is there; otherwise returns false.

(STRING-POSQ #\i "oribi") $\Longrightarrow$ 2
(STRING-POSQ #\s "oribi") $\Longrightarrow$ *false*

(STRING-REPLACE *destination source count*) $\longrightarrow$ *string*

Copies *count* characters from the *source* string to the *destination* string, destructively, and return the modified *destination*.

```
(DEFINE S (COPY-STRING "The bison"))
(STRING-REPLACE S "Any how" 3) ⟹ "Any bison"
```

(MAP-STRING *procedure string*) $\longrightarrow$ *string*

Calls *procedure* on each character in *string*, collecting the successive return values which should be characters in a new string.

(MAP-STRING CHAR-UPCASE "A grisbok") $\Longrightarrow$ "A GRISBOK"

(MAP-STRING! *procedure string*) $\longrightarrow$ *string*

Calls *procedure* on each character in *string*, storing the results which should be characters back into *string*.

(WALK-STRING *procedure string*) $\longrightarrow$ *undefined*

Calls *procedure* on each character in *string*.

12.5 String header manipulation

A *string header* is a structure of fixed size which contains a pointer into a *string text*, and a length. A string text is a vector of characters themselves. The string text is not itself a directly accessible object, but can only be manipulated via a string header. Several string headers may point into the same text. The term *string* is used to refer to a header and text considered as a whole.

(CHOPY *string*) ⟶ *string*

> Makes a new string header pointing to the same string text, and with the same length, as the header for *string*.

(CHOPY! *destination source*) ⟶ *string*

> Copies the header for the *source* string into the header for the *destination* string, which is returned.

(STRING-TAIL! *string*) ⟶ *string*

(CHDR! *string*) ⟶ *string*

> Destructively modifies *string*'s header to point to the next character in its text, and decrements its length.

```
(LET ((S (COPY-STRING "String.")))
     (CHDR! S)
     S)
     ⟹
     "tring."
```

(STRING-NTHTAIL! *string n*) ⟶ *string*

(NTHCHDR! *string* n) ⟶ *string*

> Destructive version of STRING-NTHTAIL.

12.6 Case conversion

(CHAR-UPCASE *character*) ⟶ *character*

> If *character* is a lower-case character, returns the corresponding upper-case character. Otherwise *character*, which must be a character, is returned.

(CHAR-DOWNCASE *character*) ⟶ *character*

> If *character* is an upper-case character, returns the corresponding lower-case character. Otherwise *character*, which must be a character, is returned.

(STRING-UPCASE *string*) ⟶ *string*

> Returns a copy of *string* with all lower-case characters converted to upper case.
>
> (STRING-UPCASE *string*) ≡ (MAP-STRING CHAR-UPCASE *string*)

(STRING-DOWNCASE *string*) ⟶ *string*

> Returns a copy of *string* with all upper-case characters converted to lower case.

(STRING-DOWNCASE *string*) ≡ (MAP-STRING CHAR-DOWNCASE *string*)

(STRING-UPCASE! *string*) ⟶ *string*

Destructive version of STRING-UPCASE.

(STRING-UPCASE! *string*) ≡ (MAP-STRING! CHAR-UPCASE *string*)

(STRING-DOWNCASE! *string*) ⟶ *string*

Destructive version of STRING-DOWNCASE.

(STRING-DOWNCASE! *string*) ≡ (MAP-STRING! CHAR-DOWNCASE *string*)

12.7 Digit conversion

(CHAR->DIGIT *character radix*) ⟶ *integer*

Returns the *weight* of the character when treated as a digit. *Character* must be a digit in the given *radix*.

(CHAR->DIGIT #\A 16) ⟹ 10

(DIGIT->CHAR *integer radix*) ⟶ *character*

Given a non-negative *integer* less than *radix*, returns a character (a digit) whose weight is the *integer*.

(DIGIT->CHAR 10 16) ⟹ #\A

(DIGIT *character radix*) ⟶ *integer* or *false*

If *character* is a digit, returns its weight; otherwise returns false.

(DIGIT #\5 10) ⟹ 5

12.8 ASCII conversion

(CHAR->ASCII *character*) ⟶ *integer*

Given a character, returns its ASCII representation as an integer.

(ASCII->CHAR *integer*) ⟶ *character*

Given an integer which is the ASCII code for some character, returns the character.

NUMBER-OF-CHAR-CODES ⟶ *integer*

The value of NUMBER-OF-CHAR-CODES is a number that is 1 larger than the largest value that will ever be returned by CHAR->ASCII. This may be used to make tables which are to be indexed by ASCII codes.

```
(DEFINE *TABLE* (MAKE-VECTOR NUMBER-OF-CHAR-CODES))  ⟹  vector
(VSET *TABLE* (CHAR->ASCII #\F) 'COW)                ⟹  COW
(VREF *TABLE* (CHAR->ASCII #\F))                     ⟹  COW
```

12.9 Symbols

Symbols are similar to strings, but are instantiated uniquely; only one symbol with a given print name exists. Symbols are used to identify variables, among other things. The fact that they have a convenient external representation makes them useful for many purposes.

Symbols may be coerced to strings and vice versa. If two strings are equal to each other (e.g. according to `STRING-EQUAL?`), then they will both convert to the same symbol.

`(EQ? (STRING->SYMBOL` *string1*`) (STRING->SYMBOL` *string2*`))`

if and only if

`(STRING-EQUAL?` *string1 string2*`)`

See also sections 3.4 and 14.1.

`(STRING->SYMBOL` *string*`)` $\longrightarrow$ *symbol*

Returns the symbol whose print name is equal to *string*.

```
(STRING->SYMBOL "COW")     ==>  COW
(STRING->SYMBOL "123")     ==>  \123
(STRING->SYMBOL "bison")   ==>  \b\i\s\o\n
(STRING->SYMBOL "")        ==>  #[Symbol ""]
```

Note that it is `READ-OBJECT` (page 93), not `STRING->SYMBOL`, which coerces alphabetic characters to upper case.

`(SYMBOL->STRING` *symbol*`)` $\longrightarrow$ *string*

Returns a string for which `SYMBOL->STRING` will return *symbol*.

```
(SYMBOL->STRING 'COW) ==> "COW"
```

Chapter 13

Miscellaneous features

13.1 Comments and declarations

`(COMMENT . `*comment*`)` $\longrightarrow$ *undefined* — Special form

Does nothing and returns no value of interest. `COMMENT`-expressions may be used to write comments in code. (The preferred way to write comments, however, is with the semicolon read macro character; see page 94.)

`(IGNORE . `*variables*`)` $\longrightarrow$ *undefined* — Special form

Ordinarily, programs such as compilers which manipulate source programs consider it to be an exceptional condition when a bound variable is not referenced, and may generate warning messages when they detect this condition. `IGNORE`-expressions may be used to suppress such warnings and also to request that a warning be issued if in fact there are any references to *variables*.

```
(LAMBDA (X Y) (IGNORE X) (CAR Y))
```

`(IGNORABLE . `*variables*`)` $\longrightarrow$ *undefined* — Special form

`IGNORABLE` is like `IGNORE` except that permission is *not* given to give warnings if any of *variables* actually is referenced. This is not useful for human-generated expressions, but may be useful in the expansion of a macro invocation where the macro expander may not know whether a bound variable in the expansion is referenced or not, and wants to declare that it is all right if the variable is not referenced.

13.2 Errors and dead ends

`(ERROR `*control-string* `.` *arguments*`)` $\longrightarrow$ *object*

Signals an error. *Control-string* and *arguments* should be arguments suitable for a call to `FORMAT` (page 107). The error is reported in an implementation-dependent manner, and an opportunity is provided to possibly correct or proceed from the error. (See section 19.1 for details of **T**'s handling of errors.)

```
(ERROR "cannot wash the dishes because ~A" EXCUSE-DESCRIPTION-STRING)
```

`(SYNTAX-ERROR `*control-string* `.` *arguments*`)` $\longrightarrow$ *object*

Similar to `ERROR`, but signals a syntax error. This should be called, for example, from within macro expanders when an illegal syntax is encountered.

(READ-ERROR *port control-string* . *arguments*) ⟶ *object*

Similar to ERROR, but signals a read error. This should be called, for example, from within read macros when an illegal read syntax is encountered.

(CHECK-ARG *predicate object procedure*) ⟶ *object*

Verifies that an object is of a particular type. *Predicate* should be a type predicate, and *object* can be any object. If *predicate*, when applied to *object*, returns false, then an error is signalled. If it returns true, then the CHECK-ARG returns *object*. *Procedure* is a procedure whose name will be given in any message printed by the error system.

The user interface (see section 19.1) may provide a way to supply a value to use in place of *object*. If the user attempts to correct the error in this way, then CHECK-ARG again verifies that the new value answers true to *predicate*, and returns it.

For example:

```
(DEFINE (WASH-DISH DISH)
        (LET ((DISH (CHECK-ARG DISH? DISH WASH-DISH)))
             ... ))
```

(ENFORCE *predicate value*) ⟶ *value*

Returns *value* which must answer *true* to *predicate*. If *predicate* returns *false* when applied to *value*, the effect is undefined (normally, this means that an error is signalled). If ENFORCE signals an error and enters a breakpoint, then a new value can be returned using RET. For example,

```
> (LET ((A (ENFORCE FIXNUM? 'A))) (+ A 1))
* * Error:  (ENFORCE FIXNUM? A) failed in (anonymous)
>> (RET 1)
2
>
```

(UNDEFINED-VALUE . *arguments*) ⟶ *undefined*

Has no effect and yields some undefined value. An implementation will endeavor to return some object which, when printed or otherwise displayed, will show the *arguments*. This feature may be useful in debugging, for example in tracking down the origin of the undefined value, if the value has propagated to an undesirable place.

(UNDEFINED-EFFECT . *arguments*) ⟶ *undefined*

The effect of calling UNDEFINED-EFFECT is undefined. An implementation will endeavor to signal an error condition if such a call ever occurs; however, an optimizing compiler may make use of the fact that the control path leading to a call to UNDEFINED-EFFECT will never be taken in a correctly running program, and so in some cases may eliminate the call.

13.3 Early binding

(DEFINE-CONSTANT *variable value*) ⟶ *undefined* Special form

This is semantically identical to DEFINE, but also declares that the value of the variable will not change. This might permit a compiler to perform constant-folding. Also, if the variable is defined to be a small integer, this may interact well with SELECT to obtain fast numeric dispatch on "enumerated types".

(DEFINE-INTEGRABLE *variable value*) ⟶ *undefined* Special form
(DEFINE-INTEGRABLE (*variable* . *arguments*) . *body*) ⟶ *undefined*

This is semantically identical to DEFINE, but also declares that the value of the variable is not expected to change. For example, if a reference to the variable is encountered in functional position in a call, its definition may be *integrated*, that is, substituted in-line.

13.4 Symbol generators

(GENERATE-SYMBOL *prefix*) ⟶ *symbol*

Each call to GENERATE-SYMBOL generates a unique identifier. The form which the new identifier's name takes is not defined, but the identifier is guaranteed to be different from any identifier created in any other way. The identifier's external representation will begin with *prefix*.

(CONCATENATE-SYMBOL . *things*) ⟶ *symbol*

Creates a symbol whose print name is obtained by appending the printed representations of *things* according to the DISPLAY operation.

```
(CONCATENATE-SYMBOL 'FOO- THING- 34) ⟹ FOO-THING-34
```

13.5 Combinators

(ALWAYS *value*) ⟶ *procedure*

Returns a procedure which ignores its arguments and always returns *value*.

(ALWAYS *value*) ≡ (LAMBDA X (IGNORE X) *value*)

(IDENTITY *object*) ⟶ *object*

Identity function. Returns its argument.

(PROJN *n*) ⟶ *procedure*

Returns a procedure which returns (projects) its *nth* argument, ignoring any others.

```
(PROJN 1) ≡ (LAMBDA (A B . REST) (IGNORE A REST) B)
```

(PROJ0 *object* . *rest*) ⟶ *object*

Projection function: returns its zeroth argument, ignoring the rest.

```
PROJ0 ≡ (PROJN 0)
```

PROJ0 is similar to BLOCK0 (page 37), except that it is a procedure, not a special form.

(PROJ1 *object0 object1* . *rest*) ⟶ *object1*

Returns its second argument.

(PROJ2 *object0 object1 object2* . *rest*) ⟶ *object2*

Returns its third argument.

(PROJ3 *object0 object1 object2 object3 . rest*) ⟶ *object3*

Returns its fourth argument.

(CONJOIN . *predicates*) ⟶ *predicate*

Returns a predicate which is the logical conjunction of all of the *predicates*.

((CONJOIN >0? ODD?) 13)	⟹	*true*
((CONJOIN >0? ODD?) 8)	⟹	*false*

(DISJOIN . *predicates*) ⟶ *predicate*

Returns a predicate which is the logical disjunction of all of the *predicates*.

((DISJOIN >0? ODD?) 13)	⟹	*true*
((DISJOIN >0? ODD?) 8)	⟹	*true*

(COMPLEMENT *predicate*) ⟶ *predicate*

Returns a predicate which is the logical complement of the *predicate*.

ATOM?	≡	(COMPLEMENT PAIR?)
((COMPLEMENT MEMQ?) 'A '(X Y Z))	⟹	*true*

(COMPOSE . *procedures*) ⟶ *procedure*

Returns a procedure which is the composition of the *procedures*. The last of the *procedures* may take any number of arguments, and the resulting procedure will take that same number of arguments; all the other *procedures* must take one argument.

((COMPOSE CAR CDR) '(A B)) ⟹ B
(COMPLEMENT *predicate*) ≡ (COMPOSE NOT *predicate*)
PROPER-LIST? ≡ (DISJOIN NULL? (COMPOSE NULL? CDR LASTCDR))
NTH ≡ (COMPOSE CAR NTHCDR)

(TRUE . *arguments*) ⟶ *true* — Type predicate

Ignores its arguments, and always returns true. This may be used as a predicate representing the "universal type" - the type which subsumes all objects.

(FALSE . *arguments*) ⟶ *false* — Type predicate

Ignores its arguments, and always returns false. This may be used as a predicate representing the "null type" - the type which subsumes no objects.

(TRUE? *value*) ⟶ *boolean* — Type predicate

Returns true if *value* is *some* true value, false otherwise. This is convenient where one wants to coerce a truth value to be a *standard* truth value; that is, TRUE? maps false to itself, and any true value to the standard true value (T).

```
(TRUE? object)  ≡   (NOT (FALSE? object))
(TRUE? NIL)     ⟹   false
(TRUE? T)       ⟹   true
(TRUE? 3)       ⟹   true
```

(BOOLEAN? *object*) ⟶ *boolean* — Type predicate

This returns true if *object* is either the standard true value or the standard false value.

```
(BOOLEAN? NIL)  ⟹  true
(BOOLEAN? T)    ⟹  true
(BOOLEAN? 3)    ⟹  false
```

13.6 Vectors

Vectors can be thought of as one-dimensional, zero-based arrays, or as fixed-length, random-access lists. They read and print like lists with a # in front. Like lists, but unlike numbers and strings, vectors are not self-evaluating. To write a constant vector, quote it: '#(X Y (1 2)).

(VECTOR? *object*) ⟶ *boolean* — Type predicate

Returns true if *object* is a vector.

(MAKE-VECTOR *size*) ⟶ *vector*

Returns a vector whose length is *size*. The elements are not initialized to any particular value.

VECTOR-FILL (see below) may be used to fill the vector with some useful value (such as ()).

(LIST->VECTOR *list*) ⟶ *vector*

Converts a list to a vector.

```
(LIST->VECTOR '(A B C)) ⟹ #(A B C)
```

(VECTOR->LIST *vector*) ⟶ *list*

Converts a vector to a list.

```
(VECTOR->LIST '#(A B C)) ⟹ (A B C)
```

(VECTOR-ELT *vector n*) ⟶ *object* — Settable

(VREF *vector n*) ⟶ *object* — Settable

Accesses the *nth* element of *vector* (zero-based).

```
(VECTOR-ELT '#(A B C) 1) ⟹ B
```

(VSET *vector n object*) ⟶ *object*

Sets the *nth* element of *vector* to *object*.

```
(VSET vector n object) ≡ (SET (VREF vector n) object)
```

(COPY-VECTOR *vector*) ⟶ *vector*

Makes a copy of *vector*.

`(VECTOR-FILL` *vector value*`)` $\longrightarrow$ *vector*

Sets every element of *vector* to *value*, and returns the modified *vector*.

`(VECTOR-REPLACE` *target source n*`)` $\longrightarrow$ *vector*

Sets the first *n* elements of *target* to be the same as the corresponding elements of *source*, and returns the (modified) *target*.

`(VECTOR-LENGTH` *vector*`)` $\longrightarrow$ *integer*

Returns *vector*'s length.

`(VECTOR-POS` *predicate object vector*`)` $\longrightarrow$ *integer* or *false*

Returns index of the first element *x* of *vector* such that `(`*predicate object x*`)`, or false if there is no such element.

`(VECTOR-POSQ` *object vector*`)` $\longrightarrow$ *integer* or *false*

`(VECTOR-POSQ` *object vector*`)` $\equiv$ `(VECTOR-POS EQ?` *object vector*`)`

`(WALK-VECTOR` *procedure vector*`)` $\longrightarrow$ *undefined*

Applies *procedure* to every element in *vector*.

13.7 Pools

Pools give a convenient way to perform explicit storage management in T. One may obtain an object from a pool (popping the pool's free list, or creating a new object if the free list is empty), and later return an object to a pool. By managing storage allocation in this way, the frequency of garbage collections

All pools are emptied when a garbage collection occurs. Garbage collections occur asynchronously in an implementation-dependent manner.

`(MAKE-POOL` *identification generator*`)` $\longrightarrow$ *pool*

Creates a pool. *Generator* should be a procedure, and will be called by `OBTAIN-FROM-POOL` whenever the pool's free list is empty. *Identification* is only for identification purposes, for example, when the pool is printed.

`(OBTAIN-FROM-POOL` *pool*`)` $\longrightarrow$ *object*

Obtains an object from *pool*. If the pool is empty, the pool's generator is called to obtain an object, which is then returned directly.

`(RETURN-TO-POOL` *pool object*`)` $\longrightarrow$ *undefined*

Returns *object* to *pool*. A future call to `OBTAIN-FROM-POOL` may yield *object*.

13.8 Weak pointers

(`OBJECT-HASH` *object*) $\longrightarrow$ *integer*

Returns a unique numeric identifier for *object*. That is,

(`EQ?` *object1 object2*)

if and only if

(`=` (`OBJECT-HASH` *object1*) (`OBJECT-HASH` *object2*))

Because `OBJECT-HASH` is invertible (see below), it can be used to create weak pointers to objects, that is, "pointers" or "references" which are not strong enough to prevent an object from being reclaimed by the garbage collector. This concept of weak pointer is implemented by the integers returned by `OBJECT-HASH`, which can be dereferenced by calling `OBJECT-UNHASH`.

`OBJECT-HASH` is used by standard methods for the `PRINT` operation when printing objects which have no read syntax.

(`OBJECT-UNHASH` *integer*) $\longrightarrow$ *object* or *false*

Returns the object whose unique identifier is *integer*, or false if the object is no longer accessible (e.g. due to garbage collection).

(`OBJECT-UNHASH` (`OBJECT-HASH` *object*)) $\Longrightarrow$ *object*

In **T3**, "populations" have been renamed to "weak-sets". This change was made in the belief that "weak-set" is a more intuitive name than "population". The old names are still supported, but they will be removed in a future release.

Weak-Sets provide a way to keep track of a collection of objects. They are sometimes known as *weak sets* because they behave much like sets, but an object in a weak-set may go away if the only pointer to the object is via the weak-set. The garbage collector will remove such objects from weak-sets.

(`MAKE-WEAK-SET` *identification*) $\longrightarrow$ *weak-set*

Creates a new weak-set.

(`ADD-TO-WEAK-SET` *weak-set object*) $\longrightarrow$ *undefined*

Adds *object* to *weak-set*.

(`REMOVE-FROM-WEAK-SET` *weak-set object*) $\longrightarrow$ *undefined*

Removes *object* from *weak-set*.

(`WEAK-SET->LIST` *weak-set*) $\longrightarrow$ *list*

Returns a list of all objects currently in *weak-set*. Note that as long as this list is accessible, none of the objects will be implicitly removed from the weak-set, because they will be accessible via this list.

(`WALK-WEAK-SET` *weak-set procedure*) $\longrightarrow$ *undefined*

Calls *procedure* on each member of *weak-set*.

(`WALK-WEAK-SET` *weak-set procedure*)
$\equiv$
(`WALK` *procedure* (`WEAK-SET->LIST` *weak-set*))

(`WEAK-SET-MEMBER?` *object weak-set*) $\longrightarrow$ *boolean*

`WEAK-SET-MEMBER?` returns true if *object* is a member of *weak-set*; otherwise, it returns false.

(`WEAK-SET-EMPTY?` *weak-set*) $\longrightarrow$ *boolean*

`WEAK-SET-EMPTY?` returns true if *weak-set* is empty; otherwise, it returns false.

Chapter 14

Syntax

The **T** standard environment includes routines which perform syntactic and semantic analysis of **T** programs. There are two such subsystems within **T**, corresponding to the language's two syntactic levels (see chapter 2). These are the *reader* and the *compiler*.

In an attempt to make each of these subsystems as generally useful and flexible as possible, they are not restricted to processing the language as described in this manual. Instead, they each operate with respect to parameter clusters known as *read tables*, in the case of the reader, or *syntax tables*, in the case of the compiler.

14.1 The reader

The *reader* is a procedure available in the standard environment as the value of the variable `READ-OBJECT`. Conceptually, the reader coerces a stream of characters (external representation) to a stream of objects (internal representations) via a mechanism known as *parsing*.

(`READ-OBJECT` *port read-table*) $\longrightarrow$ *object* or *end-of-file*

> `READ-OBJECT` employs the `READ-CHAR` (page 105) and `UNREAD-CHAR` (page 105) operations in order to parse an object according to the port's read table.
>
> If the port is empty, the end-of-file token is returned.
>
> `READ-OBJECT` is called by the default method for the `READ` operation (page 106), so the reader is usually invoked indirectly by calling `READ`, not by calling `READ-OBJECT` directly. When invoked from `READ`, the second argument to `READ-OBJECT` is obtained by calling the `PORT-READ-TABLE` operation on the port.

The reader works as follows:

Any whitespace characters (space, tab, newline, carriage return, line feed, or form feed) are read and ignored. A non-whitespace character is obtained; call it *c*.

If *c* is a read-macro character, the reader invokes a specialist routine to handle a syntactic construct introduced by the read-macro character.

If *c* is not a read-macro character, then characters are read and saved until a *delimiter* character is read. A delimiter character is either a whitespace character, or one of the following: `(` (left parenthesis),) (right parenthesis), `[`, `]`, {, }, or `;` (semicolon). If the sequence of characters beginning with *c* and going up to but not including the delimiter is parsable as a number, then the sequence is converted to a number, which is returned. Otherwise the sequence is converted to a symbol.

The *escape character*, backslash (\), may be used within a run of constituent characters to unusual characters in a symbol's print name. In this case, the escaped character (i.e. the character following the escape character) is treated as if it were a constituent character, and is not converted to upper case if it is a lower case letter. For example:

> `abc\;def` reads the same as `#[Symbol "ABC;DEF"]`

`\a\bcdef` reads the same as `#[Symbol "abCDEF"]`
`\12345` reads the same as `#[Symbol "12345"]`
`\'12345` reads the same as `#[Symbol "12345"']`

The following are standard read-macro characters:

`"` Doublequote: introduces a string. Characters are read until another doublequote character is found which does not immediately follow a backslash (\) and a string is returned. Within a string, backslash acts as an escape character, so that doublequotes and backslashes may appear in strings.

`'` Quote: '*object* reads the same as `(QUOTE` *object*`)`.

(Left parenthesis: begins a list.

) Right parenthesis: ends a list or vector, and is illegal in other contexts.

` Quasiquote: see section 10.4.

, Comma: this is part of the backquote syntax.

`@` At sign: this is part of the backquote syntax.

; Semicolon: introduces a comment. Characters are read and discarded until a newline is encountered, at which point the parsing process starts over.

`#` Sharp-sign: another dispatch to a specialist routine is performed according to the character following the `#`.

Standard sharp-sign forms:

#\ Character syntax. See section 12.

`#x` Hexadecimal input. An integer following the `#x` is read in base 16.

`#o` Octal input. An integer following the `#o` is read in base 8.

`#b` Binary input. An integer following the `#b` is read in base 2.

`#(... )` Vector. The elements of a vector are read between the parentheses, and the vector is returned.

`#[... ]` This syntax is used for certain kinds of re-readable objects. It also provides an alternate syntax for characters and symbols. The brackets enclose a sequence of objects; the first should be a symbol which keys the type of the resulting object, e.g. `CHAR` or `SYMBOL`. For example,

> `#[Ascii 65]` represents the same object as `#\A`
> `#[Symbol FOO]` represents the same object as `FOO`

This syntax is used by the printer when necessary, for example:

> `(STRING->SYMBOL )` ⟹ `#[Symbol ]`
> `(ASCII->CHAR 128)` ⟹ `#[Ascii 128]`

`#{... }` This is the syntax used by the printer for objects which have no reader syntax. When the reader encounters the sequence `#{` it signals an error.

14.2 Read tables and read macros

Read tables package a number of parameters for use by programs which parse, generate, or otherwise manipulate external syntax of programs and objects. In particular, every read table contains a table which maps characters to objects which describe their lexical properties.

There is a standard read table which contains the standard read syntax for all characters. In order to define nonstandard read syntax, one must create a new read table using `MAKE-READ-TABLE`, and arrange for `READ-OBJECT` to use the new read table instead of the standard read table, e.g. by doing `(SET (PORT-READ-TABLE ... ) ... )`.

`(MAKE-READ-TABLE` *read-table identification*`)` $\longrightarrow$ *new-read-table*

Creates a new read table which is a copy of *read-table*. *Identification* serves for debugging purposes only.

```
(DEFINE *MY-READ-TABLE*
        (MAKE-READ-TABLE STANDARD-READ-TABLE '*MY-READ-TABLE*))
```

`STANDARD-READ-TABLE` $\longrightarrow$ *read-table*

The standard **T** read table.

`VANILLA-READ-TABLE` $\longrightarrow$ *read-table*

The value of `VANILLA-READ-TABLE` is a read table in which all all graphic characters have ordinary non-read-macro constituent syntax, whitespace characters have whitespace syntax, and other characters (e.g. control characters) are illegal.

```
(WITH-INPUT-FROM-STRING (PORT " FOO () ")
                        (SET (PORT-READ-TABLE PORT) VANILLA-READ-TABLE)
                        (LIST (READ PORT) (READ PORT)))
⟹ (FOO #[Symbol ()])
```

`(READ-TABLE-ENTRY` *table character*`)` $\longrightarrow$ *syntax* Settable

Access *character*'s read syntax in *table*. The entry for a given character is some object which represents the character's lexical properties, for example, whether it is a constituent, whitespace, or read-macro character. For read-macro characters, the entry is a procedure for parsing the read-macro construct.

Values suitable to be stored in read tables may be obtained by accessing existing entries in the standard read table. For example:

```
(SET (READ-TABLE-ENTRY *MY-READ-TABLE* #\;)
     (READ-TABLE-ENTRY STANDARD-READ-TABLE #\:))
```

makes the read syntax of semicolon in `*MY-READ-TABLE*` be the same as the standard read syntax of colon (which is a contituent character).

To define a read macro, do

`(SET (READ-TABLE-ENTRY` *read-table character*`)` *procedure*`)`

where *procedure* is a procedure taking three arguments: a port, a character, and a read-table. The port is the port which is currently being parsed by `READ-OBJECT`. It may be passed as the port argument to input routines like `READC` and `READ-REFUSING-EOF` if the read-macro needs to parse further characters from the input port. The character is the character which caused the read macro to be invoked; that is, it is the character under which the procedure is stored in the read table. The read-table is the read table from which the procedure was fetched (i.e. the one that was originally passed to `READ-OBJECT`. Read macros which recursively invoke the reader will want to pass that read table as the second argument to `READ-OBJECT`.

Example:

```
(SET (READ-TABLE-ENTRY *MY-READ-TABLE* #\')
     (LAMBDA (PORT CH RTABLE)
             (IGNORE CH) (IGNORE RTABLE)
             (LIST 'QUOTE (READ-REFUSING-EOF PORT))))
```

Note that the standard read table and the vanilla read table are immutable, and so their entries may not be changed.

`NOTHING-READ` ⟶ *object*

Read macros should return this object, which is treated specially by the reader, if they want to return no object. For example, the semi-colon (comment) read-macro might be defined as follows:

```
(SET (READ-TABLE-ENTRY *MY-READ-TABLE* #\;)
     (LAMBDA (PORT CH RTABLE)
             (IGNORE RTABLE)
             (ITERATE LOOP ()
                      (LET ((C (READC PORT)))
                           (COND ((EOF? C) C)
                                 ((CHAR= C #\NEWLINE) NOTHING-READ)
                                 (ELSE (LOOP)))))))
```

`(DELIMITING-READ-MACRO?` *procedure*`)` ⟶ *boolean* Operation

If an object which returns true to the `DELIMITING-READ-MACRO?` operation is stored in a read table under a given character, then the reader will treat the character as a delimiter (non-constituent) character. By default, read-macro procedures return false to this predicate. Thus to make a read-macro character be a delimiting character also (as are parentheses and semicolon in the standard read table), its read table entry should handle this operation by returning true.

```
(SET (READ-TABLE-ENTRY *MY-READ-TABLE* #\~)
     (OBJECT (LAMBDA (PORT CH RTABLE)
             ... )
             ((DELIMITING-READ-MACRO? SELF) T)))
```

`(MAKE-LIST-READER)` ⟶ *list-reader*

The two procedures `MAKE-LIST-READER` and `LIST-TERMINATOR` can be used together to define read macros which behave syntactically like parentheses.

Each call to `MAKE-LIST-READER` returns an object which is a procedure of two arguments called a *list reader.* Calling `LIST-TERMINATOR` on a list reader will return another object called its *list terminator.*

List readers and terminators are suitable for entry in read tables. A list reader acts as a read macro which reads a sequence of objects, terminated by a character whose read syntax is the corresponding list terminator. For example, the standard syntax for left and right parentheses might be defined as follows:

```
(LET ((LIST-READER (MAKE-LIST-READER)))
     (SET (READ-TABLE-ENTRY STANDARD-READ-TABLE #\LEFT-PAREN)
          LIST-READER)
     (SET (READ-TABLE-ENTRY STANDARD-READ-TABLE #\RIGHT-PAREN)
          (LIST-TERMINATOR LIST-READER)))
```

Like any read-macro procedure, a list reader is a procedure of two arguments. The first argument must be a port, and the second is ignored. Thus instead of being stored in a read table, it may be called from another read-macro procedure. For example, the following makes `[... ]` an alternative read syntax for vectors:

```
(LET ((LIST-READER (MAKE-LIST-READER)))
     (SET (READ-TABLE-ENTRY *MY-READ-TABLE* #\LEFT-BRACKET)
          (OBJECT (LAMBDA (PORT CH RTABLE)
                          (IGNORE RTABLE)
                          (LIST->VECTOR (LIST-READER PORT CH)))
                  ((DELIMITING-READ-MACRO? SELF) T)))
     (SET (READ-TABLE-ENTRY *MY-READ-TABLE* #\RIGHT-BRACKET)
          (LIST-TERMINATOR LIST-READER)))
```

List readers and terminators handle the `DELIMITING-READ-MACRO?` operation by returning true.

`(LIST-TERMINATOR` *list-reader*`)` $\longrightarrow$ *list-terminator*

Given a list reader, returns its list terminator. See `LIST-READER`, above.

14.3 Standard compiler

A *compiler* is a procedure which accepts an expression and a syntax table, and returns a *compiled code* object. A compiled code object may be executed in a given environment. Note that the term "compiler" is used in a technical sense and encompasses not only compilers such as ORBIT (section 17.5) which produce machine instructions, but also programs such as the standard compiler (which is called by `EVAL`) which operate internally by producing intermediate code or by interpreting source code directly. Often programs like these are called *interpreters* instead of *compilers.*

A given **T** implementation may have several compilers. `STANDARD-COMPILER` should be one of these compilers. **T** 3.1 also provides a compiler called ORBIT, which is described in section 17.5.

`(EVAL` *expression environment*`)` $\longrightarrow$ *object*

Evaluates *expression* in *environment*. Evaluation is performed as a two-stage process: first, the standard compiler compiles the expression, producing a compiled code object; then the compiled code object is invoked in the given environment.

```
(EVAL expression environment)
      ≡
(RUN-COMPILED-CODE (STANDARD-COMPILER expression
                                      (ENV-SYNTAX-TABLE environment))
                   environment)
```

The evaluation semantics have been extended to allow the valuation of forms whose `car`'s are syntax descriptors. such a form is interpreted just as if it were a form whose `car` was a symbol whose syntax table entry was the syntax descriptor. For example,

```
(DEFINE-LOCAL-SYNTAX (FOO X)
                     ` (,(SYNTAX-TABLE-ENTRY STANDARD-SYNTAX-TABLE 'LAMBDA) () ,X))
((FOO 5)) ==> 5
```

This feature allows control over binding time for reserved words. For example, a syntax descriptor such as `FOO`, above, can be sure that its expansion will be treated as an expression that evaluates to a closure, regardless of what the syntax table entry for the symbol `lambda` is when the expansion is evaluated or otherwise analyzed.

`(STANDARD-COMPILER` *expression syntax-table*`)` $\longrightarrow$ *compiled-code*

Compiles *expression*. **T** may provide several compilers, of which `STANDARD-COMPILER` will be the one which is invoked by `EVAL`.

`(RUN-COMPILED-CODE` *compiled-code environment*`)` $\longrightarrow$ *object*

Invokes a compiled code object.

14.4 Syntax Tables

A syntax table maps symbols to *syntax descriptors.* Every syntax descriptor is itself either a macro expander, or a unique token identifying a primitive special form type.

Every locale has an associated syntax table. A locale's syntax table contains definitions of special forms which are local to the locale. Each such syntax table inherits entries lexically from the syntax tables of enclosing locales.

Macros provide a mechanism for extending the syntax of **T** by means of source-to-source transformations. As in many Lisp dialects, the macro facility in **T** provides a powerful tool for amplifying the expressiveness of the language. But like any powerful tool, macros may be abused. They may easily lead to programs that are very hard to understand.

Macros are defined by entering syntax descriptor objects known as *macro expanders* into syntax tables; see `SYNTAX-TABLE-ENTR` and `DEFINE-SYNTAX`, below. Macros may also be defined locally to a file or expression using `DEFINE-LOCAL-SYNTAX` or `LET-SYNTAX`.

Procedure integration is preferable to the use of macros in situations where either would be applicable. See `DEFINE-INTEGRABLE`, page 87.

`(ENV-SYNTAX-TABLE` *environment*`)` ⟶ *syntax-table*

Returns the syntax table associated with *environment*.

`(MAKE-SYNTAX-TABLE` *syntax-table identification*`)` ⟶ *syntax-table*

Creates a new syntax table inferior to the given syntax table. Note that syntax tables are created implicitly by `MAKE-LOCALE` (page 30).

`STANDARD-SYNTAX-TABLE` ⟶ *syntax-table*

A syntax table with entries for all standard **T** reserved words.

`STANDARD-SYNTAX-TABLE` ≡ `(ENV-SYNTAX-TABLE-ENTRY STANDARD-ENV)`

`(SYNTAX-TABLE-ENTRY` *syntax-table symbol*`)` ⟶ *descriptor* or *false* Settable

Accesses the syntax descriptor associated with *symbol* in *syntax-table*. Returns false if there is no such entry.

```
(SYNTAX-TABLE-ENTRY STANDARD-SYNTAX-TABLE 'QUOTE) ⟹ #{Syntax QUOTE}
(SYNTAX-TABLE-ENTRY STANDARD-SYNTAX-TABLE 'CAR) ⟹ false
```

Syntax table entries may be created or altered using `(SET (SYNTAX-TABLE-ENTRY ... ) ... )` or by using `DEFINE-SYNTAX`. On assignment, *descriptor* may be false, in which case *symbol* loses any syntax table entry it may have had. This allows it to be bound as a variable using `DEFINE` or `LET`, for example.

14.5 Defining syntax

`(DEFINE-SYNTAX` *symbol descriptor*`)` ⟶ *undefined*
`(DEFINE-SYNTAX (`*symbol* . *vars*`) .` *body*`)` ⟶ *undefined*

Sets *symbol*'s syntax table entry in the syntax table of the environment in which the `DEFINE-SYNTAX` form is being evaluated. The second form is an abbreviation for an equivalent expression of the first form involving `MACRO-EXPANDER`:

```
(DEFINE-SYNTAX (symbol . variables) . body)
               ≡
(DEFINE-SYNTAX symbol
               (MACRO-EXPANDER (symbol . variables) . body))
```

Macros and MACRO-EXPANDER are explained below.

As with (SET (SYNTAX-TABLE-ENTRY ...) ...), *descriptor* may be false, in which case *symbol* loses any syntax table entry it may have had. This allows it to be bound as a variable using DEFINE or LET, for example.

Note that DEFINE-SYNTAX forms have no effect at compile time. Using them indiscriminately may lead to code which behaves differently depending on what compiler is being used. For example, a use of the special form defined by a DEFINE-SYNTAX form later on in the same file in which the DEFINE-SYNTAX form occurs may be seen as a valid special form reference by the standard compiler, but may be treated as a call by ORBIT.

```
(DEFINE-SYNTAX (REPEAT N . CODE)
              ` (LET ((COUNT ,N)
                      (THUNK (LAMBDA () ,@CODE)))
                      (DO ((COUNT COUNT (- COUNT 1)))
                          ((<= COUNT 0) NIL)
                          (THUNK))))
```

14.6 Local syntax

Reserved words may be defined at compile time using LET-SYNTAX and DEFINE-LOCAL-SYNTAX. Syntax defined this way is called *local syntax* and is in effect only at compile time, not at run time.

Local syntax is block structured, much as variables are. The outermost local syntax contour is the point at which a compiler is invoked, which usually means a file boundary. Inner contours are introduced by LET-SYNTAX forms.

Put another way, a local syntax table is created whenever a compiler is invoked (LOAD, COMPILE-FILE, EVAL) and whenever a LET-SYNTAX form is compiled. Entries are created in a local syntax table at compile time for syntax defined initially by the LET-SYNTAX form and later when the compiler encounters DEFINE-LOCAL-SYNTAX forms. The syntax table is used at compile time and is otherwise unavailable.

(LET-SYNTAX *specs* . *body*) ⟶ *value-of-body*

Defines macros locally to *body*. Yields the value of *body*, an implicit block. Each *spec* should be either

(*symbol* *descriptor*)

or

((*symbol* . *vars*) . *body*)

in analogy to DEFINE-SYNTAX.

The *descriptor* and *body* forms in *specs* will not necessarily run in an environment which is at all related to the environment in which the program in which the LET-SYNTAX form occurred will be run, because compilation may occur independently of execution. ORBIT evaluates syntax-descriptors in the env-for-syntax-definition associated with the syntax table from which the descriptor was obtained, e.g. (tc-syntax-table). The standard compiler uses the locale with which the syntax table passed to it is associated. This is implementation-dependent, and subject to change. For this reason, it is best to write local macros in such a way that no free variables or special forms are used, other than those in the standard system environment, that is, those defined to be part of the **T** language.

This disclaimer does not apply to the *body* of the LET-SYNTAX form, which is evaluated (except for syntax) exactly as if the LET-SYNTAX expression were a BLOCK expression.

```
(LET-SYNTAX ((KWOTE (SYNTAX-TABLE-ENTRY STANDARD-SYNTAX-TABLE
                                         'QUOTE)))
            (KWOTE (A B C)))
            ⟹ (A B C)

(LET-SYNTAX ((SET NIL)) (LET ((SET LIST) (X 5)) (SET X 8))) ⟹ (5 8)
(LET-SYNTAX (((MAC X) ` '(X = ,X))) (MAC Y)) ⟹ (X = Y)
```

(DEFINE-LOCAL-SYNTAX *symbol descriptor*) ⟶ *undefined* Special form
(DEFINE-LOCAL-SYNTAX (*symbol* . *vars*) . *body*) ⟶ *undefined*

DEFINE-LOCAL-SYNTAX defines syntax locally to the body of the nearest enclosing LET-SYNTAX form, or, if the DEFINE-LOCAL-SYNTAX does not appear inside a LET-SYNTAX form, then to the file or outermost expression in which it occurs. Forward references are not defined to work; the DEFINE-LOCAL-SYNTAX form should appear prior to any use of *symbol* as a reserved word.

The syntax of DEFINE-LOCAL-SYNTAX is analogous to that of DEFINE-SYNTAX.

In general, DEFINE-LOCAL-SYNTAX should be used for syntax which is to be available only within the file in which it occurs. If a syntax definition is needed for several files, then they should be made available in some locale's syntax table by evaluating DEFINE-SYNTAX forms in that locale, and then that locale's syntax table should be used when compiling or loading the file (see the SYNTAX-TABLE file header clause, page 98).

> **Note:** To ease incremental debugging, the standard compiler causes syntax defined with DEFINE-LOCAL-SYNTAX to be retained indefinitely; that is, they are entered into the syntax table of the locale which was passed to LOAD. Programs should not rely on this feature, however, or code may behave differently when compiled.

14.7 Macro expanders

(MACRO-EXPANDER (*identification* . *variables*) . *body*) ⟶ *macro-expander* Special form

Yields a macro expander. A macro expander is a kind of syntax descriptor, and may therefore be stored in a syntax table. When a compiler using a symbol table S encounters a form whose car is a symbol, and the entry in S for that symbol is the object yielded by a MACRO-EXPANDER-expression, then the macro expander is invoked; that is, its *variables* are bound to the rest of form (as with one level of DESTRUCTURE binding), the *body* (an implicit block) is evaluated, and the value is returned to the compiler. The compiler then compiles that form in place of the original one.

The lexical context of *body* is that of the MACRO-EXPANDER form (augmented by the bindings of *variables*, of course), as with LAMBDA.

```
(DEFINE M (MACRO-EXPANDER (FOO X Y Z) ‘ (LIST ’FIRST ’,X ,Y ,Z)))
(INVOKE-MACRO-EXPANDER M ’(BAR QUOTED (+ 1 2) (* 3 4)))
                        ⟹ (LIST ’FIRST ’QUOTED (+ 1 2) (* 3 4))
(DEFINE L (MAKE-LOCALE STANDARD-ENV NIL))
(SET (SYNTAX-TABLE-ENTRY (ENV-SYNTAX-TABLE L) ’BAR) M)
(EVAL ’(BAR QUOTED (+ 1 2) (* 3 4)) L) ⟹ (FIRST QUOTED 3 12)
(DEFINE-SYNTAX FOO
               (MACRO-EXPANDER (FOO THING FORM)
                               ‘ (LIST ,FORM ’,THING)))
(FOO (CONS 1 2) (CONS 3 5)) ⟹ ((3 . 5) (CONS 1 2))
```

DESTRUCTURE (page 70) and quasiquote (page 71) are useful in writing macro expansion procedures, the first for taking apart the form which is to be expanded, the second for constructing the resultant code from templates.

Note that for a macro definition to take effect at compile time, it must either be present in the syntax table being used by the compiler (see page 98), or defined locally using LET-SYNTAX or DEFINE-LOCAL-SYNTAX.

(MACRO-EXPANDER? *descriptor*) ⟶ *boolean*

Returns true if *descriptor*, which must be a syntax descriptor, is a macro expander.

```
(MACRO-EXPANDER? (MACRO-EXPANDER (FOO X) X)) ⟹ true
```

(INVOKE-MACRO-EXPANDER *descriptor form*) ⟶ *new-form*

Invokes the macro expansion procedure for *descriptor*, which must be a macro expander. (See MACRO-EXPANDER, above.)

```
(INVOKE-MACRO-EXPANDER (MACRO-EXPANDER (FOO X) ‘ (LAMBDA () ,X))
                       ‘ (BAZ (+ 1 2)))
                       ⟹
(LAMBDA () (+ 1 2))
```

(MACRO-EXPAND *form syntax-table*) ⟶ *new-form*

Performs one macro expansion on the *form*, if it is a list whose car is a symbol, there is an entry in the given *syntax-table* for that symbol, and that entry is a macro expander.

Chapter 15

Ports

A *port* is any object which handles port operations. In general, ports are objects which contain pointers into sequences of objects. The port operations provide for obtaining objects from such a sequence, advancing a port's pointer, performing side effects on the sequence itself, and so forth.

T programs communicate with the external world via ports. Ports may access file systems or terminals. They may also be used to implement filters and buffers of various sorts.

Ports are manipulated using generic operations. Most of the procedures described below are generic operations, and their descriptions either specify the behavior of their default methods or of the methods for system-supplied ports. Users may create ports to their own specifications using `OBJECT`-expressions. As long as a user port supports `READ-CHAR` and `UNREAD-CHAR` (for input ports) or `WRITE-CHAR` (for output ports), most of the other I/O operations will work with them.

15.1 General

`(PORT?` *object*`)` $\longrightarrow$ *boolean* Type predicate operation

Returns true if *object* is a port.

`(INPUT-PORT?` *object*`)` $\longrightarrow$ *boolean* Type predicate operation

Returns true if *object* is an input port.

`(OUTPUT-PORT?` *object*`)` $\longrightarrow$ *boolean* Type predicate operation

Returns true if *object* is an output port.

`(INTERACTIVE-PORT?` *object*`)` $\longrightarrow$ *boolean* Type predicate operation

Returns true if *object* is an *interactive* port. An interactive port is any input port which is likely to have a human being at the other end, for example, a terminal input port.

`(EOF?` *object*`)` $\longrightarrow$ *boolean* Type predicate

Returns true if *object* is the end-of-file token. ("End-of-port" would be a more appropriate term.)

`EOF` $\longrightarrow$ *end-of-file*

This global variable holds the end-of-file token.

(WITH-OPEN-PORTS *specs* . *body*) ⟶ *object* — Special form

WITH-OPEN-PORTS has the same syntax as LET, and similar semantics. Each *spec* should be of the form (*variable port*). Each *port* expression is evaluated, and should evaluate to a port; for example, *port* would typically be an expression involving OPEN or MAYBE-OPEN. The *body*, an implicit block, is evaluated in a lexical context in which each *variable* is bound to the value of the corresponding *port* expression. The ports are closed, and the value of *body* is returned.

WITH-OPEN-PORTS is the preferred way to use port creation operations such as OPEN. It ensures that the ports will be closed, even if there is a non-local exit (a (RESET) or any other kind of throw) out of *body* or any of the *port* expressions. (See UNWIND-PROTECT, page 44.)

For an example, see OPEN, page 114.

(CLOSE *port*) ⟶ *undefined* — Operation

Closes *port*; indicates that no more input or output is intended, and that any resources associated with it may be freed.

(STRING->INPUT-PORT *string*) ⟶ *port*

Returns an input port which yields successive characters of *string* on successive READ-CHAR's.

(WITH-INPUT-FROM-STRING (*variable string*) . *body*) ⟶ *value-of-body* — Special form

Opens an input port to *string*, binds *variable* to that port, and evaluates *body*, returning whatever *body* returns.

(WITH-INPUT-FROM-STRING (*variable string*) . *body*)
≡
(WITH-OPEN-PORTS ((*variable* (STRING->INPUT-PORT *string*)))LNL . *body*)

(WITH-OUTPUT-TO-STRING *var* . *body*) ⟶ *string* — Special form

Binds *var* to an output port. The *body* (an implicit block) is evaluated, and any characters written to the port are collected in a string, which is returned as the value of the WITH-OUTPUT-TO-STRING form.

```
(WITH-OUTPUT-TO-STRING FOO (WRITE FOO '(A B))) ⟹ (A B)
```

15.2 Port switches

(TERMINAL-INPUT) ⟶ *port* — Settable

Accesses the default port for input from the terminal.

Note that an end-of-file condition on the terminal input port (see page 123) will cause the end-of-file token to be returned as the value of the next input operation on that port, but will not cause the port to become closed. This is an exception to the normal rule that no input is available from a port following an end-of-file condition on it.

(TERMINAL-OUTPUT) ⟶ *port* — Settable

Accesses the default port for output to the terminal.

(STANDARD-INPUT) ⟶ *port* — Settable

Accesses the default standard input port. Initially, this is the same as the value of `(TERMINAL-INPUT)`.

The intent is that a program could do all its input from the standard input port. By default, input would come from the terminal. But some procedure could set the standard input port to be a file, for example, and then call the program, which would automatically read from the file. This same idea is used for all the following predefined ports.

`(STANDARD-OUTPUT)` $\longrightarrow$ *port* Settable

`STANDARD-OUTPUT` accesses the default standard output port. Initially, this is the same as the value of `(TERMINAL-OUTPUT)`.

`(ERROR-OUTPUT)` $\longrightarrow$ *port* Settable

Initially, yields the same value as `(TERMINAL-OUTPUT)`.

`(DEBUG-OUTPUT)` $\longrightarrow$ *port* Settable

Initially, yields the same value as `(TERMINAL-OUTPUT)`.

15.3 Input

`(READ-CHAR` *port*`)` $\longrightarrow$ *character* or *end-of-file* Operation

`(READC` *port*`)` $\longrightarrow$ *character* or *end-of-file* Operation

Reads a single character from *port*, advances the port pointer, and returns the character read. If no character is available, then the end-of-file token is returned.

`(MAYBE-READ-CHAR` *port*`)` $\longrightarrow$ *character* or *false*

`MAYBE-READ-CHAR` when invoked on a port will return the next character if one is available; otherwise, it will return immediately with a value of false.

`(CHAR-READY?` *port*`)` $\longrightarrow$ *boolean*

`CHAR-READY?` returns true if a character is available for input; otherwise, it returns false.

`(UNREAD-CHAR` *port*`)` $\longrightarrow$ *undefined* Operation

`(UNREADC` *port*`)` $\longrightarrow$ *undefined* Operation

Backs up *port*'s pointer into its corresponding sequence of characters by one character. This causes the next `READ-CHAR` from *port* to return *character* instead of actually reading another character. *Character* is returned. `UNREAD-CHAR` is convenient for implementing one-character-lookahead scanners. Only the previous character `READ-CHAR`'ed from *port* may be put back, and it may be put back only once.

`(PEEK-CHAR` *port*`)` $\longrightarrow$ *character* or *end-of-file* Operation

`(PEEKC` *port*`)` $\longrightarrow$ *character* or *end-of-file* Operation

Returns the next character available on *port* without advancing the port's pointer. If no character is there then the end-of-file token is returned.

```
(PEEK-CHAR port) ≡
        (BLOCK0 (READ-CHAR port) (UNREAD-CHAR port))
```

(READ-LINE *port*) ⟶ *string* or *end-of-file* Operation

Reads a line of input from *port* and returns it as a string. If no input is available then the end-of-file token is returned.

(READ *port*) ⟶ *object* or *end-of-file* Operation

Reads an object from *port*. The default method simply calls READ-OBJECT, passing it the port and the port's current associated read table. See READ-OBJECT, page 93, and PORT-READ-TABLE, page 108.

(READ-REFUSING-EOF *port*) ⟶ *object*

This is like READ but should be used in cases where an end-of-file is not appropriate or is not handled, such as within the definition of a read macro.

(READ-OBJECTS-FROM-STRING *string*) ⟶ *list*

Returns a list of all the objects that could be read from the string.

```
(READ-OBJECTS-FROM-STRING "A B C ")          ⟹  (A B C)
(READ-OBJECTS-FROM-STRING " 015 ( Foo ) ")   ⟹  (15 (FOO))
(READ-OBJECTS-FROM-STRING "")                ⟹  ()
```

(CLEAR-INPUT *port*) ⟶ *undefined* Operation

Discards any buffered input for *port*. The precise action of this operation is implementation-dependent.

15.4 Output

(PRINT *object port*) ⟶ *undefined*

Prints *object* on *port* according to the current read-table. This is an operation, so objects may decide how they would like to print.

(WRITE *port object*) ⟶ *undefined* Operation

Prints the *object* on *port*. This is like PRINT with the argument order reversed. This is an operation, so particular ports may decide how they want to write objects to themselves.

(WRITE-CHAR *port character*) ⟶ *undefined* Operation

(WRITEC *port character*) ⟶ *undefined* Operation

Writes a single *character* to *port*.

(WRITE-STRING *port string*) ⟶ *undefined* Operation

(WRITES *port string*) ⟶ *undefined* Operation

Writes *string* to *port*.

(WRITE-LINE *port string*) ⟶ *undefined* Operation

Writes *string* to *port* (like WRITE-STRING), then does a NEWLINE operation.

(WRITE-SPACES *port count*) ⟶ *undefined*

Writes *count* space characters to *port*.

(DISPLAY *object port*) ⟶ *undefined*

Prints *object* on *port*, but omits any slashes, delimiters, etc., when printing strings, symbols, and characters.

(PRETTY-PRINT *object port*) ⟶ *undefined*

Pretty-prints *object* on *port*.

(NEWLINE *port*) ⟶ *undefined* Operation

Begins a new line on the given output *port*.

(FRESH-LINE *port*) ⟶ *undefined* Operation

If not at the beginning of a line, begins a new line on the given output *port*.

(SPACE *port*) ⟶ *undefined* Operation

Write whitespace to *port*. Ordinarily this will simply write a space character, but if the current output line has overflowed a "reasonable" right margin, this will do a NEWLINE.

(FORCE-OUTPUT *port*) ⟶ *undefined* Operation

Makes sure any buffered output to *port* is forced out. This is useful especially with terminal output ports. The precise behavior of this operation is implementation-dependent.

15.5 Formatted output

(FORMAT *destination control-string* . *rest*) ⟶ *string* or *undefined*

Performs formatted output. Characters in the *control-string* other than tilde (~) are simply written to *destination*. When a tilde is encountered, special action is taken according to the character following the tilde.

Destination should be one of the following:

A port. In this case, output is simply written to the port. The value returned by the call to FORMAT is undefined.

The standard true value. (This is the value of the system variable T.) This is equivalent to a port argument of (STANDARD-OUTPUT). The value returned by the call to FORMAT is undefined.

Null. In this case the output is collected in a string, as with WITH-OUTPUT-TO-STRING. This string is returned as the value of the call to FORMAT.

The FORMAT control sequences are as follows. (Case is irrelevant, so ~A and ~a behave identically.)

~A DISPLAY the next format argument.

~B Print the next argument in binary.

~D Print the next argument in decimal (radix ten).

~O Print the next argument in octal (radix eight).

~P Write the character "s" if the next argument, which must be a number, is not equal to 1 (for plurals).

~R ~nR prints the next argument in radix n.

~S PRINT the next format argument.

~T ~nT tabs to column n (HPOS).

~X Print the next argument in hexadecimal (radix sixteen).

~% Go to a new output line (NEWLINE).

~& Go to a fresh output line (FRESH-LINE).

~_ Print a space, or go to a fresh output line (SPACE).

~~ Write a tilde.

A tilde followed by any whitespace character is ignored, along with all following whitespace.

```
(FORMAT NIL "The ~s eats grass." 'ELAND) ⟹ "The ELAND eats grass."
(FORMAT NIL "The ~A eats grass." "kudu") ⟹ "The kudu eats grass."
(FORMAT NIL "~S had ~X goat ~P." '(SANDY SMITH) 31 31)
⟹ "(SANDY SMITH) had 1F goats."
```

15.6 Miscellaneous

(PORT-READ-TABLE *port*) ⟶ *read-table* Settable operation

Accesses the read table associated with *port*. See section 14.2.

(LINE-LENGTH *port*) ⟶ *integer* Settable operation

Returns the maximum width that lines read from *port* are likely to take, or that lines written to *port* ought to take.

(HPOS *port*) ⟶ *integer* Settable operation

Accesses the current horizontal position (column number) of *port*. The leftmost position on a line is column 0. When assigned, as many spaces as necessary are written to bring the horizontal position to the assigned value.

(VPOS *port*) ⟶ *integer* Settable operation

Accesses the current vertical position (line number) of *port*. The uppermost vertical position is line 0.

(MAKE-OUTPUT-WIDTH-PORT *variable* . *body*) ⟶ *integer* Settable

Binds *variable* to an output port, and evaluates *body* in the augmented lexical environment. The characters sent to the output port are not accumulated, but merely counted, and the total is returned as the value of WITH-OUTPUT-WIDTH-PORT.

(PRINTWIDTH *object*) ⟶ *integer*

Returns the number of WRITEC's which would be performed were *object* to be printed using PRINT.

(PRINTWIDTH *object*)
≡
(WITH-OUTPUT-WIDTH-PORT PORT (PRINT *object* PORT))

(DISPLAYWIDTH *object*) ⟶ *integer*

Returns the number of WRITEC's which would be performed were *object* to be printed using DISPLAY.

> (DISPLAYWIDTH *object*)
> ≡
> (WITH-OUTPUT-WIDTH-PORT PORT (DISPLAY *object* PORT)

(MAKE-BROADCAST-PORT . *output-ports*) ⟶ *port*

Returns a port which will "broadcast" all output operations to all of the *output-ports*. For example, if the port *s* is the value of a call

> (MAKE-BROADCAST-PORT *q* *r*)

then any WRITEC (WRITES, SPACE, etc.) operation to *s* will result in WRITEC operations on both *q* and *r*.

15.7 Example

Here is an example of a user-defined port. MAKE-PREFIXED-PORT returns a port which prefixes each line written to a given port with a given string.

```
(DEFINE (MAKE-PREFIXED-PORT PORT PREFIX)
        (JOIN (OBJECT NIL
                      ((NEWLINE SELF) (NEWLINE PORT) (WRITES PORT PREFIX))
                      ((PRINT SELF OPORT)
                       (FORMAT OPORT
                               #{Prefixed-port~_~S~_~S}
                               PORT
                               PREFIX)))
              PORT))
```

Chapter 16

Files

An executing **T** system interacts with the world outside by communicating with various external entities, including:

> *people*, via keyboards and displays;
>
> *file systems*, via operating system calls, possibly over a communications network;
>
> and *processes*, running on the same machine or on different machines.

The language provides simple standardized interfaces to the first two of these. Particular implementations may provide more complete interfaces.

Standard access to a terminal is provided by the ports `TERMINAL-INPUT` and `TERMINAL-OUTPUT` (page 104). Access to file systems is provided by the facilities described in this chapter.

16.1 File systems

A *file system* is a collection of named permanent objects called *files*, and a mechanism for manipulating this collection. A *file system object* is a **T** object which names or represents an actual file system. Where the distinction is unimportant, a file system object is referred to simply as a file system.

A given **T** implementation may provide access to multiple file systems; such a facility is not described in this manual. However, one may assume that at least one file system, known as the *local file system*, is accessible.

`(LOCAL-FS)` ⟶ *file-system*

> Returns the local file system object.

The following type predicates are defined to query the type of a file system.

`(AEGIS-FS?` *file-system*`)` ⟶ *boolean* — Type predicate

> Returns true if *file-system* represents an Aegis file system.

`(UNIX-FS?` *file-system*`)` ⟶ *boolean* — Type predicate

> Returns true if *file-system* represents a Unix file system.

`(VMS-FS?` *file-system*`)` ⟶ *boolean* — Type predicate

> Returns true if *file-system* represents a VMS file system.

16.2 Filenames

A *filename* is an object which names a file. Filenames are immutable record structures with the following components:

> *File system:* the file system object via which the named file is accessible.
>
> *Directory:* a collection of files within the file system to which the named file belongs.
>
> *Name:* the name of the file within the directory, or of a family of files, as further specified by the file type and generation components of the filename.
>
> *Type:* the type of the file (also known as "extension").
>
> *Generation:* a positive integer specifying a particular version or incarnation of the file. Larger generation numbers indicate newer versions.

In general, the directory, name, and type components of a file are usually symbols, the file system component is a file system object, and the generation component is an integer. The file system component may be a symbol which names a file system in an implementation-dependent manner.

Filenames may be incompletely specified; missing components are represented by having null (false) as their value. An omitted file system is usually interpreted to be the same as the local file system, an omitted directory component means the current working directory. The name component may not be omitted.

The external representation of a filename has the form

> `#[Filename` *file-system dir name type gen*`]`

where *gen* and *type* may be omitted, if null.

`(MAKE-FILENAME` *file-system dir name type gen*`)` $\longrightarrow$ *filename*
`(MAKE-FILENAME` *file-system dir name* . *type*`)` $\longrightarrow$ *filename*

> Returns a filename. The arguments become the components of the filename object. *Type* and *gen* may be omitted, in which case they default to null (absent).

`(->FILENAME` *filespec*`)` $\longrightarrow$ *filename*

> Coerces *filespec* to a filename.
>
> If *filespec* is a filename, then it is returned.
>
> If *filespec* is a list *l*, then `MAKE-FILENAME` is called, passing null as the file system argument, and the elements of the list as the rest of the arguments.
>
> If *filespec* is a symbol *x*, then it is treated the same as the list `(()` *x*`)`.
>
> If *filespec* is a string, then it is converted to a filename in an implementation-dependent way, such that `FILENAME->STRING` applied to the filename will return a string which is equal to *filespec*.
>
> For example:
>
> ```
> ->FILENAME '(MATH FACT T))   ==>   #[Filename () MATH FACT T]
> ->FILENAME '(MATH FACT))     ==>   #[Filename () MATH FACT]
> ->FILENAME 'FACT)            ==>   #[Filename () () FACT]
> ->FILENAME "fact.t")         ==>   #[Filename () () FACT T]
> ```
>
> The last example is plausible, but it will not necessarily hold in all **T** implementations, since the coercion of strings to filenames is not defined here.

`(FILENAME?` *object*`)` $\longrightarrow$ *boolean* Type predicate

Returns true if *object* is a filename.

(`FILENAME-FS` *filename*) ⟶ *file-system* or *false*

Returns the file system component of *filename*, or false (null) if it has none.

(`FILENAME-DIR` *filename*) ⟶ *symbol* or *false*

Returns the directory component of *filename*.

(`FILENAME-NAME` *filename*) ⟶ *symbol*

Returns the name component of *filename*.

```
(FILENAME-NAME '#[Filename () MATH FACT T]) ⟹ FACT
```

(`FILENAME-TYPE` *filename*) ⟶ *symbol* or *false*

Returns the type component of *filename*.

(`FILENAME-GENERATION` *filename*) ⟶ *integer* or *false*

Returns the generation component of *filename*.

(`FILENAME->STRING` *filename*) ⟶ *string*

Returns a string representing *filename* in the native syntax of *filename*'s file system (or of the local file system if *filename* is null).

If the directory component of *filename* is a symbol, then it is interpreted as a "logical name" in a manner idiosyncratic to the type of the file system:

Aegis: a **T** logical name is interpreted as being a link in the naming directory.

Unix: a **T** logical name is an environment variable.

VMS: a **T** logical name is a VMS logical name.

For example:

```
(FILENAME->STRING '#[Filename AN-AEGIS-FS MATH FACT T])
⟹ "~math/fact.t"
(FILENAME->STRING '#[Filename A-UNIX-FS MATH FACT T])
⟹ "/usr/math/fact.t"
(FILENAME->STRING '#[Filename A-VMS-FS MATH FACT T])
⟹ "MATH:FACT.T"
```

(In the Unix example, we assume that environment variable `MATH` was defined to be `/usr/math`, e.g. by a "`setenv MATH /usr/math`" shell command.)

16.3 Files

A *file* is an external permanent object stored in a file system. Files are accessed in **T** via ports. Ordinarily, files are sequences of characters, similar to strings. An input port open on an existing file delivers successive characters (or lines, or parsed objects) out of the file. An output port open on a new file deposits successive characters (or lines, or printed representations of objects) into the file.

`OPEN` and `MAYBE-OPEN` obtain ports which access files. Any port created by `OPEN` or `MAYBE-OPEN` should be closed (using the `CLOSE` operation, page 104) when no further access to the file is required. This is guaranteed if `OPEN` and `MAYBE-OPEN` are always used in conjunction with `WITH-OPEN-PORTS` (page 104), which ensures that any port opened actually gets closed, even if there is a throw out of the body of the `WITH-OPEN-PORTS` form.

(OPEN *filespec mode-list*) ⟶ *port*

Opens an external file for reading or for writing, and returns a port which accesses it. *Filespec* should be a filename or any other object which can be coerced to one (see `->FILENAME`, page 112). *Mode-list* should be a list of keywords (symbols) specifying what kind of access to the file is desired. The only keywords currently recognized are `IN`, `OUT`, and `APPEND`, indicating reading, writing, and appending, respectively. It is an error condition if the file cannot be opened for some reason.

```
(WITH-OPEN-PORTS ((PORT (OPEN file.txt '(IN))))
                 (READ-LINE PORT))
```

(MAYBE-OPEN *filespec modes*) ⟶ *port* or *false*

Like `OPEN`, but returns false if for any reason it cannot open the file.

(PORT-NAME *port*) ⟶ *filename* Operation

Returns the filename of the file on which *port* is open. This operation is not handled by any system-created ports other than those created by `OPEN` and `MAYBE-OPEN`.

(FILE-EXISTS? *filespec*) ⟶ *boolean*

Returns true if the specified file exists.

(FILE-MOVE *source-filespec dest-filespec*) ⟶ *undefined*

Moves the file named by *source-filespec* to a new location given by *dest-filespec*. In some cases, this operation can be performed without actually copying the file, for example, if the two locations are on the same "volume" of the same file system. In this case, `FILE-MOVE` is simply a rename operation. In other cases, it may be more expensive.

Bug: Not all versions of **T** 3.1 implement `FILE-MOVE`.

(FILE-DELETE *filespec*) ⟶ *undefined*

Deletes the specified file.

Bug: Not all versions of **T** 3.1 implement `FILE-DELETE`.

Chapter 17

Program structure

This chapter provides information about organizing, loading, and compiling programs.

17.1 Environment structure

Lexical environments in **T** are hierarchically arranged. Variable bindings are inherited from outer (superior) contours to inner (inferior) ones. At the top of the hierarchy is a root environment, which has no bindings in it. Inferior to that is a standard environment which has bindings for all standard system variables, for example, `CAR` and `+`. (See section 2.3.) Inferior to the standard environment are environments into which programs have been or are to be loaded. In the standard environment, the variable `STANDARD-ENV` is bound to the standard environment itself.

When a **T** system starts up, it sets up an initial environment configuration which has one environment inferior to the standard environment, called the *user environment.* The variable `USER-ENV` is bound in the standard environment to the user environment. The user environment has no variable bindings in it at first; however, the initial read-eval-print loop (section 18.3) is started in this environment, so that if no other provision is made, user global variable (i.e. definitions) will be made in the user environment.

In **T** 3.1, there is another environment called the *implementation environment* (page 130). This is the value of `T-IMPLEMENTATION-ENV` in the standard environment. The implementation environment is not inferior to the standard environment, but instead is inferior to the root.

In **T 3.0**, the initial locales look like:

```
                          <root>
                           / | \
                         /   |   \
                       /     |     \
                     /       |       \
                   /         |         \
t-implementation-env   standard-env   scheme-env
                           / | \
                         /   |   \
                       /     |     \
               user-env  orbit-env  scheme-internal-env
```

`<root>` The `<root>` locale is the conceptual root of the locale tree. It does not actually exist. The `<root>` locale is empty, it contains no variable bindings.

`t-implementation-env` The `t-implementation-env` is the environment which contains the system internals.

`standard-env` The `standard-env` is the environment defined by the T manual.

`user-env` The `user-env` is the default environment for the `read-eval-print-loop` on system startup.

`orbit-env` The `orbit-env` is the environment which contains the internals of the ORBIT compiler.

`scheme-internal-env` The `scheme-internal-env` contains the system internals for the Scheme environment.

`scheme-env` The `scheme-env` is the environment defined by Revised3 Report on Scheme.

Empty environments may be created using `MAKE-EMPTY-LOCALE` (page 30). For example:

```
(DEFINE *ALMOST-USELESS-ENV* (MAKE-EMPTY-LOCALE '*ALMOST-USELESS-ENV*))
(*DEFINE *ALMOST-USELESS-ENV* '+ +)
(*DEFINE *ALMOST-USELESS-ENV* '- -)
(EVAL '(+ 5 (- 21 13)) *ALMOST-USELESS-ENV*) ⟹ 13
```

`STANDARD-ENV` ⟶ *locale*

The value of `STANDARD-ENV` is an environment (a locale) in which all system variables have appropriate values, as described in this manual. That is, it is a *standard environment* in the sense of section 2.3.

`USER-ENV` ⟶ *locale*

The value of `USER-ENV` is an environment (a locale) inferior to `STANDARD-ENV`. It is provided by a **T** implementation as an environment in which a user may evaluate expressions and write programs. Other evaluation environments may be created inferior to `STANDARD-ENV`, however, with `MAKE-LOCALE` (page 30).

17.2 Source files

T programs are usually represented by collections of one or more text files resident in a file system. Text files containing **T** programs are called *source files.*

A source file consists of a sequence of (external representations of) **T** expressions. Source files may be *loaded* into a **T** environment. When a file is loaded, the expressions in the file are evaluated. Typically, this means that useful side-effects, such as procedure definitions, occur which then make the program available in that **T** environment.

A compilation (semantic analysis) step must occur either as a file is being loaded, or prior to loading the file. In the former case, compilation is typically performed by a compiler such as the "standard compiler" (page 97) which itself runs relatively quickly and produces intermediate code which must then be interpreted. In the latter case, an auxiliary file known as an *object file* is involved; the compilation step need only be performed once, even if the file is to be loaded many times. (The term *object file* is completely unrelated to the term *object.*)

A *file compiler* (such as ORBIT; see section 17.5) takes a source file as input and produces an object file as output. The object file may then be loaded in lieu of the source file, with the same effect.

Note: In the current implementation, the standard compiler cannot be used as a file compiler, and ORBIT cannot be used as an "on-the-fly" compiler. In principle, the two dimensions of compiler and compilation mode are orthogonal: it should be possible to use either compiler in either manner. In practice, this is not the case, but it turns out not to be too much of a problem.

17.3 File syntax

The first form in a source file must be a list whose car is the symbol `HERALD`. This form is not an expression, but rather is part of the syntax of the file. It provides information relevant to programs which operate on the source file (such as readers, compilers, and loaders). The syntax of a `HERALD`-form is as follows:

`(HERALD` *identification* `.` *items*`)`

Identification should be either `()` or a symbol identifying the file (usually the same as the root of the name of the file). It is for documentary purposes only.

Items is a sequence of lists. Each *item* should be a list beginning with a valid keyword symbol, as described below.

Example:

```
(HERALD FACT
        (READ-TABLE *MATH-READ-TABLE*)
        (ENV T (MATH MATHMACROS)))
```

`(READ-TABLE` *expression*`)` Herald item

> *Expression* should evaluate to a read table, which is used in reading the expressions which follow the `HERALD`-form from the source file. The environment in which *expression* is evaluated depends on the program processing the file (but might be, e.g., the scratch environment).
>
> If this item is absent, then expressions are read using the standard read table.

`(SYNTAX-TABLE` *expression*`)` Herald item

> *Expression* should evaluate to a syntax table, which is used in compiling (evaluating) the expressions in the source file. The environment in which *expression* is evaluated depends on the program processing the file (but might be, e.g., the scratch environment).
>
> If this item is absent, then expressions are compiled according to an appropriate syntax table: the syntax table associated with the locale into which the file is being loaded, if the file is being loaded, or the current value of `(TC-SYNTAX-TABLE)` (see below), if the file is being compiled using ORBIT .

`(ENV` *support-env-name* `.` *filespecs*`)` Herald item

> Specifies a support (early binding) environment for compiling the file. The support environment consists of the support environment named by *support-env-name*, augmented by information obtained from support files named by *filespecs*.
>
> If this item is absent, then the standard support environment, whose name is `T`, is used. The ability to create and name other support environments is not yet documented.

17.4 Loading files

`(LOAD` *filespec environment*`)` $\longrightarrow$ *undefined*

> Loads the file named by *filespec*. If the file is a source file, then each expression in the file is compiled (with the standard compiler) and run in *environment*. If the file is an object file, as produced by TC, then the compiled object code is simply run in *environment*.
>
> If no explicit file type is given in the *filespec*, then `LOAD` will load either an object file (file type `BIN`), if one exists, or a source file (file type `T`) otherwise.

`(LOAD-OUT-OF-DATE-ACTION)` $\longrightarrow$ *symbol*

> `LOAD-OUT-OF-DATE-ACTION` is a switch that controls what the loader does with a filespec without an extension. The options are:
>
> **binary** load the object file.
>
> **source** load the source file.
>
> **newer** load the most recent of object and source files.

recompile recompile the file if the source is newer than the object file.

warn give a warning if the source is newer than the object file.

query offer to recompile if the source is newer than the object file.

The default is `WARN`.

17.5 File compilation

In addition to the standard compiler invoked when `EVAL` or `LOAD` is called, **T** provides an optimizing compiler. This compiler, known as ORBIT, trades compilation speed for execution speed: `STANDARD-COMPILER` tries to compile quickly, while ORBIT tries to generate executable code which will run fast.

To help produce more efficient code, ORBIT makes assumptions about the values that some variables will have at run-time. These assumptions are called *early bindings*. For example, it will ordinarily assume that the variable `PAIR?` will have the standard `PAIR?` predicate as its top-level value. This means that if an object file produced under this assumption is loaded into a lexical environment where this is not the case, then calls to `PAIR?` will not execute the same as they would if the source file had been loaded.

Early bindings are obtained from *support environments*. The support environment to be used in compiling a file may be specified by an `ENV` clause in the file's header. (See page 117.) A support environment may contain user-defined integrable procedure and constant definitions.

Besides early binding, another source of improved efficiency is a difference in the handling of undefined effects (that is, run-time program errors). When code compiled using the standard compiler incurs an error, an error is signalled, and the error system is entered, giving the user an opportunity to debug the problem at the point where it occurs. Code compiled using ORBIT has less error-checking, so the effect of an error may go unnoticed until long after the error occurred, or a secondary error of an obscure or unexpected kind will occur.

ORBIT can be invoked from within **T** by using the following functions.

`(COMPILE-FILE` *filespec*`)` ⟶ *undefined*

Alternatively `COMFILE`. To compile a source file, say, `fact.t`, use `(COMPILE-FILE "fact")`. ORBIT writes three output files, all having the same file name as the **T** source, and distinguished by extension:

The *noise file* is a transcript of what ORBIT wrote to the terminal in the course of the compilation. ORBIT also writes some additional statistics and cross-referencing information to noise files.

The *support file* contains early binding information useful for compiling other files with ORBIT. See the `ENV` file header clause, page 117

The *object file* contains the native code for the target machine that can be loaded into **T**.

The file extensions for the output files depend on the target architecture. For MC68000, VAX, and Encore, the extensions are as follows:

Architecture	Support	Noise	Object
68000	`.mi`	`.mn`	`.mo`
VAX11	`.vi`	`.vn`	`.vo`
Encore	`.ni`	`.nn`	`.no`

`(ORBIT` *expression . locale*`)` ⟶ *undefined*

`ORBIT` does expression at a time compilation, e.g. `(ORBIT (DEFINE (F X) (+ X 1))`') will cause the expression to be compiled and loaded into the (REPL-ENV).

`(CL` *expression*`)` ⟶ *undefined*

`(CL` *expression*`)` prints out assembly code for *expression*.

`(TC-SYNTAX-TABLE)` ⟶ *syntax-table* Settable

This is the basic syntax table from which ORBIT obtains macro definitions. ORBIT obtains additional macro definitions from the support environment set up by an `(ENV ... ) HERALD` clause. Its default value is the syntax table of the user environment.

Warning: This may change in the future.

`(ENV-FOR-SYNTAX-DEFINITION` *syntax-table*`)` $\longrightarrow$ *environment* Settable

The environment in which macro definition bodies themselves are evaluated, either as a result of encountering a `DEFINE-LOCAL-SYNTAX` or `LET-SYNTAX` expression, or by loading macro definitions from a support file. ORBIT evaluates syntax-descriptors in the `ENV-FOR-SYNTAX-DEFINITION` associated with the syntax table from which the descriptor was obtained, e.g. for `(tc-syntax-table)`, the default environment is `USER-ENV`.

Chapter 18

User interface

This chapter describes the user interface to the **T** system, which is the current implementation of the **T** language. Also part of the user interface are the various debugging facilities, which are described in chapter 19.

The features described in this chapter are highly volatile and implementation-dependent.

18.1 Invoking T

To invoke a **T** system, one typically gives a command to the system command processor, as appropriate to the operating system and installation. For example, under Aegis, the following interaction might take place:

```
$ t
524280 bytes per heap, 131071 bytes reserved
T 3.1 (5) MC68000/UNIX Copyright (C) 1988 Yale University
;Loading /usr/local/lib/t/tfix5.t into T-IMPLEMENTATION-ENV
T Top level
> (list 'planner 'conniver)
(PLANNER CONNIVER)
>
```

The interaction may look slightly different in the other implementations. The above illustrates the startup sequence, which proceeds as follows:

The user invokes **T** with an appropriate command.

The **T** system identifies itself by giving the processor and operating system under which it believes itself to be running, the major and minor system version numbers, and the system edit number. The version numbers identify which release of **T** is running. The edit number is useful to the system implementors and should be provided in any bug reports sent to them.

T loads a *patch file*, if one exists. The patch file consists of forms which fix bugs in the current release of the system.

T loads a user initialization file, if one exists. The user initialization file consists of any forms which the users wants to have evaluated when **T** starts up. The location in the file system where the initialization file is found is system-dependent, but is usually the file `init.t` in the user's home directory.

T enters a *read-eval-print loop*. Read-eval-print loops are described in section 18.3.

`(STOP)` ⟶ *undefined*

Exits **T** in such a way that it may be resumed later. Control returns to the context from which **T** was invoked. Under Unix, this usually means the shell. Under VMS, this means the command interpreter (DCL). `STOP` is not defined in Aegis **T** because the Aegis environment makes it unnecessary.

(EXIT)

Exits **T**, returning control to the context in which **T** was invoked in the first place (usually a shell or other command processor). Any resources associated with the **T** process will be freed; EXIT is not reversible. A call to EXIT cannot return.

(COMMAND-LINE) ⟶ *list*

This variable has as its value the command line which was used to invoke **T**, represented as a list of strings.

18.2 Suspending T

Caution: The following is somewhat Sun specific.

A system is suspended as follows:

```
% t -h 8000000 # as big as possible
> ;; load stuff
> (gc)
> ((*value t-implementation-env 'system-suspend) filespec nil)
> (exit)
```

If a GC occurs during the suspend it will hang. This means that the heap was not big enough.

The binary distributions are a directory called "tsystem". In that directory are a bunch of files including a script called "linkt" which takes a .o file produced by suspend and creates an executable.

```
% cd tsystem
% linkt filespec.o newimage
```

If the code to be loaded before suspending a system contains DEFINE-FOREIGN forms, files containing these forms should be loaded inside of:

```
(bind (((*value t-implementation-env 'make-foreign-procedure)
        (*value t-implementation-env 'make-foreign)))
      (load file1)
      ... )
```

In this case, the linkt script must be modified to .o files that are referenced by the DEFINE-FOREIGN expressions.

System building:

It is possible to build a system from the sources. The tsystem directory contains a file, i.e. "sunbuild.t" which explains how.

18.3 Read-eval-print loops

The user usually interacts with the **T** system via a read-eval-print loop. As illustrated above, this is a command loop which repeatedly reads an expression, evaluates it, and prints the value.

The read-eval-print loop is a simple command processor; it prints a prompt, reads a command from the terminal, executes the command, then prints another prompt, ad infinitum. A *command* is any executable **T** form, and executing the command consists of evaluating the form and printing the result.

⟶ *object*

The variable `##` always has as its value the last object which was *printed* by the read-eval-print loop, that is, the value of the last expression typed by the user.

`++` ⟶ *object*

Removed in T3. The variable `++` always has as its value the last expression *read* by the read-eval-print loop. It is not assigned this value, however, until *after* the expression has been evaluated, so that a new expression may refer to the previous one.

Expressions are read from the terminal by applying the `READ` operation (actually, the value of `(REPL-READ)`; see page 124) to the terminal input port. Initially, that port's read table is the standard read table, but this may be changed using `SET`, for example:

```
(SET (PORT-READ-TABLE (TERMINAL-INPUT)) *MY-READ-TABLE*)
```

Evaluation is performed with respect to a particular variable environment and its syntax table. The read-eval-print loop may move from place to place within the environment hierarchy; one may control the evaluation environment and syntax table by setting `(REPL-ENV)`.

`(REPL-ENV)` ⟶ *environment* Settable

Accesses the environment passed to `(REPL-EVAL)` by the read-eval-print loop. Initially, this is the scratch environment (see page 116).

18.4 Command levels

Read-eval-print loops may be invoked recursively. Each currently running read-eval-print loop is said to be at a different *command level.* The initial read-eval-print loop is at *top level*, and recursive read-eval-print loops are at successively deeper levels.

The level of the current command loop is reflected in the way the loop prompts for input. At top level, the prompt is a single greater-than sign (`>`). At deeper command levels, the prompt contains as many greater-than signs as there are active read-eval-print loops; for example, `>>>` if there are two recursive read-eval-print loops beneath the top level one.

Deeper command levels are usually entered as the result of program execution errors, but may also be entered because of a keyboard interrupt or an explicit call to `BREAKPOINT`.

Interrupts: Issuing a keyboard interrupt will asynchronously enter a read-eval-print loop. One may proceed from this breakpoint, at the point where computation was interrupted, by doing `(RET)`. Keyboard interrupts are normally issued under Aegis by giving a Display Manager `DQ` command (normally assigned to control-Q), under VMS by typing control-C, or under Unix by typing the interrupt character (normally control-C or DEL).

End-of-file: An end-of-file condition on the terminal input port (see page 104) which occurs at a read-eval-print loop will cause control to transfer up one command level to the next read-eval-print loop. End-of-file can usually be generated under Aegis with the display manager `EEF` command (normally assigned to control-Z), under VMS by typing control-Z, or under Unix by typing the end of file character (normally control-D).

`(RESET)`

Transfers control directly to the top-level read-eval-print loop by performing a throw (see page 38).

`(BREAKPOINT .` *message*`)` ⟶ *object* or *undefined*

Enters a read-eval-print loop. If *message* is non-null, then it is printed on `(ERROR-OUTPUT)` using `DISPLAY`. If `(RET` *object*`)` is called, then the loop terminates and *object* is returned as the value of the call to `BREAKPOINT`. If an end-of-file condition occurs, then control is thrown to the read-eval-print loop at the next higher level, which continues.

18.5 Transcripts

A *transcript* is a record of the user's interaction with the **T** system. Transcripts are necessary in **T** implementations under operating systems which do not natively provide transcript facilities. Transcripts are not necessary, for example, when running under the Aegis Display Manager.

`(TRANSCRIPT-ON` *filename*`)` ⟶ *undefined*

> Starts writing a transcript to the specified file. All output to `(TERMINAL-OUTPUT)` and input from `(TERMINAL-INPUT)` between calls to `TRANSCRIPT-ON` and `TRANSCRIPT-OFF` is written to the file. A subsequent call to `TRANSCRIPT-OFF` will terminate the transcript, closing the file.

`(TRANSCRIPT-OFF)` ⟶ *undefined*

> Closes the active transcript file.

18.6 Customization

The various phases of the read-eval-print loop may be customized by assigning values to switches.

`(REPL-READ)` ⟶ *procedure* Settable

> Accesses the `READ` part of the read-eval-print loop. Initially, this is `READ`.

`(REPL-EVAL)` ⟶ *procedure* Settable

> Accesses the `EVAL` part of the read-eval-print loop. Initially, this is `EVAL`.

`(REPL-PRINT)` ⟶ *procedure* Settable

> Accesses the `PRINT` part of the read-eval-print loop. Initially, this is `PRINT`.

`(REPL-PROMPT)` ⟶ *procedure* Settable

> Accesses the prompting routine used in the read-eval-print loop. This should be a procedure of one argument, which is a nonnegative integer giving the command level (zero at top level, one at the next deeper level, and so on). The procedure should return a string. Initially, `(REPL-PROMPT)` is a procedure that returns `>` at the top level, `>>` at the level below that, etc.

`(LOAD-NOISILY?)` ⟶ *boolean* Settable

> If `(LOAD-NOISILY?)` is true (which it is initially), then `LOAD` and `REQUIRE` will print the value of each top-level expression in the file (standard compiler only). The output goes to the terminal output port.

`(REPL-WONT-PRINT?` *object*`)` ⟶ *boolean* Operation

> Read-eval-print loops will not print any object which answers true to this operation. Note that this is unrelated to the functioning of `PRINT` itself.

`REPL-WONT-PRINT` ⟶ *object*

> Its value is an object which answers true to the `REPL-WONT-PRINT?` predicate.
>
> `REPL-WONT-PRINT` ≡ `(OBJECT NIL ((REPL-WONT-PRINT? SELF) T))`

Chapter 19

Debugging

The facilities described in this chapter are highly implementation-specific, and subject to change without notice. Facilities described here are intended either for use directly as commands, or as utilities for user-written debugging subsystems.

19.1 Errors

When the implementation detects an error condition, a message is printed and a read-eval-print loop is entered in the dynamic context in which the error occurred. A number of facilities are available at this point for debugging and for recovering from the error.

Errors are usually detected in conditions which are left undefined by the manual (see sections 2.4 and 13.2). These conditions the following:

An unbound variable is referenced.

A procedure is called with too few or too many arguments.

A non-procedure is called.

An object of the wrong type is passed to a system procedure.

A string or vector index is out of range.

A generic operation is invoked on an object with no method to handle it.

The external representation of an object being parsed by `READ-OBJECT` or a read macro is syntactically incorrect. (This kind of error is called a *read error*.)

An object being interpreted as an expression by a compiler or a macro expander is syntactically incorrect. (This kind of error is called a *syntax error*.)

A call to `ERROR` or `UNDEFINED-EFFECT` occurs.

Once inside the read-eval-print loop, one may examine the current environment (e.g., examining variables or the stack) using the read-eval-print loop (section 18.3) or the inspector (section 19.3).

Typical actions after an error occurs :

The user corrects the error by returning a new value with `RET`.

The user goes up one command level by typing the end-of-file character (see page 123).

The user throws to top level by calling `(RESET)` (page 123).

Example:

```
> (CADR 3)
* * Error:  attempting to take CADR of 3
>> (RET '(A B))
BA
> (PLUS 3 4)
* * Error:  variable PLUS has no value
>> (RET ADD)
7
```

19.2 Debugging utilities

(TRACE *variable*) ⟶ *undefined* Special form

Reassigns *variable*, whose value should be a procedure, to be a new procedure which prints information on the (DEBUG-OUTPUT) port whenever it is called or returns a value.

(UNTRACE *variable*) ⟶ *undefined* Special form

If *variable* has a traced procedure as its value, UNTRACE restores it to its original value. Otherwise it prints a warning and does nothing.

(PP *procedure*) ⟶ *undefined* Special form

Prints the definition of *procedure* on the terminal output port. If the source code is not available, then it prints the name of a file where it can be found. (See also PRETTY-PRINT, page 107, and WHERE-DEFINED, page 129.)

(BACKTRACE) ⟶ *undefined*

Prints a one-line summary describing each continuation on the stack. See also DEBUG, page 126.

19.3 The inspector

CRAWL, also known as the *inspector*, is a stack and structure inspector. It consists of a command loop and a set of commands. The inspector keeps track of a *current object*, and a stack of objects which have previously been current objects. Some inspector commands move from one object to another.

The command loop operates as follows: the current object is "summarized" (that is, printed or displayed somehow); a prompt is printed; a line is read; and any commands on the line are executed. One can give one or more commands on a single input line.

The current object may be any object. If it is a continuation (stack frame), the name of the procedure which contains the continuation's return point is printed, and the prompt is "debug:". Otherwise, the object is printed (up to a maximum of one line of output), and the prompt is "crawl:".

The meanings of some commands vary depending on what kind of object the current object is, and not all commands are appropriate for all kinds of objects.

(DEBUG) ⟶ *undefined*

(DEBUG) enters the inspector. The current object becomes a continuation near the top of the stack (more precisely, the continuation to which the value passed to RET will be given).

`(CRAWL` *object*`)` $\longrightarrow$ *undefined*

Enters the inspector. The current object becomes *object*.

Inspector commands:

`?` *Help:* prints a list of inspector commands, with one-line summaries.

`Q` *Quit:* exits out of the inspector, usually back to the read-eval-print loop.

`U` *Up:* pops the stack of saved objects. The current object is forgotten, and the inspector moves to the previous current object.

`D` *Down:* if the current object is a continuation, moves to the next continuation deeper in the stack, that is, to the continuation that was pushed prior to the current continuation.

`X` *Exhibit:* prints useful information about the current object. The exact behavior of this varies depending on the type of the object. For structures, all of the structure's components are displayed. For continuations, any saved values are printed. Integers are printed in various radices.

For some kinds of object, such as structures and vectors, the contents of an object will be displayed in a menu form, as a sequence of lines of the form

"[*key*] *component*"

In this case, giving *key* as an inspector command will move to *component*, which then becomes the current object.

integer *Select element:* in the case of a list or vector, this moves to the appropriate component. For example, if the current object is a list, then `3` as an inspector command is the same as `A CADDDR`: it moves to the list's fourth element.

selector *Select component:* if the current object is a structure, typing a selector name will move to that component of the structure.

`B` *Breakpoint:* enters a read-eval-print loop. Executing `(RET)` will return back into the inspector. The read-eval-print loop will execute either in `(REPL-ENV)`, or, if the current object is a continuation or procedure created by interpreted code (i.e. code compiled by the standard compiler, as opposed to TC), in the lexical environment of the object. Inside the breakpoint loop, the system variable `*OBJ*` is bound to the current object.

`C` *Crawl:* moves to a new object. The object is obtained by evaluating an input expression, which is read either from the command line or in response to a prompt. The environment of the evaluation is one appropriate to the current object, as with the `B` command.

`E` *Evaluate:* evaluates an expression. The expression is read and evaluated as with the `C` command. The result of the evaluation is printed, but the current object remains the same.

`A` *Apply:* applies a procedure to the current object, and move to the result of that call. An expression evaluating to the procedure is read as with the `C` command.

`M` *Macroexpand:* performs one macro expansion on the current object, which should be a list, and the current object becomes the result of the macro expansion.

`P` *Pretty-print:* prints the current object using the pretty printer. If the current object is a continuation, then this command tries to print the expression to which the value returned by the continuation is to be supplied.

`R` *Return:* if the current object is a continuation, returns a value to it. The value is obtained as with the `C`, `E`, and `A` commands.

`W` *Where-defined:* prints the result of calling `WHERE-DEFINED` on the current object.

`V` *Unit:* moves to a template's or procedure's unit. Units are not documented, but the `X` command works with them. This command is intended primarily for the use of the **T** implementors.

The control stack which one inspects by invoking the inspector with (DEBUG) is a sequence of continuations, or stack frames. These differ from activation records in implementations of languages such as C or Lisp in that they represent future computations rather than past calls. A full call history is not easily available in tail-recursive languages such a **T** and Scheme. This is implementation-dependent, of course, and future **T** implementations may maintain call histories for debugging purposes.

Each continuation represents a control point which will make use of the value returned to the continuation. Usually these control points correspond to the arguments in a call, the predicate in an IF or COND, or any but the last subform of a block. For example, when evaluating a call, a continuation is constructed for each argument in the call, because these values will be used when the call actually occurs. In the example below, the continuations into FACT were mostly delivering values to the second argument position in the call to *.

The three-column synopsis that the inspector prints for continuations is the same as that printed by BACKTRACE (page 126). The first column is the name of the procedure into which control will return, or (anonymous) if no procedure name is available (as for procedures created by anonymous top-level LAMBDA-expressions instead of by DEFINE). The second column is the name of the source file containing the procedure, and the third column is relevant source code, if available.

Here is a sample interaction with the inspector. Commentary is in Roman font on the right. The terms "frame", "stack frame", and "continuation" are used interchangeably.

```
> (define (fact n) Define factorial function.
(cond ((= n 0) i) Bug: i instead of 1.
(else (* n (fact (- n 1))))))
#{Procedure 120 FACT }
> (fact 4)
** Error:  variable I is unbound Error detected by interpreter.
>> (debug)
#{Continuation 121} Current object is this frame.
BIND-INTERNAL THROW Internal to the implementation.
debug:  d Go down one frame.
#{Dynamic-state-transition 122} Also internal
debug:  d
#{Continuation 123}
FACT () I Okay
debug:  d
#{Continuation 124}
FACT () (* N (FACT (- N 1)))
debug:  e n Evaluate N in this frame.
1 Value is 1.
debug:  d
#{Continuation 125}
FACT () (* N (FACT (- N 1)))
debug:  e n Value is 2 in this frame.
2 debug:  d d Go down two frames.
#{Continuation 126}
FACT () (* N (FACT (- N 1)))
debug:  d
#{Continuation 127}
READ-EVAL-PRINT-LOOP REPL FACTs caller (i.e. top level).'
debug:  u u u u u Up five frames.
#{Continuation 123}
FACT () I
debug:  r 1 Return the value 1 to this frame.
24 Execution proceeds
> 24 comes out.
```

19.4 Debugging primitives

This section describes routines which may be useful in writing debugging aids. Note that they are not part of the language, and therefore should be avoided in "ordinary" programs. Relying on these routines may lead to programs which behave differently depending on how the programs are compiled (TC or standard compiler), or which fail to work across releases.

`(WHERE-DEFINED` *object*`)` ⟶ *filename* Operation

Tries to find a filename for the source file wherein *object* (usually a procedure) is defined.

`(IDENTIFICATION` *object*`)` ⟶ *symbol* or *false* Operation

If appropriate, this returns a symbol naming a variable which, in some environment, might be defined to be the *object*. If no such identification is appropriate, `IDENTIFICATION` returns false. This behavior is heuristic, not contractual; for no value is `IDENTIFICATION` required to return non-null.

```
(IDENTIFICATION CADR)                ⟹  CADR
(LET ((X CADR)) (IDENTIFICATION X))  ⟹  CADR
```

`(ARGSPECTRUM` *procedure*`)` ⟶ *pair* Operation

Returns an *argspectrum* for *procedure*. An argspectrum is a pair `(`*min* `.` *max*`)` which describes the number of arguments that *procedure* expects to receive when called. *Min* is always an integer giving the minimum number of arguments expected; *max* is either an integer giving a maximum, or it is `()`, meaning that exactly *min* arguments are required, or it is `T`, meaning that any number of arguments (but at least *min*) are acceptable.

`(DISCLOSE` *procedure*`)` ⟶ *list* or *false*

Attempts to reconstruct a source expression from which *procedure* may have been compiled. This may work for code compiled using the standard compiler, but is likely to fail to work for code compiled using ORBIT. Returns false if no source code can be obtained.

`(GET-ENVIRONMENT` *procedure*`)` ⟶ *environment* or *false*

Attempts to reconstruct an environment in which *procedure* may have been loaded (or run). This may work for code compiled using the standard compiler, but is likely to fail to work for code compiled using TC. Returns false if no environment can be obtained.

`(STRUCTURE-TYPE` *object*`)` ⟶ *stype* or *false*

If *object* is a structure, this returns the structure type of which it is an instance. If *object* is not a structure, it returns false. `STRUCTURE-TYPE` is to be used with care since it may violate the data protection otherwise provided by structure types. That is, anyone who is given a structure may find out about its internals.

`(WALK-SYMBOLS` *procedure*`)` ⟶ *undefined*

Calls *procedure* on every accessible symbol.

19.5 Miscellaneous

`*T-VERSION-NUMBER*` ⟶ *integer*

An integer which gives the version number of the currently running **T** implementation. The integer has the form

```
(+ (* major-version-number 10) minor-version-number)
```

For example, in **T3.1**, `*T-VERSION-NUMBER*` is `31`. Knowledge of the version number may be useful in dealing with incompatibilities between **T** releases, so that programs may conditionally adjust their state according to the version, and thus be able to run in both older and newer releases.

Caution: This may soon change to `T-VERSION-NUMBER`.

`T-IMPLEMENTATION-ENV` ⟶ *locale*

This environment contains variables internal to the implementation of **T**.

```
(IMPORT T-IMPLEMENTATION-ENV %%PAIR-TAG)
```

`ORBIT-ENV` ⟶ *locale*

This environment contains variables internal to the implementation of ORBIT.

`(GC)` ⟶ *undefined*

Invokes the garbage collector. Garbage collection is a low-level process by which the memory used by objects which are no longer accessible is reclaimed for use by new objects. Ordinarily, garbage collection is invoked asynchronously as the need arises, so explicit calls to `GC` are unnecessary.

A side-effect of garbage collection is that any ports created by `OPEN` which are both inaccessible and still open, are closed.

See also section 13.8.

`(GC-STATS)` ⟶ *undefined*

Prints some statistics about the most recent garbage collection.

`(GC-NOISILY?)` ⟶ *boolean* Settable

Switch, initially true. If true, then the garbage collector will print messages when invoked. If false, it will do its work silently.

`(TIME` *expression count*`)` ⟶ *undefined*

`(TIME` *expression count*`)` computes *expression count* times and prints the virtual time used. The *count* argument is optional.

Appendix A

Foreign-function Interface

The interface between T3 and the local operating system is the `define-foreign` special form:

(DEFINE-FOREIGN *T-name*
(*foreign-name parameters*)
return-type)

DEFINE-FOREIGN defines a *foreign procedure*, i.e. a **T** procedure which will call a procedure defined by the operating system or in another language.

T-name is the name of the **T** procedure being defined.

foreign-name is the name of the foreign procedure to which the *T-name* corresponds.

parameters specifies the representation of the parameters to the foreign procedure or function.

return-type indicates the representation of the value returned by the foreign procedure.

```
parameter       ⟹  (parameter-type foreign-type [parameter-name])
parameter-type  ⟹  { in | out | in/out | var | ignore }
foreign-type    ⟹  { rep/integer |
                     rep/integer-8-s |
                     rep/integer-8-u |
                     rep/integer-16-s |
                     rep/integer-16-u |
                     rep/value |
                     rep/extend |
                     rep/extend-pointer |
                     rep/string |
                     rep/string-pointer }
parameter-name  ⟹  symbol used for identification
return-type     ⟹  Aegis: { foreign-type | ignore | rep/address }
                     Unix: { foreign-type | ignore }
```

For example, on the Apollo a procedure to do block reads from a stream would be defined as follows:

```
(DEFINE-FOREIGN aegis-read
                (stream_$get_buf (in rep/integer-16-u stream-id)
                                 (in rep/string bufptr)
                                 (in rep/integer buflen)
```

```
                                 (ignore rep/integer retptr)
                                 (out rep/integer retlen)
                                 (ignore rep/extend seek-key)
                                 (out rep/integer status))
                ignore)
```

The following code will use `AEGIS-READ` to read in a string from standard input:

```
(let ((stream 0)
     (buf (make-string 128)))
(receive (5 status) (aegis-read stream buf 128 nil nil nil nil)
         (cond ((= 0 status)
                (set (string-length buf) 5)
                5)
               (error ... ))))
```

On a Unix machine a similar procedure would be defined as,

```
(define-foreign unix-read-extend (read (in rep/integer)
                                       (in rep/string)
                                       (in rep/integer))
                rep/integer)
```

To read a string from standard input on Unix the **T** code would look something like:

```
(let ((buf (make-string 128)))
     (receive (5 status) (unix-read 0 buf 128)
              (cond ((> 0 status )
                     (set (string-length buf) 5)
                     5)
                    (error ... ))))
```

A.1 Foreign Type Specification

The *foreign-type* tells the compiler how to interpret a **T** data type in order to pass it to the foreign call. The general categories of Pascal data types are numeric, string, record, enumerated, set of.

	Pascal Type	T3 Type	Foreign Type Spec
Numeric			
	integer8	fixnum	rep/integer-8-s
	binteger	fixnum	rep/integer-8-u
	integer16	fixnum	rep/integer-16-s
	pinteger	fixnum	rep/integer-16-u
	integer	fixnum	rep/integer
	linteger	fixnum	rep/integer
	real		unimplemented
	double	flonum	rep/extend
String			
	string	string	rep/string
	string	text	rep/extend
	univ_pointer	string	rep/string-pointer
	univ_pointer	text	rep/extend-pointer
Record			
	record	extend	rep/extend
Miscellaneous			
	char	char	rep/char
	boolean	boolean	rep/integer-8-s

Beware that if a **T** string is being used as an out parameter the offset field of the string must be 0 (the string must never have been `chdr!`'ed).

Record structures are represented by byte-vectors of the appropriate size.

A.2 Pascal (Apollo) Enumerated Types

Pascal enumerated types are defined using the `DEFINE-ENUMERATED` special form:

`(DEFINE-ENUMERATED` *type-name* {*element*}* `)` ⟶ *undefined* Special form

Here *type-name* is just for identification, and the *elements* are the enumerated types. For example,

```
(define-enumerated ios_$create_mode_t
                   ios_$no_pre_exist_mode
                   ios_$preserve_mode
                   ios_$recreate_mode
                   ios_$truncate_mode
                   ios_$make_backup_mode
                   ios_$loc_name_only_mode
                   )
```

The foreign procedure is called with the enumerated type name just as in Pascal.

A.3 Pascal Sets (Apollo)

The Pascal type **set-of** is defined using the `DEFINE-SET-OF` special form:

`(DEFINE-SET-OF` *type-name* {*element*}* `)` ⟶ *undefined* Special form

There, again, *type-name* is just for identification, and the *elements* are the names of the set members. For example,

```
(define-set-of ios_$put_get_opts_t
               ios_$cond_opt
               ios_$preview_opt
               ios_$partial_record_opt
               ios_$no_rec_bndry_opt
               )
```

A.4 Returned Values and Out Parameters

For languages which have output parameters, e.g. Pascal, multiple values are returned. The first value is the return-value of the foreign procedure, unless it is of *return-type* `ignore`, followed by the out parameters. Thus a call to the **T** procedure `AEGIS-READ`, defined above, would return two values: *retlen* and *status*. For a Pascal procedure the return spec will always be `ignore`. The argument to a foreign procedure should usually be of type `ignore` if it is an out parameter to the foreign procedure that is bigger than a longword. Also, the value of any out parameters which are not needed can be specified as `ignore`.

Pascal functions which return addresses **must** have *return-type* of type *rep/address*. If this value is passed to another foreign call it should be with *rep/integer*.

`DEFINE-FOREIGN` does not allocate storage for out parameters. This means that you must allocate your own object and pass it to the foreign procedure even if it is only an out parameter. If it is an out parameter which is other than an integer then its *foreign-type* should be `ignore` and the variable passed in should be used to reference the parameter.

Appendix B

Libraries

This appendix describes various software packages which are not officially part of the **T** language but which are of general utility. Some are available within the implementations directly, while others must be loaded from external files using `LOAD`.

B.1 Tables

T3.1 contains generalized hash tables. A table associates a *key* with a *value*. `MAKE-HASH-TABLE` is the most general way to make a hash table. In addition, the most common types of tables have been predefined.

> **Caution:** This feature is experimental and may go away or change in future releases.

Tables should be used in place of property lists.

(`MAKE-HASH-TABLE` *type? hash comparator gc? id*) $\longrightarrow$ *table*

`MAKE-HASH-TABLE` creates a table which associates keys to values. Any object may be a key or a value.

type? — is a predicate. All keys in the table must answer true to the predicate *type?*.

hash — is a procedure from keys to fixnums which is used to hash the table entries.

comparator — is an equality predicate on keys.

gc? — is a boolean value which specifies whether the hash procedure is dependent on the memory location(s) occupied by the object, i.e. whether or not the table must be rehashed after a garbage collection.

id — is an identifier used by the print method of the table.

(`HASH-TABLE?` *object*) $\longrightarrow$ *boolean*

`HASH-TABLE?` returns true if the *object* is a hash table.

(`TABLE-ENTRY` *table key*) $\longrightarrow$ *object* Settable

`TABLE-ENTRY` returns the *object* associated with the *key* in the *table* if there is an entry for *key*, otherwise returns *false*.

(`WALK-TABLE` *proc table*) $\longrightarrow$ *undefined*

`WALK-TABLE` invokes *proc*, a procedure of two arguments, on each *keyvalue* association in the *table*. Note that it is an error to perform any operations on the table while walking it.

The following common table types have been predefined as follows:

(MAKE-TABLE . *id*) ⟶ *table*

> MAKE-TABLE creates a *table* in which any object can be a *key* and EQV? is used as the equality predicate on *keys*.

(TABLE? *object*) ⟶ *boolean*

> TABLE? returns true if the *object* is an EQ? table.

(MAKE-STRING-TABLE . *id*) ⟶ *table*

> MAKE-STRING-TABLE creates a *table* in which the *keys* must be *strings* and STRING-EQUAL? is used as the equality predicate on *keys*.

(STRING-TABLE? *object*) ⟶ *boolean*

> STRING-TABLE? returns true if the *object* is a *string-table*.

(MAKE-SYMBOL-TABLE . *id*) ⟶ *symbol-table*

> MAKE-SYMBOL-TABLE creates a table in which the keys must be *symbols* and EQ? is used as the equality predicate on keys.

(SYMBOL-TABLE? *object*) ⟶ *boolean*

> SYMBOL-TABLE? returns true if the *object* is a *symbol-table*.

B.2 Random Integers

(MAKE-RANDOM *seed*) ⟶ *thunk*

> MAKE-RANDOM takes a *seed* which is a fixnum and returns a *thunk*. The thunk returns a new pseudo-random *integer*, x, in the range
>
> MOST-NEGATIVE-FIXNUM $\leq x \leq$ MOST-POSITIVE-FIXNUM
>
> each time it is invoked.

B.3 List utilities

(MEM *predicate object list*) ⟶ *list*

> Returns the first *tail* of *list* such that (*predicate object* (CAR *tail*)).
>
> (MEM EQ? 'A '(B A D E)) ⟹ (A D E)

(MEMQ *object list*) ⟶ *list*

> (MEMQ *object list*) ≡ (MEM EQ? *object list*)

(ANY *predicate . lists*) ⟶ *object*

Calls the *predicate* on successive elements of the *lists*, stopping as soon as the *predicate* returns a value other than false. This value is returned as `ANY`'s value.

```
(ANY NUMBER? '(A B 3 FOO))                      ⟹   true
(ANY NUMBER? '(A B C FOO))                      ⟹   false
(ANY < '(1 2 3) '(6 6 6))                       ⟹   true
(ANY MEMQ '(A B C) '((X B) (U B Z) (C D E F)))  ⟹   (B Z)
```

`(ANYCDR` *predicate . lists*`)` ⟶ *object*

Like `ANY`, but the *predicate* is called on successive tails of the *lists* instead of successive elements.

`(EVERY` *predicate . lists*`)` ⟶ *object*

Calls the *predicate* on successive elements of the *lists*, stopping as soon as the *predicate* returns false. If for every element *predicate* returns true, `EVERY` returns the value of the last call.

```
(EVERY NUMBER? '(1 2 3 -15.7)) ⟹ true
(EVERY NUMBER? '(1 2 3 FOO)) ⟹ false
(EVERY ASSQ '(A B C)
'(((A X) (X Y)) ((A J) (B K)) ((C D) (T U))))
⟹ (C D)
```

`(EVERYCDR` *predicate . lists*`)` ⟶ *object*

Like `EVERY`, but the *predicate* is called on successive tails of the *lists* instead of successive elements.

`(EVERYCDR?` *predicate . lists*`)` ⟶ *boolean*

`(EVERYCDR?` *predicate . lists*`)` ≡ `(TRUE? (EVERYCDR` *predicate . lists*`))`

`(ANYCDR?` *predicate . lists*`)` ⟶ *boolean*

`(ANYCDR?` *predicate . lists*`)` ≡ `(TRUE? (ANYCDR` *predicate . lists*`))`

`(POS` *predicate object list*`)` ⟶ *integer* or *false*

Returns the position of the first item in *list* such that (*predicate object item*), or false if there is no such item.

`(POSQ` *object list*`)` ⟶ *integer* or *false*

`(POSQ` *object list*`)` ≡ `(POS EQ?` *object list*`)`

`(APPEND-REVERSE` *list ending*`)` ⟶ *list*

```
(APPEND-REVERSE list ending) ≡
                (APPEND (REVERSE list) ending)
(APPEND-REVERSE '(A B C) '(D E)) ⟹ (C B A D E)
```

`(APPEND-REVERSE!` *list ending*`)` ⟶ *list*

```
(APPEND-REVERSE! list ending) ≡
(APPEND! (REVERSE! list) ending)
```

B.4 Type-specific arithmetic

A *fixnum* is an integer whose magnitude lies within an implementation-dependent range. This range is guaranteed to all integers which are acceptable as array, string, or vector indices. **T** guarantees that fixnums will have at least 24 bits. In **T3.1**, this range is the half-open interval

$[2^{-29}, 2^{29})$.

(FIXNUM? *object*) ⟶ *boolean* Type predicate

Returns true if *object* is a fixnum.

MOST-POSITIVE-FIXNUM ⟶ *integer*

Returns the largest integer value within the fixnum range. No integer answering true to `FIXNUM?` is greater than `MOST-POSITIVE-FIXNUM`.

MOST-NEGATIVE-FIXNUM ⟶ *integer*

Returns the smallest integer value within the fixnum range. No integer answering true to `FIXNUM?` is less than `MOST-NEGATIVE-FIXNUM`.

```
(FIXNUM? x)
         ≡
(AND (INTEGER? X)
     (>= x MOST-NEGATIVE-FIXNUM)
     (<= x MOST-POSITIVE-FIXNUM))
```

The following procedures are defined for performing type-restricted arithmetic. These procedures should be considered specializations of their corresponding generic arithmetic procedures. They assume restrictions on the types of their arguments and results. The prefix `FX` means "fixnum-specific" and `FL` means "flonum-specific." For example,

```
(FX+ x y)
≡ (ENFORCE FIXNUM? (+ (ENFORCE FIXNUM? x) (ENFORCE FIXNUM? y)))
```

T will implement calls to these type-specific procedures in a more efficient manner than calls to the corresponding generic procedures.

The following list catalogs the available routines. In most case the effect of the procedure should be analogous to the example above. The one exception is `FX/`, which is a specialization of the truncated (integer) division routine `QUOTIENT`, not of the division routine `/`.

(FX+ *fixnum1 fixnum2*)	⟶	*fixnum*	
(FL+ *fixnum1 fixnum2*)	⟶	*flonum*	
(FX- *fixnum1 fixnum2*)	⟶	*fixnum*	
(FL- *flonum1 flonum2*)	⟶	*flonum*	
(FX* *fixnum1 fixnum2*)	⟶	*fixnum*	
(FL* *fixnum1 fixnum2*)	⟶	*flonum*	
(FX/ *fixnum1 fixnum2*)	⟶	*fixnum*	
(FL/ *flonum1 flonum2*)	⟶	*flonum*	
(FX= *fixnum1 fixnum2*)	⟶	*boolean*	
(FL= *flonum1 flonum2*)	⟶	*boolean*	
(FX< *fixnum1 fixnum2*)	⟶	*boolean*	
(FL< *flonum1 flonum2*)	⟶	*boolean*	
(FX> *fixnum1 fixnum2*)	⟶	*boolean*	
(FL> *flonum1 flonum2*)	⟶	*boolean*	
(FXN= *fixnum1 fixnum2*)	⟶	*boolean*	
(FLN= *flonum1 flonum2*)	⟶	*boolean*	
(FX>= *fixnum1 fixnum2*)	⟶	*boolean*	
(FL>= *flonum1 flonum2*)	⟶	*boolean*	
(FX<= *fixnum1 fixnum2*)	⟶	*boolean*	
(FL<= *flonum1 flonum2*)	⟶	*boolean*	
(FIXNUM-REMAINDER *fixnum1 fixnum2*)	⟶	*fixnum*	
(FX-REM *fixnum1 fixnum2*)	⟶	*fixnum*	**T3**: ≡ FIXNUM-REMAINDER
(FIXNUM-ODD? *fixnum*)	⟶	*boolean*	
(FIXNUM-EVEN? *fixnum*)	⟶	*boolean*	
(FIXNUM-ABS *fixnum*)	⟶	*fixnum*	
(FIXNUM-MIN *fixnum1 fixnum2*)	⟶	*fixnum*	
(FIXNUM-MAX *fixnum1 fixnum2*)	⟶	*fixnum*	
(FIXNUM-LOGAND *fixnum1 fixnum2*)	⟶	*fixnum*	
(FIXNUM-LOGIOR *fixnum1 fixnum2*)	⟶	*fixnum*	
(FIXNUM-LOGNOT *fixnum*)	⟶	*fixnum*	
(FIXNUM-ASHR *fixnum*)	⟶	*fixnum*	
(FIXNUM-ASHL *fixnum*)	⟶	*fixnum*	
(FIXNUM->FLONUM *fixnum*)	⟶	*flonum*	
(FLONUM->FIXNUM *flonum*)	⟶	*fixnum*	

Appendix C

Other Lisps

T is a dialect of Lisp derived from Scheme and influenced by Common Lisp and NIL. **T3** provides an environment with a Scheme emulation.

C.1 Scheme Environment

Scheme is specified by Revised3 Report on Scheme. **T3** provides the environment `SCHEME-ENV` which supports some subset of Scheme.

The Scheme interpreter is entered using `SCHEME-RESET` and `SCHEME-BREAKPOINT`, and is left using `T-RESET`.

`(SCHEME-BREAKPOINT)` $\longrightarrow$ *undefined*

> Enters a Scheme read-eval-print-loop in the Scheme environment. Similar to the **T** procedure `BREAKPOINT`.

`(SCHEME-RESET)` $\longrightarrow$ *undefined*

> Enters a top level Scheme read-eval-print-loop in the Scheme environment. This is equivalent to doing `SCHEME-BREAKPOINT` followed by `RESET`.

`(T-RESET)` $\longrightarrow$ *undefined*

> Enters a top level **T** read-eval-print-loop in the **T** `USER-ENV`.

When in Scheme, `RESET` is rebound to `SCHEME-RESET`.

`(RESET)` $\longrightarrow$ *undefined*

> Transfers control directally to the top-level Scheme read-eval-print loop by performing a throw.

Switching between `SCHEME-ENV` and `USER-ENV` does not alter the bindings made in `SCHEME-ENV`.

The value of symbols in the `SCHEME-ENV` can be accessed from other environments using `*VALUE`, but it is not posible to do the reverse from `SCHEME-ENV`.

Appendix D

ASCII character conversion

The ASCII character set has nothing inherently to do with the **T** language or its implementations. Internally, any implementation is free to use any convenient encoding for characters, for example, a modified ASCII (such as that of the Lisp Machine), or EBCDIC. However, the language does provide conversions between characters and ASCII codes (`CHAR->ASCII` and `ASCII->CHAR`), so this correspondence is, for the purposes of those routines, a part of the language definition. The table below is also provided as a general reference, because people using **T** are also likely to be using computers, and this information may often be useful in that context.

8	10	16	character	8	10	16	character
0	0	0	`NULL`	40	32	20	`SPC`
1	1	1		41	33	21	!
2	2	2		42	34	22	”
3	3	3		43	35	23	#
4	4	4		44	36	24	$
5	5	5		45	37	25	%
6	6	6		46	38	26	&
7	7	7	`BELL`	47	39	27	’
10	8	8	`BS`	50	40	28	(
11	9	9	`TAB`	51	41	29	)
12	10	A	`LFD`	52	42	2A	*
13	11	B		53	43	2B	+
14	12	C	`FORMFEED`	54	44	2C	,
15	13	D	`RET`	55	45	2D	-
16	14	E		56	46	2E	.
17	15	F		57	47	2F	/
20	16	10		60	48	30	0
21	17	11		61	49	31	1
22	18	12		62	50	32	2
23	19	13		63	51	33	3
24	20	14		64	52	34	4
25	21	15		65	53	35	5
26	22	16		66	54	36	6
27	23	17		67	55	37	7
30	24	18		70	56	38	8
31	25	19		71	57	39	9
32	26	1A		72	58	3A	:
33	27	1B	`ESC`	73	59	3B	;
34	28	1C		74	60	3C	<
35	29	1D		75	61	3D	=
36	30	1E		76	62	3E	>
37	31	1F		77	63	3F	?

8	10	16	character	8	10	16	character
100	64	40	@	140	96	60	`
101	65	41	A	141	97	61	a
102	66	42	B	142	98	62	b
103	67	43	C	143	99	63	c
104	68	44	D	144	100	64	d
105	69	45	E	145	101	65	e
106	70	46	F	146	102	66	f
107	71	47	G	147	103	67	g
110	72	48	H	150	104	68	h
111	73	49	I	151	105	69	i
112	74	4A	J	152	106	6A	j
113	75	4B	K	153	107	6B	k
114	76	4C	L	154	108	6C	l
115	77	4D	M	155	109	6D	m
116	78	4E	N	156	110	6E	n
117	79	4F	O	157	111	6F	o
120	80	50	P	160	112	70	p
121	81	51	Q	161	113	71	q
122	82	52	R	162	114	72	r
123	83	53	S	163	115	73	s
124	84	54	T	164	116	74	t
125	85	55	U	165	117	75	u
126	86	56	V	166	118	76	v
127	87	57	W	167	119	77	w
130	88	58	X	170	120	78	x
131	89	59	Y	171	121	79	y
132	90	5A	Z	172	122	7A	z
133	91	5B	[	173	123	7B	{
134	92	5C	\	174	124	7C	|
135	93	5D	]	175	125	7D	}
136	94	5E	^	176	126	7E	~
137	95	5F	_	177	127	7F	DEL

Appendix E

Friendly advice

E.1 Comparison with other Lisp dialects

Some of the terminology may take some getting used to. We always say *procedure* instead of *function*, for example. *Special form* and *reserved word* have special meaning. *Syntax* usually refers not to what the reader does but to what the compiler does.

As in Common Lisp, but unlike most familiar Lisp dialects besides Scheme, **T** is lexically scoped.

As in Scheme, there is full support for lexical closures, and tail-recursive calls are reliably performed as jumps.

Generic operations are like message-passing.

See also the equivalences appendix for some rough functional analogues.

E.2 Incompatibilities

The empty list is distinguished from the symbol whose print name is `NIL`. That is, `(NEQ? NIL NIL)`'. The *value* of the variable `NIL` is the empty list. The empty list is the same as the logical false value currently, but these two will be differentiated in the future.

All "global" variables must be declared using `LSET` before they are assigned using `SET` or `BIND`. This is quite unlike most Lisp dialects where the first `SETQ` causes a variable to come into existence.

T has no `FEXPR` or `NLAMBDA` mechanism. Their effect may be accomplished using procedures or macros.

There are no `SPECIAL` declarations and no implicit dynamic binding. `BIND` must be used explicitly when a variable is to be dynamically bound.

`COND` and `CASE` don't yield nil in the fall-through case; they yield undefined values.

`RETURN` and `GO`-tags are not supported in `DO`. Lisp `RETURN` may be simulated using `(CATCH RETURN ... )`.

`NTH` is incompatible with Maclisp's function of the same name. Maclisp's takes the index as the first argument and the list as the second, instead of the other way around.

`LAST` returns the last *element* of the list, not the last *pair*, as in Maclisp. Maclisp's `LAST` is like **T**'s `LASTCDR`.

`APPEND` isn't defined to copy all but the last list. I.e. the language definition allows the implementation to yield `X`, and not a copy of it, for `(APPEND X ()`').

`PUSH`'s syntax is incompatible with that of `PUSH` in Maclisp and Common Lisp.

Locatives aren't as cheap as they are in Lisp Machine Lisp; creating a locative may involve consing.

Appendix F

Future work

This appendix catalogs some ideas about **T**'s future development. It is provided as a stimulus for user input, and as an assurance that these problems are known and are being addressed.

F.1 Language design problems

A system for managing multiple namespaces - loading files into appropriate environments, and communicating names from one module to another in a controlled way - is sorely needed.

Better aggregate structures are needed (arrays, hash tables).

A condition/error signalling system is needed.

Each special form should be marked as being either primitive or a macro. This would allow users to write robust program-manipulating programs using only the released language.

There should be a way to remove bindings from locales.

There ought to be able to have variables named by objects other than symbols. Maybe there ought to be a generalized `INTERN` which creates a unique instance of any object, or something like OWL's `UCONS` (unique cons) primitive.

Should reintroduce `DEFINE-LOCALE`?

`MEMQ`, `ASSQ`, `POSQ`, etc. are inconsistent with the other aggregate accessing routines in that the aggregate argument follows the index argument and not vice versa.

Ideas for list routines: `SUBSET`, `FILTER`, `FIND`, set operations.

Need a term more specific than "tree" for whatever those things are.

The tree manipulation routines (e.g. `ALIKEV?`) should be user-extensible. Big problem with what `TREE-HASH` should do with unusual leaf nodes.

There should be a general way to do generic dispatches based on more than one argument.

There should be a non-side-effecting way to define methods for structures.

`SYNONYM` is a misleading name.

`SETTER` is inadequate. Why not `PUSHER`, `SWAPPER`, etc.? Is there any way to do this that's at the same time convenient, general, and efficient?

`CASE` and `SELECT` should use `EQUIV?` instead of `EQ?`.

Method-clauses aren't incrementally redefinable. What to do about this. Objects handling many operations become quite unmanageable. Maybe this is an implementation and editor problem, not a language problem.

The looping/iteration constructs are maybe a little too simple. Figure out some better ones without compromising principles, if possible. Waters' `LetS` looks sort of good.

`CHECK-ARG` is ad hoc. Need type inference and other compiler smarts. Need better type declaration syntax.

There should be case-ignoring character and string comparison predicates.

`STRING-POSQ` is not a good name.

`STRING-REPLACE` should probably be called `STRING-REPLACE!`, or flushed. Similarly with `VECTOR-FILL`, `VECTOR-REPLACE`.

The name `DIGIT?` is nconsistent with `DIGIT->CHAR` - does the term *digit* mean a character or an integer?

What about `UPPERCASE?` ⟶ `UPPER-CASE?`? Ugh.

Enumerated types.

Infinities?

Should `NOT-ZERO?` and friends be renamed to be `NONZERO?` etc.?

Divulge expression syntax for quasiquote?

Maybe release `BOUND?`.

Need a way to make read tables be read-only.

Need `LET-SYNTAX-TABLE` and `LET-READ-TABLE` features?

Maybe flush the random not-parsable-as-number-implies-symbol syntax feature. It can be a real pain to catch typos in numbers.

I/O system improvements needed:

> Streams are pretty random. How does a stream differ from a sequence? There should be ways to coerce from streams to (infinite) sequences, and vice versa. Maybe the term *stream* should follow Sussman's usage (infinte sequence), not Common Lisp's (pointer into same).
>
> The `FORMAT` sublanguage is random and unextensible.
>
> Block-mode and/or "binary" I/O, for database or whatever applications.
>
> There ought to be a way to make the printer complain if it comes across an object which isn't re-readable.
>
> Ought to be a way to establish the line-length of a stream.
>
> `HPOS` and `VPOS` aren't precisely defined.
>
> `SPACE` operation needs a better definition.

Structure package improvements needed:

> `DEFINE-STRUCTURE-TYPE` features: initializers, variant record types, arguments to constructor procedure, type-restricted fields, alternate name construction.
>
> Maybe structures should have an official external syntax.
>
> Structures ought to be callable.
>
> `COPY-STRUCTURE` and `COPY-STRUCTURE!` ought to take as an additional argument the structure type to which the structure belongs. Alternatively, a copying procedure could be associated with the structure type object itself.
>
> Maybe there ought to be a way to test initializedness of structure components.

F.2 Common Lisp influence

The design and implementation of **T** began around the same time that the Common Lisp design effort started in earnest. Some of the goals of the projects have been similar; in many cases, the **T** designers left certain problems for the Common Lisp designers to work out, intentionally ignoring many sticky issues, such as numeric routines, sequences, and arrays, in order to work on other areas.

Now that the Common Lisp language has matured, and portable implementations are approaching reality, two distinct integration projects are in order: importing ideas and facilities from Common Lisp into **T** (with appropriate customization); and building a Common Lisp emulation package in **T**.

T itself will never contain Common Lisp as a subset, but it should be suitable as an implementation language for a Common Lisp. This will probably take the form of alternate evaluator, reader, printer, and namespace, together with a Common Lisp-to-**T** translator.

The following Common Lisp features, at least, should be incorporated into **T** in some form:

> Arrays.
>
> Sequences. (These would clean up a lot of the current clutter with the string and vector routines, but introduce a new level of implementation hair.)
>
> Multiple values. These can be simulated now with continuation-passing, but the syntax is clumsy.
>
> Numeric routines. Ratios and complexes. Multiple floating point precisions.
>
> `SHIFTF` and `ROTATEF`. (These are generalized versions of **T** `SWAP` and `EXCHANGE`.)
>
> Hash tables.
>
> `FORMAT` enhancements: floating-point formats, etc.
>
> File dates. Time and date manipulation. Time and date parsing.

F.3 Bugs in the implementation

This implementation (**T3.1**) doesn't implement everything described in this manual.

> Not implemented: `ATAN2`, `ROUND`, `PORT-POSITION`, and `TRUNCATE`.

Many things are implemented incorrectly.

> Deficiencies in syntax environments (they don't exist per se) and in locales (shadowing loses).
>
> `GENERATE-SYMBOL`: The implementation generates obscure names, but doesn't guarantee uniqueness. That is, the symbols are interned.
>
> `ANY?` and `EVERY?` of more than one list don't work.
>
> `GET` and `PUT` work only on symbols.
>
> `READ-LINE` and `UNREAD-CHAR` interact badly.
>
> There is no way to trace an operation.

Many things are implemented less efficiently than they ought to be.

> The garbage collector is slow. This should be fixed.
>
> There ought to be an option to invoke the standard compiler on demand, as code is run, instead of all at once, as code is loaded.
>
> `BIND` is implemented in a pretty cons-intensive way; there's a bigger performance penalty for dynamic binding than one would find in other Lisp implementations.
>
> "Lexprs" are inefficient because they always cons a list for the rest-variable.
>
> The i/o system doesn't do any buffering; this really slows down reading and printing a lot, most notably the loading of code files.

Operation dispatch could be done using hash tables. `DEFINE-OPERATION` needs to do early binding. This is feasible, but the implementation could become quite hairy.

Locales are permanent (!), and forward references to shadowed variables will lose big, especially with the standard compiler.

The interpreter conses a lot more than it ought to. It ought to do some small amount of closure analysis, or else use McDermott's dynamic migration hack.

The combinator routines are pretty useless since TC doesn't know anything about them. This should be fixed. For example, `((COMPOSE CAR CDR) (A B C)')` is evaluated, even in compiled code, by actually consing a procedure, and then invoking it.

Many things in the system are implemented less smoothly than they ought to be.

Many system routines still don't check their argument types.

Redefining `DEFINE-CONSTANT`'ed and `DEFINE-INTEGRABLE`'d should report an error condition.

Newline characters following input lines come out at the wrong place in transcript files.

`LOAD` ought to be smarter about file dates, checking for recompilation, etc.

Interpreted `QUOTE` ought to enforce read-only-ness.

There ought to be a way to enforce the downwards-only-ness of escape procedures.

Missing features.

Help system.

More system self-knowledge: who calls x, who does x call, type analysis, etc. Masterscope-like stuff.

There should be a version of `TRACE` which *destructively* modifies a procedure so that it acts traced.

Appendix G

Notes on the 4th edition

This document is still far from finished. Future work required includes the following projects.

The preface ought to have a paragraph or two describing the organization of the manual.

The "language overview" and "language principles" sections need to be expanded. There should be a description of the memory model of objects, in which objects are modeled by pointers into memory.

The form of procedure and special form description should be explained.

The section on types needs expansion and examples.

Tail recursion should be explained.

`MAKE-ECHO-PORT` should be described.

The external syntax of numbers should be described. Chapter 14 needs to give more detail.

Need to document `HANDLER` so that people know what to set a structure handler to. Need to describe keyword clauses in `OBJECT` syntax.

The introduction to the port chapter needs work. Also, port positions should be documented and implemented.

There should be a section giving a brief description of how the implementation works at the machine level. A full description should be relegated to an independent document, which should be written.

Need to discuss the VM system and reorganize the presentation of the user interface. Many more remarks on error handling and debugging are in order.

Bibliography

[RA82] Jonathan A. Rees and Norman I. Adams IV. T: a dialect of lisp or, LAMBDA: the ultimate software tool. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 114–122, August 1982.

[SS78] Guy Lewis Steele Jr. and Gerald Jay Sussman. *The Revised Report on Scheme, a Dialect of Lisp.* Technical Report, AI Lab, MIT, Cambridge, Mass., January 1978.

[Ste84] Guy L. Steele Jr. *Common Lisp: The Language.* Digital Press, 1984.

[Whi79] Jon L. White. Nil: a perspective. In *1979 Macsyma Users' Conference Proceedings*, Macsyma Users' Conference, Washington, D.C., June 1979.

[WM81] Daniel Weinreb and David Moon. *Lisp Machine Manual.* Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massacusetts, third edition edition, 1981.

Index