Copyright 1989 Digital Equipment Corporation.
Distributed only by permission of Digital Equipment Corporation.

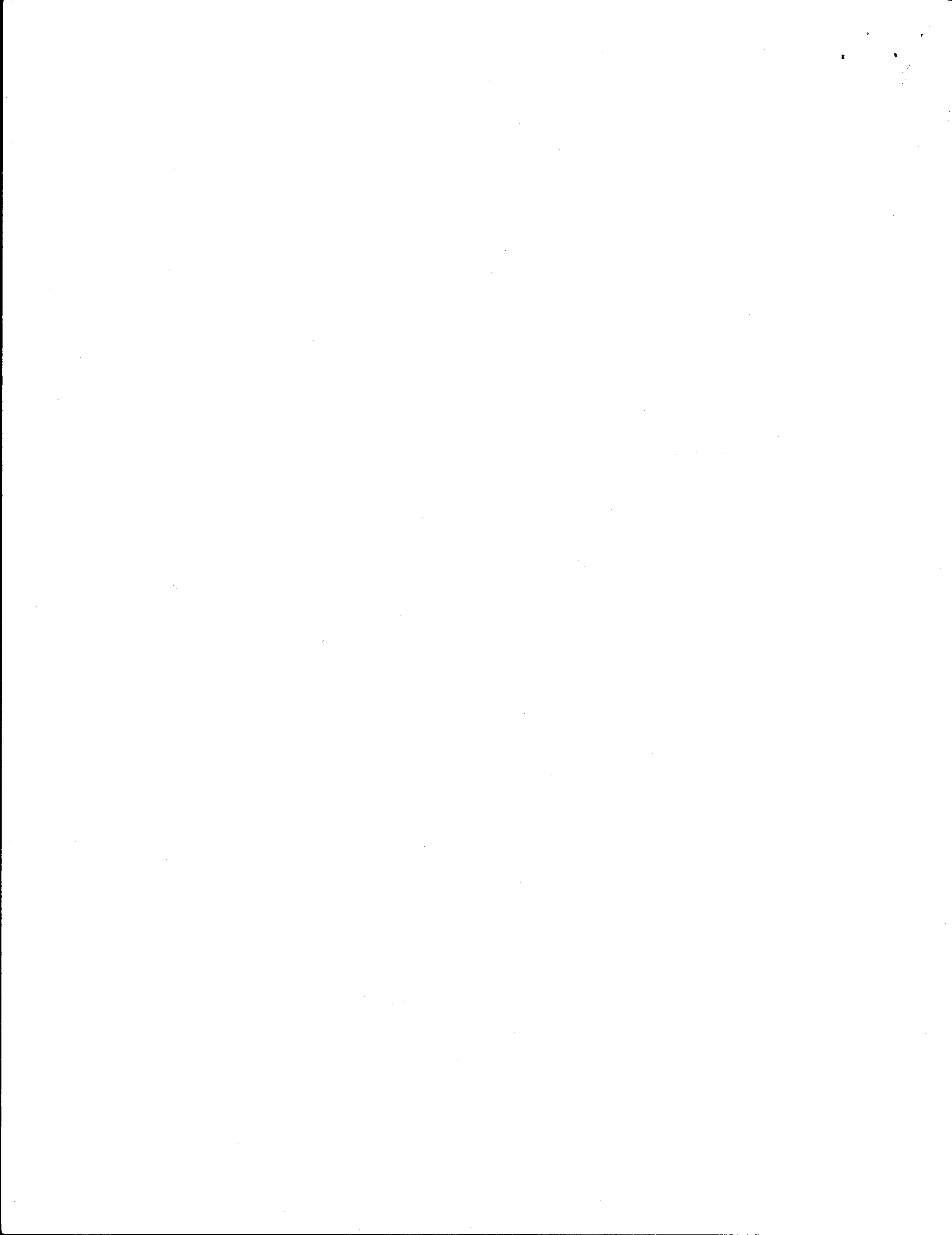
Last modified on Thu Jan 26 15:30:29 PST 1989 by glassman
modified on Sun Aug 16 16:17:02 1987 by ellis

Tinylisp Reference Manual

p.20

Table of Contents

1. Introduction 1
2. Some Examples of the Language 1
3. Symbolic Expressions 2
3.1. Whitespace and Comments 3
3.2. Numbers 3
3.3. Characters and Texts 4
3.4. Lists and Vectors 5
3.5. Symbols and Modules 5
3.6. Extended Object Syntax 6
4. Tinylisp Expressions 6
5. Formatting Style 8
6. Control Flow 8
7. Binding and Assignment 9
8. Procedures 11
9. Modules 13
10. Quoting and Backquoting 14
11. Predicates 15
12. Basic Datatypes 15
12.1. Types 16
12.2. Characters 17
12.3. Booleans 17
12.4. Numbers 17
12.5. Texts 19
12.6. Readers and Writers 20
12.7. Lists 20
12.8. Vectors 20
12.9. CharSets 21
12.10. Tables 22
12.11. Records 23
13. Exceptions 23
14. Threads and Synchronization 24
15. FOR 25
16. Dynamic Variables 28
17. Reading and Printing Symbolic Expressions 30
18. Evaluation and Read-Eval-Print Loops 31
19. Source and Object Files 31
20. Debugging 32
21. Defining Special Forms 34
22. Finding Your Way Around the Built-in Modula-2+ Packages 35
23. Including Tinylisp in an Application 37
23.1. An Example 38
23.2. The Declaration Language 38
23.3. Declaration Forms 39
23.4. Contents of the .def file 42
24. Tinylisp Performance 42
Index 44



1. Introduction

Tinylisp is a language intended for "programming-in-the-small" in SRC's Modula-2+ environment. It is a lexically scoped Lisp implemented as a package that can be bound into any Modula-2+ application, providing that application with instant programmability. The Ivy text editor uses Tinylisp to implement its predefined commands and to allow users to write their own commands; future applications may include a shell based on "vbtkit" dialogs.

The Tinylisp language itself is a small, modern Lisp that provides a fairly rich set of traditional control and data structures (including threads), with two-level naming based on modules. All the basic Modula-2+ packages are directly accessible from Tinylisp, including Text, List, Table, Thread, FileStream, Rd, Wr, OS, Time, Math, and RegExpr.

Tinylisp can directly manipulate integers, characters, booleans, longreals, and any opaque-ref types provided by the particular application, and Tinylisp can call procedures that traffic in these types. Using `compile_tli`, a stub generator similar to RPC's `flume`, application implementers define which of the application's procedures and datatypes will be accessible from Tinylisp. It is the responsibility of the application implementer to define Modula-2+ interfaces that are suitable for programming-in-the-small.

This document is not a tutorial; it assumes familiarity with Modula-2+ and its environment and some passing familiarity with Lisps. While reading, I suggest that you run `tinylisp`, a program which repeatedly reads Tinylisp expressions, evaluates them, and prints the results. Try experimenting with simple expressions and procedures as you read about the language.

2. Some Examples of the Language

Later sections provide a more exact definition of the language. Here are just a few examples to give the flavor of the language and its relationship to Modula-2+.

First, the classic factorial:

```
(DEFINE (Fact n)
    (IF (<= n 0)
        1
     ELSE
        (* n (Fact (- n 1)))))
```

Here's a procedure which enumerates through a list of texts, and returns (as a list) all those texts which contain the character 'A':

```
(DEFINE (FindA texts)
    (FOR (text IN texts)
         (WHEN (>= (Text.FindChar text 0 'A') 0))
         (LIST text)))
```

This next procedure copies one file to another, translating all characters to uppercase, and returning the output file on success or nil if the files couldn't be read or written:

```
(DEFINE (UpperFile inFile outFile)
    (TRY
        (LET (rd (FileStream.OpenRead  () inFile)
              wr (FileStream.OpenWrite () outFile))
            (TRY
                (LOOP
                    (Wr.PutChar
                        wr
                        (Char.ToUpper (Rd.GetChar rd))))
             EXCEPT Rd.EndOfFile)
            (Wr.Close wr)
            (Rd.Close rd)
           outFile)
     EXCEPT OS.Error
        ()))
```

You can find larger examples of Tinylisp in `/proj/packages/tinylisp/tl` and in Ivy.

3. Symbolic Expressions

Tinylisp source programs are represented syntactically using *symbolic expressions*. A symbolic expression is a Lisp-like data structure composed of integers, characters, booleans, reals, texts, symbols (Lisp-like atoms), modules (collections of symbols), lists, vectors, and any other ref types supplied by Modula-2+ clients. Symbolic expressions can be read and printed using the same syntax. Printing an expression and then reading it back in will produce an identical or isomorphic symbolic expression. The Sx interface completely defines symbolic expressions, and provides `Read` and `Print` procedures.

While we normally talk about the printed representation of symbolic expressions, it's important to remember that Tinylisp itself is defined in terms of the expressions themselves, not their printed representation.

If you are familiar with Lisps and their s-expression syntax, you may wish to skip this section on the first reading.

The basic symbolic expression types are represented using the Modula-2+ ref types:

integer	`Ref.Integer`
longreal	`Ref.LongReal`
boolean	`Ref.Boolean`
character	`Ref.Char`
text	`Text.T`
symbol	`SxSymbol.T`
module	`SxModule.T`
list	`List.T`
vector	`Ref.Vector`

Examples of symbolic expressions:

```
13        -4.0e9       "hello world"       'a'      Wire.T
(10 23 45)
[15 32 -4]
(Employee (Salary 10000) (Pension 4a) (Name "John R. Ellis"))
```

Overview of symbolic expression syntax:

```
-23              a decimal integer
0f3H             a hex integer
2.3e9            a longreal
#True            the boolean TRUE
#False           the boolean FALSE
#Undefined       the undefined value
'a'              a character
"hello"          a text
Hello            a symbol in the current module
X.Y              a public symbol Y in the module X
X..Z             a public or private symbol Z in the module X
X.               the module X
()               nil
(e... )          a list of expressions
[e... ]          a vector of expressions
{Wire 3}         read/print syntax for client-supplied types
#<Rd "a.out">    print syntax for client-supplied types that can't be read
#| ... |#        block comment
#" ... "#        a block text, with newlines allowed
```

3.1. Whitespace and Comments

Arbitrary amounts of whitespace between expressions (spaces, tabs, newlines, returns, formfeeds, comments) are ignored.

Block comments are indicated with:

```
#| This comment can go anywhere |#
```

Block comments nest.

3.2. Numbers

Decimal integers have syntax similar to Modula-2+, except they include an optional sign:

```
integer        -> ["-" | "+"] digit+
digit          -> "0" | "1" | ... | "9"
hexinteger     -> ["-" | "+"] digit hexdigit* ("H" | "h")
hexdigit       -> digit | "a" | "A" | ... | "f" | "F"
```

Examples:

```
23    -1    0FF3AH    -4FH    0FFFFFFFFH
```

(An unsigned hex integer represents a 32-bit unsigned quantity; a "-" in front of a hex integer yields the two's complement of the unsigned quantity.)

Longreals have the syntax:

```
longreal    -> ["-"|"+"] digit+ "." digit* [exponent]
            -> ["-"|"+"] "." digit+ [exponent]
            -> ["-"|"+"] digit+ exponent
exponent    -> ("E"|"e") [["-" | "+"] digit+]
```

Examples:

```
1.5    -.3    12e5    -12e-11
```

3.3. Characters and Texts

Characters have Modula-2+ syntax:

```
'a'    '\n'    '\''
```

The set of recognized escapes is:

`\n`	newline
`\t`	tab
`\r`	return
`\f`	formfeed
`\b`	backspace
`\e`	escape
`\ddd`	the ASCII character represented by octal digits ddd
`\\`	the character \
`\'`	the character '
`\"`	the character "
`\x`	the character "x" for any other graphic x

Texts also have Modula-2+ syntax:

```
"hello world"
```

Texts have the same escapes as characters. Only printable ASCII characters are allowed in texts and characters (`' '` .. `'~'`); to get any other character into a text, use the \ escape.

Block texts have the syntax:

```
#"Hello world"#
```

Block texts may extend across lines (the newlines will be included as part of the actual text); the only recognized escape is `\"#`, which includes a `"#` in the text. Block texts are useful for constructing texts that otherwise would have multiple levels of escaping (for example, RegExpr patterns).

3.4. Lists and Vectors

Lists are indicated with parentheses surrounding an arbitrary number of sub-expressions separated by whitespace:

```
(1 2 3)        (1 ('a' 'b') "hello" 3.0)
```

NIL is represented using the empty list:

```
()
```

Vectors are like lists, but using brackets:

```
[1 2 3]     [(1 2) 3 [4 5] ]
```

The empty vector [] is not the same as NIL; it is a vector with 0 elements.

3.5. Symbols and Modules

Symbols are like traditional Lisp atoms, unique objects with print names stored in lookup tables. Printing a symbol and then reading it back in will yield the exact same symbol. Any sequence of non-white-space characters that aren't otherwise interpreted as integers, reals, delimiters, etc. are taken to be a symbol name:

```
Hello    funny-bone    &    -    +    !=
```

Special characters can be included in a symbol's name by using \, which just forces the following character to be treated as if it were a letter:

```
A\ B     \3\1
```

The first example is the symbol with the name "A B", the second is the symbol with the name "31", not the integer 31. (The use of \ in symbols is different from its use in texts.)

To avoid symbol name conflicts between applications, there are multiple naming spaces for symbols called *modules*. A module is simply a named table mapping names onto symbols. Some symbols are *public*, accessible outside the module, and others are *private*, accessible only in that module. The fully qualified read/print name of the public symbol `Y` in the module `X` is:

```
X.Y
```

The full read/print name of the private symbol `Z` in the module `X` is:

```
X..Z
```

The module `X` itself is referenced as:

```
X.
```

Module names have the same syntax rules as symbol names. There is a separate, flat space of module names. An attempt to reference a module that doesn't exist is an error.

For fully qualified symbol names of the form `X.Y` or `X..Y`, module `X` must exist and contain a symbol `Y`, and in the case of `X.Y`, the symbol must be public. The form `X..Y` allows access to both public and private symbols in `X`.

Symbolic expressions are always read relative to a *current module*. An unqualified symbol name (one without an explicit module) is looked up in the current module, and if not found, a new private symbol of that name is created in the module.

Modules can inherit other modules for the purpose of name lookup. Unqualified symbol names are looked up first in the current module, then in its ancestors (in depth first order). If the the name is not found, a new symbol of that name is created in the current module.

A symbol `X.Y` can be *imported* into some other module `M`, so that the symbol can be named with both `X.Y` and `M.Y`. But the symbol continues to be owned by `X`, and it will always print as `X.Y`.

Printing of expressions is always done relative to a current module. The module name of a symbol `X.Y` will be omitted if that symbol is accessible from the current module, that is, if the reader, given the same current module and the input "Y", would return the symbol `X.Y`.

It is possible to have symbols owned by no module (for example, if a symbol is deleted from a module). Such symbols print as:

```
.X
```

For more details on symbolic expressions and symbols and modules, see the Sx, SxModule, and SxSymbol interfaces.

3.6. Extended Object Syntax

The `{...}` notation provides a way of smoothly extending the syntax to include other ref types. For example, suppose I have a Modula-2+ ref type called Wire that has two parts, a name and a length. I could define a read/print syntax for Wires that looks like:

```
{Wire name "ground" length 35}
```

Using the facilities in SxSyntaxTable, my program could register a print procedure for the type Wire that would print Wires out using the above syntax, and it could register a "curly" read procedure to read in the same syntax.

For some kinds of objects, a read syntax doesn't make sense. Such objects are printed with the `<...>` notation. For example, readers (`Rd.Ts`) can't be read, so they print as:

```
#<Rd.T 0dff3a04H>
```

The reader will raise an error if it tries to read such an object.

4. Tinylisp Expressions

Tinylisp is an expression language; every construct returns a value and may be used wherever an expression is allowed.

Tinylisp expressions are represented as symbolic expressions (see section 3, page 2).

The simplest Tinylisp expression is a constant, any symbolic expression that is not a list or symbol. Examples:

integers:	`32   -15   0fH`
booleans:	`#True   #False`
chars:	`'a'   '\n'`
longreals:	`3.   4.5e9   -1.2334e-4`

texts: `"Hello world\n"`

Symbols represent the names of Tinylisp variables:

```
x
Employees.table
+
```

Variables are bound to locations where values can be stored, and the value of a variable is the current contents of its location. Several Tinylisp forms create lexically nested variable scopes.

If a list begins with one of a small set of reserved symbols, it is interpreted as a *special form* with special evaluation rules. Examples:

```
(QUOTE (1 2 3))
(IF boolean 3 ELSE 4)
(PROC (a b) (+ a b))
```

By convention, the names of special forms are in uppercase.

All other lists represent procedure application:

```
(f e...)
```

The value is the result of applying the value of expression `f`, which should be a procedure, to the values of the expressions `e...`. The expressions `f` and `e...` are always evaluated, but the evaluation order is unspecified. If `f` doesn't evaluate to a procedure, or if the wrong number of arguments are given, an exception will be raised.

Examples of procedure application:

```
(+ 3 4 5)
(Wr.PutChar wr 'a')
(Initialize)
((Table.Get table keyword) arg1 arg2)
```

By convention, each word in a procedure name is capitalized.

Both Tinylisp and Modula-2+ are strongly typed, but unlike Modula-2+, typechecking in Tinylisp occurs at runtime when expressions are evaluated. The exception `System.NarrowFault` is raised whenever a value of the wrong type is encountered during evaluation. In this manual, I use the phrase "evaluates to a boolean" or "evaluates to an integer" to indicate that a specific type is expected.

Because Tinylisp is built on top of Modula-2+ and can call arbitrary Modula-2+ procedures, any Modula-2+ exception can be raised during evaluation.

A simple meta-syntax is used below to describe valid Tinylisp expressions. As an example, look at the syntax for `IF`:

```
(IF b1 e1... [ELSIF bi ei...]... [ELSE en...] )
```

Uppercase names like `IF` represent themselves; lowercase names like `b1` and `e1` are placeholders for arbitrary Tinylisp expressions. Brackets enclose optional items, and "`...`" means the previous item can occur 0 or more times.

Many of the examples given show both the expression and the result of its evaluation; to avoid confusion, I use "=>" as a shorthand for "evaluates to the value":

```
(+ 7 2)              =>  9
(List.Tail '(a b c)) =>  (b c)
```

5. Formatting Style

Lisp syntax can be just as readable as Algol-style syntax, provided it is formatted properly. The examples given for each construct defined below illustrate proper style. Also, the Ivy text editor provides a Tinylisp pretty-print command that implements an acceptable style; use it.

6. Control Flow

`(IF b1 e1... [ELSIF bi ei...]... [ELSE en...] )`

A traditional if-elsif-else. The expressions `bi` are evaluated to booleans in turn until one evaluates to true, and then the corresponding `ei` are evaluated in order. If no `bi` evaluates to true and the `ELSE` is present, the `en` are evaluated. The value of the `IF` is the value of the last `ei` or `en` evaluated; it is undefined if no `ei` are evaluated. Examples:

```
(IF (< x 2) (:= x 4) (P x y))
(IF (< x 2)
     (:= x 4)
 ELSIF (= x 2)
     (:= x 5)
     (P x y)
 ELSIF (= x 3)
     (Q x y)
 ELSE
     (:= x (+ x 1)))
```

`(& e...)`

Conditional "and". Returns true if all the expressions evaluate to true, false otherwise. Evaluation of the expressions goes from left to right and stops as soon as one evaluates to false. Example:

```
(& (> x 4) isFirst (TestFlag y))
```

`(| e...)`

Conditional "or". Returns false if all the expressions evaluate to false, true otherwise. Evaluation of the expressions goes from left to right and stops as soon as one evaluates to true.

```
(CASE e
    [(v v... => e...)]...
    [(=> e...)] )
```

Evaluates `e` and then each expression `v` in turn, until the first one that is equal (=) to `e`, or until the default arm is reached. The corresponding `e...` of the selected arm are then evaluated, and the value of the last is the value of the `CASE`. An exception is raised if `e` doesn't equal any `v` and there is no default arm. Example:

```
(CASE (Fact n)
    (0 1 2 =>
        (+ n 1))
    (4 =>
        (:= n 4)
        (F n x))
    (=>
        (* n n)))
```

```
(CASEQ e
    [(v v... => e...)]...
    [(=> e...)] )
```

Like **CASE**, except that == is used instead of =.

```
(LOOP [label] e...)
```

Repeatedly executes the expressions **e...**. Executing **EXIT** terminates the loop, supplying a value for the loop expression. Example:

```
(LOOP
    (IF (Rd.EOF rd) (EXIT))
    (Wr.PutChar wr (Rd.GetChar rd)))
```

```
(EXIT)
(EXIT e)
(EXIT label e)
```

Exits the innermost loop with the value of **e**, or the named loop if a **label** is given. If no **e** is given, the value is undefined.

```
(ASSERT b e...)
```

If **b** evaluates to false, the expressions **e...** are evaluated, and an assertion-failed exception is raised with the resulting list of values. Example:

```
(ASSERT (== 1 ()) "The list 1 is non-nil" 1)
```

See also **FOR**, section 15, page 25.

7. Binding and Assignment

```
(LET ( [var v]... )
    e...)
```

Defines a lexical scope containing the new variables **var** initialized to the values **v**, which are evaluated outside of the scope in unspecified order. The expressions **e...** are evaluated in the scope, the value of the last one becoming the value of the **LET**. Example:

```
(:= x 4)
(LET (x (+ 1 2)
      y (+ x 1))
    (+ x y))          =>  8
```

```
(LET* ( [var v]... )
    e...)
```

Like **LET** except that each **v** is evaluated in the scope of the preceding variables **var**. Equivalent to:

```
(LET (var1 v1)
    (LET (var2 v2)
        ...
            (LET (varn vn)
                e...) ))
```

Example:

```
(:= x 4)
(LET* (x (+ 1 2)
       y (+ x 1))
    (+ x y))          =>  7
```

```
(LET  ( [pattern v]... )
    e...)
(LET* ( [pattern v]... )
    e...)
```

In addition to simple variable binding, `LET` and `LET*` also provide a very simple form of list pattern matching, called *destructuring*. For example:

```
(LET ((x y) `(3 4))
    (+ x y))
```

evaluates to 7; `x` is bound to the first element of `(3 4)`, and `y` to the second.

The patterns can be

A simple variable, which is initialized to the corresponding value.

`()`, representing a dummy variable or placeholder that will match any corresponding value.

A list of patterns `(p1 p2 ... pn)`. The expression `v` must evaluate to an n-element list, and the sub-patterns `pi` are matched against the corresponding elements.

A list of patterns `(p1 p2 ... pn-1 @ pn)`. The expression `v` must evaluate to a list of at least `n-1` elements. The sub-patterns `p1` through `pn-1` are matched against the first `n-1` elements, and `pn` is matched against the rest of the list (which may be empty).

Example:

```
(LET ((x y @ rest) `(a b c d))
    (List.List rest y x))

=> ((c d) b a)
```

`(:= var e)`

Sets the value of the variable `var` to be the value of `e`, and returns that value.

`(:= form e)`

Syntactic sugar for updating aggregate structures. The recognized forms and their equivalents are:

```
(:= (List.First l) e) -> (List.SetFirst l e)
```

```
(:= (List.Tail l)  e) -> (List.SetTail l e)
(:= (List.Nth l i) e) -> (List.SetNth l i e)
(:= (List.NthTail l i) e) -> (List.SetNthTail l i e)
(:= (@ v i) e  )-> (Vector.Set v i e)
(:= (record-accesser r) e)
```

-> updates a field of a record **r** with the value **e**, returning **e**.

`(:= $x e)` -> sets the value of the dynamic variable **$x** to be **e**, returning **e**.

(DSET pattern e)

Evaluates **e**, assigns to the variables in **pattern** the matching values in **e**, and returns the value of **e**. The patterns are the same as in **LET**. Example:

```
(DSET (x y @ z) '(1 2 3 4))  =>  (1 2 3 4)
x                            =>  1
y                            =>  2
z                            =>  (3 4)
```

(SET var e)

Assigns the variable **var** the value of expression **e**. This is a primitive form for building constructs like **:=**.

(DEFINE var e)

Assigns the value of **e** to the global variable **var**, whose name must be a symbol in the current module. (As a convention, **DEFINE** should be used in preference over **:=** to initialize global variables; also, **DEFINE** prevents you from redefining a pre-defined name such as **Read**.)

8. Procedures

(PROC (var...) e...)

Evaluates to an unnamed procedure with formal parameters **var...** (which must be symbols) and body **e...**. When invoked, the procedure evaluates the expressions **e...** and returns the value of the last one. The body of **PROC** is closed over the outermost, top-level scope, not over any enclosing procedures or **LET**s. That is, any free names inside the body always refer to global variables (that's why **PROC** is not called **LAMBDA**). Example:

```
(PROC (x y)
    (P x y)
    (IF (<= x y) x ELSE y))
```

(PROC var e...)

Evaluates to an unnamed procedure that accepts any number of formal parameters. The list of parameters is bound to the single formal **var** (a symbol). For example, this procedure takes any number of arguments and returns their sum:

```
(PROC l
    (LET (sum 0)
        (LOOP
            (IF (== l ()) (EXIT sum))
            (:= sum (+ sum (List.First l)))
            (:= l (List.Tail l)))))
```

(RETURN)
(RETURN e)

Returns from the lexically enclosing procedure with the value of **e** (undefined if **e** isn't given). Normally it isn't necessary to use **RETURN**, since the value of a procedure is the value of the last expression in its body.

(DEFINE (p var...) e...)

Defines the global variable **p** (a symbol) to be a named procedure with formal parameters **var...** and body **e...**. The name **p** must be a symbol in the current module. Example:

```
(DEFINE (PrintObject wr object)
    (IF (== wr ()) (:= wr $so))
    (PrintF wr "#<Object %t>" (ObjectName object))
    ())
```

(DEFINE (p var1 ... varn-1 [@ varn]) e...)

Defines a procedure with possibly a variable number of arguments, allowing formal parameter destructuring (as in **LET**). Formal parameters **var1** through **varn-1** are destructured-bound to the first **n-1** arguments (**n-1 >= 0**), and **varn** is bound to the list of the remaining arguments. This is equivalent to:

```
(DEFINE p
        (PROC temp
            (LET ((var1 ... varn-1 [@ varn]) temp)
                e...)))
```

For example:

```
(DEFINE (AddTime (secs1 usecs1) (secs2 usecs2))
    `(,(+ secs1 secs2) ,(+ usecs1 usecs2)))

(Time.Now)          =>  (555401027 388671)
(AddTime (Time.Now) `(1 500000))
                    =>  (555401028 888671)
```

(Apply p l)
(Apply p e1 e2... en l)

The first form applies procedure **p** to arguments taken from list **l**:

```
(p l1 l2... ln)
```

where **l** has the form **(l1 l2... ln)**.

The second form applies procedure **p** to:

```
(p e1 e2... en l1 l2... ln)
```

Examples:

```
(Apply + ())                 =>  0
(Apply + '(1 2 3))           =>  6
(Apply + 1 2 3 ())           =>  6
(Apply + 1 2 3 '(4 5 6))     =>  21
```

9. Modules

All identifier names in Tinylisp are represented as symbols. To avoid the name-conflict problems of a flat name space, symbolic expressions provide *modules*, named collections of symbols. Superficially, modules provide the same two-level naming of identifiers that Modula-2+ provides. Reread the section on symbolic expressions for the basic concepts of symbols and modules. See SxModule for other operations on modules.

Tinylisp has two distinguished modules, `Lisp.` and `Work.`. `Lisp.` contains the special forms and procedures that form the core of Tinylisp (for example, `IF`, +, and `:=`); by convention, every other Tinylisp module inherits `Lisp.`, allowing those names to be referenced without qualification. The reader, printer, and read-eval-print loop use `Work.` as the default current module; `Work.` thus serves as a workspace that won't affect other modules.

The dynamic variable `$module` contains the current module used by the Tinylisp reader and printer (initially `Work.`). You can change the current module of a read-eval-print loop using the `MODULE` form (preferable to using `:=`).

`(MODULE name [INHERITS module module...])`

> Creates a module with the given **name** (which must be a text) if it doesn't already exist and makes it the current module. By default, the module inherits symbols from `Lisp.`.
>
> If the optional `INHERITS` is present, then the module inherits symbols from the given modules. (If the module previously existed, then its list of inherited modules is changed to those given.)

`(PUBLIC symbol...)`

> Marks each of the symbols as public, allowing them to be accessed from outside the module by using their fully qualified form, e.g. `Paragraph.Fill`. The symbols must be local to the current module.

`(IMPORT symbol...)`

> Imports each of the external symbols into the current module, obviating the need to fully qualify them within the current module. The symbols continue to be owned by their original module.

`(SHADOW symbol...)`

> If the symbols do not already exist locally in the current module, then they are created, possibly "shadowing" any inherited symbols of the same name.
>
> For example, the procedure `Lisp.Load` is accessible simply as `Load` within most modules, since most modules inherit `Lisp.` by default. But if a module tries to define its own procedure `Load`, an error will be raised, since `Load` is not local to that module. So the module must first do `(SHADOW Load)` to create a local symbol of the same name. This indicates to readers that a globally accessible name has been redefined in

this module.

Here's an example of a trivial module that has one public procedure, one private procedure, and two private global variables.

```
(MODULE "Stack")

(PUBLIC Push)

(DEFINE stack (Vector.New 10))
(DEFINE top   -1)

(DEFINE (Push x)
    (ASSERT (! (Full)) "Stack overflow")
    (:= top (+ top 1))
    (:= ( stack top) x)
    x)

(DEFINE (Full)
    (= top (Vector.High stack)))
```

10. Quoting and Backquoting

`(QUOTE e)`

Evaluates to the symbolic expression **e** itself. This provides constant constant lists and symbols, which otherwise would be interpreted according to the normal Tinylisp evaluation rules:

```
(QUOTE (a b c))    =>  (a b c)
(QUOTE employee)   =>  employee
```

`BACKQUOTE` (`` ` ``) is usually preferable to `QUOTE`.

```
`e    (BACKQUOTE e)
,x    (UNQUOTE x)
,@l   (UNQUOTE-SPLICING l)
```

`BACKQUOTE` provides an easy, template-based method of constructing lists and vectors. The expression `` `e `` is eqivalent to `(QUOTE e)` if it doesn't contain any occurrences of `,x` or `,@l`.

Each occurrence of the form `,x` within the expression `` `e `` is replaced by the value of the expression `x`. And each occurrence of the form `,@l` is replaced by the value of the expression `l`, which must be a list, with its parenthesis "stripped away" before insertion.

Examples:

```
(:= a 3)                         =>  3
(:= b `(4 5 6))                  =>  (4 5 6)
`(a ,a (,(+ a 4) "hello")  =>  (a 3 (7 "hello"))
`(1 2 3 ,@b 7)                   =>  (1 2 3 4 5 6 7)
`[,b ,@b "fun"]                  =>  [(4 5 6) 4 5 6 "fun"]
```

`` `e `` is equivalent to `(BACKQUOTE e)`, `,x` to `(UNQUOTE x)`, and `,@l` to `(UNQUOTE-SPLICING l)`.

While the `{...}` syntax for objects such as records and other aggregate types may occur inside backquote templates, the forms `,x` and `,@l` may

not occur inside those objects.

11. Predicates

`(! x)`

Evaluates **x** to a boolean and returns its negation.

As in most languages, including Modula-2+, there are two common kinds of "equality", structural isomorphism and object identity.

`(= x y)`
`(List.Equal x y)`

Compares the structure of **x** with the structure of **y**, returning true iff one of the following is true:

`(== x y)` (**x** and **y** are the same object)

x and **y** are lists or vectors whose elements are =

x and **y** are texts with the same characters

x and **y** are booleans, integers, characters, or longreals with the same value

x is a longreal and **y** is an integer that, converted to a longreal, has the same value (or vice versa)

`(!= x y)`

`(! (= x y)).`

`(== x y)`

Returns true iff **x** and **y** are the same identical object, that is, if their representation is the same Modula-2+ ref. (This is the same as Modula-2+ =.) == is currently an order of magnitude faster than =. Nil, booleans, symbols, and characters are represented as unique objects, so == may be used to compare them. But integers, longreals, and texts do not have unique representations, so = should be used to compare them.

`(!== x y)`

`(! (== x y)).`

12. Basic Datatypes

12.1. Types

Types are first-class objects in Tinylisp. The basic symbolic expression and Tinylisp types are:

```
Boolean.T        SpecialForm.T
Char.T           SxModule.T
Exception.T      SxSymbol.T
Integer.T        Text.T
List.T           Thread.T
LongReal.T       Type.T
Nil.T            Undefined.T
Procedure.T      Vector.T
```

(Actually, these are global variables whose values are the types, but for convenience we use the global variable name.)

The type of `#Undefined` is `Undefined.T`. The type of `()` is `Nil.T`.

`(Type.Of e)`

Returns the type of `e`.

`(NARROW e t)`

Evaluates `e` and `t`, raising an exception if `e` is non-nil and isn't of type `t`. Returns `e`.

`(NARROWN e t)`

Evaluates `e` and `t`, raising an exception if `e` doesn't have type `t`. If `e` is nil, then `t` must be `Nil.T`. Returns `e`.

```
(TYPECASE e
    [(t t... => e...)]...
    [(=> e...)] )
```

Equivalent to:

```
(CASEQ (Type.Of e)
    [(t t... => e...)]...
    [(=> e...)] )
```

Unlike the Modula-2+ `TYPECASE`, nil doesn't automatically go to the first arm; use an explicit `Nil.T` to catch nil. Example:

```
(TYPECASE x
    (Nil.T =>
        ())
    (Integer.T =>
        (+ x x))
    (List.T =>
        (Recurse (List.First x))
        (Recurse (List.Tail x))))
```

12.2. Characters

`Char.T`

`Char.First`
`Char.Last`

The first and last characters (in ASCII order).

`(Char.Ord char)`

Returns the ASCII value of `char`.

`(Char.Val i)`

Interprets integer `i` as an ASCII code and returns the corresponding character.

`(Char.ToUpper char)`

Returns the uppercase version of `char` if it is lowercase, returns `char` otherwise.

`(Char.ToLower char)`

Returns the lowercase version of `char` if it is uppercase, returns `char` otherwise.

`(Char.ToControl char)`

Returns the control-character version of `char`, equivalent to `(Val (BitAnd (Ord char) 01FH))`.

12.3. Booleans

`Boolean.T`

`#True`

`#False`

12.4. Numbers

Tinylisp provides two number types, integers and longreals. Integers are 32-bit Modula-2+ `INTEGER`s, and longreals are 64-bit Modula-2+ `LONGREAL`s.

`Integer.T`

`LongReal.T`

`Integer.First`
`Integer.Last`

The smallest and largest integers.

Unless stated otherwise, the following procedures operate on both integers and longreals. If the operands are all integer, the result is an integer. If the operands are all longreals or mixed integers and longreals, the integers are converted to longreals and the result is a longreal.

`(+ x...)`

Addition. `(+)` returns 0.

`(* x...)`

Multiplication. `(*)` returns 1.

`(- x...)`

Subtraction.
`(-)` returns 0.
`(- e)` returns `0-e` (negation).
`(- e1 e2 ... en)` returns `e1-e2-...-en`.

`(/ x...)`

Division.
`(/)` returns 1.
`(/ e)` returns `1/e`.
`(/ e1 e2 ... en)` returns `e1 / e2 / ... / en`.

`(Min x...)`

Minimum. `(Min)` returns `Integer.Last`.

`(Max x...)`

Maximum. `(Max)` returns `Integer.First`.

`(Rem x y)`

The remainder of `|x| / |y|`, with the sign of the result the same as `x`.
If `x` or `y` are longreals, they are truncated to integers first.

`(Mod x y)`

Mathematical mod:

if `y # 0`, `x - y * Floor( x / y )`

if `y = 0`, `x`.

`(Abs x)`

The absolute value of `x`.

`(Floor x)`

The largest integer `i` such that `i <= x`.

`(Ceiling x)`

The smallest integer `i` such that `i >= x`.

`(Trunc x)`

`x` rounded towards 0.

`(Round x)`

`x` rounded to the nearest integer.

`(Float x)`

`x` converted to longreal.

```
(< x y)
(<= x y)
(= x y)
(>= x y)
(> x y)
```

Arithmetic comparisons.

The following bit operations operate on the 32-bit two's complement represention of integers and return integers:

`(BitNot x)`

Ones complement of `x`.

`(BitAnd x y)`

Logical-and of `x` and `y`.

`(BitOr x y)`

Logical-or of `x` and `y`.

`(BitXor x y)`

Logical-exclusive-or of `x` and `y`.

`(BitShiftLeft x n)`

`x` shifted left by `n` bits, with zeroes inserted on the right.

`(BitShiftRight x n)`

`x` shifted right by `n` bits, with zeroes inserted on the left.

`(BitExtract n o s)`

Returns bits `o` through `o+s-1` as a two's complement integer.

`(BitInsert n o s d)`

Returns `d` with bits `o` through `o+s-1` replaced by the `s` least significant bits of `n`.

12.5. Texts

See the Text interface for full details. Differences in the Tinylisp interface:

`Text.T`

`(Text.Cat text...)`

Concatenates any number of texts.

`(Text.Compare text1 text2 [ignoreCase])`

Compares the texts `text1` and `text2`, returning one of the symbols `Base.Lt`, `Base.Eq`, `Base.Gt`.

12.6. Readers and Writers

All the procedures from Rd and Wr are available to Tinylisp; see those interfaces for details.

12.7. Lists

See the List interface for full details. Differences in the Tinylisp interface:

`List.T`

`(List.List e...)`

Returns a list of any number of arguments.

`(List.Append l...)`

Appends any number of lists.

`(List.AppendD l...)`

Destructively appends any number of lists.

`(List.TTail l)`

`(List.Tail (List.Tail l))`

`(List.TTTail l)`

`(List.Tail (List.Tail (List.Tail l)))`

`(List.SetFirst l e)`

Destructively sets the first element of a list to be `e`.

`(List.SetTail l e)`

Destructively sets the tail of a list to be `e`.

```
(List.Sort  l [compareProc compareProcArg])
(List.SortD l [compareProc compareProcArg])
```

Sorts (or destructively sorts) a list `l`. The `compareProc` should return one of the symbols `Base.Lt`, `Base.Eq`, `Base.Gt`. See the List interface for more details.

12.8. Vectors

Vectors are one-dimensional 0-based arrays (Modula-2+ `Ref.Vectors`).

`Vector.T`

```
(@ v i)
(Vector.Get v i)
```

Returns element `i` from vector `v`, raising `System.NarrowFault` if `i` isn't an integer or is out of range.

```
(:= (@ v i) e)
(Vector.Set v i e)
```

Sets the value of element `i` of vector `v` to be `e`.

`(Vector.New size)`

Returns a new vector of `size` (an integer) elements.

`(Vector.Number v)`

The number of elements in vector `v`.

`(Vector.High v)`

`(- (Vector.Number v) 1)`

`(Vector.Copy v)`

Returns a new vector with the same elements as `v`.

`(Vector.Expand v size)`

Returns a new vector with `(Max size (Number v))` elements, with the first `(Number v)` elements copied from `v`.

12.9. CharSets

CharSets are Modula-2+ `Ref.CharSets`. You construct a CharSet with a pair of curlies (see section 3.6, page 6) and a text that specifies the members. If you need to specify hyphen explicitly as part of the character set, you have to put it first in the text (and that initial hyphen can't begin a sequence).

`{CharSet.T "aeiou"}`

`{CharSet.T "a-z"}`

`{CharSet.T "-,.\"'"}`

`(CharSet.Incl charSet char)`

Returns a new CharSet containing all the elements of `charSet` plus `char`.

`(CharSet.Excl charSet char)`

Returns a new CharSet containing all the elements of `charSet` except `char`.

`(CharSet.Union charSetA charSetB)`

Returns a new CharSet containing all the elements of `charSetA` and `charSetB`.

`(CharSet.Difference charSetA charSetB)`

Returns a new CharSet containing all the elements in `charSetA` but not in `charSetB`.

`(CharSet.Intersection charSetA charSetB)`

Returns a new CharSet containing all the elements in both `charSetA` and `charSetB`.

`(CharSet.SymDifference charSetA charSetB)`

Returns a new CharSet containing all the elements in either `charSetA` or `charSetB` but not in both.

`(CharSet.Equals charSetA charSetB)`

Returns `#True` if all the elements in `charSetA` are also elements of `charSetB`; otherwise `#False`.

(CharSet.Distinct charSetA charSetB)

Returns **#True** if at least one element is in either of the two CharSets but not in both; otherwise **#False**.

(CharSet.Subset charSetA charSetB)

Returns **#True** if every element of **charSetA** is also an element of **charSetB**; otherwise **#False**.

(CharSet.Superset charSetA charSetB)

Returns **#True** if every element of **charSetB** is also an element of **charSetA**; otherwise **#False**.

(CharSet.In charSet char)

Returns **#True** if **char** is an element of **charSet**; otherwise **#False**.

12.10. Tables

The Tinylisp interface to the Table package is slightly different from the Modula-2+ interface because of **VAR** parameters. See Table for details on these procedures and on the other procedures not described here.

Table.T

(Table.New [hashProc hashProcArg compareProc compareProcArg initialSize maxChainLength])

Creates a new **Table.T**, by default using the procedures **List.Hash** and **List.Compare** to compare arbitrary s-expression keys using isomorphic structure equality (=).

(Table.Get table key)

Returns the value associated with **key** in the table, returning **#Undefined** if not found.

(Table.Put table key value)

Puts a key/value pair in the table, returning true iff the key was previously in the table.

(Table.Delete table key)

Deletes a key/value pair from the table, returning the old value if the key was present, **#Undefined** if not.

Table.RefHash
Table.RefCompare

Tables created with these hash and compare procedures will object equality (==) for comparing keys.

12.11. Records

Though it is not possible to access Modula-2+ record types directly, Tinylisp does provide a simple way to define new record types that can be used only from within Tinylisp.

(DEFINE-RECORD r field... [(field...)])

Defines the global variable r to be a record type with the given field names; r must a symbol in the current module, and the field names must also be symbols. A special-form accesser whose name is r:field is defined for each field, and a special form r:New is defined for creating new instances of that type.

Example:

```
(DEFINE-RECORD Employee name salary age)

(:= e (Employee:New))              =>  {Employee}
(:= (Employee:name e) "John")      =>  "John"
(:= (Employee:age e) 30)           =>  30
e
    =>  {Employee name "John" age 30}
(Employee:name e)                  => "John"
(Employee:salary e)                => ()
```

The r:New form allows the specification of one or more field names with associated initial values; missing fields are initialized to nil. Example:

```
(Employee:New name "John" age (+ 30 1))
    => {Employee name "John" age 31}
```

The optional (field...) at the end of a DEFINE-RECORD form specifies which fields, if any, should be suppressed when records of that type are printed. Normally, the printer only suppresses nil-valued fields.

13. Exceptions

A Tinylisp exception is a first-class object of type Exception.T. Otherwise they have the same semantics as Modula-2+ exceptions.

(DEFINE-EXCEPTION exception)
(DEFINE-EXCEPTION exception ())

Defines a new, named exception and assigns it to the global variable exception, whose name must a symbol in the current module. The second form defines an exception with a single parameter. Example:

```
(DEFINE-EXCEPTION IllegalCharacter)
```

(RAISE exception)
(RAISE exception e)

Evaluates exception to an exception and raises it. The value e is evaluated and passed as a parameter to the exception if it requires one.

(TRY e... FINALLY f...)

Like the Modula-2+ TRY FINALLY. The expressions e... are evaluated in turn, and the value of the last one is returned. The expressions f.. are evaluated on exit from the TRY, even if the exit is caused by a

procedure **RETURN**, a loop **EXIT**, or an exception. Example:

```
(LET (rd (FileStream.Open () "/etc/passwd"))
    (TRY
        (ReadPasswordFile rd)
     FINALLY
        (Rd.Close rd)))
```

```
(TRY
    e...
 [EXCEPT exception
    e...]...
 [EXCEPT (exception exception... ())
    e...]...
 [EXCEPT (exception exception... var)
    e...]...)
```

Similar to the Modula-2+ **TRY EXCEPT** statement. The expressions in the body are evaluated, and if no exceptions are raised, the **TRY** exits with the value of the last expression. If one of the **EXCEPT** clauses catches a raised exception, then the expressions of that clause are evaluated and the **TRY** exits with the value of the last one. The second form of the **EXCEPT** clause allows multiple exceptions to be named on the same **EXCEPT** clause. The third form allows the specification of a formal parameter that will be bound to the argument of the caught exception; that parameter is lexically scoped over that one **EXCEPT** clause only.

The `exceptions` must be symbols naming exceptions, or else the symbol **ELSE**, which catches all exceptions.

Example:

```
(TRY
    (:= wr (FileStream.OpenWrite () "/tmp/output"))
    (LOOP (Wr.PutChar wr (Rd.GetChar rd)))
 EXCEPT Rd.EndOfFile
    (Wr.Close wr)
 EXCEPT (OS.Error ec)
    (PrintF $se "Error opening file: %t\n"
            ( OS.errMessage ec)))
```

```
(TRY e... PASSING (exception exception...))
```

Evaluates the expressions `e...` and returns the value of the last one. If any exceptions are raised, all except the named ones are converted to `System.Fail`.

14. Threads and Synchronization

Tinylisp provides threads, mutexes, and condition variables, and the operations in the Thread interface. Differences in the Tinylisp interface:

`Thread.T`

`Thread.Mutex`

`Thread.Condition`

`(LOCK m e...)`

Like the Modula-2+ **LOCK** statement. Evaluates **m** to a mutex, acquires the mutex, evaluates the expressions **e...**, releases the mutex, and returns the value of the last expression. Equivalent to:

```
(LET (temp m)
    (TRY
        (Thread.Acquire temp)
        e...
     FINALLY
        (Thread.Release temp)))
```

(Thread.NewMutex)

Returns a new, initialized mutex.

(Thread.NewCondition)

Returns a new, initialized condition.

(Thread.GetCPUTime thread)

Returns the CPU time of a thread as a pair **(seconds microseconds)**.

(Thread.AllThreads)

Returns all the existing threads as a **List.T**.

15. FOR

(FOR [label] clause...)

A higher-level looping construct for uniformly manipulating sequences (lists, vectors, texts, readers, integer ranges, or client-supplied sequences). One or more of the clauses below can be present, and they can be composed. **EXIT** can be used to terminate the loop prematurely but not to return a value -- only the result clauses can return a value for the loop. Read the examples at the end first to get the flavor.

Don't try to fit a square peg into a round hole -- if a complicated loop doesn't immediately fit into the **FOR** sequence idiom, use **LOOP** instead.

***

Initialization clauses:

(VAR [var v]...)

Initializes the new variables **var** to the values **v**, evaluating each **v** in the scope of the previous **vars** (like **LET***). The variables exist only in the scope of the **FOR**.

***

Iteration clauses: These iterate a control variable through the elements of a sequence. The **FOR** expression defines a new lexical scope in which the control variables are implicitly declared; anywhere a variable name **var** is allowed, a general destructuring pattern may be given. If more than one iteration clause is present, the sequences are iterated in parallel. At the top of each iteration, all the sequences are tested for termination, and if any of them have terminated the **FOR** terminates; only if none of the sequences have terminated will the control variables be stepped to the next elements of the sequences.

(x := s [TO f] [BY d])

> Iterates **x** through the sequence s, s + d, s + 2d, ..., s + nd such that s + nd <= f. **TO f** defaults to **TO Integer.Last**. **BY d** defaults to **BY 1**. All expressions are evaluated once on entry to the **FOR**.

(x := s [DOWN-TO f] [BY d])

> Iterates **x** through the sequence s, s - d, s - 2d, ..., s - nd such that s - nd >= f. **f** defaults to **Integer.First**, **d** defaults to 1 (**d** should always be positive).

(x IN l)
(x IN-VECTOR v)
(x IN-TEXT t)
(x IN-RD rd)

> Steps **x** through the elements of a list, the characters of a text or **Rd.T**, or the elements of a vector (all evaluated once on entry to the **FOR**). For **IN** and **IN-VECTOR**, **x** may be a destructuring pattern (see **LET**).

***

Body clauses:

(WHILE e)

> Stops iteration when **e** is false.

(UNTIL e)

> Stops iteration when **e** is true.

(DO e...)

> Evaluates the expressions **e...** each time through the loop.

(WHEN e)

> If **e** is false, then execution of any succeeding **DO**, **WHILE**, **UNTIL**, or result-producing clauses is suppressed for the current iteration of the loop only.

(BIND [var v]...)

> Declares **var** as a local variable and sets it to **e** at this point each time through the loop. **var** may be a destructuring pattern. Equivalent to:

```
(VAR var v)
(DO (DSET var v))
```

***

Result-producing clauses. At most one may be given; if none is given, the result of the `FOR` is `()`.

`(RESULT e...)`

When iteration stops, evaluates `e...` and returns the value of the last one.

`(LIST e)`

Evaluates `e` each time through the loop, returning a list of the values.

`(VECTOR e)`

Evaluates `e` each time through the loop, returning a vector of the values.

`(TEXT e)`

Evaluates `e` to a character each time through the loop, returning a text of those characters.

`(SOME e)`

Evaluates `e` to a boolean each time through the loop, exiting with true the first time `e` evaluates to true. Returns false if every `e` evaluated to false.

`(EVERY e)`

Evaluates `e` to a boolean each time through the loop, exiting with false the first time `e` evaluates to false. Returns true if every `e` evaluated to true.

`(REDUCE f i e)`

Returns `(... (f (f (f i e1) e2) e3) ...)` as the value of the loop, where where `ei` is the value of `e` each time through the loop. `f` is evaluated once on entry.

***

The clauses may occur in any order, though they will be implicitly regrouped according to this order:

`VAR` clauses

Iterator clauses

Body clauses

Result clauses

Expressions in the clauses are evaluated in the scope of all the preceding `VAR` and iterator control variables of the loop (in the regrouped order).

***

Examples:

```
(FOR (i := 1 TO 5) (LIST (* i i)))
=>  (1 4 9 16 25).
```

```
(FOR (x IN l)
     (i := 0)
     (WHEN (!== x ()))
     (LIST i))
```

steps x through the elements of list l in parallel with stepping i from 0, returning a list of those i for which x is non-nil.

```
(FOR (x IN-VECTOR v) (DO
    (Test x)
    (Print x))
```

steps x through the elements of vector v, evaluating the DO body each time through.

```
(FOR ((x y) IN '((a 1) (b 2) (c 3)))
     (LIST (List.List y x)))
=>  ((1 a) (2 b) (3 c)).
```

```
(FOR (c IN-RD rd)
     (WHEN (!= c ' '))
     (TEXT c))
```

reads the characters from rd and puts all except blanks into a text.

```
(FOR (x := 1)
     (WHILE (Pred x))
     (REDUCE + 0 (* x x)))
```

steps x from 1 until (Pred x) is false, returning the sum of the squares of all preceding x.

16. Dynamic Variables

In addition to providing lexically scoped variables, Tinylisp also provides dynamically scoped variables. Such variables are referenced by preceding them with a $:

```
$x
```

Each thread has its own set of nested dynamic scopes, and a single, shared, outermost global scope includes the all the threads' scopes. DYNAMIC-BIND creates new, nested scopes. The binding of a variable $x is taken from the most recent dynamically enclosing DYNAMIC-BIND that binds $x, or the global scope if there is no such enclosing DYNAMIC-BIND.

Dynamic variables have a limited usefulness for conveniently passing around thread-specific global values such as $so (standard output). It costs about two orders of magnitude more to reference a dynamic variable compared to a lexical variable. The dynamic variables used by Tinylisp itself are:

`$si`	standard input
`$so`	standard output
`$se`	standard error
`$module`	the current module
`$elision`	the amount of elision the read-eval-print loops `Eval.Loop` and `Debug` should use when printing values
`$r`	the last value printed by the read-eval-print loops

`(DYNAMIC-BIND ([$var v]...)`
`e...)`

Creates a new dynamic scope for the current thread in which the dynamic variables `$var` are initialized to the values of the corresponding expressions `v` (which are evaluated outside of the scope of the `DYNAMIC-BIND`). The expressions `e...` are evaluated, the dynamic scope removed, and the value of the last expression returned.

Example:

```
(DEFINE (P)
    (DYNAMIC-BIND ($z 4)
        `(,$x ,$y ,$z)))

(:= $x 1)                          => 1
(DYNAMIC-BIND ($y 2
               $z 3)
    (P))                           => (1 2 4)
```

`(DynamicValue.Get symbol thread)`

Returns the current binding in the given thread of the dynamic variable whose name is `symbol`. If the thread is nil, it defaults to `Thread.Self`. `$x` is syntactic sugar for:

```
(DynamicValue.Get `x ())
```

`(DynamicValue.Put symbol e thread)`

Assigns the current binding in the given thread of the dynamic variable whose name is `symbol` to be the value `e`. If the thread is nil, it defaults to `Thread.Self`. The syntactic sugar:

```
(:= $x e)
```

is equivalent to:

```
(DynamicValue.Put `x e ())
```

`(DynamicValue.CopyBindings fromThread toThread)`

A new thread created by `Thread.Fork` inherits only the global dynamic scope. This operation copies the current dynamic scopes from `fromThread` to `toThread`. If either is nil, it defaults to `Thread.Self`. The scopes are copied, so if the source thread then exits a dynamic scope, that won't affect the scopes of destination thread (and vice versa). However, the variable locations (the slots to which the variables are bound) are shared, so assignments to a particular variable binding in one thread will be visible to the other.

This operation costs at least as much as a `Thread.Fork`, maybe

considerably more.

17. Reading and Printing Symbolic Expressions

The following procedures read and write symbolic expressions; see the Sx interfaces for full details and other procedures. In all the procedures, the `module` parameter defaults to `$module`, and `syntaxTable` defaults to `TinylispSyntaxTable`. Elisions (`Sx.Elision`) are represented in Tinylisp as either nil, meaning no elision, or a list `(depth length)`.

`(Read rd [module syntaxTable])`

Reads the next symbolic expression from the reader `rd`. Raises `Rd.EndOfFile` if there are no more expressions in the reader, and `ReadError` on any kind of syntax error.

`(ReadDelimitedList rd char [module syntaxTable])`

Repeatedly reads symbolic expressions from the reader `rd` until `char` is encountered, returning the expressions in a list. There may be whitespace between the last expression and `char`. Raises `ReadError` if any syntax error is encountered or end-of-file is encountered at any point.

`(ScanF rd format [module syntaxTable])`

Formatted input, returning the results in a list. The format codes are similar to those of Modula-2+'s `Rd.ScanF`, plus an additional `%r` which calls `Read` to read an s-expression.

`(Print wr value [elision module syntaxTable])`

Prints a symbolic expression to `wr`, which must be an `FWr` (a formatted writer). The default values of `$so` and `$se` are `FWrs`.

`(PP wr value [elision module syntaxTable])`

Same as `Print`, except that `wr` may be an arbitrary writer, and the output is always terminated with a newline.

`(PrintList wr value [elision module syntaxTable])`

Prints a list without using any client-supplied print procedure defined for that list.

`(PrintF wr format arg... [elision module syntaxTable])`

Formatted output using format strings similar to those of Modula-2+'s `Wr.PrintF`, plus an additional format code `%p` for printing an arbitrary value with `Print`, plus additional codes for controlling `FWrs` (formatted writers). See the Sx interface for more details.

18. Evaluation and Read-Eval-Print Loops

(Eval.Loop [si so se prompt])

Invokes a read-eval-print loop using the given i/o streams and prompt, which default to `$si`, `$se`, `$so`, and > . The loop repeatedly reads an expression from the input, evaluates it, and prints its result.

Each expression evaluation occurs in a new thread. If an error occurs during evaluation (an unhandled exception or trap) or a Pause is executed, the Tinylisp debugger is invoked automatically on the thread. If the application has enabled control-C interrupts, then typing control-C during the evaluation will invoke the debugger on the thread.

The dynamic variable `$elision` specifies the elision to use when printing values (see `Print`); this defaults to `(20 200)`. The variable `$r` is set to be the value of the last evaluated expression.

(Eval sexpr)

Evaluates `sexpr`, which should be a Tinylisp s-expression. Equivalent to:

```
(Apply (Procedure.Compile `(PROC () ,expression))
       ())
```

Example:

```
(Eval `(+ x y))
```

19. Source and Object Files

Tinylisp does not have a separate compiler and interpreter -- all source expressions are compiled on-the-fly. When a source file is loaded (read in to Tinylisp), Tinylisp automatically creates a corresponding object file, which can be loaded much faster. By convention, source files end in ".tl", object files end in ".to".

Object files normally don't contain the source s-expressions of defined procedures (this saves space and time). Without the source s-expressions available, `Debug` can show only the name of a procedure and its arguments, not the expressions inside the procedure.

(Load filename [retainSource])

Loads a source or object file, reading and evaluating each expression in the file, returning the name of the file, and creating an object file if necessary.

The action taken depends on the form of `filename` (a text):

`myfile.tl`	The source file is loaded, and the object file `myfile.to` is created.
`myfile.to`	The object file is loaded.
`myfile`	The newer of `myfile.tl` and `myfile.to` is loaded, and `myfile.to` is created if `myfile.tl` is newer.

`Load` won't create an object file (and it won't complain) if it doesn't have sufficient file-access privileges.

If an object file is created and `retainSource` is true, the source s-expressions for procedures will be included in the object file. By default, the source is not stored in the object files; this saves time and space, but it also prevents the debugger from showing the source inside those procedures.

```
(LoadSource filename
    [objectFilename [retainSource [errorWr [skipFirstLine]]]])
```

Reads and evaluates the source expressions in `filename`. If `objectFilename` is non-nil, creates an object file with that name, raising `OS.Error` if it doesn't have write access. If `retainSource` is true, then source s-expressions for procedures are retained in the object file. If `errorWr` is non-nil, syntax and compilation errors are printed to it. If `skipFirstLine` is true, then the first input line is skipped (for shell scripts).

```
(LoadObject filename)
```

Loads an object file.

20. Debugging

Tinylisp provides a simple set of tools for debugging, including a same-address-space debugger, rudimentary methods of setting breakpoints, and the tried-and-true print statement (combined with very fast turnaround).

The debugger lets you examine, continue, and destroy threads containing Tinylisp procedure calls. From the debugger you can examine variable values and see the s-expression source location of each procedure on a stack. When Tinylisp is bound into an application, threads that encounter an error (an unhandled exception or trap) simply suspend themselves, without stopping the entire address space and waiting for Loupe.

Every expression evaluated by a read-eval-print loop is forked off as a separate thread, and the loop will invoke the debugger on that thread automatically if it suspends because of an unhandled exception or other error. Also, typing control-C to the read-eval-print loop will suspend the current thread and invoke the debugger on it.

Users can also invoke the debugger explicitly to examine other threads. (The read-eval-print loop will tell you if there are other threads needing debugging.)

The debugger has no problems suspending a thread currently executing a Tinylisp procedure. But if the thread's current procedure is a Modula-2+ procedure, the debugger must wait until the thread returns to a Tinylisp procedure. This of course, could take a long time; so the debugger has an `alert` command which alerts a running thread, politely asking it to stop. All Modula-2+ packages which could possibly take a very long time to execute should respond to such alerts; but it is likely that many don't.

You can use Loupe freely to examine a program containing Tinylisp. However, beware that when a thread gets a trap or unhandled exception, all other threads continue executing. Also, Loupe doesn't know how to print stack frames for Tinylisp procedures; but there is a file of Loupe macros that provide minimal tools for examining such frames and printing s-expressions. Within Loupe do:

```
<"/proj/packages/tinylisp/tinylisp.lp"
call help()
```

(Debug [thread [si so se prompt]])

Explicitly invokes the Tinylisp debugger. If a thread is specified, it becomes the debugger's current thread; otherwise the debugger selects one of the stopped threads needing debugging. Normally `$si`, `$se`, and `$so` are used as the i/o streams, but these can be supplied explicitly.

The `Debug` commands are typical stack-debugging commands; do `help` to see specifics.

Unlike Loupe, the debugger does not allow you to set arbitrary breakpoints or singlestep through an expression. You can however, set breaks on entry to and exit from a procedure by invoking `Break`, and you can pause procedures at selected points by explicitly inserting calls to `Pause`. And of course, you can insert print statements.

(Break [proc [inExpr [outExpr]]])

Sets a breakpoint on entry to and exit from the given procedure; the calling thread will suspend (notify `Debug`) on entrance to the procedure and on exit. If the s-expression `inExpr` is supplied, then the procedure breaks on entry only when that Tinylisp expression evaluates to true; `inExpr` is evaluated in the context of the formal parameters of the procedure. If the s-expression `outExpr` is supplied, then the procedure breaks on exit only when the expression evaluates to true; `outExpr` is evaluated in a context containing the formal parameters and the special variable `RESULT` containing the procedure's return value.

Example:

```
(DEFINE (Fact n)
    (IF (<= n 0) 1 ELSE (* n (Fact (- n 1)))))

(Break Fact '(<= n 4))
```

You can resume threads from breaks by using the debugger's `continue` command.

If no arguments are given, then a list of all the procedures currently having breaks is returned.

(Unbreak proc...)

Removes the entry breakpoints from all of the given procedures. If no arguments are given, then all breakpoints are removed.

(Pause e...)

Evaluates the expressions `e...` and then suspends the procedure and thread executing the Pause. The values of `e...` are displayed when the debugger examines that thread. The debugger's `continue` command resumes threads suspended by `Pause`.

21. Defining Special Forms

A *special form* is an expression that doesn't have the normal procedure-call evaluation semantics; examples include `IF`, `PROC`, `LOOP`, and `:=`. There are only a small number of primitive special forms in the language; all the other special forms are defined as source-to-source expansions that occur at compile time (so-called "macros").

The expansions are defined by Modula-2+ or Tinylisp procedures. When the Tinylisp compiler encounters a form whose first element is a symbol whose value is a special form, then the expansion procedure is called on the form; the result returned by the expansion procedure is then recursively expanded until no more expansions can occur.

`(SpecialForm.New name proc arg)`

> Defines a special form whose name is the symbol `name` and whose expansion is defined by `proc`. To expand an occurrence of a special form `f`, the compiler calls
>
> ```
> (proc arg f)
> ```
>
> and uses the result as the expansion.

`(SOURCE e...)`

> The debugger shows the source of a procedure before special-form expansion, and it needs help to identify in the fully expanded procedure which forms are original and which are the result of expansions. A special form should wrap each source expression `e` in an expansion with `(SOURCE e)`. This tells the debugger that `e` was in the original form and not generated as part of the expansion.

`(SyntaxError sexpr)`

> This exception should be raised by an expansion procedure whenever it finds an s-expression `sexpr` that's syntactically incorrect.

As an example, here's a special form `(Time e)` that times the evaluation of `e`, returning a list of the result of `e` and its time. `(Time e)` expands to:

```
(LET* (time    (Time.Now)
       result  (SOURCE e))
       (s m)   (Time.Subtract (Time.Now) time))
     (List.List result (+ s (* m 1e-6))))
```

The definition of `Time` is:

```
(DEFINE Time
    (SpecialForm.New
        'Time
        (PROC (arg form)
            (IF (!= 2 (List.Length form))
                (RAISE SyntaxError form))
            (LET (tempTime    (SxModule.GenerateSymbol
                                  $module "time")
                  tempResult (SxModule.GenerateSymbol
                                  $module "time"))
                `(LET* (,tempTime
                            (Time.Now)
                        ,tempResult
                            (SOURCE ,@(List.Tail form))
                        (s m)
                            (Time.Subtract (Time.Now)
                                           ,tempTime))
                    (List.List ,tempResult
                               (+ s (* m 1e-6))))))
        ()))
```

Because special forms define new syntax with new evaluation rules, they are both extremely powerful and quite dangerous if misused. If in doubt, don't use them.

Implementation restriction: If a special form is redefined after a procedure using it is defined (compiled), then the debugger may not correctly show source locations within that procedure.

22. Finding Your Way Around the Built-in Modula-2+ Packages

The power of Tinylisp derives from its ability to invoke many Modula-2+ packages directly. Most of the commonly used packages in "srclib" are built in to standard Tinylisp configurations:

FWr, List, LocalPipe, Math, NullIO, Params, Random, RegExpr, Table, Text, Rd, Thread, Time, TimeConv, Tty, UnixFile, Wr

(This list is approximate, since the particular Tinylisp configuration in an application may include more or less.)

You should rely directly on the Modula-2+ interfaces themselves for documentation. Usually, the Tinylisp interface is identical. But because Tinylisp deals with only a subset of the full range of Modula-2+ types and doesn't have the notion of VAR parameters, some procedures will have small differences in argument types and results.

For example, a Modula-2+ `Time.T` is a record of two integers, but Tinylisp can't deal with Modula-2+ records directly. So for simplicity, the Tinylisp versions of the procedures deal in two-element lists of the form `(seconds microseconds)`.

Unfortunately, with our limited resources, it isn't possible for us to document all these minor differences here. (And applications like Ivy will provide interfaces to their own packages.) But no worries, mate: A few simple rules and tools will help you quickly and reliably find out what you need to know.

First, to see if a procedure is in Tinylisp at all, get to a read-eval-print loop and type the name of the procedure or module:

```
Time.Add
```

If you see something like:

```
#<Procedure.T Time.Add 2>
```

then you'll know the procedure is defined and takes 2 arguments. Similarly, to see if a module is present, type the name of the module (with a trailing dot):

```
Time.
```

and Tinylisp will give a read error if the module doesn't exist.

If you're not sure of the name or its spelling, use `Apropos`:

`(Apropos text)`

> Returns a list of all the public symbols that have `text` as part of their name or their module's name. Case distinctions are ignored. (I.e. this is `grep` over the all the public symbols.) Example:
>
> ```
> (Apropos "char")
> => (Char.First Char.Last Char.Ord Char.T
>     Char.ToControl Char.ToLower Char.ToUpper
>     Char.Val Rd.CharsReady Rd.FindChar
>     Rd.GetChar Rd.UngetChar
>     SxSyntaxTable.CharQuote Text.FindChar
>     Text.FromChar Text.GetChar Wr.PutChar)
> ```

Suppose you know the Modula-2+ procedure is present in Tinylisp, but you're not sure if the Tinylisp interface is any different. If the procedure accepts and returns only refs, integers, cardinals, numeric subranges, characters, booleans, reals, longreals, enumerated types, or procedures, and it has no VAR parameters, then the interface will be similar. If the Modula-2+ procedure has default parameters, they can be defaulted from Tinylisp as well.

Tinylisp provides the following automatic conversions (with full, safe checking):

> Tinylisp integers, booleans, longreals, and characters are converted to Modula-2+ `INTEGER`, `BOOLEAN`, `REAL`, `LONGREAL`, and `CHAR`.
>
> Tinylisp integers are converted to Modula-2+ numeric subranges and cardinals.
>
> A resulting Modula-2+ `INTEGER`, `BOOLEAN`, `REAL`, `LONGREAL`, or `CHAR` is converted to its Tinylisp equivalent.
>
> Enumerated-type parameters and results are represented in Tinylisp as symbols of the same name. For example, `Text.Compare` returns one of the symbols `Base.Lt`, `Base.Eq`, or `Base.Gt`. Remember to use a backquote in front of the symbols passed to enumerated-type parameters.

For most casual use, simply try invoking the procedure from the read-eval-print loop.

If the procedure deals with non-Tinylisp types or you want to know for sure what the interface is, you must look at the special Tinylisp-Modula-2+ interface files that define the Modula-2+ names accessible from Tinylisp.

By convention, these interface files end in ".tli" (Tiny Lisp Interface) and are stored with the other interface files in `/proj/{topaz,ultrix}/{friends,public}`.

Three interfaces define the standard Tinylisp namespace:

`TLInit.tli`	The names forming the core of the Tinylisp language itself.
`TLLibInit.tli`	The standard srclib Modula-2+ packages accessible from Tinylisp.
`TLOSInit.tli`	The OS interface and friends.

The complete interface specification language is defined in section 23, but the interfaces should be fairly perspicuous to anyone familiar with both Tinylisp and Modula-2+.

For example, consider `Text.Compare`. In `TLLibInit.tli` we find the line

```
(MODULE Text)
```

and a little later:

```
(PROCEDURE Compare ("Text.T" "Text.T" & BOOLEAN)
    (ENUM "Base.Comparison"))
```

This defines the Tinylisp interface to `Text.Compare`. It declares the procedure to take two texts and an optional boolean, and converts the result, a Modula-2+ enumerated type `Base.Comparison`, into one of the Tinylisp symbols `Base.Lt`, `Base.Eq`, or `Base.Gt`.

23. Including Tinylisp in an Application

Tinylisp is implemented as a Modula-2+ library that can be bound into any application program.

Your application should provide a set of Modula-2+ interfaces containing the procedures and opaque-ref types that are to be referenced from Tinylisp. These interfaces should be designed specifically for programming-in-the-small; that is, there should be only a very few exported types, simple procedures with few arguments, and reliance on the primary s-expression types and opaque-ref types. Interfaces that are suitable for programming-in-the-large via Modula-2+ are probably not suitable for programming-in-the-small.

Tinylisp proper deals only with the s-expression types (see Sx): integer, character, boolean, longreal, list, vector, symbol, module, and all application-supplied opaque-ref types. So your interfaces should traffic only in those types or reals or enumerated types (if they don't, you'll have to do more work writing procedures that convert between Tinylisp and Modula-2+).

To make the procedures and types accessible from Tinylisp, you'll have to declare them using a special-purpose declaration language in a `.tli` file, say `MyApplInit.tli`. Compiling `MyApplInit.tli` with `compile_tli` produces `MyApplInit.def`, `MyApplInit.mod`, and `MyApplInitAs.as`, which should then be compiled and linked with the application.

Along with `MyApplInit.o` and `MyApplInitAs.o`, you should link the application with at least the following libraries:

```
dump.a   dynamic.a   vaxinstr.a   regexpr.a   tinylisp.a
```

The application initializes Tinylisp by first calling:

```
Tinylisp.Initialize();
MyApplInit.Initialize();
```

It can then load Tinylisp source files or invoke a read-eval-print loop using the procedures in the Tinylisp interface.

The rest of this section describes the declaration language used in `.tli` files.

23.1. An Example

Though this description is rather long, in fact it isn't hard to declare typical Modula-2+ interfaces. Here's a fragment of a `.tli` file declaring the Text interface:

```
(MODULE Text Lisp)

(TYPE T)
(EXCEPTION EndOfFile)
(EXCEPTION ScanFailed)

(PROCEDURE FromChar (CHAR) "Text.T")
(PROCEDURE Length   ("Text.T")                        INTEGER)
(PROCEDURE IsEmpty  ("Text.T")                        BOOLEAN)
(PROCEDURE SubText  ("Text.T" INTEGER & INTEGER) "Text.T")
```

This declares a Tinylisp module `Text` containing a number of initialized symbols corresponding to the Modula-2+ interface of the same name. The `TYPE` declaration specifies that the Tinylisp symbol `Text.T` should be bound to the Modula-2+ opaque-ref type of the same name. The `EXCEPTION` declarations declare `Text` exceptions that Tinylisp should recognize. And the `PROCEDURE` declarations specify the procedures that should be accessible from Tinylisp, declaring their formal parameter types and any conversions that should be performed on the actual parameters.

The stub generated for this `.tli` file and bound into the application creates the Tinylisp module and initializes its symbols to the appropriate types, exceptions, and procedures. For the procedure declarations, there are stub procedures which dynamically check the number and types of arguments passed in from Tinylisp, converting them if necessary, before calling the corresponding Modula-2+ procedures.

For complete examples, see `/proj/packages/tinylisp/*.tli` or `/proj/packages/ivy/*.tli`.

23.2. The Declaration Language

The input `.tli` files generally contain a mixture of declarations and Modula-2+ statements. Lines beginning with "(" in the first column (but not "(*") are processed as declarations represented as s-expressions (see Sx). All other lines are passed on into the generated `.mod` or `.def` file unchanged. This allows you to specify your own Modula-2+ conversion procedures right in the `.tli` file.

The beginning of the `.tli` file should always contain a Modula-2+ `IMPORT` statement importing all the interfaces referenced in the `.tli` file.

By convention, Sx symbols in the declarations represent the names of Tinylisp symbols, while quoted texts represent Modula-2+ names:

```
tlname  -> Symbol
m2+name -> Text
```

The declarations establish a correspondence between Tinylisp symbols and Modula-2+ names and specify the automatic conversions to be performed between Tinylisp and Modula-2+ procedures. By default, the symbols will be matched with Modula-2+ names of the same name; in the example above, the Tinylisp symbol `Text.T` corresponds to the Modula-2+ name `Text.T`. This default can be overridden by using a **namepair**:

```
namepair -> (tlname m2+name)
         -> tlname
```

For example:

```
(MODULE Text)
...
(PROCEDURE Length ("Text.T") INTEGER)
(PROCEDURE (GetChar "MyText.GetChar") ("Text.T") CHAR)
```

declares the Tinylisp symbol `Text.Length` to correspond to the Modula-2+ procedure `Text.Length`, whereas the symbol `Text.GetChar` corresponds with the Modula-2+ procedure `MyText.GetChar`.

Though Tinylisp proper deals only with ref types, the declaration language performs a limited number of conversions between standard Modula-2+ types and their s-expression representation. These types are named in declarations as a **basetype**:

```
basetype -> REFANY | INTEGER | LONGREAL | CHAR | BOOLEAN
```

23.3. Declaration Forms

(MODULE tlname [parent-tlname...])

Declares a new Tinylisp module with the given parents. Successive Tinylisp names will be qualified relative to that module. Example:

```
(MODULE Text Lisp)
```

(SYMBOL namepair)

Declares a new Tinylisp symbol of the given name. If an explicit Modula-2+ name is given in the **namepair**, then the symbol's value is initialized to the value of the Modula-2+ variable of that name. Examples:

```
(SYMBOL PROC)
(SYMBOL (si "Stdio.stdin"))
```

The second example creates the symbol `si` and initializes its value to to the value of the Modula-2+ variable `Stdio.stdin`.

(TYPE namepair)

Declares a new Tinylisp type corresponding to the Modula-2+ ref type. Examples:

```
(TYPE T)
(TYPE (RegExpr "REPrivate.RegExpr"))
```

(EXCEPTION namepair [basetype])

Declares a Tinylisp exception; if **basetype** is given, then the Modula-2+ exception is assumed to take one argument of that type; the basetype `INTEGER` can be used for Modula-2+ exceptions which return enumerated

or subrange types. Warning: `compile_tli` currently doesn't verify that the declaration matches the actual type of the Modula-2+ exception. Examples:

```
(EXCEPTION EndOfFile)
(EXCEPTION (SyntaxError "TLProcedure.SyntaxError")
           REFANY)
```

`(ENUM m2name tlname...)`

Makes an enumerated type available (after a fashion) to Tinylisp. `m2name` is the name of a Modula-2+ enumerated type, and the `tlname`'s should be the elements of that type, in order.

In Tinylisp, enumerated values are represented by symbols of the same name. This declaration creates those symbols and establishes the correspondence between the enumerated type and the symbols for later procedure declarations. Example:

```
(ENUM "Base.Comparison" Lt Eq Gt)
```

From Tinylisp, you would call a procedure expecting a `Base.Comparison` by passing one of the symbols `Base.Lt`, `Base.Eq`, or `Base.Gt` (remember to backquote literal symbols).

```
(PROCEDURE namepair (formal... [& formal...]) result)
    formal -> basetype | m2name | REAL | (VAL m2name) |
              (PROCEDURE m2name) | (ENUM m2name) |
              (SET set-m2name enum-m2name [base-m2name])
    result -> basetype | m2name | REAL | (ORD m2name) | number |
              () | (ENUM m2name) |
              (SET set-m2name enum-m2name [base-m2name])
```

This is the most complicated declaration, specifying a procedure its argument and result types, and any conversions that should be done between Tinylisp and Modula-2+ representations.

An `&` in the parameter list indicates that the succeeding parameters are optional and will be defaulted if not supplied on a call. The number of optional parameters should match the declaration of the Modula-2+ procedure.

The specified formal and result types needn't match exactly the types of the Modula-2+ procedure, but they must be compatible. For example, a type of `INTEGER` can be specified for a Modula-2+ formal parameter that's a numeric subrange. Typechecking, narrowing, and bounds-checking of actual arguments is done at run time -- type errors raise `System.NarrowFault`.

Details about formal and result specifications:

`basetype -> REFANY | INTEGER | LONGREAL | CHAR | BOOLEAN`

Specifies that the Tinylisp (Sx) representations of parameter values are converted to these Modula-2+ types, or vice versa for results. No conversion is done for `REFANY`.

`m2name`

Specifies that a parameter value should be narrowed to this Modula-2+ ref type (e.g. `Text.T`), or that a result is of this type.

`REAL`

Specifies that a longreal parameter should be converted to the Modula-2+ type **REAL**, or that the Modula-2+ result type is **REAL** and should be converted to a Tinylisp longreal.

(ENUM m2name)

Specifies that the parameter or result is of the specified Modula-2+ enumeration type (which should have been declared using the **ENUM** declaration above). A parameter value is expected to be one of the Tinylisp symbols of the enumeration, and `NarrowFault` is raised otherwise. A result is converted from the enumeration type to the corresponding symbol.

(SET set-m2name enum-m2name [base-m2name])

Sets of enumerated types can be passed between Tinylisp and Modula-2+ by representing them as lists of symbols. **SET** declares such a conversion; **set-m2name** is the Modula-2+ name of the set type, **enum-m2name** is the Modula-2+ name of the element type, and **base-m2name** is the Modula-2+ name of the base enumeration type (if **enum-m2name** is a subrange). The base enumeration type should have been declared using the **ENUM** declaration above.

(VAL m2name)

Specifies that a Tinylisp integer parameter value should be converted to the specified Modula-2+ enumeration type using **VAL**.

(ORD m2name)

Specifies that a Modula-2+ enumerated-type result should be converted to a Tinylisp integer using **ORD**.

(PROCEDURE m2name)

Specifies that a formal parameter should be the given Modula-2+ procedure type. Tinylisp procedures can be passed to such parameters (without conversion of any sort), provided that the Modula-2+ procedure type accepts only ref parameters and either has no result or returns a ref type. (Warning: `compile_tli` can't verify this.)

()

Specifies that the Modula-2+ procedure doesn't return a result, so nil will be returned to Tinylisp.

number

Specifies that the Tinylisp result will be the **number**-th parameter value (1-based).

(PROCEDURE namepair)

This form of procedure declaration performs no conversions; the Modula-2+ procedure is called directly. This is useful for providing n-ary procedures. The Modula-2+ procedure should have the type:

```
PROCEDURE( VAR ARRAY OF REFANY ): REFANY;
```

The arguments are passed in the open array.

(SPECIAL-FORM namepair)

Initializes the Tinylisp symbol to be a special form, whose expansion procedure is the corresponding Modula-2+ name.

(EXPORT module-tlname tlname...)

Imports the given Tinylisp symbols `tlname...` into the module `module-tlname`, using `SxModule.Import`. The symbols then belong to both their original module and the new one.

(FOR-MOD)
(FOR-INIT)
(FOR-BEGIN)
(FOR-DEF)

These control the disposition of succeeding non-declaration lines occuring in the `.tli` file, sending them to the current position in the `.mod` file, the `Initialize` procedure of the `.mod`, the module's main body, or the `.def` file respectively. By default, such lines go to the current position in the `.mod`.

23.4. Contents of the .def file

The .def file generated by `compile_tli` contains an `Initialize` procedure which should be called by the application to initialize the stubs. It also exports one Modula-2+ variable for each module and symbol created by the stub. This allow the application to refer to the modules and symbols efficiently, without calling `SxModule` operations to do name lookups.

24. Tinylisp Performance

Unlike other Lisps, Tinylisp is not interpreted; instead, it has an on-the-fly compiler that compiles expressions and procedures directly into machine code. This results in performance significantly better than other comparable systems (Emacs Mocklisp, for example).

Only the basic primitive forms are open-compiled (compiled inline):

> `IF, &, |, CASE, CASEQ, LOOP, EXIT, ASSERT, LET, LET*,`
> `:=` (for local variables only), `DSET, PROC, RETURN, QUOTE,`
> `!, ==, !==, TYPECASE, TRY, FOR`

Everything else is closed-compiled as procedure calls to Modula-2+ procedures. Tinylisp-to-Tinylisp and Modula-2+-to-Tinylisp procedure calls cost about the same as Modula-2+-to-Modula-2+ calls, but Tinylisp-to-Modula-2+ calls cost about twice as much, since Modula-2+ doesn't do dynamic argument checking.

Arithmetic, being closed compiled, is much more expensive than in Modula-2+, but much less expensive than, say, Mocklisp or the shell language.

If you're curious about the quality of Tinylisp-generated code for some procedure `MyProcedure`, do:

```
(Procedure.Disassemble MyProcedure $so)
```

So far, Ivy hasn't encountered any serious performance problems using Tinylisp as a high-level control language manipulating the efficient Modula-2+ primitives provided

by Ivy.

Index

Text.T 19

! 15
!= 15
!== 15

#<...> 6
#False 2, 17
#True 2, 17
#Undefined 2

$elision 28, 30
$module 13, 28, 30
$r 28
$se 28
$si 28
$so 28

& 8

()
  See instead: nil

* 17

+ 17

- 18

.tl 31
.tli 36, 37, 38
.to 31

/ 18

:= 10
  FOR 26

< 18
<= 18

= 15, 18
== 15
=> 7

> 18
>= 18

@ 10, 20

‘ 14

Abs 18
actual
  parameter 11
alert 32
and
  boolean 8
application 7
  interfacing 37

Apply 12
Apropos 36
arithmetic
  performance 42
arithmetic operations 17
ASSERT 9
assignment 9
  destructuring 11
  dynamic variable 28
  variable 10
atom
  See instead: symbol

BACKQUOTE 14
backquoting 14
Base.Eq 20
Base.Gt 20
Base.Lt 20
BIND
  FOR 26
binding 9
  dynamic variable 28
bit operations 19
BitAnd 19
BitExtract 19
BitInsert 19
BitNot 19
BitOr 19
BitShiftLeft 19
BitShiftRight 19
BitXor 19
block comment 3
block text
  syntax 4
boolean 2, 17
  and 8
  or 8
  Ref.Boolean 2
boolean predicates 15
Boolean.T 16, 17
Break 33
breakpoints 32
BY
  FOR 26

CASE 8
CASEQ 9
Ceiling 18
char
  Ref.Char 2
Char.First 17
Char.Last 17
Char.Ord 17
Char.T 16, 17
Char.ToControl 17
Char.ToLower 17
Char.ToUpper 17
Char.Val 17
character 2, 17
  syntax 4

character set 21
CharSet.Difference 21
CharSet.Distinct 21
CharSet.Equals 21
CharSet.Excl 21
CharSet.In 22
CharSet.Incl 21
CharSet.Intersection 21
CharSet.Subset 22
CharSet.Superset 22
CharSet.SymDifference 21
CharSet.T 21
CharSet.Union 21
closure 11
comment 3
comparison 15
  arithmetic 18
compile_tli 1, 37
compiler 42
compiling 31, 34
condition variable 24
constant 6
  list 14
  symbol 14
control flow 8
control-C 31, 32
conversions
  Modula-2+ 36
Curly 6
current module 5, 13, 28, 30

datatype
  See instead: type
Debug 28, 31, 33
debugging 32
  special forms 34
declarations
  TLI 38
DEFINE 11, 12
DEFINE-EXCEPTION 23
DEFINE-RECORD 23
destructuring 10, 26
  assignment 11
  procedure parameters 12
DO
  FOR 26
DOWN-TO
  FOR 26
DSET 11
dynamic binding 28
dynamic scope 28
dynamic variable 28
DYNAMIC-BIND 29
DynamicValue.CopyBindings 29
DynamicValue.Get 29
DynamicValue.Put 29

efficiency 42
elision 28, 30
ELSE 8
ELSIF 8
Emacs 42
ENUM
  TLI declaration 40, 41
equality 15
errors 32
Eval 31
Eval.Loop 28, 31
evaluation 31
  expression 6
  order 7
EVERY
  FOR 27
examples 1
EXCEPT
  TRY 24
exception 7, 23
  TLI declaration 39
  unhandled 32
Exception.T 16
EXIT 9, 25
EXPORT
  TLI declaration 42
expression 6
  evaluation 6
  symbolic 2
  tinylisp 6
extended objects
  syntax 6

factorial 1
FINALLY
  TRY 23
Float 18
Floor 18
flow of control
  See instead: control flow
FOR 25
FOR-BEGIN
  TLI declaration 42
FOR-DEF
  TLI declaration 42
FOR-INIT
  TLI declaration 42
FOR-MOD
  TLI declaration 42
formal
  parameter 11
formatted input 30
formatted output 30
formatting 8
function
  See instead: procedure

IF 8
IMPORT 13
importing
  symbol 6, 13
IN
  FOR 26
IN-RD
  FOR 26
IN-TEXT
  FOR 26

IN-VECTOR
  FOR 26
indenting 8
indexing 20
inheritance 5
INHERITS 13
integer 2, 17
  Ref.Integer 2
  syntax 3
Integer.First 17
Integer.Last 17
Integer.T 16, 17
integers
  iterating 26
interfacing
  application 37
  Modula-2+ 35, 37
interpreter 31, 42
iterating
  integers 26
  lists 26
  readers 26
  texts 26
  vectors 26
iteration 9, 25
Ivy 1, 8, 42

LET 9, 10
LET* 9, 10
lexical scope 9
library 35, 37
Lisp. 13
list 2, 20
  constant 14
  FOR 27
  List.T 2
  syntax 5
List.Append 20
List.AppendD 20
List.Equal 15
List.List 20
List.SetFirst 20
List.SetTail 20
List.Sort 20
List.SortD 20
List.T 16, 20
  list 2
List.TTail 20
List.TTTail 20
lists
  iterating 26
Load 31
loading files 31
LoadObject 32
LoadSource 32
LOCK 24
logical operations 19
longreal 2, 17
  Ref.LongReal 2
  syntax 3
LongReal.T 16, 17
LOOP 9, 25

Loupe 32

Max 18
meta-syntax
  syntax 7
Min 18
Mocklisp 42
Mod 18
Modula-2+ 1
  calling 35, 37
  conversions 36
  interfacing 35, 37
  packages 35, 37
module 2, 5, 13
  importing a symbol 6
  inheritance 5
  SxModule.T 2
  syntax 5
  TLI declaration 39
  See also: current module
mutex 24

NARROW 16
NARROWN 16
negation 15
nil 2
Nil.T 16
number 17
  syntax 3

object file 31
or
  boolean 8
ORD
  TLI declaration 41
order
  evaluation 7

packages
  Modula-2+ 35, 37
parameter
  actual 11
  formal 11
PASSING
  TRY 24
pattern matching 10
Pause 33
performance 42
PP 30
predicates 15
pretty printing 8
Print 2, 30
PrintF 30
PrintList 30
private
  symbol 5
PROC 11
procedure 11
  application 7
  TLI declaration 40, 41
  variadic 11
procedure call

performance 42
Procedure.T 16
programming in the small 1
public 13
symbol 5, 13

QUOTE 14
quoting 14

RAISE 23
Rd.T 20
Read 2, 30
read-eval-print loop 13, 31
ReadDelimitedList 30
reader 20
readers
iterating 26
REAL
TLI declaration 40
record 23
REDUCE
FOR 27
ref type 2, 37
syntax 6
Ref.Boolean
boolean 2
Ref.Char
char 2
Ref.Integer
integer 2
Ref.LongReal
longreal 2
Ref.Vector
vector 2
Rem 18
RESULT
FOR 27
RETURN 12
Round 18

s-expression
See instead: symbolic expression
ScanF 30
scope 11, 26
dynamic 28
lexical 9
sequence 25
SET 11
TLI declaration 41
set, of characters 21
SHADOW 13
shadowing
symbol 13
SOME
FOR 27
sorting 20
SOURCE 34
source file 31
special form 7, 34
SPECIAL-FORM
TLI declaration 41
SpecialForm.New 34
SpecialForm.T 16
srclib 35
standard error 28
standard input 28
standard output 28
string
See instead: text
stub 37
style 8
suspension 32
Sx 2, 6, 37
SxModule 6
SxModule.T 16
module 2
SxSymbol 6
SxSymbol.T 16
symbol 2
SxSyntaxTable 6
symbol 2, 5
as a variable 7
constant 14
importing 6, 13
private 5
public 5, 13
shadowing 13
SxSymbol.T 2
syntax 5
TLI declaration 39
unowned 6
symbolic expression 2, 6, 37
printing 30
reading 30
synchronization 24
syntax 2
extended objects 6
meta-syntax 7
special forms 34
SyntaxError 34
System.NarrowFault 7

table 22
Table.Delete 22
Table.Get 22
Table.New 22
Table.Put 22
Table.RefCompare 22
Table.RefHash 22
Table.T 22
template
destructuring 10
See instead: BACKQUOTE
text 2, 19
FOR 27
syntax 4
Text.T 2
Text.Cat 19
Text.Compare 19
Text.T 16
text 2
texts
iterating 26
thread 24, 32

Thread.AllThreads 25
Thread.Condition 24
Thread.GetCPUTime 25
Thread.Mutex 24
Thread.NewCondition 25
Thread.NewMutex 25
Thread.T 16, 24
TLI
  declarations 38
TLInit.tli 36
TLLibInit.tli 36
TLOSInit.tli 36
TO
  FOR 26
trap 32
Trunc 18
TRY
  EXCEPT 24
  FINALLY 23
  PASSING 24
type 15
  TLI declaration 39
Type.Of 16
Type.T 16
TYPECASE 16
typechecking 7

Unbreak 33
undefined 2
Undefined.T 16
UNQUOTE 14
UNQUOTE-SPLICING 14
UNTIL
  FOR 26

VAL
  TLI declaration 41
VAR
  FOR 25
variable 7
  assignment 10
  dynamic 28
  lexical 9
variadic
  procedure 11
vector 2, 20
  FOR 27
  Ref.Vector 2
  syntax 5
Vector.Copy 21
Vector.Expand 21
Vector.Get 20
Vector.High 21
Vector.New 20
Vector.Number 21
Vector.T 16, 20
vectors
  iterating 26

WHEN
  FOR 26
WHILE
  FOR 26
whitespace 3
Work. 13
Wr.T 20
writer 20

| 8