T

Version 3.0

Release Notes

1 Introduction

T version 3.0 is a new implementation of the T language and system. T3.0 is more efficient and more correct than previous versions. T3.0 was implemented by Norman Adams, David Kranz, Richard Kelsey, James Philbin, and Jonathan Rees.

orbit, the new optimizing compiler, is the most significant difference between version 3 and previous versions. orbit generates much better code than tc, the old compiler. It is also more correct than tc; at the time of release there were no known compiler bugs. The compiler is now built into the system; there is no seperate compiler. tc is no longer available.

The system as a whole is much faster. This is in part due to orbit and in part due to the fact that major portions of the system have been rewritten. The i/o subsystem, in particular, is much faster.

These release notes supercede the manual where applicable. A new version of the manual will be available at some future date.

Bug reports should be sent to t3-bugs.yale.edu.

2 Notational Conventions

Throughout this document the following conventions are used:

f ... g Curly braces group together whatever they enclose.

[...] Square brackets indicate that what they enclose is optional.

$^{\Lambda}$ Braces or brackets followed by a star, e.g. f ... g$^{\Lambda}$, indicate zero or more occurences of the enclosed item.

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

j A vertical bar is used to separate alternative in a braced or bracketed group, e.g. fthing$_1$ j ... j thing$_n$g.

=) is read as "evaluates to" and is used to indicate the values of various expressions in the language, e.g. (+ 1 2) =) 3.

J is read as "is equivalent to" and is used to indicate that one expression "is equivalent to" another, i.e that they have the same meaning. For example, (+ 1 2) J (+ 2 1).

Γ! is read as "has changed to" and is used to specify things that have changed from T2 to T3.

2.1 Margin Notes

Margin notes in this document give information about the type of difference between T3.0 and previous versions of T. There are seven categories of difference:

Fixed indicates errors or bugs in previous releases that have been fixed.

Extended indicates that some additional functionality has been added to the feature thus marked.

Added indicates some additional functionality that has been added to the language.

Experimental indicates new features of the language which are included on an experimental basis. Experimental features may be removed from the language at some future release.

Changing indicates a change in functionality. The previous functionality is supported in the current release but will be removed in some future release. Any code which relies on aspects of the language which are changing should be modified as soon as possible.

Changed indicates an incompatibility between T3.0 and previous versions of T. Code which uses a changed feature will either not work in T3.0 or will not work in previous versions of T.

Removed indicates functionality that has been removed from the current release, and is no longer supported. Any code which relies on removed features will no longer work.

3 Lambda

fixed

lambda-bindings no longer 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 manual (4^{th} edition). In T2 tc, the old compiler, complied with the manual but the standard compiler did not. Thus,

(let ((set list) (x 5)) (set x 8)) => 8 not (5 8)

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

(let ((set list) (x 5)) ((block set) x 8)) => (5 8)

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

4 Argument Evaluation

The evaluation order of arguments in a procedure call is undefined. This is not a language change. In code compiled by both the standard-compiler and tc the evaluation order of arguments in a combination is left to right. 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 forms whose clauses contain order dependent side effects. Because tc and the standard compiler evaluated the clauses of let forms of this sort in sequential order they produced the expected value. orbit will usually not produce the expected value. let* should be used to ensure sequential evaluation order.

5 Multiple Values

Version 3.0 of 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.

experimental

(return fvalueg$^{\Lambda}$) procedure
=> fvalueg$^{\Lambda}$

return returns its arguments as the value(s) of the current expression. In order to access

the value(s) of a return expression the value(s) must be bound to identifiers using either receive or receive-values.

For example,

(lambda () (return 1 2 3)) => 1 2 3

where "=> 1 2 3" denotes evaluates to the three values 1, 2, and 3.

Like procedures, continuations have certain expactations 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 only needs to understand these cases:

1. 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

2. In a block (begin), 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.

3. 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 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.

(receive-values receiver sender) procedure

=> 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 (identg$^{\Lambda}$) expression fbodyg$^{\Lambda}$) syntax

=> value of body

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 value(s) 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)
```

Other forms have been extended in T3.0 to allow multiple return values:

(catch identifier fbodyg$^{\Lambda}$) syntax extended

=> value of body

The identifier is bound to the continuation of the catch form, which is now an n-ary procedure. This means that catch forms can ruturn multiple values. The continuation can be invoked only during the dynamic extent of the catch form (see section 21). In T2 the continuation was a procedure of one argument. For example,

```
(catch x (list 1 (x 2 3) 4)) =>  2 3
```

(ret fvalueg$^{\Lambda}$) procedure extended

=> fvalueg$^{\Lambda}$

Returns zero or more values as the value of the current read-eval-print loop.

Note: Multiple values are implemented efficiently. It may be more efficient to use multiple values than to pass continuations.

6 Side Effects

6.1 LSET

changing 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 version 3.0 lset will continue to return a value.

6.2 SET

changing The value of the set special form is undefined, and 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. In version 3.0 set will continue to return the value being assigned to the location, but an error will be signalled in the future.

6.3 MODIFY

changing 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 version 3.0 modify will continue to return a value.

7 Canonical Boolean Values

added There is now a read syntax for canonical true and false: #F reads as the canonical false object, and #T reads as the canonical true object.

```
(true? '#t)                    =>  true
(false? (car '( #f #t))) =>  true
```

(list #f #t) is not defined, and is probably a syntax error, whereas (list '#f '#t) evaluates to (() #T).

7.1 False and the Empty List

changing

In T3.0 the canonical false value #F is not necessarily the same object as the empty list, (). nil is bound to #F. For example,

```
(cond ((cdr '(a)) 1)
      (else 2))
```

may return 2 in a future release.

In T3.0 false and the empty list will continue to be the same object, for compatibility with previous versions, but this will change in a future release. As long as #F and () evaluate to the same object null? and not will continue to be isomorphic; however, null? should be used to test for the empty list, and not should be used to test for false.

changing

It is now an error to take the car or cdr of the empty list, (). Again, for compatibility with previous versions, in T3.0 the car and cdr of () will continue to evaluate to (), but an error will be signalled in a future release.

changing

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, nil or '#F for false values.

8 Objects

fixed

The object system has been made more efficient. join now works on procedures and objects created by the object special form. It does not yet work on structures or primitive objects such as numbers and symbols.

8.1 Synonym

removed

The synonym special form has been removed.

9 Locale

removed

The locale special form has been removed from the language; however, make-locale and friends are still available.

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

10 Declare

added

A new special form declare has been added. Its syntax has not yet been released, but users should be aware that it is a reserved word.

11 Undefined Values

changing

Most procedures and special forms that have undefined values now return either no value or an explicit undefined value. See page 66 of the manual (4^{th} edition). For example, cond if no clause is selected returns an undefined value.

12 Streams and Ports

changing

In T3.0 "streams" have be renamed to "ports". This was done for compatibility with scheme and to avoid incompatibility with the use of the term stream in Structure and Interpretation of Computer Programs by Abelson and Sussman.

In accordance with this naming convention the following procedures have been renamed:

stream?	Γ!	port?
input-stream?	Γ!	input-port?
output-stream?	Γ!	output-port?
interactive-stream?	Γ!	interactive-port?
stream-read-table	Γ!	port-read-table
stream-filename	Γ!	port-name
make-output-width-stream	Γ!	make-output-width-port
make-broadcast-stream	Γ!	make-broadcast-port

13 Weak Sets

"Populations" have been renamed to "weak-sets". This change was made in the belief that "weak-set" is a more intuitive name then "population". The old names are still supported, but they will be removed in a future release. changing

make-population	Γ!	make-weak-set
population?	Γ!	weak-set?
add-to-population	Γ!	add-to-weak-set!
remove-from-population	Γ!	remove-from-weak-set!
population-¿list	Γ!	weak-set-¿list
walk-population	Γ!	walk-weak-set

In addition, two new procedures on weak-sets have been added.

(weak-set-member? object weak-set) procedure added

⇒ boolean

weak-set-member? returns true if object is a member of weak-set; otherwise, it returns false.

(weak-set-empty? weak-set) procedure added

⇒ boolean

weak-set-empty? returns true if weak-set is empty; otherwise, it returns false.

14 Syntax

define-macro has been removed from the language. Use define-syntax, define-local-syntax, and let-syntax instead. removed

The syntax of special forms is checked more thoroughly than in previous releases. Some expressions that did not cause syntax errors in previous versions of T will cause errors in T3.0. For example, changed

(lambda ()) ⇒ syntax error

In previous versions of T this invalid expression would return the value (). In T3.0 it generates an error.

changed tc-macro-definition-env has been eliminated. 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).

14.1 Read Syntax

changed Read syntax procedures now take three arguments instead of two. The first two arguments are as before; the third 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.

Note: The hack in T2.8, in which (set (read-table-entry ...) proc) would convert proc from a two-argument procedure to a three-argument procedure which ignores its third argument, has been removed.

14.1.1 Character Read Syntax

changing The #[Char ...] read syntax for characters has been changed to #[Ascii ...].

#[Char ...] Γ! #[Ascii ...]

14.2 Syntax Descriptors

extended The evaluation semantics have been extended to allow the evaluation 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.

15 Quasiquote

backquote has been renamed to quasiquote, and the backquote character, ` , now reads as quasiquote. The semantics of nested quasiquote have changed to conform with the Revised^3 Report on the Algorithmic Language Scheme. This change should not cause you problems. If you need a more thorough explanation of this change, contact the implementors. changed

quasiquote now works on vectors. Thus, extended

`#(1 2 ,(+ 1 2)) ⇒ #(1 2 3)

16 Structures

Structures can now be defined with methods. extended

(define-structure-type typename fcomponentsg$^+$ fmethodsg$^\Lambda$) syntax

⇒ stype

typename and components are handled as before. methods is an optional list of method clauses. For example,

```
(define-structure-type employee
  name
  age
  salary
  (((human? self) t)
   ((print self stream)
    (format stream "#fEmployee (~a) ~ag"
            (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.

Structures cannot yet be joined to other objects.

17 Miscellaneous

17.1 Numbers

added (rational? obj) procedure

⇒ boolean

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

added (truncatenumber) procedure

⇒ boolean

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.

17.2 Global Variables

changing The *...* convention for global variables has been changed. *...* now indicates a global, mutable variable, i.e. bound by lset. The *'s have been removed from global constants. Thus the following name changes have been made:

standard-read-table	Γ!	standard-read-table
vanilla-read-table	Γ!	vanilla-read-table
standard-syntax-table	Γ!	standard-syntax-table
eof	Γ!	eof
repl-wont-print	Γ!	repl-wont-print
number-of-char-codes	Γ!	number-of-char-codes
nothing-read	Γ!	nothing-read
standard-env	Γ!	standard-env
t-implementation-env	Γ!	t-implementation-env
scratch-env	Γ!	user-env
tc-env	Γ!	orbit-env
t-version-number	Γ!	t-version-number

17.3 Miscellaneous Name Changes

changing The following names have been changed in T3.0:

div	Γ!	quotient
div2	Γ!	quotient&remainder
min-fixnum	Γ!	most-negative-fixnum
max-fixnum	Γ!	most-positive-fixnum
fxrem	Γ!	fx-rem
comfile	Γ!	compile-file

17.4 Command Line

The global variable *command-line* has been replaced by command-line which is a nullary procedure that returns the command line that was used to invoke the system. The value returned by command-line is a list of strings. Thus, changed

```
*command-line* ⌡ (command-line)
```

17.5 Mutable Handlers

The unreleased feature of T2 that allowed handlers for structures to be mutated no longer exists. Any code using handle-stype, get-method, set-method, etc. will no longer work, but join now works efficiently; see section 8. changed

17.6 Property Lists

In T3.0 symbols no longer have property lists. Tables, see 19, provide a superset of the functionality of property lists and do not involve global state as do property lists. changed

17.7 Symbol Tables

Symbol tables as defined in T2 have been removed from the language. They have been replaced by a generalized hash table facility; see section 19. The following procedures are now defunct: changed

make-symbol
the-symbol-table
intern
really-intern
interned
interned?
walk-symbol-table

17.8 Any and Every

fixed any, any?, anycdr, anycdr?, every, every?, everycdr, everycdr? now work as advertised in the manual (4^{th} edition).

18 Other Changes

added (enforce predicate value) procedure

⇒ value

enforce returns value which must answer true to predicate. enforce is used to ensure that value is of type predicate. 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
¿
```

fixed (generate-symbolprefix) procedure

⇒ symbol

generate-symbol now ensures that the symbol returned is unique, in the sense that it was not previously interned , during the current session. Note, however, that symbols generated using generate-symbol which are written to a file during one session and then read during another session are not guaranteed to be unique.

changed ^ has been removed.

changed ## replaces **.

changed ++ has been removed.

19 Tables

experimental T3.0 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.

Note: Tables should be used in place of property lists.

(make-hash-table type? hash comparator gc? id) procedure

=) 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) predicate

=) boolean

hash-table returns true if the object is a hash table.

(table-entrytable key) settable

=) object

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) procedure

=) undefined

walk-table invokes procedure, a procedure of two arguments, on each key, value 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) procedure

⇒ 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) procedure

⇒ boolean

table? returns true if the object is an eq? table.

(make-string-table . id) procedure

⇒ 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) procedure

⇒ boolean

string-table? returns true if the object is a string-table.

(make-symbol-table . id) procedure

⇒ 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) procedure

⇒ boolean

symbol-table? returns true if the object is a symbol-table.

20 Random Integers

experimental (make-randomseed) procedure

=) 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 ¡= x ¡= most-positive-fixnum each time it is invoked.

21 call-with-current-continuation

(call-with-current-continuationproc) procedure experimental

=) value-of-proc

The procedure call-with-current-continuation packages up the current continuation as an "escape procedure" and passes it as an argument to procedure. procedure 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 Revised[3] Report on the Algorithmic Language Scheme.

22 Input and Output

(maybe-read-charport) procedure experimental

=) 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) procedure experimental

=) boolean

char-ready? returns true if a character is available for input; otherwise, it returns false.

23 Scheme

added

scheme is an embedded language in T3. For more information on Scheme see the Revised^3 Report on the Algorithmic Language Scheme. There are two ways to invoke the Scheme interpreter:

(scheme-breakpoint) procedure

⇒ undefined

scheme-breakpoint enters a Scheme read-eval-print-loop in the Scheme environment. This is similar to the T procedure breakpoint.

(scheme-reset) procedure

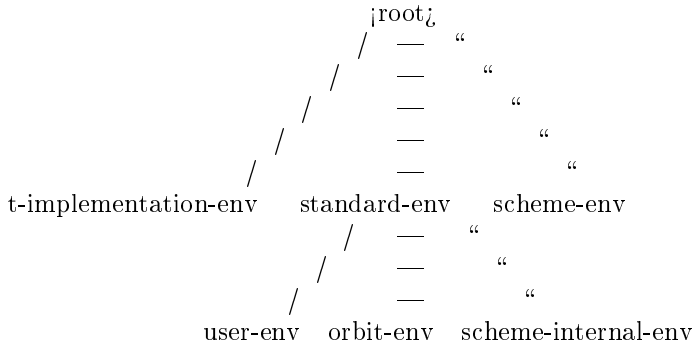
⇒ undefined

scheme-reset enters a top level Scheme read-eval-print-loop in the Scheme environment. This is similar to doing reset in the standard-env, with the exception that the read-eval-print-loop is an evaluator for Scheme.

24 The Initial Locales

extended

When the T system starts up the locale structure looks as follows:



¡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 Revised^3 Report on the Algorithmic Language Scheme.

25 Foreign Procedure Calls

The interface between T3 and the local operating system is the define-foreign special form:

(define-foreign T-name (foreign-name parameters$^+$) return-type) syntax

=⟩ undefined

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        Γ!  (parameter-type foreign-type [parameter-name])

parameter-type Γ!  f in — out — in/out — var — ignore g

foreign-type    Γ!  f 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  g

parameter-name Γ!  symbol used for identification

return-type     Γ!  Aegis: f foreign-type — ignore — rep/address g
                      Unix:  f foreign-type — ignore g
```

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 (len status) (aegis-read stream buf 128 nil nil nil nil)
    (cond ((= 0 status)
           (set (string-length buf) len)
           len)
          (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 (len status) (unix-read 0 buf 128)
    (cond ((¿ 0 status )
           (set (string-length buf) len)
           len)
          (error ...))))
```

25.1 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.

25.2 Pascal (Apollo) Enumerated Types

Pascal enumerated types are defined using the define-enumerated special form:

(define-enumerated type-name felementg$^{\Lambda}$) syntax

=) undefined

where 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.

25.3 Pascal Sets (Apollo)

The Pascal type set-of is defined using the define-set-of special form:

(define-set-of type-name felementg$^{\Lambda}$) syntax

=) undefined

where, 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
                    )
```

25.4 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.