STELLA Manual

Painless symbolic programming with
delivery in Common-Lisp, C++ and Java

Edition 1.0

This manual describes
STELLA 3.1 or later.

The STELLA Development Team

USC Information Sciences Institute

1 Introduction

This document describes the STELLA programming language. STELLA stands for Strongly-TypEd, Lisp-like LAnguage. It is an object-oriented language that strongly supports symbolic programming tasks. We developed it, since none of the currently "healthy" languages such as C++ or Java adequately support symbolic programming. While Common-Lisp would probably still be today's language of choice for many symbolic programming applications, its dwindling vendor support and user base make it more and more difficult to justify its use.

When we started the development of the PowerLoom knowledge representation system in 1995 we were faced with exactly this problem. PowerLoom had to be delivered in C++, but it was simply incoceivable to write such a large symbolic programming application directly in C++. The solution was to invent a new programming language we called STELLA and write PowerLoom in STELLA instead.

STELLA is a strongly typed, object-oriented, Lisp-like language specifically geared to support artificial intelligence applications. STELLA preserves those features of Common Lisp deemed essential for symbolic programming such as built-in support for dynamic data structures, heterogeneous collections, first-class symbols, powerful iteration constructs, name spaces, an object-oriented type system with a simple meta-object protocol, exception handling, language extensibility through macros and automatic memory management. Maybe the biggest difference between STELLA and Common Lisp is that STELLA is strongly typed. All externally visible interfaces such as slots, function parameters and return values, etc. have to be explicitly typed. Internal objects such as local variables, however, are mostly typed implicitly supported by type inference. This in conjunction with a powerful type coercion mechanism significantly reduces the number of explicit type information that needs to be supplied by the programmer compared to languages such as C++ or Java.

STELLA programs are first translated into a target language such as Common Lisp, C++ or Java, and then compiled with the native target language compiler to generate executable code. The language constructs of STELLA are restricted to those that can be translated fairly directly into native constructs of the intended target languages. This allows STELLA to be translated into efficient, conventional and readable Lisp, C++ and Java code. The resulting native code can be understood and to some extent even maintained by programmers who don't know STELLA, and it can easily be interfaced with other programs not written in STELLA.

As of Fall 2000, we have programmed approximately 100,000 lines of STELLA code - about 50% for the STELLA kernel itself and the other 50% for the PowerLoom knowledge representation system and related systems. Our subjective experience has been that it is only slightly more difficult to write and debug a STELLA program than a Lisp program, and that the inconvenience of having to supply some type information is much outweighed by the benefits such as catching many errors during compile time instead of at run time.

The biggest benefit, however, seems to be that we can still leverage all the incremental code development benefits of Lisp, since we use the Common Lisp-based version of STELLA for prototyping. This allows us to incrementally define and redefine functions, methods and classes and to inspect, debug and fix incorrect code on the fly. Even the most sophisticated

C++ or Java IDE's don't yet seem to support this fully incremental development style, i.e., a change in a class (every change in Java is a change to a class) still requires recompilation and restart of the application. But it is the restart that can be the most time consuming if one debugs a complex application that takes a significant time to reach a certain state!

Once a STELLA program has matured, it can be translated into C++ or Java to gain extra efficiency, to deliver it as a stand-alone application, or to link it with other programs.

1.1 Credits and History

Bob MacGregor invented STELLA in 1995 to implement the PowerLoom knowledge representation system. He wrote most of the first Lisp-based kernel system of STELLA and still occasionally writes extensions or provides fixes. Today he is primarily a STELLA user writing his own applications.

Hans Chalupsky completed the first full STELLA bootstrap (STELLA translating itself) in Spring 1996, and then went on to deal with all the changes necessary to handle the many C++ and Java idiosyncrasies that were discovered when the first versions of these translators came online. He is currently one of the principal maintainers of STELLA supporting the STELLA code analyzer and the Lisp and C++ translators.

Eric Melz wrote the first version of the C++ translator under very trying circumstances (i.e., at a stage where the STELLA language changed under him on a daily basis). He got the first C++ version of STELLA running in the Fall of 1996.

Tom Russ wrote the Java translator and got the first Java version of STELLA running in Spring 1999. He is currently one of the principal maintainers of STELLA supporting the STELLA code analyzer and the Lisp and Java translators. He is also still active writing occasional extensions such as the STELLA XML parser.

2 Installation

2.1 System Requirements

To install and use STELLA you'll approximately need the following amounts of disk space:

7 MB for the tar-red or zip-ped archive file

35 MB for the untarred sources, tanslations, compiled Java files and documentation

8 MB to compile a Lisp version

11 MB to compile the C++ version (without -g)

3 MB to compile the Java version (already included)

This means that you will need approximately 55 MB to work with one Lisp, one C++ and one Java version of STELLA in parallel. If you also want to experiment with the Lisp translation variant that uses structures instead of CLOS instances to implement STELLA objects, then you will need an extra 8 MB to compile that.

The full STELLA development tree is quite large, since for every STELLA source file there are three to four translated versions and as many compiled versions thereof. The actual STELLA libraries that you have to ship with an application, however, are quite small. For example, the Java jar file 'stella.jar' is only 2 MB including Java sources. Eliminating the Java sources cuts that down to about 1 MB! The dynamic C++ library 'libstella.so' compiled on a Linux platform is about 4 MB. Additionally, if you don't need all the different translations of STELLA, you can delete some of the versions to keep your development tree smaller (See Section 2.7 [Removing Unneeded Files], page 6).

To run the Lisp version of STELLA you need an ANSI Common-Lisp (or at least one that supports CLOS and logical pathnames). We have successfully tested STELLA with Allegro-CL 4.2, 4.3, 5.0 and 6.0, Macintosh CL 3.0 and 4.0, Lucid CL 4.1 (plus the necessary ANSI extensions and Mark Kantrowitz's logical pathnames implementation) and the freely available CMUCL 18c. Our main development platform is Allegro CL running under Sun Solaris and Linux RedHat, so, the closer your environment is to ours, the higher are the chances that everything will work right out of the box. Lisp development under Windows should also be no problem.

To run the C++ version of STELLA you need a C++ compiler such as g++ that supports templates and exception handling. We have successfully compiled and run STELLA with g++ 2.96 under Linux Redhat 7.0 and 7.2, and with CygWin 5.0 under Windows 2000 (CygWin provides a very Unix-like environment). We have not yet tried to run the C++ version fully natively under Windows. The main portability issue is the garbage collector. It is supposed to be very portable and run natively on Windows platforms, but we have never verified that.

For the Java version you will need Java JDK 1.2 or later. To get reasonable performance, you should use JDK 1.3 or later. We've run the Java version of STELLA on a variety of platforms without any problems.

Any one of the Lisp, C++ or Java implementations of STELLA can be used to develop your own STELLA code and translate it into all three languages, but the most convenient development environment is the one based on Lisp. If you use the C++ or Java version, translating and using your own STELLA macros is possible but not yet very well supported.

2.2 Unpacking the Sources

Uncompress and untar the file 'stella-X.Y.Z.tar.gz' (or unzip the file 'stella-X.Y.Z.zip') in the parent directory of where you want to install STELLA ('X.Y.Z' are place holders for the actual version numbers). This will create the STELLA tree in the directory 'stella-X.Y.Z/'. All pathnames mentioned below will be relative to that directory which we will usually refer to as the "STELLA directory".

2.3 Lisp Installation

To install the Lisp version startup Lisp and load the file 'load-stella.lisp' with:

```
(CL:load "load-stella.lisp")
```

The first time around this will compile all Lisp-translated STELLA files before they are loaded. During subsequent sessions the compiled files will be loaded right away.

If you want to use the version that uses Lisp structs instead of CLOS objects to implement STELLA objects do the following:

```
(CL:setq cl-user::*load-cl-struct-stella?* CL:t)
(CL:load "load-stella.lisp")
```

Alternatively, you can edit the initial value of the variable *load-cl-struct-stella?* in the file 'load-stella.lisp'. Using structs instead of CLOS objects greatly improves slot access speed, however, it may cause problems with incremental re-definition of STELLA classes. It is therefore recommended to only use this for systems that are in or near the production stage.

Once all the files are loaded, you should see a message like this:

```
Initializing STELLA...
STELLA 3.1 (patch-level 0) loaded.
Type '(in-package "STELLA")' to execute STELLA commands.
USER(2):
```

To reduce startup time, you might want to create a Lisp image that has all of STELLA preloaded.

Now type

```
(in-package "STELLA")
```

to enter the STELLA Lisp package where all the STELLA code resides.

IMPORTANT: All unqualified Lisp symbols in this document are assumed to be in the STELLA Lisp package. Moreover, the STELLA package does **NOT** inherit anything from the COMMON-LISP package (see the file 'sources/stella/cl-lib/cl-setup.lisp' for the few exceptions), hence, you have to explicitly qualify every Lisp symbol you want to use with

`CL:`. For example, to get the result of the previous evaluation you have to type `CL:*` instead of `*`.

2.4 C++ Installation

To compile the C++ version of STELLA change to the native C++ directory and run `make`:

```
% cd native/cpp/stella
% make
```

This will compile all STELLA files, the garbage collector and generate a static or dynamic '`libstella`' library file in the directory '`native/cpp/lib`' which can later be linked with your own C++-translated STELLA (or other) code. To test whether the compilation was successful you can run STELLA from the same directory like this:

```
% ./stella
Welcome to STELLA 3.1 (patch-level 0)
Running kernel startup code...
Initializing symbol tables...
Initializing quoted constants...
Initializing global variables...
Creating class objects...
Finalizing classes...
Creating method objects...
Finalizing methods...
Running non-phased startup code...
Starting up translators...
Bye!
```

This will simply run various STELLA startup code and exit. See Section 4.1.2 [Hello World in C++], page 14, to see how you can use the STELLA C++ executable to translate STELLA code.

2.5 Java Installation

Nothing needs to be done to install the Java version. Since Java class files are platform independent, they are already shipped with the STELLA distribution and can be found in the directory '`native/java`' and its subdirectories. Additionally, they have been collected into the file '`stella.jar`' in the STELLA directory. To try out the Java version of STELLA run the following in the STELLA directory:

```
% java -jar stella.jar
Welcome to STELLA 3.1 (patch-level 0)
Running kernel startup code...
Initializing symbol tables...
Initializing quoted constants...
Initializing global variables...
Creating class objects...
Finalizing classes...
```

```
Creating method objects...
Finalizing methods...
Running non-phased startup code...
Starting up translators...
Bye!
```

Similar to the C++ executable, this will simply run various STELLA startup code and exit. See Section 4.1.3 [Hello World in Java], page 16, to see how you can use the STELLA Java executable to translate STELLA code.

2.6 X/Emacs Setup

STELLA development is very similar to Lisp development, and it is best done in an X/Emacs-based Lisp development environment such as the Allegro-CL Emacs interface plus Allegro Composer, or ILISP. If you do use X/Emacs with the Allegro CL interface, add the following to your '.emacs' or '.xemacs/init.el' file:

```
(setq auto-mode-alist
      (cons '("\\.ste$" . fi:common-lisp-mode) auto-mode-alist))
```

If you are using the Allegro CL interface, you might want to install the file 'emacs/fi-stella.el', since it sets up proper indentation for STELLA code and makes looking up STELLA definitions via the `C-c .` or `M-.` commands work better. Look at the file 'emacs/fi-stella.el' for specific installation instructions.

2.7 Removing Unneeded Files

To save disk space you can remove files that you don't need. For example, if you are not interested in the C++ version of STELLA, you can delete the directory 'native/cpp'. Similarly, you can remove 'native/java' to eliminate all Java-related files. You could do the same thing for the Lisp directory 'native/lisp', but that would eliminate your ability to develop any new STELLA code! Finally, if you don't need any of the STELLA sources, you can delete the directory 'sources/stella'. If you don't need local copies of the STELLA documentation, you can delete parts or all of the 'sources/stella/doc' directory.

3 The STELLA Language

3.1 Language Overview

STELLA is a strongly typed, object-oriented, Lisp-like language. STELLA programs are first translated into either Common Lisp, C++, or Java, and then compiled with any conventional compiler for the chosen target language to generate executable code. Over 95% of the STELLA system is written in STELLA itself, the rest is written in target-language-specific native code.

The design of STELLA borrows from a variety of programming languages, most prominently from Common Lisp, and to a lesser degree from other object-oriented languages such as Eiffel, Sather, and Dylan. Since STELLA has to be translatable into C++ and Java, various restrictions of these languages also influenced its design.

In the following, we assume that the reader is familiar with basic Common Lisp concepts, and has at least some familiarity with C++ or Java. Let us start with a cursory overview of STELLA's main features:

Syntax: STELLA uses a parenthesized, uniform expression syntax similar to Lisp. Most definitional constructs and control structures are similar to their Common Lisp analogues with variations to support types.

Type system: STELLA is strongly typed and supports efficient static compilation similar to C++. Types are required for the arguments and return values of functions and methods, for global variables, and for slot definitions. Local, lexically scoped variables can be typed implicitly by relying on type inference.

Object system: Types are organized into a single inheritance class hierarchy. Restricted multiple inheritance is allowed via mixin classes. Dynamic method dispatch is based on the runtime type of the first argument (similar to C++ and Java). Slots can be static (native) or dynamic. Dynamic slots can be defined at runtime and do not occupy any space until they are filled. Slots can have both initial and default values, and demons can be triggered by slot accesses. A meta-object protocol allows the control of object creation, initialization, termination, and destruction.

Control structure: Functions and methods are distinguished. They can have multiple (zero or more) return values and a variable number of arguments. Lisp-style macros are supported to facilitate syntax extensions. Expressions and statements are distinguished. Local variables are lexically scoped, but dynamically scoped variables (specials) are also supported. STELLA has an elegant, uniform, and efficient iteration mechanism plus a built-in protocol for iterators. An exception mechanism can be used for error handling and non-local exits.

Symbolic programming: Symbols are first-class objects, and extensive support for dynamic datatypes such as cons-trees, lists, sets, association lists, hash tables, extensible vectors, etc., is available. A backquote mechanism facilitates macro writing and code generation. Interpreted function call, method call, slot access, and object creation is supported, and a restricted evaluator is also available.

Name spaces: Functions, methods, variables, and classes occupy separate name spaces (i.e., the same name can be used for a function and a class). A hierarchical module system compartmentalizes symbol tables and supports large-scale programming.

Memory management: STELLA relies on automatic memory management via a garbage collector. For Lisp and Java the native garbage collector is used. For the C++ version of STELLA we use the Boehm- Weiser conservative garbage collector with good results. Various built-in support for explicit memory management is also available.

The Common Lisp features most prominently absent from STELLA are anonymous functions via lambda abstraction, lexical closures, multi-methods, full-fledged eval (a restricted evaluator is available), optional and keyword arguments, and a modifiable readtable. STELLA does also not allow dynamic re/definition of functions and classes, even though the Lisp-based development environment provides this facility (similar to Dylan). The main influences of C++ and Java onto STELLA are the strong typing, limited multiple inheritance, first-argument polymorphism, and the distinction between statements and expressions.

3.2 Basic Data Types (tbw)

To be written.

3.3 Control Structure (tbc)

To be completed.

3.3.1 Conditionals

STELLA conditionals are very similar to those found in Common-Lisp. The main difference is that most STELLA conditionals are statements and therefore do not return a value. For this reason, a C++-style `choose` directive has been added to the language to allow value conditionalization based on a boolean expression.

if *condition then-statement else-statement*   Statement

    Evaluate the boolean expression *condition*. If the result is true execute *then-statement*, otherwise, execute *else-statement*. Note that unlike the Common-Lisp version of `if` the *else-statement* is not optional in STELLA. Example:

```
(if (> x y)
    (print "x is greater than y" EOL)
  (print "x is less than or equal to y" EOL))
```

when *condition statement...*   Statement

    Evaluate the boolean expression *condition*. Only if the result is true execute the *statement*'s in the body. Example:

```
(when (symbol? x)
  (print "x is a symbol, ")
  (print "its name is " (symbol-name (cast x SYMBOL)) EOL))
```

unless *condition statement...*   Statement

> Evaluate the boolean expression *condition*. Only if the result is false execute the
> *statement*'s in the body. Therefore, (`unless` *test* ...) is equivalent to (`when` (`not`
> *test*) ...). Example:

```
(unless (symbol? x)
  (print "x is not a symbol, ")
  (print "hence, its name is unknown" EOL))
```

cond *clause...*   Statement

> `cond` is a conditional with an arbitrary number of conditions each represented by a
> *clause*. Each `cond` clause has to be of the following form:

> (*condition statement...*)

> The first *clause* whose *condition* evaluates to true will be selected and its *statement*'s
> will be executed. Each clause can have 0 or more statements. The special condition
> `otherwise` always evaluates to true and can be used for the catch-all case. Example:

```
(cond ((symbol? x)
       (print "x is a symbol" EOL))
      ((cons? x)
       (print "x is a cons" EOL))
      (otherwise
       (print "x is an object" EOL)))
```

choose *condition true-expression false-expression*   Expression

> Evaluate the boolean expression *condition*. If the result is true return the value of
> *true-expression*, otherwise, return the value of *false-expression*. STELLA computes
> the most specific common supertype of *true-expression* and *false-expression* and uses
> that as the type returned by the `choose` expression. If no such type exists, a trans-
> lation error will be signaled. Example:

```
(setq face (choose happy? :smile :frown))
```

case *expression clause...*   Statement

> Each `case` clause has to be of one of the following forms:

> (*key statement...*)
> ((*key...*) *statement...*)

> `case` selects the first *clause* whose *key* (or one of the listed *key*'s) matches the result
> of *expression* and executes the clause's *statement*'s. Each `case` *key* has to be a
> constant such as a number, character, string, symbol, keyword or surrogate. Keys are
> compared with `eql?` (or `string-eql?` for strings). All keys in a `case` statement have
> to be of the same type. The special key `otherwise` can be used to catch everything.
> It is a run-time error if no clause with a matching key exists. Therefore, a STELLA
> `case` without an `otherwise` clause corresponds to a Common Lisp `ecase`. An empty
> `otherwise` clause can always be specified via (`otherwise NULL`). Example:

```
(case car-make
  ("Yugo"
   (setq price :cheap))
  ("VW"
```

```
        (setq price :medium))
       (("Ferrari" "Rolls Royce")
        (setq price :expensive))
       (otherwise
        (setq price :unknown)))
```

typecase *expression clause. . .*   Statement
    Each `typecase` clause has to be of one of the following forms:

        (*type* *statement. . .*)
        ((*type. . .*) *statement. . .*)

`typecase` selects the first *clause* whose *type* (or one of the listed *type*'s) equals or
is a supertype of the run-time type of the result of *expression* and then executes the
clause's *statement*'s. Therefore, `typecase` can be used to implement a type dispatch
for cases where the run-time type of an expression can be different from the static
type known at translation time. Currently, the static type of *expression* is required
to be a subtype of `OBJECT`.

Each *type* expression has to be a symbol describing a simple type (i.e., parametric
or anchored types are not allowed). Similar to `case`, the special key `otherwise` can
be used to catch everything. It is a run-time error if no clause with a matching type
exists. Therefore, a STELLA `typecase` without an `otherwise` clause corresponds to
a Common Lisp `etypecase`. An empty `otherwise` clause can always be specified via
`(otherwise NULL)`. `typecase` does allow the value of *expression* to be undefined, in
which case the `otherwise` clause is selected. Example:

```
        (typecase (first list)
          (CONS
           (print "it is a cons"))
          ((SYMBOL KEYWORD)
           (print "it is a symbol"))
          (STANDARD-OBJECT
           (print "it is a regular object"))
          (otherwise NULL))
```

Note that in the example above it is important to list `STANDARD-OBJECT` after `SYMBOL`
and `CONS`, since it subsumes the preceding types. Otherwise, it would always shadow
the clauses with the more specific types.

The semantics of `typecase` is slightly extended for the case where *expression* is a
local variable. In that case each reference to the variable within a `typecase` clause
is automatically casted to the appropriate narrower type. For example, in the code
snippet below method calls such as `first` or slot accesses such as `symbol-name` are
translated correctly without needing to explicitly downcast `x` which is assumed to be
of type `OBJECT`:

```
        (typecase x
          (CONS
           (print "it is a cons with value " (first x)))
          ((SYMBOL KEYWORD)
           (print "it is a symbol with name " (symbol-name x)))
          (STANDARD-OBJECT
```

```
        (print "it is a regular object"))
      (otherwise NULL))
```

Since the `typecase` *expression* has to be a subtype of `OBJECT`, a `typecase` cannot be used to test against literal types such as `STRING` or `INTEGER`. If such type names are encountered as keys in a `typecase`, they are automatically converted to their wrapped version, e.g., `STRING-WRAPPER`, `INTEGER-WRAPPER`, etc.

3.4 Functions (tbw)

To be written.

3.5 Classes (tbw)

To be written.

3.6 Methods (tbw)

To be written.

3.7 Macros (tbw)

To be written.

3.8 Modules (tbw)

To be written.

4 Programming in STELLA

4.1 Hello World in STELLA

Included with the STELLA distribution is a simple Hello World application that shows you how to organize your own STELLA code and build a working STELLA application. The sources for the Hello World system consist of the following files:

```
sources/systems/hello-world-system.ste
sources/hello-world/file-a.ste
sources/hello-world/file-b.ste
```

STELLA organizes code modules with a simple system facility. Translation always operates on a complete system, so you always need to create a system definition for the STELLA files comprising your application (somewhat similar to what you would put in a Unix Makefile).

For the Hello World system the system definition already exists and resides in the file 'sources/systems/hello-world-system.ste'. By default, STELLA looks in the directory 'sources/systems' to find the definition of a particular system. 'hello-world-system.ste' defines two things:

(1) The HELLO-WORLD module which defines a namespace for all objects in the Hello World systems. STELLA modules are mapped onto corresponding native namespace constructs, i.e., Lisp packages, C++ namespaces or Java packages. The exact mapping for each language can be defined via the keyword options :lisp-package, :cpp-package and :java-package in the module definition, for example:

```
(defmodule "HELLO-WORLD"
  :lisp-package "STELLA"
  :cpp-package "hello_world"
  :java-package "edu.isi.hello_world"
  :uses ("STELLA"))
```

The :uses directive tells STELLA from what other modules this one inherits.

(2) The actual system definitions defining what source files comprise the system, and what parent systems this one depends on, plus a variety of other options:

```
(defsystem HELLO-WORLD
  :directory "hello-world"
  :required-systems ("stella")
  :cardinal-module "HELLO-WORLD"
  :production-settings (1 0 3 3)
  :development-settings (3 2 3 3)
  :files ("file-a"
          "file-b"))
```

4.1.1 Hello World in Lisp

To generate a Lisp translation of Hello World you can use either the Lisp, C++ or Java version of STELLA. Before you can translate you have to make sure the following native directories exist:

```
native/lisp/hello-world/
bin/acl5.0/hello-world/
```

The directory 'native/lisp/hello-world/' will hold the Lisp translations of the corresponding STELLA source files. The directory 'bin/acl5.0/hello-world/' will hold the compiled Lisp files if you are using Allegro CL 5.0. If you are using a different Lisp, one of the other binary directories as defined in the top-level file 'translations.lisp' will be used. The directory 'bin/lisp/hello-world/' will be used as a fall-back if your version of Lisp is not yet handled in 'translations.lisp'.

If you create your own system, you will need to create those directories by hand (future versions of STELLA might do that automatically). For the Hello World system these directories already exist.

To generate a Lisp translation of Hello World using Lisp startup a Lisp version of STELLA (see). The following idiom will then translate the system into Lisp and also Lisp-compile and load it. The first argument to `make-system` is the name of the system, and the second argument indicates into what language it should be translated:

```
STELLA(3): (make-system "hello-world" :common-lisp)
Processing '/tmp/stella-3.1.0/sources/hello-world/file-a.ste':
*** Pass 1, generating objects...
Processing '/tmp/stella-3.1.0/sources/hello-world/file-b.ste':
*** Pass 1, generating objects...
   .......................................
;;; Writing fasl file
;;;   /tmp/stella-3.1.0/bin/acl5.0/hello-world/startup-system.fasl
;;; Fasl write complete
; Fast loading
;    /tmp/stella-3.1.0/bin/acl5.0/hello-world/startup-system.fasl
CL:T
STELLA(4):
```

After the system is loaded you can call its `main` function:

```
STELLA(10): (main)
Hello World A
Hello World B
bye
()
STELLA(11):
```

Using `main` in the Lisp version will not always make sense, since you can call any function directly at the Lisp top level, but both C++ and Java always need a `main` function as a top-level entry point.

While this would be somewhat unusual, you could also generate the Lisp translation using the C++ or Java version of STELLA. The easiest way to do that is to run the `stella` script in the STELLA directory like this:

```
% ./stella -e '(make-system "hello-world" :common-lisp)'
Welcome to STELLA 3.1 (patch-level 0)
Processing 'sources/hello-world/file-a.ste':
*** Pass 1, generating objects...
Processing 'sources/hello-world/file-b.ste':
*** Pass 1, generating objects...
   ..............................................
Translating 'sources/hello-world/file-a.ste' to 'Common Lisp'...
Writing 'native/lisp/hello-world/file-a.lisp'...
Translating 'sources/hello-world/startup-system.ste' to 'Common Lisp'...
Writing 'native/lisp/hello-world/startup-system.lisp'...
```

The `-e` command line option is used to evaluate an evaluable STELLA command. Conveniently, `make-system` is such a command, so you can supply a `make-system` form to the C++ or Java version of STELLA just as you would do in Lisp. Note the extra quotes around the expression to protect the characters from interpretation by the Unix shell.

To compile and load the translated Lisp files into Lisp you then have to startup a Lisp version of STELLA and call `make-system` again which now will only compile and load the necessary files, since the translations have already been generated in the previous step.

4.1.2 Hello World in C++

To generate a C++ translation of Hello World you can use either the Lisp, C++ or Java version of STELLA. Before you can translate you have to make sure the following native directory exists:

`native/cpp/hello-world/`

The directory '`native/cpp/hello-world/`' will hold the C++ translations of the corresponding STELLA source files. If you create your own system, you will need to create this directory by hand (future versions of STELLA might do that automatically). For the Hello World system the directory already exist.

To generate a C++ translation of Hello World using Lisp startup a Lisp version of STELLA (see Section 2.3 [Lisp Installation], page 4). The following idiom will then translate the system into C++. The first argument to `make-system` is the name of the system, and the second argument indicates into what language it should be translated:

```
STELLA(4): (make-system "hello-world" :cpp)
Processing '/tmp/stella-3.1.0/sources/hello-world/file-a.ste':
*** Pass 1, generating objects...
Processing '/tmp/stella-3.1.0/sources/hello-world/file-b.ste':
*** Pass 1, generating objects...
   ..............................................
Writing '/tmp/stella-3.1.0/native/cpp/hello-world/file-b.hh'...
Writing '/tmp/stella-3.1.0/native/cpp/hello-world/file-b.cc'...
Translating '/tmp/stella-3.1.0/sources/hello-world/startup-system.ste'.
Writing '/tmp/stella-3.1.0/native/cpp/hello-world/startup-system.hh'...
Writing '/tmp/stella-3.1.0/native/cpp/hello-world/startup-system.cc'...
:VOID
STELLA(5):
```

Alternatively, you can generate the translation using the C++ or Java version of STELLA. The easiest way to do that is to run the `stella` script in the STELLA directory like this:

```
% ./stella -e '(make-system "hello-world" :cpp)'
Welcome to STELLA 3.1 (patch-level 0)
Processing 'sources/hello-world/file-a.ste':
*** Pass 1, generating objects...
Processing 'sources/hello-world/file-b.ste':
*** Pass 1, generating objects...
   ..............................................
Writing 'native/cpp/hello-world/file-b.hh'...
Writing 'native/cpp/hello-world/file-b.cc'...
Translating 'sources/hello-world/startup-system.ste'.
Writing 'native/cpp/hello-world/startup-system.hh'...
Writing 'native/cpp/hello-world/startup-system.cc'...
```

The `-e` command line option is used to evaluate an evaluable STELLA command. Conveniently, `make-system` is such a command, so you can supply a `make-system` form to the C++ or Java version of STELLA just as you would do in Lisp. Note the extra quotes around the expression to protect the characters from interpretation by the Unix shell.

Different from Lisp, neither of the above idioms will compile and load the generated C++ code. Instead you have to use the Unix '`make`' facility to compile and link the C++ sources. First change into the native '`hello-world`' directory and then call `make` (**important**: the generated Makefiles currently require the GNU version of `make`):

```
% cd native/cpp/hello-world/
% make
g++ -w -g -O2  -DSTELLA_USE_GC -I../stella/cpp-lib/gc/include \
    -c -I.. main.cc
g++ -w -g -O2  -DSTELLA_USE_GC -I../stella/cpp-lib/gc/include \
    -c -I.. file-a.cc
g++ -w -g -O2  -DSTELLA_USE_GC -I../stella/cpp-lib/gc/include \
    -c -I.. file-b.cc
g++ -w -g -O2  -DSTELLA_USE_GC -I../stella/cpp-lib/gc/include \
    -c -I.. startup-system.cc
   .................................
g++ -dynamic  -L../stella/cpp-lib/gc -Xlinker -rpath -Xlinker \
      '../lib:/tmp/stella-3.1.0/native/cpp/lib' \
      main.o -o hello-world \
      -L../lib -lhello-world -L../lib -lstella -lgc -lm
```

The first time around this will also compile the C++ version of STELLA and the C++ garbage collector and create a STELLA library file. Future builds of the Hello World and other systems will use the STELLA library file directly. To run the Hello World system simply run the '`hello-world`' executable that was built in the previous step:

```
% ./hello-world
Hello World A
Hello World B
bye
```

4.1.3 Hello World in Java

To generate a Java translation of Hello World you can use either the Lisp, C++ or Java version of STELLA. Before you can translate you have to make sure the following native directory exists:

```
native/java/edu/isi/hello-world/
```

The directory 'native/java/edu/isi/hello-world/' will hold the Java translations of the corresponding STELLA source files. If you create your own system, you will need to create this directory by hand (future versions of STELLA might do that automatically). For the Hello World system the directory already exist.

Note that following Java convention we use the package edu.isi.hello_world to hold the Hello World system. This was specified via the :java-package option in the definition of the HELLO-WORLD module. Also note that we use hello_world instead of hello-world as the package name, since a dash cannot legally appear as part of a Java identifier.

To generate a Java translation of Hello World using Lisp startup a Lisp version of STELLA (see Section 2.3 [Lisp Installation], page 4). The following idiom will then translate the system into Java. The first argument to make-system is the name of the system, and the second argument indicates into what language it should be translated:

```
STELLA(5): (make-system "hello-world" :java)
Processing '/tmp/stella-3.1.0/sources/hello-world/file-a.ste':
*** Pass 1, generating objects...
    ...........................................
Writing '/tmp/stella-3.1.0/native/java/hello_world/Startup_Hello_...
:VOID
STELLA(6):
```

Alternatively, you can generate the translation using the C++ or Java version of STELLA. The easiest way to do that is to run the stella script in the STELLA directory like this:

```
% ./stella -e '(make-system "hello-world" :java)'
Welcome to STELLA 3.1 (patch-level 0)
Processing 'sources/hello-world/file-a.ste':
*** Pass 1, generating objects...
Processing 'sources/hello-world/file-b.ste':
*** Pass 1, generating objects...
    ...........................................
Writing 'native/java/edu/isi/hello_world/HelloWorld.java'...
Writing 'native/java/edu/isi/hello_world/StartupFileA.java'...
Writing 'native/java/edu/isi/hello_world/StartupFileB.java'...
Writing 'native/java/edu/isi/hello_world/StartupHelloWorldSystem.java'...
```

The -e command line option is used to evaluate an evaluable STELLA command. Conveniently, make-system is such a command, so you can supply a make-system form to the C++ or Java version of STELLA just as you would do in Lisp. Note the extra quotes around the expression to protect the characters from interpretation by the Unix shell.

Different from Lisp, neither of the above idioms will compile and load the generated C++ code. Instead you have to use the Java compiler to compile and Java to run the compiled Java sources. First change into the top-level native Java directory 'native/java' and then compile and run the Hello World system like this:

```
% cd native/java/
% javac edu/isi/hello_world/*.java
% java edu.isi.hello_world.HelloWorld
Hello World A
Hello World B
bye
```

It is not necessary to Java-compile STELLA first, since STELLA already ships with a Java compilation of the STELLA system.

4.2 Incrementally Developing STELLA Code

The preferred method of STELLA code development is to use a Lisp-based version of STELLA for all the prototyping and testing, since that allows you to exploit most (or all) of the rapid-prototyping advantages of Lisp. Once a system has reached a certain point of stability, it can be translated into C++ or Java for delivery or to interface it with other C++ or Java code.

In the following, we assume an X/Emacs-based Lisp development environment such as the Allegro CL Emacs interface, where Lisp is run in an Emacs subprocess, and Lisp source can be compiled and evaluated directly from the source buffers. By "Lisp buffer" we mean the listener buffer in which Lisp is actually running, and by "source buffer" we mean a buffer that is used to edit a file that contains STELLA source.

Included in the distribution is the Hello World system comprised of the files

```
sources/systems/hello-world-system.ste
sources/hello-world/file-a.ste
sources/hello-world/file-b.ste
```

To get started, simply add your code to either 'file-a.ste' or 'file-b.ste', since all the necessary definitions and directories for these files are already set up properly. See section ??? on how to setup your own system.

Make sure the Hello World system is loaded into Lisp by doing the following:

```
(make-system "hello-world" :common-lisp)
```

This will make sure that the system definition is loaded and the necessary module definition is evaluated.

Now suppose you add the following function to 'file-a.ste':

```
(defun (factorial INTEGER) ((n INTEGER))
  (if (eql? n 0)
      (return 1)
    (return (* n (factorial (1- n))))))
```

There are various options for translating and evaluating this definition. For example, you can simply remake the complete system similar to what you would do for a C++ or Java program:

```
(make-system "hello-world" :common-lisp)
```

This will retranslate the modified files, recompile them and reload them into your Lisp image.

Instead of retranslating and recompiling everything, you can incrementally evaluate the definition of `factorial` from your Emacs-to-Lisp interface. Simply put your cursor somewhere inside the definition in the source buffer and evaluate it by typing `M-C-x`. This translates the STELLA code into Lisp and compiles (or evaluates) the resulting Lisp code. Now you can actually try it out in the Lisp buffer, for example:

```
STELLA(4): (factorial 6)
720
```

Finally, instead of evaluating the definition in the source buffer, you can also enter it directly at the Lisp prompt with the same effect.

The way this works is that the Lisp symbol `stella::defun` is actually bound to a Lisp macro that calls all the necessary translation machinery to convert the STELLA `defun` into Lisp code. Look at the file 'sources/stella/cl-lib/stella-to-cl.ste' for the complete set of such macros. This might be a bit confusing, since there are now three different bindings (or meanings) of `defun`:

1. The STELLA operator `defun` used to define STELLA functions.

2. The Lisp macro `stella::defun` that resides in the STELLA Lisp package and is only available for convenience in Lisp versions of STELLA.

3. The Lisp macro `CL:defun` which is the standard Common Lisp macro used to define Lisp functions.

We'll try to explicitly qualify which meaning is used wherever there might be some doubt which one is meant. In general, every unqualified symbol mentioned below is either part of the STELLA language or resides in the STELLA Lisp package.

Since a newly-written STELLA function might have errors, it is prudent to first only translate it without actually executing the result of the translation. In the source buffer you can do that by macro-expanding the `defun`. For example, if you use the Allegro CL interface you would position the cursor on the opening parenthesis of the `defun` and then type `M-M`. Any errors discovered by the STELLA translator are reported in the Lisp buffer window. The expansion will be a `CL:progn` that contains the translated definition as the first element plus various startup-time (initialization) code following it.

In the Lisp buffer you can achieve a similar effect with the `lptrans` macro. For example, executing

```
(lptrans
 (defun (factorial INTEGER) ((n INTEGER))
   (if (eql? n 0)
       (return 1)
     (return (* n (factorial (1- n)))))))
```

in the Lisp buffer first Lisp-translates the definition, and then prints the translation. To see the C++ translation you can use `cpptrans`, calling `jptrans` will generate the Java translation.

You can also use `lptrans/cpptrans/jptrans` to translate code fragments that are not top-level definitions such as `defun` and its friends. For example:

```
STELLA(8): (lptrans
             (foreach element in (list 1 2 3)
                 do (print element EOL)))
```

```
(CL:LET* ((ELEMENT NULL)
          (ITER-003
           (%THE-CONS-LIST (LIST (WRAP-INTEGER 1) (WRAP-INTEGER 2)
                                 (WRAP-INTEGER 3)))))
  (CL:LOOP WHILE (CL:NOT (CL:EQ ITER-003 NIL)) DO
          (CL:PROGN (SETQ ELEMENT (%%VALUE ITER-003))
                    (SETQ ITER-003 (%%REST ITER-003)))
          (%%PRINT-STREAM (%NATIVE-STREAM STANDARD-OUTPUT)
                          ELEMENT EOL)))
()
STELLA(9): (cpptrans
             (foreach element in (list 1 2 3)
                 do (print element EOL)))
{ Object* element = NULL;
  Cons* iter004 = list(3, wrapInteger(1), wrapInteger(2),
                          wrapInteger(3))-> theConsList;

  while (!(iter004 == NIL)) {
    element = iter004->value;
    iter004 = iter004->rest;
    cout << element << endl;
  }
}
:VOID
STELLA(10): (jptrans
             (foreach element in (list 1 2 3)
                 do (print element EOL)))
{ Stella_Object element = null;
  Cons iter005 = Stella.list
                   (Stella_Object.cons
                     (Stella.wrapInteger(1),
                      Stella_Object.cons
                        (Stella.wrapInteger(2),
                         Stella_Object.cons
                           (Stella.wrapInteger(3),
                            Stella.NIL)))).theConsList;

  while (!(iter005 == Stella.NIL)) {
    {
      element = iter005.value;
      iter005 = iter005.rest;
    }
    java.lang.System.out.println(element);
  }
}
:VOID
```

The use of `lptrans` is really necessary here, since there is no Lisp macro `foreach` that knows how to translate STELLA `foreach` loops (those Lisp macros only exist for top-level

definition commands such as `defun`). In order to translate such code fragments without error messages, they need to be self-contained, i.e., all referenced variables have to be either bound by a surrounding `let`, or they must be globally defined variables. Otherwise, the STELLA translator will generate various "undefined variable" error messages.

You can use the STELLA Lisp macro `eval` (i.e., `stella::eval` not `CL:eval`) to actually execute such a code fragment. For example:

```
STELLA(11): (eval
              (foreach element in (list 1 2 3)
                   do (print element EOL)))
|L|1
|L|2
|L|3
()
```

This translates the loop and executes the result, which prints the wrapped numbers (hence, the `|L|` prefix) to standard output. The `()` at the end is the resulting Lisp value returned by the loop (in Lisp everything returns a value, even though for STELLA `foreach` is a statement, not an expression).

Make it a habit to wrap `eval` around any STELLA code you incrementally evaluate in the Lips buffer. This makes sure that all the arguments to a function, etc., are translated into the appropriate STELLA objects. For example, evaluating

```
(eval (list :a :b :c))
```

in the Lisp buffer generates a STELLA list that points to the STELLA keywords `:a`, `:b` and `:c`. If you don't use `eval`, for example,

```
(list :a :b :c)
```

a STELLA list containing the Lisp keywords ':a', ':b' and ':c' will be created. Lisp keywords are a completely different data structure than STELLA keywords, and any STELLA code expecting a STELLA keyword but finding a Lisp keyword will break, since Lisp keywords are not a legal STELLA data structure. Unfortunately, such cases can be very confusing, since Lisp and STELLA keywords look/print exactly alike.

`eval` is also necessary to access STELLA symbols and surrogates in the Lisp buffer. For example, to access a STELLA symbol, you can use `quote` (again, this is the STELLA `quote` not `CL:quote`):

```
(eval (quote foo))
```

This returns the STELLA symbol `foo`. We explicitly used `quote` here, since code typed at the Lisp prompt is first passed through the Lisp reader before the STELLA translator sees it, and the default Lisp reader interprets the ' character differently than the STELLA reader. Within a STELLA file you can use the syntax `'foo`, since it will be read directly by the STELLA reader that knows how to interpret it correctly.

`lptrans`, `cpptrans` and `jptrans` are evaluable STELLA commands that can also be evaluated by the C++ and Java version of STELLA. For example, to generate a Java translation of a little STELLA code fragment you could run the `stella` script in the STELLA directory like this (the output below has been additionally indented by hand for clarity):

```
% ./stella -e '(jptrans\
              (foreach element in (list 1 2 3)\
                   do (print element EOL)))'
```

```
Welcome to STELLA 3.1 (patch-level 0)
{ Stella_Object element = null;
  Cons iter001 = Stella.list
                   (Stella_Object.cons
                     (Stella.wrapInteger(1),
                      Stella_Object.cons
                        (Stella.wrapInteger(2),
                         Stella_Object.cons
                           (Stella.wrapInteger(3),
                            Stella.NIL)))).theConsList;

  while (!(iter001 == Stella.NIL)) {
    {
      element = iter001.value;
      iter001 = iter001.rest;
    }
    java.lang.System.out.println(element);
  }
}
```

4.3 Performance Hints

Here are a few things to watch out for once you get serious about the performance of your translated STELLA programs:

Safety checks: The STELLA variable `*safety*` controls whether certain safety code is added to your translated STELLA program. For Lisp translations it also controls whether `cast`'s will be translated into run-time type checks or not. There is no run-time type checking performed in C++. In Java native casts will always perform runtime type tests. The default `*safety*` level is 3 which enables the translation of all `safety` clauses with level 3 or lower. A safety level of 1 or lower disables the generation of calls to the `cast` function in Lisp. `cast` performs run-time type checks which are somewhat expensive. However, you should not disable run-time type checking in Lisp until you have fully debugged your program. Once you are confident that your program works correctly, you can set `*safety*` to 0 before you translate it. That way you will avoid the generation and execution of any safety code at all. All of the core STELLA system was translated with `*safety*` set to 1.

Quoted cons trees: Access to quoted constants that are not symbols is somewhat slow, since it currently uses hashing to find them in a table. Hence, access to quoted constants such as `(quote (foo bar fum))` should be avoided in inner loops. Access to quoted symbols such as `(quote foo)` is fast and does not cause any performance problems. The use of `quote` for constant cons trees is rare in STELLA (and somewhat deprecated), which is the reason why this mechanism is not all that well supported. Future versions of STELLA might re-implement the handling of constants and alleviate this performance problem.

Equality tests: The standard equality test in STELLA is `eql?`, which the translator will translate into the most efficient equality test for the particular types of operands (`eql?` is somewhat similar to the Lisp function `CL:eql` with the exception of comparing strings). If the translator can determine that at least one of the operands is a subtype of STANDARD-

OBJECT, it will translate the test into a fast pointer comparison with the Lisp function `CL:eq` or the C++/Java `==` operator. However, if both operands are of type OBJECT, they might be wrapped literals such as wrapped integers or strings. In that case the equality test translates into a call to the function `eql?` which in turn uses method calls to handle comparison of different types of wrapped literals (two wrapped literals are equal if their wrapped content is equal). This is of course a lot less efficient than a simple pointer comparison. It also means that if you can restrict the type of a variable that will be tested with `eql?` to STANDARD-OBJECT, you probably should do so for performance reasons.

Type tests: Run-time type tests as used implicitly within a `typecase` or explicitly with functions such as `cons?` have to use a call to the method `primary-type`. Hence, in performance-critical portions of your code you should try to keep the number of such tests as small as possible.

Wrapping and unwrapping literals: The STELLA translator automatically wraps (or objectifies) literals such as numbers or strings when they are stored in a variable or slot of type OBJECT. Similarly, it unwraps wrapped literals automatically to operate on the literal directly. This is very convenient, since it relieves the programmer from having to perform these conversions by hand and makes the code less cluttered. For example, consider the following code fragment:

```
(let ((l (cons "foo" nil))
      (x (concatenate "bar" (first l))))
  (print x EOL)))
```

Here is its C++ translation:

```
{ Cons* l = cons(stringWrapLiteral("foo"), NIL);
  char* x = stringConcatenate
             ("bar", ((StringWrapper*) (l->first()))->wrapperValue, 0);

  cout << x << endl;
}
```

Notice how the string literal `"foo"` is first wrapped so it can be inserted into the CONS list `l` and then automatically unwrapped in the call to `concatenate`. While this is very convenient, it does cause a certain overhead that should be avoided in performance critical loops, etc. In such situations, it often helps to use auxiliary variables of the appropriate literal type to avoid unnecessary wrap/unwrap operations.

Lisp-style property lists: Lisp programs often use property lists for fast retrieval of information that is linked to symbols. To support the easy translation of existing Lisp programs that use this paradigm into STELLA, a similar mechanism implemented by the functions `symbol-value`, `symbol-plist`, and `symbol-property` is available that preserves the performance benefits of this storage scheme (see the file `sources/stella/symbols.ste`). However, property lists do not fit the object-oriented programming paradigm supported by STELLA, and, hence, are frowned upon.

Compiler optimization: The optimization settings used with the native Lisp or C++ compiler can greatly influence performance results. In particular, using high optimization settings with the Lisp compiler can greatly improve slot access time on STELLA objects.

4.3.1 Lisp Performance Hints

The standard Lisp implementation for STELLA objects are CLOS objects, since CLOS provides the most natural Lisp implementation for the STELLA object system. However, there is a price to pay, since in Lisp slot access on CLOS objects is a lot slower than slot access on structs. For example, in Allegro CL 4.3, the access to the `value` slot of a STELLA CONS cell takes about 4 times longer on a CLOS object implementation of `CONS` than on a struct implementation. Unfortunately, the struct implementation itself takes about 3 times longer than calling `CL:car` on a Lisp cons, which is why we are actually using Lisp conses as the Lisp implementation for STELLA `CONS`es. Note, that in the C++ and Java translation these slot-access performance problems are nonexistent.

In order to get the maximum performance out of the Lisp version of STELLA, you can tell the translator to use structs as the implementation for STELLA objects. It does so by using `CL:defstruct` instead of `CL:defclass` and dispatches methods directly on the structure object.

To use the struct translation scheme evaluate

```
(set-stella-feature :use-common-lisp-structs)
```

before you translate a STELLA system. This will generate translated files with a `.slisp` extension. Make sure that after you translated all the files you are interested in, you disable the above feature with

```
(unset-stella-feature :use-common-lisp-structs)
```

Otherwise, subsequent incremental translations in that Lisp image might fail, since different translation schemes cannot be mixed. If you already are using the struct version of STELLA, all systems will be translated in struct mode by default.

To use the struct translation of your system you have to use the struct version of STELLA. To do so do the following:

```
(CL:setq cl-user::*load-cl-struct-stella?* CL:t)
(CL:load "load-stella.lisp")
```

Alternatively, you can edit the initial value of the variable `*load-cl-struct-stella?*` in the file 'load-stella.lisp' (see also Section 2.3 [Lisp Installation], page 4).

The reasons why the struct translation scheme is not enabled by default are the following:

Incremental redefinition of STELLA classes does not redefine any objects created with the old definition, and, hence, slot accessors might simply break or retrieve the value of a different slot when applied to such an old object. The programmer therefore has to be very careful when redefining a STELLA class while in struct mode. This means, that you should view the usage of the struct-translation scheme for Lisp as a kind of delivery option, similar to translating into C++. Part of the reason why slot access on CLOS object is expensive is the indirection machinery that allows redefinition of classes and their associated instances. This is great for code development, but the flexibility and expense is usually not needed or warranted for delivered code.

The performance trade-offs between CLOS and struct versions might be different in different versions of Lisp. For example, in older version of Allegro CL slot access on structs was fast, but method dispatch was significantly slower than for CLOS objects which eliminated some/all of the performance gains.

5 Library Classes (tbw)

To be written.

6 Library Functions

6.1 Basic Constants and Predicates

true : BOOLEAN   Constant
    Represents the boolean true truth value.

false : BOOLEAN   Constant
    Represents the boolean false truth value.

null? $((x$ OBJECT$))$: BOOLEAN   Method
    Return true if x is undefined (handled specially by all translators).

null? $((x$ SECOND-CLASS-OBJECT$))$: BOOLEAN   Method
    Return true if x is undefined (handled specially by all translators).

null? $((x$ NATIVE-VECTOR$))$: BOOLEAN   Method
    Return true if x is undefined (handled specially by all translators).

null? $((x$ STRING$))$: BOOLEAN   Method
    Return true if x is undefined (handled specially by all translators).

null? $((x$ MUTABLE-STRING$))$: BOOLEAN   Method
    Return true if x is undefined (handled specially by all translators).

null? $((x$ CHARACTER$))$: BOOLEAN   Method
    Return true if x is undefined (handled specially by all translators).

null? $((x$ CODE$))$: BOOLEAN   Method
    Return true if x is undefined (handled specially by all translators).

null? $((x$ INTEGER$))$: BOOLEAN   Method
    Return true if x is undefined (handled specially by all translators).

null? $((x$ FLOAT$))$: BOOLEAN   Method
    Return true if x is undefined (handled specially by all translators).

defined? $((x$ OBJECT$))$: BOOLEAN   Method
    Return true if x is defined (handled specially by all translators).

defined? $((x$ SECOND-CLASS-OBJECT$))$: BOOLEAN   Method
    Return true if x is defined (handled specially by all translators).

defined? ((x SECOND-CLASS-OBJECT)) : BOOLEAN   Method
  Return true if x is defined (handled specially by all translators).

defined? ((x STRING)) : BOOLEAN   Method
  Return true if x is defined (handled specially by all translators).

defined? ((x MUTABLE-STRING)) : BOOLEAN   Method
  Return true if x is defined (handled specially by all translators).

defined? ((x CHARACTER)) : BOOLEAN   Method
  Return true if x is defined (handled specially by all translators).

defined? ((x CODE)) : BOOLEAN   Method
  Return true if x is defined (handled specially by all translators).

defined? ((x INTEGER)) : BOOLEAN   Method
  Return true if x is defined (handled specially by all translators).

defined? ((x FLOAT)) : BOOLEAN   Method
  Return true if x is defined (handled specially by all translators).

eq? ((x UNKNOWN) (y UNKNOWN)) : BOOLEAN   Function
  Return true if x and y are literally the same object (or simple number). Analogue to
  the Common Lisp EQL and C++ and Java's ==.

eql? ((x OBJECT) (y OBJECT)) : BOOLEAN   Function
  Return true if x and y are `eq?` or equivalent literals such as strings that also might be
  wrapped in non-identical wrappers. For the case where x or y are plain literals such as
  strings or integers, the STELLA translator substitutes the equality test appropriate
  for the particular target language and does not actually call this function. For cases
  where x or y are known to be of type STANDARD-OBJECT, the STELLA translator
  substitutes the faster `eq?` test inline.

equal? ((x OBJECT) (y OBJECT)) : BOOLEAN   Function
  Return true if x and y are `eql?` or considered equal by a user-defined `object-equal?`
  method. This implements a fully extensible equality test similar to Java's `equals`
  method.

object-equal? ((x OBJECT) (y OBJECT)) : BOOLEAN   Method
  Return true if x and y are `eq?`.

object-equal? ((x WRAPPER) (y OBJECT)) : BOOLEAN   Method
  Return true if x and y are literal wrappers whose literals are considered `eql?`.

6.2 Numbers

pi : FLOAT   Constant
> A float approximation of the mathematical constant pi.

+ (&rest (*arguments* NUMBER)) : NUMBER   Function
> Return the sum of all *arguments*.

- ((*x* NUMBER) &rest (*arguments* NUMBER)) : NUMBER   Function
> If only *x* was supplied return the result of 0 - *x*. Otherwise, return the result of (...((*x*
> - arg1) - arg2) - ... - argN).

* (&rest (*arguments* NUMBER)) : NUMBER   Function
> Return the product of all *arguments*.

/ ((*x* NUMBER) &rest (*arguments* NUMBER)) : NUMBER   Function
> If only *x* was supplied return the result of 1 / *x*. Otherwise, return the result of (...((*x*
> / arg1) / arg2) / ... / argN).

1+ ((*expression* OBJECT)) : OBJECT   Macro
> Add 1 to *expression* and return the result.

1- ((*expression* OBJECT)) : OBJECT   Macro
> Subtract 1 from *expression* and return the result.

++ ((*place* OBJECT) &body (*increment* CONS)) : OBJECT   Macro
> Increment the value of *place* and return the result. *place* can be either a variable
> name or a slot reference. Increment by the optional *increment* (which can be a float)
> or 1 otherwise.

− ((*place* OBJECT) &body (*decrement* CONS)) : OBJECT   Macro
> Decrement the value of *place* and return the result. *place* can be either a variable
> name or a slot reference. Decrement by the optional *decrement* (which can be a float)
> or 1 otherwise.

= ((*x* NUMBER) (*y* NUMBER)) : BOOLEAN   Function
> Return true if *x* and *y* are numbers of exactly the same magnitude.

< ((*x* NUMBER) (*y* NUMBER)) : BOOLEAN   Function
> Return true if *x* is less than *y*.

<= ((*x* NUMBER) (*y* NUMBER)) : BOOLEAN   Function
> Return true if *x* is less than or equal to *y*.

>= ((*x* NUMBER) (*y* NUMBER)) *:* BOOLEAN   Function
       Return true if *x* is greater than or equal to *y*.

> ((*x* NUMBER) (*y* NUMBER)) *:* BOOLEAN   Function
       Return true if *x* is greater than *y*.

zero? ((*x* INTEGER)) *:* BOOLEAN   Function
       Return true if *x* is 0.

plus? ((*x* INTEGER)) *:* BOOLEAN   Function
       Return true if *x* is greater than 0.

even? ((*x* INTEGER)) *:* BOOLEAN   Function
       Return true if *x* is an even number.

odd? ((*x* INTEGER)) *:* BOOLEAN   Function
       Return true if *x* is an odd number.

mod ((*x* INTEGER) (*modulo* INTEGER)) *:* INTEGER   Function
       Return the result of *x* mod *modulo*.

ceiling ((*n* NUMBER)) *:* INTEGER   Function
       Return the smallest integer >= *n*.

floor ((*n* NUMBER)) *:* INTEGER   Function
       Return the biggest integer <= *n*.

round ((*n* NUMBER)) *:* INTEGER   Function
       Round *n* to the closest integer and return the result.

abs ((*x* INTEGER)) *:* INTEGER   Method
       Return the absolute value of *x*.

abs ((*x* FLOAT)) *:* FLOAT   Method
       Return the absolute value of *x*.

min ((*x* INTEGER) (*y* INTEGER)) *:* INTEGER   Function
       Return the minimum of *x* and *y*. If either is NULL, return the other.

max ((*x* INTEGER) (*y* INTEGER)) *:* INTEGER   Function
       Return the maximum of *x* and *y*. If either is NULL, return the other.

sqrt ((*n* FLOAT)) *:* FLOAT   Function
       Return the square root of *n*.

exp ((*x* FLOAT)) : FLOAT   Function
        The natural exponentiation function e^x.

log ((*n* FLOAT)) : FLOAT   Function
        Return the natural logarithm (base e) of *n*.

sin ((*n* FLOAT)) : FLOAT   Function
        Return the sine of *n* radians.

cos ((*n* FLOAT)) : FLOAT   Function
        Return the cosine of *n* radians.

tan ((*n* FLOAT)) : FLOAT   Function
        Return the tangent of *n* radians.

random ((*n* INTEGER)) : INTEGER   Function
        Generate a random integer in the interval [0..n-1]. *n* must be `<=` 2^15.

integer-to-string ((*i* INTEGER)) : STRING   Function
        Print *i* to a string and return the result. This is more efficient than using a string
        stream.

string-to-integer ((*string* STRING)) : INTEGER   Function
        Convert a *string* representation of an integer into an integer.

float-to-string ((*f* FLOAT)) : STRING   Function
        Print *f* to a string and return the result. This is more efficient than using a string
        stream.

string-to-float ((*string* STRING)) : FLOAT   Function
        Convert a *string* representation of a float into a float.

format-float ((*f* FLOAT) (*nDecimals* INTEGER)) : STRING   Function
        Print *f* in fixed-point format with *nDecimals* behind the decimal point and return
        the result as a string.

wrap-integer ((*value* INTEGER)) : INTEGER-WRAPPER   Function
        Return a literal object whose value is the INTEGER *value*.

unwrap-integer ((*wrapper* INTEGER-WRAPPER)) : INTEGER   Function
        Unwrap *wrapper* and return the result. Return NULL if *wrapper* is NULL.

wrap-float ((*value* FLOAT)) : FLOAT-WRAPPER   Function
        Return a literal object whose value is the FLOAT *value*.

unwrap-float ((*wrapper* FLOAT-WRAPPER)) : FLOAT   Function
        Unwrap *wrapper* and return the result. Return NULL if *wrapper* is NULL.

6.3 Characters

character-code ((*ch* CHARACTER)) *:* INTEGER   Function
    Return the 8-bit ASCII code of *ch* as an integer.

code-character ((*code* INTEGER)) *:* CHARACTER   Function
    Return the character encoded by *code* (0 <= *code* <= 255).

digit-character? ((*ch* CHARACTER)) *:* BOOLEAN   Function
    Return TRUE if *ch* represents a digit.

letter-character? ((*ch* CHARACTER)) *:* BOOLEAN   Function
    Return TRUE if *ch* represents a letter.

upper-case-character? ((*ch* CHARACTER)) *:* BOOLEAN   Function
    Return TRUE if *ch* represents an upper-case character.

lower-case-character? ((*ch* CHARACTER)) *:* BOOLEAN   Function
    Return TRUE if *ch* represents a lower-case character.

white-space-character? ((*ch* CHARACTER)) *:* BOOLEAN   Function
    Return TRUE if *ch* is a white space character.

character-downcase ((*ch* CHARACTER)) *:* CHARACTER   Function
    If *ch* is lowercase, return its uppercase version, otherwise, return *ch* unmodified.

character-upcase ((*ch* CHARACTER)) *:* CHARACTER   Function
    If *ch* is uppercase, return its lowercase version, otherwise, return *ch* unmodified. If only the first character of a sequence of characters is to be capitalized, `character-capitalize` should be used instead.

character-capitalize ((*ch* CHARACTER)) *:* CHARACTER   Function
    Return the capitalized character for *ch*. This is generally the same as the uppercase character, except for obscure non-English characters in Java. It should be used if only the first character of a sequence of characters is to be capitalized.

character-to-string ((*c* CHARACTER)) *:* STRING   Function
    Convert *c* into a one-element string and return the result.

wrap-character ((*value* CHARACTER)) *:* CHARACTER-WRAPPER   Function
    Return a literal object whose value is the CHARACTER *value*.

unwrap-character ((*wrapper* CHARACTER-WRAPPER)) *:* CHARACTER   Function
    Unwrap *wrapper* and return the result. Return NULL if *wrapper* is NULL.

6.4 Strings

string-eql? ((*x* STRING) (*y* STRING)) *:* BOOLEAN   Function
    Return true if *x* and *y* are equal strings or are both undefined. This test is substituted automatically by the STELLA translator if `eql?` is applied to strings.

string-equal? ((*x* STRING) (*y* STRING)) *:* BOOLEAN   Function
    Return true if *x* and *y* are equal strings ignoring character case or are both undefined.

string-compare ((*x* STRING) (*y* STRING) (*case-sensitive?* BOOLEAN)) *:*   Function
       INTEGER
    Compare *x* and *y* lexicographically, and return -1, 0, or 1, depending on whether *x* is less than, equal, or greater than *y*. If *case-sensitive?* is true, then case does matter for the comparison

string< ((*x* STRING) (*y* STRING)) *:* BOOLEAN   Function
    Return true if *x* is lexicographically < *y*, considering case.

string<= ((*x* STRING) (*y* STRING)) *:* BOOLEAN   Function
    Return true if *x* is lexicographically <= *y*, considering case.

string>= ((*x* STRING) (*y* STRING)) *:* BOOLEAN   Function
    Return true if *x* is lexicographically >= *y*, considering case.

string> ((*x* STRING) (*y* STRING)) *:* BOOLEAN   Function
    Return true if *x* is lexicographically > *y*, considering case.

string-less? ((*x* STRING) (*y* STRING)) *:* BOOLEAN   Function
    Return true if *x* is lexicographically < *y*, ignoring case.

string-less-equal? ((*x* STRING) (*y* STRING)) *:* BOOLEAN   Function
    Return true if *x* is lexicographically <= *y*, ignoring case.

string-greater-equal? ((*x* STRING) (*y* STRING)) *:* BOOLEAN   Function
    Return true if *x* is lexicographically >= *y*, ignoring case.

string-greater? ((*x* STRING) (*y* STRING)) *:* BOOLEAN   Function
    Return true if *x* is lexicographically > *y*, ignoring case.

all-upper-case-string? ((*s* STRING)) *:* BOOLEAN   Function
    Return TRUE if all letters in *s* are upper case.

all-lower-case-string? ((*s* STRING)) *:* BOOLEAN   Function
    Return TRUE if all letters in *s* are lower case.

make-string ((*size* INTEGER) (*initchar* CHARACTER)) : STRING   Function
    Return a new string filled with *size initchar*s.

make-mutable-string ((*size* INTEGER) (*initchar* CHARACTER)) :   Function
       MUTABLE-STRING
    Return a new mutable string filled with *size initchar*s.

make-raw-mutable-string ((*size* INTEGER)) : MUTABLE-STRING   Function
    Return a new uninitialized mutable string of *size*.

first ((*self* STRING)) : CHARACTER   Method
    Return the first character of *self*.

first ((*self* MUTABLE-STRING)) : CHARACTER   Method
    Return the first character of *self* (settable via `setf`).

second ((*self* STRING)) : CHARACTER   Method
    Return the second character of *self*.

second ((*self* MUTABLE-STRING)) : CHARACTER   Method
    Return the second character of *self* (settable via `setf`).

third ((*self* STRING)) : CHARACTER   Method
    Return the third character of *self*.

third ((*self* MUTABLE-STRING)) : CHARACTER   Method
    Return the third character of *self* (settable via `setf`).

fourth ((*self* STRING)) : CHARACTER   Method
    Return the fourth character of *self*.

fourth ((*self* MUTABLE-STRING)) : CHARACTER   Method
    Return the fourth character of *self* (settable via `setf`).

fifth ((*self* STRING)) : CHARACTER   Method
    Return the fifth character of *self*.

fifth ((*self* MUTABLE-STRING)) : CHARACTER   Method
    Return the fifth character of *self* (settable via `setf`).

nth ((*self* STRING) (*position* INTEGER)) : CHARACTER   Method
    Return the character in *self* at *position*.

nth ((*self* MUTABLE-STRING) (*position* INTEGER)) : CHARACTER   Method
    Return the character in *self* at *position*.

rest ((*self* STRING)) *:* STRING   Method
       Not documented.

length ((*self* STRING)) *:* INTEGER   Method
       Return the length of the string *self*.

length ((*self* MUTABLE-STRING)) *:* INTEGER   Method
       Return the length of the string *self*.

member? ((*self* STRING) (*char* CHARACTER)) *:* BOOLEAN   Method
       Not documented.

position ((*string* STRING) (*character* CHARACTER) (*start* INTEGER)) *:*   Method
       INTEGER
       Return the position of *character* within *string* (counting from zero); or return NULL
       if *character* does not occur within *string*. If *start* was supplied as non-NULL, only
       consider the substring starting at *start*, however, the returned position will always be
       relative to the entire string.

string-search ((*string* STRING) (*substring* STRING) (*start* INTEGER)) *:*   Function
       INTEGER
       Return start position of the left-most occurrence of *substring* in *string*, beginning
       from *start*. Return NULL if it is not a substring.

copy ((*string* STRING)) *:* STRING   Method
       Return a copy of *string*.

string-upcase ((*string* STRING)) *:* STRING   Function
       Return an upper-case copy of *string*.

string-downcase ((*string* STRING)) *:* STRING   Function
       Return a lower-case copy of *string*.

string-capitalize ((*string* STRING)) *:* STRING   Function
       Return a capitalized version of *string*.

concatenate ((*string1* STRING) (*string2* STRING)   Method
       &rest (*otherStrings* STRING)) *:* STRING
       Return a new string representing the concatenation of *string1*, *string2*, and *other-*
       Strings. The two mandatory parameters allow us to optimize the common binary
       case by not relying on the somewhat less efficient variable arguments mechanism.

subsequence ((*string* STRING) (*start* INTEGER) (*end* INTEGER)) *:* STRING   Method
       Return a substring of *string* beginning at position *start* and ending up to but not
       including position *end*, counting from zero. An *end* value of NULL stands for the rest
       of the string.

remove ((*string* STRING) (*char* CHARACTER)) *:* STRING   Method
 Remove all occurences of *char* from *string*.

substitute ((*self* STRING) (*newChar* CHARACTER) (*oldChar* CHARACTER))   Method
  : STRING
 Substitute all occurences of *oldChar* with *newChar* in the string *self*.

substitute ((*self* MUTABLE-STRING) (*newChar* CHARACTER)   Method
  (*oldChar* CHARACTER)) *:* MUTABLE-STRING
 Substitute all occurences of *oldChar* with *newChar* in the string *self*.

replace-substrings ((*string* STRING) (*new* STRING) (*old* STRING)) *:*   Function
  STRING
 Replace all occurrences of *old* in *string* with *new*.

insert-string ((*source* STRING) (*start* INTEGER) (*end* INTEGER)   Function
  (*target* MUTABLE-STRING) (*target-index* INTEGER) (*case-conversion* KEYWORD))
  : INTEGER
 Inserts characters from *source* begining at *start* and ending at *end* into *target* starting
 at *target-index*. If *end* is `null`, then the entire length of the string is used. The
 copy of characters is affected by the *case-conversion* keyword which should be one of
 :UPCASE :DOWNCASE :CAPITALIZE :PRESERVE.

 The final value of target-index is returned.

wrap-string ((*value* STRING)) *:* STRING-WRAPPER   Function
 Return a literal object whose value is the STRING *value*.

wrap-mutable-string ((*value* MUTABLE-STRING)) *:*   Function
  MUTABLE-STRING-WRAPPER
 Return a literal object whose value is the MUTABLE-STRING *value*.

unwrap-string ((*wrapper* STRING-WRAPPER)) *:* STRING   Function
 Unwrap *wrapper* and return the result. Return NULL if *wrapper* is NULL.

unwrap-mutable-string ((*wrapper* MUTABLE-STRING-WRAPPER)) *:*   Function
  MUTABLE-STRING
 Unwrap *wrapper* and return the result. Return NULL if *wrapper* is NULL.

string-to-mutable-string ((*s* STRING)) *:* MUTABLE-STRING   Function
 Copy *s* into a mutable string with the same content. In Lisp and C++ this simply
 copies *s*.

mutable-string-to-string ((*s* MUTABLE-STRING)) *:* STRING   Function
 Convert *s* into a regular string with the same content. In Lisp and C++ this is a no-op.

integer-to-string ((*i* INTEGER)) *:* STRING   Function
> Print *i* to a string and return the result. This is more efficient than using a string
> stream.

string-to-integer ((*string* STRING)) *:* INTEGER   Function
> Convert a *string* representation of an integer into an integer.

float-to-string ((*f* FLOAT)) *:* STRING   Function
> Print *f* to a string and return the result. This is more efficient than using a string
> stream.

string-to-float ((*string* STRING)) *:* FLOAT   Function
> Convert a *string* representation of a float into a float.

format-float ((*f* FLOAT) (*nDecimals* INTEGER)) *:* STRING   Function
> Print *f* in fixed-point format with *nDecimals* behind the decimal point and return
> the result as a string.

character-to-string ((*c* CHARACTER)) *:* STRING   Function
> Convert *c* into a one-element string and return the result.

stringify ((*expression* OBJECT)) *:* STRING   Function
> Print *expression* onto a string and return the result. Printing is done with `*printReadably?*`
> set to true and with `*printPretty?*` set to false.

stringify-in-module ((*tree* OBJECT) (*module* MODULE)) *:* STRING   Function
> Stringify a parse *tree* relative to *module*, or `*module*` if no module is specified.

unstringify ((*string* STRING)) *:* OBJECT   Function
> Read a STELLA expression from *string* and return the result. This is identical to
> `read-s-expression-from-string`.

unstringify-in-module ((*string* STRING) (*module* MODULE)) *:* OBJECT   Function
> Unstringify relative to *module*, or `*MODULE*` if no module is specified.

6.5 `CONS` Lists and Trees

nil *:* CONS   Variable
> Not documented.

empty? ((*self* CONS)) *:* BOOLEAN   Method
> Return `true` iff *self* equals `nil`.

non-empty? ((*self* CONS)) *:* BOOLEAN   Method
> Return `true` iff *self* is not equal to `nil`.

nil? ((*x* OBJECT)) *:* BOOLEAN   Function
      Return `true` iff *x* equals `nil`.

equal-cons-trees? ((*tree1* OBJECT) (*tree2* OBJECT)) *:* BOOLEAN   Function
      Return `true` iff the cons trees *tree1* and *tree2* are structurally equivalent. Uses an
      `eql?` test.

object-equal? ((*tree1* CONS) (*tree2* OBJECT)) *:* BOOLEAN   Method
      Return `true` iff the cons trees *tree1* and *tree2* are structurally equivalent. Uses `equal?`
      to test equality of subtrees.

cons ((*value* OBJECT) (*rest* CONS)) *:* CONS   Function
      Return a cons record that points to *value* and *rest*.

first ((*self* CONS)) *:* (LIKE (ANY-VALUE SELF))   Method
      Return the first element of *self*. The first element of *self* can be set with `setf`. Note
      that (`first NIL`) = `null`.

second ((*self* CONS)) *:* (LIKE (ANY-VALUE SELF))   Method
      Return the second element of *self*. The second element of *self* can be set with `setf`.
      Note that (`second NIL`) = `null`.

third ((*self* CONS)) *:* (LIKE (ANY-VALUE SELF))   Method
      Return the third element of *self*. The third element of *self* can be set with `setf`. Note
      that (`third NIL`) = `null`.

fourth ((*self* CONS)) *:* (LIKE (ANY-VALUE SELF))   Method
      Return the fourth element of *self*. The fourth element of *self* can be set with `setf`.
      Note that (`fourth NIL`) = `null`.

fifth ((*self* CONS)) *:* (LIKE (ANY-VALUE SELF))   Method
      Return the fifth element of *self*. The fifth element of *self* can be set with `setf`. Note,
      that (`fifth NIL`) = `null`.

nth ((*self* CONS) (*position* INTEGER)) *:* (LIKE (ANY-VALUE SELF))   Method
      Return the element of *self* at *position*. The nth element of *self* can be set with `setf`.
      Note, that (`nth NIL <pos>`) = `null`.

nth-rest ((*self* CONS) (*position* INTEGER)) *:* (LIKE SELF)   Method
      Apply `rest` *position* times to *self*.

last ((*self* CONS)) *:* (LIKE (ANY-VALUE SELF))   Method
      Return the last element of *self*.

but-last ((*self* CONS)) *:* (ITERATOR OF (LIKE (ANY-VALUE SELF)))   Method
      Generate all but the last element of the cons list *self*.

last-cons ((*self* CONS)) : (CONS OF (LIKE (ANY-VALUE SELF)))   Function
> Return the last cons of *self*.

length ((*self* CONS)) : INTEGER   Method
> Return the length of the CONS list *self*.

member? ((*self* CONS) (*object* OBJECT)) : BOOLEAN   Method
> Return `true` iff *object* is a member of the cons list *self* (uses an `eql?` test).

memb? ((*self* CONS) (*object* OBJECT)) : BOOLEAN   Method
> Return `true` iff *object* is a member of the cons list *self* (uses an `eq?` test).

position ((*self* CONS) (*object* OBJECT) (*start* INTEGER)) : INTEGER   Method
> Return the position of *object* within the cons-list *self* (counting from zero); or return
> `null` if *object* does not occur within *self* (uses an `eql?` test). If *start* was supplied as
> non-'null', only consider the sublist starting at *start*, however, the returned position
> will always be relative to the entire list.

reverse ((*self* CONS)) : (LIKE SELF)   Method
> Destructively reverse the members of the cons list *self*.

remove ((*self* CONS) (*value* OBJECT)) : (LIKE SELF)   Method
> Destructively remove all entries in the cons list *self* that match *value*. Unless the
> remaining list is `nil`, insure that the cons that heads the list is unchanged.

remove-duplicates ((*self* CONS)) : (LIKE SELF)   Method
> Destructively remove duplicates from *self* and return the result. Removes all but the
> first occurrence of items in the list. Preserves the original order of the remaining
> members. Runs in linear time.

remove-if ((*self* CONS) (*test?* FUNCTION-CODE)) : (LIKE SELF)   Method
> Destructively removes all members of the cons list *self* for which *test?* evaluates to
> `true`. `test` takes a single argument of type OBJECT and returns `true` or `false`.
> Returns a cons list. In case the first element is removed, the return result should be
> assigned to a variable.

substitute ((*self* CONS) (*inValue* OBJECT) (*outValue* OBJECT)) : CONS   Method
> Destructively replace each appearance of *outValue* by *inValue* in the cons list *self*.

concatenate ((*list1* CONS) (*list2* CONS) &rest (*otherLists* CONS)) : CONS   Method
> Return a cons list consisting of the concatenation of *list1*, *list2*, and *otherLists*. The
> operation is destructive wrt all but the last list argument which is left intact. The two
> mandatory parameters allow us to optimize the common binary case by not relying
> on the somewhat less efficient variable arguments mechanism.

append ((*consList1* CONS) (*consList2* CONS)) : CONS   Function
> Return a cons list representing the concatenation of *consList1* and *consList2*. The concatenation is NOT destructive.

prepend ((*self* CONS) (*list1* CONS)) : CONS   Method
> Return a cons list consisting of the concatenation of *list1* and *self*. A copy of *list1* is prepended to *self*. This operation results in structure sharing of *self*; to avoid this, *self* should not be pointed to by anything other than the tail of the prepended copy.

pushq ((*variable* SYMBOL) (*value* OBJECT)) : OBJECT   Macro
> Push *value* onto the cons list *variable*.

pushq-new ((*variable* SYMBOL) (*value* OBJECT)) : OBJECT   Macro
> Push *value* onto the cons list *variable*, if its not there already.

popq ((*variable* SYMBOL)) : OBJECT   Macro
> Pops a value from the cons list *variable*.

cons-list (&rest (*values* OBJECT)) : CONS   Function
> Return a cons list containing *values*, in order.

list* (&rest (*values* OBJECT)) : CONS   Function
> Return a list of conses that make up the list *values*, terminated by the last value rather than by `nil`. Assumes that at least one value is passed in.

copy-cons-list ((*self* CONS)) : (LIKE SELF)   Function
> Return a copy of the cons list *self*.

copy-cons-tree ((*self* OBJECT)) : (LIKE SELF)   Function
> Return a copy of the cons tree *self*.

substitute-cons-tree ((*tree* OBJECT) (*newValue* OBJECT)   Function
>   (*oldValue* OBJECT)) : OBJECT
> Destructively replace each appearance of *oldValue* by *newValue* in the cons tree *tree*. Return the tree. Uses an `eql?` test.

search-cons-tree? ((*tree* OBJECT) (*value* OBJECT)) : BOOLEAN   Function
> Return `true` iff the value *value* is embedded within the cons tree *tree*. Uses an `eql?` test.

tree-size ((*self* OBJECT)) : INTEGER   Function
> Not documented.

safe-tree-size ((*tree* CONS)) : INTEGER STRING   Function
> Not documented.

consify ((*self* CONS)) *:* (CONS OF (LIKE (ANY-VALUE SELF)))   Method
      Return *self*.

allocate-iterator ((*self* CONS)) *:* (CONS-ITERATOR OF (LIKE   Method
        (ANY-VALUE SELF)))
      Not documented.

next? ((*self* CONS-ITERATOR)) *:* BOOLEAN   Method
      Not documented.

sort ((*self* CONS) (*predicate* FUNCTION-CODE)) *:* (CONS OF (LIKE   Method
        (ANY-VALUE SELF)))
      Perform a stable, destructive sort of *self* according to *predicate*, and return the result. If *predicate* has a `<` semantics, the result will be in ascending order. It is not guaranteed that *self* will point to the beginning of the sorted result. If *predicate* is `null`, a suitable `<` predicate is chosen depending on the first element of *self*, and it is assumed that all elements of *self* have the same type (supported element types are GENERALIZED-SYMBOL, STRING, INTEGER, and FLOAT).

map-null-to-nil ((*self* CONS)) *:* CONS   Function
      Return `nil` iff *self* is `null` or *self* otherwise.

printpretty? : BOOLEAN   Special Variable
      If `true` conses will be pretty printed.

printreadably? : BOOLEAN   Special Variable
      If `true` conses will be printed as readable Stella code.

printprettycode? : BOOLEAN   Special Variable
      When `true` pretty-print Stella and translated code. Since (Lisp) pretty-printing is somewhat slow, turning this off speeds up file translation, but it also makes translated output very unreadable.

6.6 Lists

nil-list : LIST   Variable
      Not documented.

defined-list? ((*self* LIST)) *:* BOOLEAN   Function
      Return TRUE unless *self* is NULL or the `NIL-LIST`.

null-list? ((*self* LIST)) *:* BOOLEAN   Function
      Return TRUE iff *self* is NULL or the `NIL-LIST`.

empty? ((*self* LIST)) *:* BOOLEAN   Method
>   Return TRUE if the list *self* has no members.

non-empty? ((*self* LIST)) *:* BOOLEAN   Method
>   Return TRUE if the list *self* has at least one member.

object-equal? ((*x* LIST) (*y* OBJECT)) *:* BOOLEAN   Method
>   Return TRUE iff the lists *x* and *y* are structurally equivalent. Uses `equal?` to test
>   equality of elements.

list (&rest (*values* OBJECT)) *:* LIST   Function
>   Return a list containing *values*, in order.

first ((*self* LIST)) *:* (LIKE (ANY-VALUE SELF))   Method
>   Return the first item in the list *self*, or NULL if empty.

second ((*self* LIST)) *:* (LIKE (ANY-VALUE SELF))   Method
>   Return the second item in the list *self*, or NULL if empty.

third ((*self* LIST)) *:* (LIKE (ANY-VALUE SELF))   Method
>   Return the third item in the list *self*, or NULL if empty.

fourth ((*self* LIST)) *:* (LIKE (ANY-VALUE SELF))   Method
>   Return the fourth item in the list *self*, or NULL if empty.

fifth ((*self* LIST)) *:* (LIKE (ANY-VALUE SELF))   Method
>   Return the fifth item in the list *self*, or NULL if empty.

nth ((*self* LIST) (*position* INTEGER)) *:* (LIKE (ANY-VALUE SELF))   Method
>   Return the nth item in the list *self*, or NULL if empty.

rest ((*self* LIST)) *:* (CONS OF (LIKE (ANY-VALUE SELF)))   Method
>   Return a cons list of all but the first item in the list *self*.

last ((*self* LIST)) *:* (LIKE (ANY-VALUE SELF))   Method
>   Return the last element of *self*.

but-last ((*self* LIST)) *:* (ITERATOR OF (LIKE (ANY-VALUE SELF)))   Method
>   Generate all but the last element of the list *self*.

length ((*self* LIST)) *:* INTEGER   Method
>   Not documented.

member? ((*self* LIST) (*object* OBJECT)) *:* BOOLEAN   Method
>   Return TRUE iff *object* is a member of the list *self* (uses an `eql?` test).

memb? ((*self* LIST) (*object* (LIKE (ANY-VALUE SELF))))) : BOOLEAN   Method
> Return TRUE iff *object* is a member of the cons list *self* (uses an `eq?` test).

position ((*self* LIST) (*object* OBJECT) (*start* INTEGER)) : INTEGER   Method
> Return the position of *object* within the list *self* (counting from zero); or return NULL
> if *object* does not occur within *self* (uses an `eql?` test). If *start* was supplied as non-
> NULL, only consider the sublist starting at *start*, however, the returned position will
> always be relative to the entire list.

insert ((*self* LIST) (*value* (LIKE (ANY-VALUE SELF)))) :   Method
> Add *value* to the front of the list *self*.

push ((*self* LIST) (*value* (LIKE (ANY-VALUE SELF)))) :   Method
> Add *value* to the front of the list *self*.

insert-new ((*self* LIST) (*value* (LIKE (ANY-VALUE SELF)))) :   Method
> Add *value* to the front of the list *self* unless its already a member.

insert-last ((*self* LIST) (*value* (LIKE (ANY-VALUE SELF)))) :   Method
> Insert *value* as the last entry in the list *self*.

reverse ((*self* LIST)) : (LIKE SELF)   Method
> Reverse the members of *self* (in place).

remove ((*self* LIST) (*value* (LIKE (ANY-VALUE SELF)))) : (LIKE SELF)   Method
> Destructively remove all entries in *self* that match *value*.

remove-duplicates ((*self* LIST)) : (LIKE SELF)   Method
> Destructively remove duplicates from *self* and return the result. Preserves the original
> order of the remaining members.

remove-deleted-members ((*self* LIST)) : (LIKE SELF)   Method
> Not documented.

remove-if ((*self* LIST) (*test?* FUNCTION-CODE)) : (LIKE SELF)   Method
> Destructively remove all members of the list *self* for which *test?* evaluates to TRUE.
> `test` takes a single argument of type OBJECT and returns TRUE or FALSE. Returns
> *self*.

pop ((*self* LIST)) : (LIKE (ANY-VALUE SELF))   Method
> Remove and return the first element in the list *self*. Return NULL if the list is empty.

substitute ((*self* LIST) (*inValue* OBJECT) (*outValue* OBJECT)) : (LIKE   Method
>   SELF)
> Destructively replace each appearance of *outValue* by *inValue* in the list *self*.

concatenate ((*list1* LIST) (*list2* LIST) &rest (*otherLists* LIST)) : LIST   Method
  Copy *list2* and all *otherLists* onto the end of *list1*. The operation is destructive
  wrt *list1*, but leaves all other lists intact. The two mandatory parameters allow us
  to optimize the common binary case by not relying on the somewhat less efficient
  variable arguments mechanism.

prepend ((*self* LIST) (*list2* LIST)) : (LIKE SELF)   Method
  Copy *list2* onto the front of the list *self*. The operation is destructive wrt *self*, but
  leaves *list2* intact.

copy ((*self* LIST)) : (LIST OF (LIKE (ANY-VALUE SELF)))   Method
  Return a copy of the list *self*. The conses in the copy are freshly allocated.

clear ((*self* LIST)) :   Method
  Make *self* an empty list.

consify ((*self* LIST)) : (CONS OF (LIKE (ANY-VALUE SELF)))   Method
  Return a list of elements in *self*.

allocate-iterator ((*self* LIST)) : (LIST-ITERATOR OF (LIKE (ANY-VALUE   Method
      SELF)))
  Not documented.

next? ((*self* LIST-ITERATOR)) : BOOLEAN   Method
  Not documented.

sort ((*self* LIST) (*predicate* FUNCTION-CODE)) : (LIST OF (LIKE   Method
      (ANY-VALUE SELF)))
  Perform a stable, destructive sort of *self* according to *predicate*, and return the result.
  If *predicate* has a < semantics, the result will be in ascending order. If *predicate* is
  NULL, a suitable < predicate is chosen depending on the first element of *self*, and it
  is assumed that all elements of *self* have the same type (supported element types are
  GENERALIZED-SYMBOL, STRING, INTEGER, and FLOAT).

map-null-to-nil-list ((*self* LIST)) : LIST   Function
  Return NIL-LIST iff *self* is NULL or *self* otherwise.

6.7 Property and Key-Value Lists

empty? ((*self* PROPERTY-LIST)) : BOOLEAN   Method
  Not documented.

non-empty? ((*self* PROPERTY-LIST)) : BOOLEAN   Method
  Not documented.

object-equal? ((x PROPERTY-LIST) (y OBJECT)) : BOOLEAN   Method
        Return TRUE if x and y represent the same set of key/value pairs..

length (($self$ PROPERTY-LIST)) : INTEGER   Method
        Not documented.

lookup (($self$ PROPERTY-LIST) (key (LIKE (ANY-KEY SELF)))) : (LIKE   Method
        (ANY-VALUE SELF))
        Not documented.

insert-at (($self$ PROPERTY-LIST) (key (LIKE (ANY-KEY SELF)))   Method
        ($value$ (LIKE (ANY-VALUE SELF)))) :
        Insert the entry <'key', $value$> into the property list $self$. If a previous entry existed
        with key key, that entry is replaced.

remove-at (($self$ PROPERTY-LIST) (key (LIKE (ANY-KEY SELF)))) :   Method
        OBJECT
        Remove the entry that matches the key key. Return the value of the matching entry,
        or NULL if there is no matching entry. Assumes that at most one entry matches key.

copy (($self$ PROPERTY-LIST)) : (LIKE SELF)   Method
        Return a copy of the list $self$. The conses in the copy are freshly allocated.

clear (($self$ PROPERTY-LIST)) :   Method
        Make $self$ an empty property list.

allocate-iterator (($self$ PROPERTY-LIST)) : (PROPERTY-LIST-ITERATOR   Method
        OF (LIKE (ANY-KEY SELF)) (LIKE (ANY-VALUE SELF)))
        Not documented.

next? (($self$ PROPERTY-LIST-ITERATOR)) : BOOLEAN   Method
        Not documented.

kv-cons ((key OBJECT) ($value$ OBJECT) ($rest$ KV-CONS)) : KV-CONS   Function
        Create, fill-in, and return a new KV-CONS.

copy-kv-cons-list (($kvconslist$ KV-CONS)) : KV-CONS   Function
        Return a copy of the cons list consList.

empty? (($self$ KEY-VALUE-LIST)) : BOOLEAN   Method
        Not documented.

non-empty? (($self$ KEY-VALUE-LIST)) : BOOLEAN   Method
        Not documented.

object-equal? ((*x* KEY-VALUE-LIST) (*y* OBJECT)) *:* BOOLEAN   Method
    Return TRUE if *x* and *y* represent the same set of key/value pairs.

length ((*self* KEY-VALUE-LIST)) *:* INTEGER   Method
    Not documented.

lookup ((*self* KEY-VALUE-LIST) (*key* (LIKE (ANY-KEY SELF)))) *:* (LIKE   Method
    (ANY-VALUE SELF))
    Not documented.

reverse ((*self* KEY-VALUE-LIST)) *:* (LIKE SELF)   Method
    Destructively reverse the members of the list *self*.

insert-at ((*self* KEY-VALUE-LIST) (*key* (LIKE (ANY-KEY SELF)))   Method
    (*value* (LIKE (ANY-VALUE SELF)))) *:*
    Insert the entry <'key', *value*> into the association *self*. If a previous entry existed
    with key *key*, that entry is replaced.

remove-at ((*self* KEY-VALUE-LIST) (*key* (LIKE (ANY-KEY SELF)))) *:*   Method
    OBJECT
    Remove the entry that matches the key *key*. Return the value of the matching entry,
    or NULL if there is no matching entry. Assumes that at most one entry matches *key*.

insert-entry ((*self* KEY-VALUE-LIST) (*key* (LIKE (ANY-KEY SELF)))   Method
    (*value* (LIKE (ANY-VALUE SELF)))) *:*
    Insert an entry <'key',*value*> to *self* unless an identical entry already exists. This can
    generate duplicate entries for *key*.

remove-entry ((*self* KEY-VALUE-LIST) (*key* (LIKE (ANY-KEY SELF)))   Method
    (*value* (LIKE (ANY-VALUE SELF)))) *:*
    Remove the entry that matches <'key',*value*>. Assumes that more than one entry can
    match *key*.

push ((*self* KEY-VALUE-LIST) (*value* KV-CONS)) *:*   Method
    Make *value* be the new first element of *self*. Note that the `rest` slot of *value* should be
    `null`, since it will be overwritten. This might duplicate an existing entry. If a previous
    entry existed with the same key as *value*, that entry is retained, but shadowed by this
    new entry.

kv-push ((*self* KEY-VALUE-LIST) (*key* (LIKE (ANY-KEY SELF)))   Method
    (*value* (LIKE (ANY-VALUE SELF)))) *:*
    Add a new entry <'key', *value*> to the front of the association *self*. This might
    duplicate an existing entry. If a previous entry existed with key *key*, that entry is
    retained, but shadowed by this new entry.

pop ((*self* KEY-VALUE-LIST)) *:* (LIKE (ANY-VALUE SELF))   Method
      Remove and return the value of the first element of the kv-list *self*. It does NOT
      return the KV-CONS object. Return `null` if the list is empty.

copy ((*self* KEY-VALUE-LIST)) *:* (LIKE SELF)   Method
      Return a copy of the kv-list *self*. The kv-conses in the copy are freshly allocated.

clear ((*self* KEY-VALUE-LIST)) *:*   Method
      Make *self* an empty dictionary.

consify ((*self* KEY-VALUE-LIST)) *:* (CONS OF (LIKE (ANY-VALUE SELF)))   Method
      Return a list of key-value pairs in *self*.

allocate-iterator ((*self* KEY-VALUE-LIST)) *:* (KV-LIST-ITERATOR OF   Method
        (LIKE (ANY-KEY SELF)) (LIKE (ANY-VALUE SELF)))
      Not documented.

next? ((*self* KV-LIST-ITERATOR)) *:* BOOLEAN   Method
      Not documented.

6.8 Vectors

empty? ((*self* VECTOR)) *:* BOOLEAN   Method
      Return `true` if *self* has length 0.

non-empty? ((*self* VECTOR)) *:* BOOLEAN   Method
      Return `true` if *self* has length > 0.

object-equal? ((*x* VECTOR) (*y* OBJECT)) *:* BOOLEAN   Method
      Return TRUE iff the vectors *x* and *y* are structurally equivalent. Uses `equal?` to test
      equality of elements.

vector (&rest (*values* OBJECT)) *:* VECTOR   Function
      Return a vector containing *values*, in order.

first ((*self* VECTOR)) *:* (LIKE (ANY-VALUE SELF))   Method
      Not documented.

second ((*self* VECTOR)) *:* (LIKE (ANY-VALUE SELF))   Method
      Not documented.

third ((*self* VECTOR)) *:* (LIKE (ANY-VALUE SELF))   Method
      Not documented.

fourth ((*self* VECTOR)) *:* (LIKE (ANY-VALUE SELF))   Method
     Not documented.

fifth ((*self* VECTOR)) *:* (LIKE (ANY-VALUE SELF))   Method
     Not documented.

nth ((*self* VECTOR) (*position* INTEGER)) *:* (LIKE (ANY-VALUE SELF))   Method
     Not documented.

last ((*self* VECTOR)) *:* (LIKE (ANY-VALUE SELF))   Method
     Return the last item in the vector *self*.

but-last ((*self* VECTOR)) *:* (ITERATOR OF (LIKE (ANY-VALUE SELF)))   Method
     Generate all but the last element of the vector *self*.

length ((*self* VECTOR)) *:* INTEGER   Method
     Not documented.

member? ((*self* VECTOR) (*object* OBJECT)) *:* BOOLEAN   Method
     Not documented.

position ((*self* VECTOR) (*object* OBJECT) (*start* INTEGER)) *:* INTEGER   Method
     Return the position of *object* within the vector *self* (counting from zero); or return
     `null` if *object* does not occur within *self* (uses an `eql?` test). If *start* was supplied as
     non-'null', only consider the portion starting at *start*, however, the returned position
     will always be relative to the entire vector.

insert-at ((*self* VECTOR) (*offset* INTEGER)   Method
          (*value* (LIKE (ANY-VALUE SELF)))) *:*
     Not documented.

copy ((*self* VECTOR)) *:* (VECTOR OF (LIKE (ANY-VALUE SELF)))   Method
     Return a copy of the vector *self*.

clear ((*self* VECTOR)) *:*   Method
     Not documented.

resize-vector ((*self* VECTOR) (*size* INTEGER)) *:*   Function
     Change the size of *self* to *size*. If *size* is smaller than the current size of *self* the
     vector will be truncated. Otherwise, the internal array of *self* will be grown to *size*
     and unused elements will be initialized to NULL.

consify ((*self* VECTOR)) *:* (CONS OF (LIKE (ANY-VALUE SELF)))   Method
     Return a list of elements in *self*.

insert-at ((*self* EXTENSIBLE-VECTOR) (*offset* INTEGER)   Method
      (*value* (LIKE (ANY-VALUE SELF)))) :
    Not documented.

insert ((*self* VECTOR-SEQUENCE) (*value* (LIKE (ANY-VALUE SELF)))) :   Method
    Append *value* to the END of the sequence *self*. Resize the array if necessary.

remove ((*self* VECTOR-SEQUENCE) (*value* (LIKE (ANY-VALUE SELF)))) :   Method
      VECTOR-SEQUENCE
    Remove *value* from the sequence *self*, and left shift the values after it to close the gap.

length ((*self* VECTOR-SEQUENCE)) : INTEGER   Method
    Not documented.

6.9 Hash Tables

lookup ((*self* HASH-TABLE) (*key* (LIKE (ANY-KEY SELF)))) : (LIKE   Method
      (ANY-VALUE SELF))
    Not documented.

insert-at ((*self* HASH-TABLE) (*key* (LIKE (ANY-KEY SELF)))   Method
      (*value* (LIKE (ANY-VALUE SELF)))) :
    Not documented.

remove-at ((*self* HASH-TABLE) (*key* (LIKE (ANY-KEY SELF)))) :   Method
    Not documented.

lookup ((*self* STRING-HASH-TABLE) (*key* STRING)) : (LIKE (ANY-VALUE   Method
      SELF))
    Not documented.

insert-at ((*self* STRING-HASH-TABLE) (*key* STRING) (*value* OBJECT)) :   Method
    Not documented.

remove-at ((*self* STRING-HASH-TABLE) (*key* STRING)) :   Method
    Not documented.

lookup ((*self* STRING-TO-INTEGER-HASH-TABLE) (*key* STRING)) : INTEGER   Method
    Not documented.

insert-at ((*self* STRING-TO-INTEGER-HASH-TABLE) (*key* STRING)   Method
      (*value* INTEGER)) :
    Not documented.

lookup ((*self* INTEGER-HASH-TABLE) (*key* INTEGER)) *:* (LIKE (ANY-VALUE   Method
       SELF))
     Not documented.

insert-at ((*self* INTEGER-HASH-TABLE) (*key* INTEGER) (*value* OBJECT)) *:*   Method
     Not documented.

insert-at ((*self* FLOAT-HASH-TABLE) (*key* FLOAT) (*value* OBJECT)) *:*   Method
     Not documented.

6.10 Iterators

empty? ((*self* ITERATOR)) *:* BOOLEAN   Method
     Return TRUE if the sequence represented by *self* has no elements. Side-effect free.

member? ((*self* ITERATOR) (*value* OBJECT)) *:* BOOLEAN   Method
     Iterate over values of *self*, returning TRUE if one of them is `eql` to 'value.

length ((*self* ABSTRACT-ITERATOR)) *:* INTEGER   Method
     Iterate over *self*, and count how many items there are. Bad idea if *self* iterates over
     an infinite collection, since in that case it will run forever.'

pop ((*self* ITERATOR)) *:* (LIKE (ANY-VALUE SELF))   Method
     Return the first item of the sequence represented by *self*, or NULL if it is empty.
     Destructively uses up the first iteration element.

advance ((*self* ITERATOR) (*n* INTEGER)) *:* (LIKE SELF)   Method
     Return *self* after skipping over the first *n* elements in the (remainder of the) iteration.

concatenate ((*iterator1* ITERATOR) (*iterator2* ITERATOR)   Method
       &rest (*otherIterators* ITERATOR)) *:* ALL-PURPOSE-ITERATOR
     Return an iterator that first generates all values of *iterator1*, then those of *iterator2*,
     and then those of all *otherIterators*. The generated values can be filtered by supplying
     a filter function to the resulting iterator.

consify ((*self* ITERATOR)) *:* (CONS OF (LIKE (ANY-VALUE SELF)))   Method
     Return a list of elements generated by *self*.

next? ((*self* ALL-PURPOSE-ITERATOR)) *:* BOOLEAN   Method
     Apply the stored `next?` function to *self*.

6.11 Symbols

lookup-symbol ((*name* STRING)) *:* SYMBOL   Function
    Return the first symbol with *name* visible from the current module.

intern-symbol ((*name* STRING)) *:* SYMBOL   Function
    Return a newly-created or existing symbol with name *name*.

unintern-symbol ((*self* SYMBOL)) *:*   Function
    Remove *self* from its home module and the symbol table.

lookup-symbol-in-module ((*name* STRING) (*module* MODULE)   Function
        (*local?* BOOLEAN)) *:* SYMBOL
    Return the first symbol with *name* visible from *module*. If *local?* only consider
    symbols directly interned in *module*. If *module* is `null`, use `*MODULE*` instead.

intern-symbol-in-module ((*name* STRING) (*module* MODULE)   Function
        (*local?* BOOLEAN)) *:* SYMBOL
    Look for a symbol named *name* in *module* (if *local?* do not consider inherited mod-
    ules). If none exists, intern it locally in *module*. Return the existing or newly-created
    symbol.

intern-derived-symbol ((*baseSymbol* GENERALIZED-SYMBOL)   Function
        (*newName* STRING)) *:* SYMBOL
    Return a newly-created or existing symbol with name *newName* which is interned in
    the same module as *baseSymbol*.

visible-symbol? ((*self* SYMBOL)) *:* BOOLEAN   Function
    Return `true` if *self* is visible from the current module.

lookup-visible-symbols-in-module ((*name* STRING)   Function
        (*module* MODULE) (*enforceShadowing?* BOOLEAN)) *:* (CONS OF SYMBOL)
    Return the list of symbols with *name* visible from *module*. More specific symbols
    (relative to the module precedence order defined by `visible-modules`) come earlier in
    the list. If *module* is `null`, start from `*MODULE*` instead. If *enforceShadowing?* is true,
    do not return any symbols that are shadowed due to some :SHADOW declaration.

lookup-surrogate ((*name* STRING)) *:* SURROGATE   Function
    Return the first surrogate with *name* visible from the current module.

intern-surrogate ((*name* STRING)) *:* SURROGATE   Function
    Return a newly-created or existing surrogate with name *name*.

unintern-surrogate ((*self* SURROGATE)) *:*   Function
    Remove *self* from its home module and the surrogate table.

lookup-surrogate-in-module ((*name* STRING) (*module* MODULE)   Function
      (*local?* BOOLEAN)) : SURROGATE
    Return the first surrogate with *name* visible from *module*. If *local?* only consider
    surrogates directly interned in *module*. If *module* is `null`, use `*MODULE*` instead.

intern-surrogate-in-module ((*name* STRING) (*module* MODULE)   Function
      (*local?* BOOLEAN)) : SURROGATE
    Look for a symbol named *name* in *module* (if *local?* do not consider inherited mod-
    ules). If none exists, intern it locally in *module*. Return the existing or newly-created
    symbol.

intern-derived-surrogate ((*baseSymbol* GENERALIZED-SYMBOL)   Function
      (*newName* STRING)) : SURROGATE
    Return a newly-created or existing surrogate with name *newName* which is interned
    in the same module as *baseSymbol*.

visible-surrogate? ((*self* SURROGATE)) : BOOLEAN   Function
    Return `true` if *self* is visible from the current module.

lookup-visible-surrogates-in-module ((*name* STRING)   Function
      (*module* MODULE) (*enforceShadowing?* BOOLEAN)) : (CONS OF SURROGATE)
    Return the list of surrogates with *name* visible from *module*. More specific surrogates
    (relative to the module precedence order defined by `visible-modules`) come earlier
    in the list. If *module* is `null`, start from `*MODULE*` instead. If *enforceShadowing?*
    is true, do not return any surrogates that are shadowed due to some :SHADOW
    declaration.

lookup-keyword ((*name* STRING)) : KEYWORD   Function
    Return the keyword with *name* if it exists.

intern-keyword ((*name* STRING)) : KEYWORD   Function
    Return a newly-created or existing keyword with name *name*. Storage note: a COPY
    of *name* is stored in the keyword

gensym ((*prefix* STRING)) : SYMBOL   Function
    Return a transient symbol with a name beginning with *prefix* and ending with a
    globally gensym'd integer.

local-gensym ((*prefix* STRING)) : SYMBOL   Function
    Not documented.

symbol-plist ((*symbol* SYMBOL)) : CONS   Function
    Return the property list of *symbol*. The `symbol-plist` of a symbol can be set with
    `setf`. IMPORTANT: Property list are modified destructively, hence, if you supply it
    as a whole make sure to always supply a modfiable copy, e.g., by using `bquote`.

symbol-property ((*symbol* SYMBOL) (*key* STANDARD-OBJECT)) :   Function
       OBJECT

    Return the property of *symbol* whose key is `eq?` to *key*. Symbol properties can be
    set with `setf`.

symbol-value ((*symbol* SYMBOL)) : OBJECT   Function

    Return the value of *symbol*. Note, that this value is not visible to code that references
    a variable with the same name as *symbol*. The `symbol-value` is simply a special
    property that can always be accessed in constant time. The `symbol-value` of a
    symbol can be changed with `setf`.

symbolize ((*surrogate* SURROGATE)) : SYMBOL   Function

    Convert *surrogate* into a symbol with the same name and module.

6.12 Context and Modules

get-stella-context ((*pathName* STRING) (*error?* BOOLEAN)) :   Function
       CONTEXT

    Return the context located at *pathName*, or `null` if no such context exists. If *error?*
    is `true`, throw an exception if no context is found, otherwise silently return `null`.

clear-context ((*self* CONTEXT)) :   Function

    Destroy all objects belonging to *self* or any of its subcontexts.

within-context ((*contextForm* OBJECT) &body (*body* CONS)) : OBJECT   Macro

    Execute *body* within the context resulting from *contextForm*.

destroy-context ((*self* CONTEXT)) :   Method

    Make the translator happy.

destroy-context ((*self* STRING)) :   Method

    Destroy the context *self*, and recursively destroy all contexts that inherit *self*.

change-context ((*context* CONTEXT)) : CONTEXT   Method

    Change the current context to be the context *context*.

change-context ((*contextName* STRING)) : CONTEXT   Method

    Change the current context to be the context named *contextName*.

cc (&rest (*name* NAME)) : CONTEXT   Command

    Change the current context to the one named *name*. Return the value of the new
    current context. If no *name* is supplied, return the pre-existing value of the current
    context. `cc` is a no-op if the context reference cannot be successfully evaluated.

defmodule ((*name* NAME) &rest (*options* OBJECT)) :   Command
    Define (or redefine) a module named *name*. The accepted syntax is:

```
(defmodule <module-name>
    [:documentation <docstring>]
    [:includes {<module-name> | (<module-name>*)}]
    [:uses {<module-name> | (<module-name>*)}]
    [:lisp-package <package-name-string>]
    [:java-package <package-specification-string>]
    [:cpp-namespace <namespace-name-string>]
    [:java-catchall-class
    [:api? {TRUE | FALSE}]
    [:case-sensitive? {TRUE | FALSE}]
    [:shadow (<symbol>*)]
    [:java-catchall-class <class-name-string>]
    [<other-options>*])
```

name can be a string or a symbol.

Modules include objects from other modules via two separate mechanisms: (1) they inherit from their parents specified via the `:includes` option and/or a fully qualified module name, and (2) they inherit from used modules specified via the `:uses` option. The main difference between the two mechanisms is that inheritance from parents is transitive, while uses-links are only followed one level deep. I.e., a module A that uses B will see all objects of B (and any of B's parents) but not see anything from modules used by B. Another difference is that only objects declared as public can be inherited via uses-links (this is not yet enforced). Note that - contrary to Lisp - there are separate name spaces for classes, functions, and variables. For example, a module could inherit the class `CONS` from the `STELLA` module, but shadow the function of the same name.

The above discussion of `:includes` and `:uses` semantics keyed on the inheritance/visibility of symbols. The PowerLoom system makes another very important distinction: If a module `A` is inherited directly or indirectly via `:includes` specification(s) by a submodule `B`, then all definitions and facts asserted in `A` are visible in `B`. This is not the cases for `:uses`; the `:uses` options does not impact inheritance of propositions at all.

The list of modules specified in the `:includes` option plus (if supplied) the parent in the path used for *name* become the new module's parents. If no `:uses` option was supplied, the new module will use the `STELLA` module by default, otherwise, it will use the set of specified modules. If `:case-sensitive?` is supplied as TRUE, symbols in the module will be interned case-sensitively, otherwise (the default), they will be converted to uppercase before they get interned. Modules can shadow definitions of functions and classes inherited from parents or used modules. Shadowing is done automatically, but generates a warning unless the shadowed type or function name is listed in the `:shadow` option of the module definition .

Examples:

```
(defmodule "PL-KERNEL/PL-USER"
  :uses ("LOGIC" "STELLA")
  :package "PL-USER")
```

```
(defmodule PL-USER/GENEALOGY)
```
The remaining options are relevant only for modules that contain STELLA code. Modules used only to contain knowledge base definitions and assertions have no use for them:

The keywords `:lisp-package`, `:java-package`, and `:cpp-package` specify the name of a native package or name space in which symbols of the module should be allocated when they get translated into one of Lisp, Java, or C++. By default, Lisp symbols are allocated in the `STELLA` package, and C++ names are translated without any prefixes. The rules that the STELLA translator uses to attach translated Java objects to classes and packages are somewhat complex. Use :java-package option to specify a list of package names (separated by periods) that prefix the Java object in this module. Use :java-catchall-class to specify the name of the Java class to contain all global & special variables, parameter-less functions and functions defined on arguments that are not classes in the current module. The default value will be the name of the module.

When set to TRUE, the :api? option tells the PowerLoom User Manual generator that all functions defined in this module should be included in the API section. Additionally, the Java translator makes all API functions `synchronized`.

get-stella-module ((*pathName* STRING) (*error?* BOOLEAN)) : MODULE   Function
Return the module located at *pathName*, or `null` if no such module exists. The search looks at ancestors and top-most (cardinal) modules. If *error?* is `true`, throw an exception if no module is found.

find-or-create-module ((*pathname* STRING)) : MODULE   Function
Return a module located at *pathname* if one exists, otherwise create one

clear-module (&rest (*name* NAME)) :   Command
Destroy all objects belonging to module *name* or any of its children. If no *name* is supplied, the current module will be cleared after confirming with the user. Important modules such as STELLA are protected against accidental clearing.

destroy-module ((*self* MODULE)) :   Function
Destroy the module *self*, and recursively destroy all contexts that inherit *self*.

destroy-context ((*self* MODULE)) :   Method
Destroy the context *self*, and recursively destroy all contexts that inherit *self*.

visible-modules ((*from* MODULE)) : (ITERATOR OF MODULE)   Function
Return an iterator that generates all modules visible from module *from*. The generated modules are generated from most- to least-specific and will start with the module *from*.

within-module ((*moduleForm* OBJECT) &body (*body* CONS)) : OBJECT   Macro
Execute *body* within the module resulting from *moduleForm*. `*module*` is an acceptable *moduleForm*. It will locally rebind `*module*` and `*context*` and shield the outer bindings from changes.

in-module ((*name* NAME)) *:* MODULE   Command
    Change the current module to the module named *name*.

change-module ((*module* MODULE)) *:* MODULE   Method
    Change the current module to be the module *module*.

change-module ((*moduleName* STRING)) *:* MODULE   Method
    Change the current module to be the module named *moduleName*.

create-world ((*parentContext* CONTEXT) (*name* STRING)) *:* WORLD   Function
    Create a new world below the world or module *parentContext*. Optionally, specify a
    name.

push-world () *:* WORLD   Function
    Spawn a new world that is a child of the current context, and change the current
    context to the new world.

pop-world () *:* CONTEXT   Function
    Destroy the current world and change the current context to be its parent. Return
    the current context. Nothing happens if there is no current world.

destroy-context ((*self* WORLD)) *:*   Method
    Destroy the context *self*, and recursively destroy all contexts that inherit *self*.

within-world ((*worldForm* OBJECT) &body (*body* CONS)) *:* OBJECT   Macro
    Execute *body* within the world resulting from *worldForm*.

6.13 Input and Output

read-s-expression ((*stream* INPUT-STREAM)) *:* OBJECT BOOLEAN   Function
    Read one STELLA s-expression from *stream* and return the result. Return `true` as
    the second value on EOF.

read-s-expression-from-string ((*string* STRING)) *:* OBJECT   Function
    Read one STELLA s-expression from *string* and return the result.

read-line ((*inputStream* INPUT-STREAM)) *:* STRING BOOLEAN   Function
    Read one line from *inputStream* and return the result. Return `true` as the second
    value on EOF.

read-character ((*inputStream* INPUT-STREAM)) *:* CHARACTER   Function
        BOOLEAN
    Read one character from *inputStream* and return the result. Return `true` as the
    second value on EOF.

unread-character ((*ch* CHARACTER) (*inputStream* INPUT-STREAM)) :   Function
  Unread *ch* from *inputStream*. Signal an error if *ch* was not the last character read.

y-or-n? ((*message* STRING)) : BOOLEAN   Function
  Read a line of input from STANDARD-INPUT and return `true` if the input was `y`
  or `false` if the input was `n`. Loop until either `y` or `n` was entered. If *message* is
  non-'null' prompt with it before the input is read. See also special variable `*USER-`
  `QUERY-ACTION*`.

yes-or-no? ((*message* STRING)) : BOOLEAN   Function
  Read a line of input from STANDARD-INPUT and return `true` if the input was `yes`
  or `false` if the input was `no`. Loop until either `yes` or `no` was entered. If *message* is
  non-'null' prompt with it before the input is read. See also special variable `*USER-`
  `QUERY-ACTION*`.

6.14 Files

probe-file? ((*fileName* FILE-NAME)) : BOOLEAN   Function
  Return true if file *fileName* exists. Note that this does not necessarily mean that the
  file can also be read.

file-write-date ((*fileName* FILE-NAME)) : CALENDAR-DATE   Function
  Return the time at which file *fileName* was last modified or NULL if that cannot be
  determined.

file-length ((*fileName* FILE-NAME)) : INTEGER   Function
  Return the length of file *fileName* in bytes or NULL if that cannot be determined.
  Note that this will currently overrun for files that are longer than what can be repre-
  sented by a STELLA integer.

copy-file ((*fromFile* FILE-NAME) (*toFile* FILE-NAME)) :   Function
  Copy file *fromFile* to file *toFile*, clobbering any data already in *toFile*.

delete-file ((*fileName* FILE-NAME)) :   Function
  Delete the file *fileName*.

directory-file-name ((*directory* FILE-NAME)) : FILE-NAME   Function
  Return *directory* as a file name, i.e., without a terminating directory separator.

directory-parent-directory ((*directory* FILE-NAME) (*level* INTEGER))   Function
   : FILE-NAME
  Return the *level*-th parent directory component of *directory* including the final direc-
  tory separator, or the empty string if *directory* does not have that many parents.

file-name-as-directory ((*file* FILE-NAME)) : FILE-NAME   Function
> Return *file* interpreted as a directory, i.e., with a terminating directory separator. If *file* is the empty string simply return the empty string, i.e., interpret it as the current directory instead of the root directory.

file-name-directory ((*file* FILE-NAME)) : FILE-NAME   Function
> Return the directory component of *file* including the final directory separator or the empty string if *file* does not include a directory.

file-name-without-directory ((*file* FILE-NAME)) : FILE-NAME   Function
> Not documented.

file-name-without-extension ((*file* FILE-NAME)) : FILE-NAME   Function
> Not documented.

file-extension ((*file* FILE-NAME)) : STRING   Function
> Not documented.

file-base-name ((*file* FILE-NAME)) : FILE-NAME   Function
> Not documented.

absolute-pathname? ((*pathname* STRING)) : BOOLEAN   Function
> Not documented.

logical-host? ((*host* STRING)) : BOOLEAN   Function
> Not documented.

logical-pathname? ((*pathname* STRING)) : BOOLEAN   Function
> Not documented.

translate-logical-pathname ((*pathname* STRING)) : STRING   Function
> Not documented.

directory-separator () : CHARACTER   Function
> Not documented.

directory-separator-string () : STRING   Function
> Not documented.

6.15 Dates and Times

get-current-date-time () *:* INTEGER INTEGER INTEGER KEYWORD   Function
        INTEGER INTEGER INTEGER INTEGER
    Returns the current time in UTC as multiple values of year month day day-of-week
    hour minute second millisecond. Currently millisecond will always be zero (even in
    Java where it is technically available).

get-local-time-zone () *:* FLOAT   Function
    Returns the current time zone offset from UTC as a float.

float.get-calendar-date   ???
    Not yet implemented.

float.get-time   ???
    Not yet implemented.

make-current-date-time () *:* CALENDAR-DATE   Function
    Create a calendar date with current time and date.

make-date-time ((*year* INTEGER) (*month* INTEGER) (*day* INTEGER)   Function
      (*hour* INTEGER) (*minute* INTEGER) (*second* INTEGER) (*millis* INTEGER)
      (*timezone* FLOAT)) *:* CALENDAR-DATE
    Create a calendar date with the specified components. *year* must be the complete
    year (i.e., a year of 98 is 98 A.D in the 1st century). *timezone* is a real number in the
    range -12.0 to +14.0 where UTC is zone 0.0; The number is the number of hours to
    add to UTC to arrive at local time.

parse-date-time ((*date-time-string* STRING) (*start* INTEGER)   Function
      (*end* INTEGER) (*error-on-mismatch?* BOOLEAN)) *:* DECODED-DATE-TIME
    Tries very hard to make sense out of the argument *date-time-string* and returns a
    time structure if successful. If not, it returns `null`. If *error-on-mismatch?* is true,
    parse-date-time will signal an error instead of returning `null`. Default values are
    00:00:00 UTC on the current date

decode-calendar-date ((*date* CALENDAR-DATE) (*timezone* FLOAT)) *:*   Method
      DECODED-DATE-TIME
    Returns a decoded time object for *date* interpreted in *timezone timezone* is the num-
    ber of hours added to UTC to get local time. It is in the range -12.0 to +14.0 where
    UTC is zone 0.0

encode-calendar-date ((*time-structure* DECODED-DATE-TIME)) *:*   Method
      CALENDAR-DATE
    Returns a calendar date object for *time-structure*.

calendar-date-to-string ((*date* CALENDAR-DATE) (*timezone* FLOAT)) *:*   Function
       STRING
    Returns a string representation of *date* adjusted for *timezone*

string-to-calendar-date ((*input* STRING)) *:* CALENDAR-DATE   Function
    Returns a calendar date object representing the date and time parsed from the *input*
    string. If no valid parse is found, `null` is returned.

relative-date-to-string ((*date* RELATIVE-DATE)) *:* STRING   Function
    Returns a string representation of *date*

compute-calendar-date ((*julian-day* INTEGER)) *:* INTEGER INTEGER   Function
       INTEGER KEYWORD
    Returns the YEAR, MONTH, DAY, DAY-OF-WEEK on which the given *julian-day*
    begins at noon.

compute-day-of-week ((*yyyy* INTEGER) (*mm* INTEGER)   Function
       (*dd* INTEGER)) *:* KEYWORD
    Returns the day of the week for yyyy-mm-dd.

compute-day-of-week-julian ((*julian-day* INTEGER)) *:* KEYWORD   Function
    Returns the day of the week for julian-day

compute-julian-day ((*yyyy* INTEGER) (*mm* INTEGER) (*dd* INTEGER)) *:*   Function
       INTEGER
    Returns the Julian day that starts at noon on yyyy-mm-dd. *yyyy* is the year. *mm* is
    the month. *dd* is the day of month. Negative years are B.C. Remember there is no
    year zero.

compute-next-moon-phase ((*n* INTEGER) (*phase* KEYWORD)) *:*   Function
       INTEGER FLOAT
    Returns the Julian Day and fraction of day of the Nth occurence since January 1,
    1900 of moon PHASE. PHASE is one of :NEW-MOON, :FIRST-QUARTER, :FULL-
    MOON, :LAST-QUARTER

decode-time-in-millis ((*time* INTEGER)) *:* INTEGER INTEGER INTEGER   Function
       INTEGER
    Returns multiple values of hours, minutes, seconds, milliseconds for *time* specified in
    milliseconds.

julian-day-to-modified-julian-day ((*julian-day* INTEGER)) *:*   Function
       INTEGER
    Returns the modified Julian day during which *julian-day*starts at noon.

modified-julian-day-to-julian-day ((*modified-julian-day* INTEGER))   Function
       : INTEGER
    Returns the modified Julian day during which `julian-day`starts at noon.

time-add (($t1$ DATE-TIME-OBJECT) ($t2$ DATE-TIME-OBJECT)) :   Function
      DATE-TIME-OBJECT
> Add $t1$ to $t2$. If one of $t1$ or $t2$ is a calendar date, then the result is a calendar date. If both $t1$ and $t2$ are relative dates, then the result is a relative date. $t1$ and $t2$ cannot both be calendar dates.

time-divide (($t1$ RELATIVE-DATE) ($t2$ OBJECT)) : OBJECT   Function
> Divides the relative date $t1$ by $t2$. $t2$ must be either a relative date or a wrapped number. If $t2$ is a relative date, then the return value will be a wrapped float. If $t2$ is a wrapped number, then the reutrn value will be a relative date.

time-multiply (($t1$ OBJECT) ($t2$ OBJECT)) : RELATIVE-DATE   Function
> Multiplies a relative date by a wrapped number. One of $t1$ or $t2$ must be a relative date and the other a wrapped number.

time-subtract (($t1$ DATE-TIME-OBJECT) ($t2$ DATE-TIME-OBJECT)) :   Function
      DATE-TIME-OBJECT
> Subtract $t2$ from $t1$. If $t1$ is a calendar date, then $t2$ can be either a calendar date (in which case the return value is a relative date) or it can be a relative date (in which case the return value is a calendar date). If $t1$ is a relative date, then $t2$ must also be a relative date and a relative date is returned.

get-ticktock () : TICKTOCK   Function
> Return the current CPU time. If the current OS/Language combination does not support measuring of CPU time, return real time instead. Use `ticktock-difference` to measure the time difference between values returned by this function. This is an attempt to provide some platform independent support to measure (at least approximately) consumed CPU time.

ticktock-difference (($t1$ TICKTOCK) ($t2$ TICKTOCK)) : FLOAT   Function
> The difference in two TICKTOCK time values in seconds where $t1$ is the earlier time. The resolution is implementation dependent but will normally be some fractional value of a second.

ticktock-resolution () : FLOAT   Function
> The minimum theoretically detectable resolution of the difference in two TICKTOCK time values in seconds. This resolution is implementation dependent. It may also not be realizable in practice, since the timing grain size may be larger than this resolution.

sleep (($seconds$ FLOAT)) :   Function
> The program will sleep for the indicated number of seconds. Fractional values are allowed, but the results are implementation dependent: Common Lisp uses the fractions natively, Java with a resolution of 0.001, and C++ can only use integral values.

6.16 XML Support

get-xml-attributes ((*form* CONS)) : OBJECT   Macro
    Returns a CONS consisting of the attributes for the XML expression *form*. May be
    `nil`.

get-xml-tag ((*form* CONS)) : OBJECT   Macro
    Returns the XML tag object from the XML expression *form*

make-xml-element ((*name* STRING) (*namespace-name* STRING)   Function
        (*namespace* STRING)) : XML-ELEMENT
    Creates and interns an XML element object *name* using *namespace-name* to refer
    to *namespace*. If *namespace* is `null`, then the element will be interned in the null
    namespace. *namespace* must otherwise be a URI.

make-xml-global-attribute ((*name* STRING)   Function
        (*namespace-name* STRING) (*namespace* STRING)) : XML-GLOBAL-ATTRIBUTE
    Creates and interns an XML global attribute object with *name* using *namespace-name*
    to refer to *namespace*. *namespace*must be a URI.

make-xml-local-attribute ((*name* STRING) (*element* XML-ELEMENT))   Function
        : XML-LOCAL-ATTRIBUTE
    Make an XML-LOCAL-ATTRIBUTE named *name* associated with *element*

print-xml-expression ((*stream* OUTPUT-STREAM)   Function
        (*xml-expression* CONS)) :
    Prints *xml-expression* on *stream*. It is assumed that the *xml-expression* is a well-
    formed CONS-list representation of an XML form. It expects a form like that form
    returned by `read-XML-expression`.
    Also handles a list of xml forms such as that returned by `XML-expressions`

read-xml-expression ((*stream* INPUT-STREAM)   Function
        (*startTagName* STRING)) : OBJECT BOOLEAN
    Read one balanced XML expression from *stream* and return its s-expression represen-
    tation (see `xml-token-list-to-s-expression`). If *startTagName* is non-'null', skip
    all tags until a start tag *startTagName* is encountered. XML namespaces are ignored
    for outside of the start tag. Return `true` as the second value on EOF.
    CHANGE WARNING: It is anticipated that this function will change to a) Properly
    take XML namespaces into account and b) require XML element objects instead of
    strings as the second argument. This change will not be backwards-compatible.

reset-xml-hash-tables () :   Function
    Resets Hashtables used for interning XML elements and global attribute objects. This
    will allow garbage collection of no-longer used objects, but will also mean that newly
    parsed xml elements and global attributes will not be eq? to already existing ones
    with the same name.

xml-attribute? ((*item* OBJECT)) : BOOLEAN   Function
    Return `true` if *item* is an XML attribute object

xml-cdata-form? ((*form* OBJECT)) : BOOLEAN   Function
    Return `true` if *form* is an CONS headed by a CDATA tag

xml-cdata? ((*item* OBJECT)) : BOOLEAN   Function
    Return `true` if *item* is an XML CDATA tag object

xml-declaration? ((*item* OBJECT)) : BOOLEAN   Function
    Return `true` if *item* is an XML declaration object

xml-element? ((*item* OBJECT)) : BOOLEAN   Function
    Return `true` if *item* is an XML element object

xml-expressions ((*stream* INPUT-STREAM) (*regionTag* OBJECT)) :   Function
    XML-EXPRESSION-ITERATOR
    Return an XML-expression-iterator (which see) reading from *stream*. *regionTag* can
    be used to define delimited regions from which expressions should be considered (use
    s-expression representation to specify *regionTag*, e.g., (KIF (:version "1.0")). The
    tag can be an XML element object, a symbol, a string or a cons. If the tag is a cons
    the first element can also be (name namespace) pair.

xml-tag-case ((*item* OBJECT) &body (*clauses* CONS)) : OBJECT   Macro
    A case form for matching *item* against XML element tags. Each element of *clauses*
    should be a clause with the form ("tagname" ...) or (("tagname" "namespace-uri")
    ...) The clause heads can optionally be symbols instead of strings. The key forms the
    parameters to the method `xml-element-match?`, with a missing namespace argument
    passed as NULL.

xml-token-list-to-s-expression ((*tokenList* STELLA-TOKEN)) :   Function
    OBJECT
    Convert the XML *tokenList* into a representative s-expression and return the result.
    Every XML tag is represented as a cons-list starting with the tag name as its header,
    followed by a possibly empty list of keyword value pairs representing tag attributes,
    followed by a possibly empty list of content expressions which might themselves be
    XML expressions. For example, the expression

    <a a1=v1 a2='v2'> foo <b a3=v3/> bar

    becomes

    (a (:a1 "v1" :a2 "v2") "foo" (b (:a3 "v3")) "bar")

    when represented as an s-expression. The tag names `?`, `!` and `[` are special and mark
    processing instruction tags (`?`), delcaration tags (`!`) and CDATA and conditional tags
    (`[`). For these tags the second element in the list is the actual tag name, and the
    third element is the unparsed data string following the name (including whitespace).

xml-attribute-match? ((*attribute* XML-GLOBAL-ATTRIBUTE)   Method
(*name* STRING) (*namespace* STRING) (*element-name* STRING)
(*element-namespace* STRING)) *:* BOOLEAN

Returns `true` if *attribute* is a global XML attribute with the name *name* in namespace *namespace*. Note that *namespace* is the full URI, not an abbreviation. Also, *namespace* may be `null`, in which case `tag` must not have a namespace associated with it. This method ignores *element-name* and *element-namespace*.

xml-attribute-match? ((*attribute* XML-LOCAL-ATTRIBUTE)   Method
(*name* STRING) (*namespace* STRING) (*element-name* STRING)
(*element-namespace* STRING)) *:* BOOLEAN

Returns `true` if *attribute* is a local XML attribute with the name *name* inside an XML element matching *element-name* in namespace *element-namespace* (see the method `xml-element-match?`. Note that *element-namespace* is the full URI, not an abbreviation. *element-namespace* may be `null`, in which case the XML element must not have a namespace associated with it. This method ignores *namespace*

xml-element-match? ((*tag* XML-ELEMENT) (*name* STRING)   Method
(*namespace* STRING)) *:* BOOLEAN

Returns `true` if *tag* is an XML element with the name *name* in namespace *namespace*. Note that *namespace* is the full URI, not an abbreviation. Also, *namespace* may be `null`, in which case *tag* must not have a namespace associated with it.

6.17 Miscellaneous

This is a catch-all section for functions and methods that haven't been categorized yet into any of the previous sections. They are in random order and many of them will never be part of the official STELLA interface. So beware!

operating-system () *:* KEYWORD   Function

Not documented.

activate-demon ((*demon* DEMON)) *:*   Function

Install *demon* in the location(s) specified by its internal structure.

active? ((*self* POLYMORPHIC-RELATION)) *:* BOOLEAN   Method

True if *self* or a superslot of *self* is marked active.

add-hook ((*hookList* HOOK-LIST) (*hookFunction* SYMBOL)) *:*   Function

Insert the function named *hookFunction* into *hookList*.

add-trace (&rest (*keywords* GENERALIZED-SYMBOL)) *:* LIST   Command

Enable trace messages identified by any of the listed *keywords*. After calling (`add-trace <keyword>`) code guarded by (`trace-if <keyword>` ...) will be executed when it is encountered.

all-classes ((*module* MODULE) (*local?* BOOLEAN)) *:* (ITERATOR OF   Function
      CLASS)
      Iterate over all classes visible from *module*. If *local?*, return only classes interned in
      module. If *module* is null, return all classes interned everywhere.

all-contexts () *:* (ITERATOR OF CONTEXT)   Function
      Return an iterator that generates all contexts.

all-defined? (&body (*forms* CONS)) *:* OBJECT   Macro
      Evaluate each of the forms in *forms*, and return TRUE if none of them are NULL.

all-functions ((*module* MODULE) (*local?* BOOLEAN)) *:* (ITERATOR OF   Function
      FUNCTION)
      Iterate over all functions visible from *module*. If *local?*, return only functions bound to
      symbols interned in *module*. If *module* is null, return all functions defined everywhere.

all-included-modules ((*self* MODULE)) *:* (ITERATOR OF MODULE)   Function
      Generate a sequence of all modules included by *self*, inclusive, starting from the
      highest ancestor and working down to *self* (which is last).

all-methods ((*module* MODULE) (*local?* BOOLEAN)) *:* (ITERATOR OF   Function
      METHOD-SLOT)
      Iterate over all methods visible from *module*. If *local?*, return only methods interned
      in *module*. If *module* is null, return all methods interned everywhere.

all-modules () *:* (ITERATOR OF MODULE)   Function
      Return an iterator that generates all modules.

all-public-functions ((*module* MODULE) (*local?* BOOLEAN)) *:*   Function
      (ITERATOR OF FUNCTION)
      Iterate over all functions visible from *module*. If *local?*, return only functions bound to
      symbols interned in *module*. If *module* is null, return all functions defined everywhere.

all-public-methods ((*module* MODULE) (*local?* BOOLEAN)) *:*   Function
      (ITERATOR OF METHOD-SLOT)
      Iterate over all public methods visible from *module*. If *local?*, return only methods
      interned in *module*. If *module* is null, return all methods interned everywhere.

all-slots ((*module* MODULE) (*local?* BOOLEAN)) *:* (ITERATOR OF SLOT)   Function
      Iterate over all slots visible from *module*. If *local?*, return only methods interned in
      module. If *module* is null, return all methods interned everywhere.

all-subcontexts ((*context* CONTEXT) (*traversal* KEYWORD)) *:*   Function
      (ALL-PURPOSE-ITERATOR OF CONTEXT)
      Return an iterator that generates all subcontexts of `self` (not including `self`) in the
      order specified by *traversal* (one of :preorder, :inorder, or :postorder).

all-surrogates ((*module* MODULE) (*local?* BOOLEAN)) : (ITERATOR OF   Function
      SURROGATE)
    Iterate over all surrogates visible from *module*. If *local?*, return only surrogates
    interned in *module*. If *module* is null, return all surrogates interned everywhere.

all-symbols ((*module* MODULE) (*local?* BOOLEAN)) : (ITERATOR OF   Function
      SYMBOL)
    Iterate over all symbols visible from *module*. If *local?*, return only symbols interned
    in *module*. If *module* is null, return all symbols interned everywhere.

all-variables ((*module* MODULE) (*local?* BOOLEAN)) : (ITERATOR OF   Function
      GLOBAL-VARIABLE)
    Iterate over all variables visible from *module*. If *local?*, return only variables bound to
    symbols interned in *module*. If *module* is null, return all variables defined everywhere.

allocate-iterator ((*self* ABSTRACT-ITERATOR)) : (LIKE SELF)   Method
    Iterator objects return themselves when asked for an iterator (they occupy the same
    position as a collection within a `foreach` statement).

allocation ((*self* STORAGE-SLOT)) : KEYWORD   Method
    Return the most specific :allocation facet, or :instance if all inherited values are NULL.

apply ((*code* FUNCTION-CODE) (*arguments* (CONS OF OBJECT))) :   Function
      OBJECT
    Apply *code* to *arguments*, returning a value of type OBJECT.

apply-boolean-method ((*code* METHOD-CODE)   Function
      (*arguments* (CONS OF OBJECT))) : BOOLEAN
    Apply *code* to *arguments*, returning a value of type BOOLEAN.

apply-float-method ((*code* METHOD-CODE)   Function
      (*arguments* (CONS OF OBJECT))) : FLOAT
    Apply *code* to *arguments*, returning a value of type FLOAT.

apply-integer-method ((*code* METHOD-CODE)   Function
      (*arguments* (CONS OF OBJECT))) : INTEGER
    Apply *code* to *arguments*, returning a value of type INTEGER.

apply-method ((*code* METHOD-CODE) (*arguments* (CONS OF OBJECT)))   Function
      : OBJECT
    Apply *code* to *arguments*, returning a value of type OBJECT.

apply-string-method ((*code* METHOD-CODE)   Function
      (*arguments* (CONS OF OBJECT))) : STRING
    Apply *code* to *arguments*, returning a value of type STRING.

break-program ((*message* STRING)) :   Function
    Interrupt the program and print *message*. Continue after confirmation with the user.

call-clear-module (&rest (*name* NAME)) :   Command
    Destroy all objects belonging to module *name* or any of its children. If no *name* is
    supplied, the current module will be cleared after confirming with the user. Important
    modules such as STELLA are protected against accidental clearing.

cast ((*value* OBJECT) (*type* TYPE)) : OBJECT   Function
    Perform a run-time type check, and then return *value*.

cl-slot-value ((*object* OBJECT) (*slotName* STRING)   Function
        (*dontConvert?* BOOLEAN)) : LISP-CODE
    Lookup slot *slotName* on *object* and return the lispified slot value (see `lispify`).
    If *dontConvert?* is TRUE, the returned slot value will not be lispified. Generate a
    warning if no such slot exists on *object*. In a call directly from Lisp *slotName* can
    also be supplied as a Lisp symbol.

cl-slot-value-setter ((*object* OBJECT) (*slotName* STRING)   Function
        (*value* LISP-CODE) (*dontConvert?* BOOLEAN)) : LISP-CODE
    Lookup slot *slotName* on *object* and set its value to the stellafied *value* (see
    `stellafy`). If *dontConvert?* is TRUE, *value* will not be stellafied before it gets
    assigned. Generate a warning if no such slot exists on *object*, or if *value* has the
    wrong type. In a call directly from Lisp *slotName* can also be supplied as a Lisp
    symbol.

cl-translate-file ((*file* FILE-NAME) (*relative?* BOOLEAN)) :   Function
    Translate a Stella *file* to Common-Lisp. If *relative?*, concatenate root directory to
    file.

cl-translate-system ((*system-name* STRING)) :   Function
    Translate a Stella system named *system-name* to Common Lisp.

cleanup-unfinalized-classes () :   Function
    Remove all finalized classes from `*UNFINALIZED-CLASSES*`, and set `*NEWLY-UNFINALIZED-`
    `CLASSES?*` to `false`.

clear ((*self* SET)) :   Method
    Make *self* an empty set.

clear-recycle-list ((*list* RECYCLE-LIST)) :   Function
    Reset *list* to its empty state.

clear-recycle-lists () :   Function
    Reset all currently active recycle lists to their empty state.

clear-trace () :   Command
      Disable all tracing previously enabled with `add-trace`.

close-all-files () :   Function
      Close all currently open file streams. Use for emergencies or for cleanup.

close-stream ((*self* STREAM)) :   Function
      Close the stream *self*.

coerce-&rest-to-cons ((*restVariable* SYMBOL)) : OBJECT   Macro
      Coerce the argument list variable *restVariable* into a CONS list containing all its
      elements (uses argument list iteration to do so). If *restVariable* already is a CONS
      due to argument listification, this is a no-op.

coerce-to-symbol ((*name* NAME)) : GENERALIZED-SYMBOL   Function
      Return the (generalized) symbol represented by *name*. Return `null` if *name* is unde-
      fined or does not represent a string.

collect ((*collectvariable* SYMBOL) &body (*body* CONS)) : OBJECT   Macro
      Use a VRLET to collect values. Input has the form (`collect <x> in <expression>`
      `where (<test> <x>))`.

collection-valued? ((*self* SLOT)) : BOOLEAN   Method
      True if slot values are collections.

command? ((*method* METHOD-SLOT)) : BOOLEAN   Function
      Return `true` if *method* is an evaluable command.

component? ((*self* STORAGE-SLOT)) : BOOLEAN   Method
      True if fillers of this slot are components of the owner slot, and therefore should be
      deleted if the owner is deleted.

compose-namestring   Function
      ((*name-components* (CONS OF STRING-WRAPPER)) &rest (*options* OBJECT)) :
      STRING
      name-components is a cons to be processed into a namestring. `:prefix` and `:suffix`
      are strings that will NOT be case-converted. `:case` is one of :UPCASE :TitleCase
      :titleCaseX :downcase :Capitalize default is :TitleCase `:separator` is a string that
      should separate word elements. It does not separate the prefix or suffix. Default is
      `""` `:translation-table` should be a STRING-HASH-TABLE hash table that strings
      into their desired printed representation as a string. In general the argument will be
      strings, but that is not strictly necessary.

compose-namestring-full ((*strings* (CONS OF STRING-WRAPPER))   Function
      (*prefix* STRING) (*suffix* STRING) (*outputcase* KEYWORD)
      (*outputseparator* STRING) (*translationtable* STRING-HASH-TABLE)
      (*useacronymheuristics?* BOOLEAN)) : STRING
      Non-keyword version of `compose-namestring`, which will probably be easier to use
      when called from non-Lisp languages.

configure-stella ((*file* FILE-NAME)) *:*   Function
      Perform STELLA run-time configuration. If supplied, load the configuration file *file*
      first which should be supplied with a physical pathname.

consify ((*self* OBJECT)) *:* CONS   Method
      If `object` is a CONS, return it. Otherwise, return a singleton cons list containing it.

continuable-error (&body (*body* CONS)) *:* OBJECT   Macro
      Signal error message, placing non-string arguments in quotes.

copy ((*self* SET)) *:* (SET OF (LIKE (ANY-VALUE SELF)))   Method
      Return a copy of the set *self*. The conses in the copy are freshly allocated.

cpp-translate-system ((*systemName* STRING)) *:*   Function
      Translate the system *systemName* to C++.

cpptrans ((*statement* OBJECT)) *:*   Command
      Translate *statement* to C++ and print the result.

create-object ((*type* TYPE) &rest (*initial-value-pairs* OBJECT)) *:*   Function
        OBJECT
      Funcallable version of the `new` operator. Return an instance of the class named by
      type. If *initial-value-pairs* is supplied, it has to be a key/value list similar to what's
      accepted by `new` and the named slots will be initialized with the supplied values.
      Similar to `new`, all required arguments for *type* must be included. Since all the slot
      initialization, etc. is handled dynamically at run time, `create-object` is much slower
      than `new`; therefore, it should only be used if *type* cannot be known at translation
      time.

deactivate-demon ((*demon* DEMON)) *:*   Function
      Detach *demon* from the location(s) specified by its internal structure.

decompose-namestring ((*namestring* STRING)   Function
        &rest (*options* OBJECT)) *:* (CONS OF STRING-WRAPPER)
      Keyword options: :break-on-cap one of :YES :NO :CLEVER default is :CLEVER
      :break-on-number one of :YES :NO :CLEVER default is :CLEVER :break-on-
      separators string default is "-_ "
      DECOMPOSE-NAMESTRING returns a cons of STRING-WRAPPERS that are the
      decomposition of the input STRING. The arguments are used as follows: *namestring*
      is the input string. :break-on-cap is a keyword controlling whether changes in capital-
      ization is used to indicate word boundaries. If :YES, then all capitalization changes
      delineate words. If :CLEVER, then unbroken runs of capitalized letters are treated
      as acronyms and remain grouped. If :NO or NULL, there is no breaking of words
      based on capitalization. :break-on-number is a flag controlling whether encountering
      a number indicates a word boundary. If :YES, then each run of numbers is treated
      as a word separate from surrounding words. If :CLEVER, then an attempt is made

to recognize ordinal numbers (ie, 101st) and treat them as separate words. If :NO or NULL, there is no breaking of words when numbers are encountered. :break-on-separators A string of characters which constitute word delimiters in the input word. This is used to determine how to break the name into individual words. Defaults are space, - and _.

decompose-namestring-full ((*namestring* STRING)   Function
        (*break-on-cap* KEYWORD) (*break-on-number* KEYWORD)
        (*break-on-separators* STRING)) *: (CONS OF STRING-WRAPPER)*
Non-keyword version of `decompose-namestring`, which will probably be easier to use when called from non-Lisp languages.

default-form ((*self* STORAGE-SLOT)) *: OBJECT*   Method
Returns the current value of default expression when the slot has not been assigned a value.

defdemon ((*name* STRING-WRAPPER) (*parameterstree* CONS)   Macro
        &body (*optionsandbody* CONS)) *: OBJECT*
Define a demon *name* and attach it to a class or slot.

define-demon ((*name* STRING) &rest (*options* OBJECT)) *: DEMON*   Function
Define a class or slot demon. Options are :create, :destroy, :class, :slot, :guard?, :code, :method, :inherit?, and :documentation.

define-logical-host-property ((*host* STRING) (*property* KEYWORD)   Function
        (*value* OBJECT)) *:*
Define *property* with *value* for the logical host *host*. As a side-effect, this also defines *host* as a logical host (both *property* and *value* can be supplied as NULL). If :ROOT-DIRECTORY is specified, all pathnames with *host* are assumed to be relative to that directory (even if they are absolute) and will be rerooted upon translation. :ROOT-DIRECTORY can be a logical or physical pathname. If :LISP-TRANSLATIONS is specified, those will be used verbatimely as the value of (`CL:logical-pathname-translations host`) if we are running in Lisp, which allows us to depend on the native `CL:translate-logical-pathname` for more complex translation operations.

define-module ((*name* STRING) (*options* CONS)) *: MODULE*   Function
Define or redefine a module named *name* having the options *options*. Return the new module.

define-stella-class ((*name* TYPE) (*supers* (LIST OF TYPE))   Function
        (*slots* (LIST OF SLOT)) (*options* KEYWORD-KEY-VALUE-LIST)) *: CLASS*
Return a Stella class with name *name*. Caution: If the class already exists, the Stella class object gets redefined, but the native C++ class is not redefined.

define-stella-method-slot ((*inputname* SYMBOL) (*returntypes* CONS)   Function
    (*function?* BOOLEAN) (*inputParameters* CONS)
    (*options* KEYWORD-KEY-VALUE-LIST)) : METHOD-SLOT
    Define a new Stella method object (a slot), and attach it to the class identified by
    the first parameter in *inputParameters*.

defined? ((*x* NATIVE-VECTOR)) : BOOLEAN   Method
    Return true if *x* is defined (handled specially by all translators).

defmain ((*varList* CONS) &body (*body* CONS)) : OBJECT   Macro
    Defines a function called MAIN which will have the appropriate signature for the tar-
    get translation language. The signature will be: C++: public static int main (int v1,
    char** v2) {<body>} Java: public static void main (String [] v2) {<body>} Lisp: (de-
    fun main (&rest args) <body>) The argument *varList* must have two symbols, which
    will be the names for the INTEGER argument count and an array of STRINGs
    with the argument values. It can also be empty to indicate that no command line
    arguments will be handled. The startup function for the containing system will auto-
    matically be called before *body* is executed unless the option :STARTUP-SYSTEM?
    was supplied as FALSE. There can only be one DEFMAIN per module.

defsystem ((*name* SYMBOL) &rest (*options* OBJECT)) :   Command
    SYSTEM-DEFINITION
    Define a system of files that collectively define a Stella application. Required options
    are: :directory – the path from the Stella root directory to the directory containing the
    system files. Can be a string or a list of strings (do not include directory separators).
    :files – a list of files in the system, containing strings and lists of strings; the latter
    defines exploded paths to files in subdirectories. Optional options are: :required-
    systems – a list of systems (strings) that should be loaded prior to loading this system.
    :cardinal-module – the name (a string) of the principal module for this system.

deleted? ((*self* OBJECT)) : BOOLEAN   Method
    Default `deleted?` method which always returns FALSE. Objects that inherit DYNAMIC-
    SLOTS-MIXIN also inherit the dynamically-allocated slot `deleted-object?` which
    is read/writable with specializations of this method.

describe ((*name* OBJECT) &rest (*mode* OBJECT)) :   Command
    Print a description of an object in :verbose, :terse, or :source modes.

describe-object ((*self* OBJECT) (*stream* OUTPUT-STREAM)   Method
    (*mode* KEYWORD)) :
    Prints a description of *self* to stream *stream*. *mode* can be :terse, :verbose, or :source.
    The :terse mode is often equivalent to the standard print function.

destroy-class ((*self* CLASS)) :   Method
    Destroy the Stella class *self*. Unfinalize its subclasses (if it has any).

destroy-class-and-subclasses ((*self* CLASS)) *:*   Function
    Destroy the Stella class *self* and all its subclasses.

destructure-defmethod-tree ((*method-tree* CONS)   Function
        (*options-table* KEY-VALUE-LIST)) *:* OBJECT CONS CONS
    Return three parse trees representing the name, parameters, and code body of the
    parse tree *method-tree*. Fill *options-table* with a dictionary of method options. Stor-
    age note: Options are treated specially because the other return values are subtrees
    of *method-tree*, while *options-table* is a newly-created cons tree. Note also, the pa-
    rameter and body trees are destructively removed from *method-tree*.

dictionary ((*collectiontype* TYPE)   Function
        &rest (*alternatingkeysandvalues* OBJECT)) *:* (ABSTRACT-DICTIONARY OF
        OBJECT OBJECT)
    Return a dictionary containing `values`, in order.

direct-super-classes ((*self* CLASS)) *:* (ITERATOR OF CLASS)   Method
    Returns an iterator that generates all direct super classes of *self*.

disable-memoization () *:*   Command
    Enable memoization and use of memoized expression results.

disabled-stella-feature? ((*feature* KEYWORD)) *:* BOOLEAN   Function
    Return true if the STELLA *feature* is currently disabled.

drop-hook ((*hookList* HOOK-LIST) (*hookFunction* SYMBOL)) *:*   Function
    Remove the function named *hookFunction* from *hookList*.

drop-trace (&rest (*keywords* GENERALIZED-SYMBOL)) *:* LIST   Command
    Disable trace messages identified by any of the listed *keywords*. After calling (`drop-`
    `trace <keyword>`) code guarded by (`trace-if <keyword> ...`) will not be executed
    when it is encountered.

either ((*value1* OBJECT) (*value2* OBJECT)) *:* OBJECT   Macro
    If *value1* is defined, return that, else return *value2*.

empty? ((*self* SET)) *:* BOOLEAN   Method
    Return `true` if the set *self* has no members.

enable-memoization () *:*   Command
    Enable memoization and use of memoized expression results.

enabled-stella-feature? ((*feature* KEYWORD)) *:* BOOLEAN   Function
    Return true if the STELLA *feature* is currently enabled.

error (&body (*body* CONS)) : OBJECT   Macro
>   Signal error message, placing non-string arguments in quotes.

evaluate ((*expression* OBJECT)) : OBJECT   Function
>   Evaluate the expression *expression* and return the result. Currently, only the evalu-
>   ation of (possibly nested) commands and global variables is supported. The second
>   return value indicates the actual type of the result (which might have been wrapped),
>   and the third return value indicates whether an error occurred during the evaluation.
>   Expressions are simple to program in Common Lisp, since they are built into the
>   language, and relatively awkward in Java and C++. Users of either of those languages
>   are more likely to want to call `evaluate-string`.

evaluate-string ((*expression* STRING)) : OBJECT   Function
>   Evaluate the expression represented by *expression* and return the result. This is
>   equivalent to (`evaluate (unstringify expression)`).

exception-message ((*e* NATIVE-EXCEPTION)) : STRING   Function
>   Accesses the error message of the exception *e*.

extension ((*self* CLASS)) : CLASS-EXTENSION   Method
>   Return the nearest class extension that records instances of the class *self*.

finalize-classes () :   Function
>   Finalize all currently unfinalized classes.

finalize-classes-and-slots () :   Function
>   Finalize all currently unfinalized classes and slots.

finalize-slots () :   Function
>   Finalize all currently unfinalized slots.

first-defined (&body (*forms* CONS)) : OBJECT   Macro
>   Return the result of the first form in *forms* whose value is defined or NULL otherwise.

flush-output ((*self* OUTPUT-STREAM)) :   Function
>   Flush all buffered output of *self*.

format-with-padding ((*input* STRING) (*length* INTEGER)   Function
>   (*padchar* CHARACTER) (*align* KEYWORD) (*truncate?* BOOLEAN)) : STRING
>   Formats *input* to be (at least) *length* long, using *padchar* to fill if necessary. *align*
>   must be one of :LEFT, :RIGHT, :CENTER and will control how *input* will be justified
>   in the resulting string. If *truncate?* is true, then then an overlength string will be
>   truncated, using the opposite of *align* to pick the truncation direction.

free ((*self* ACTIVE-OBJECT)) :   Method
>   Remove all pointers between *self* and other objects, and then deallocate the storage
>   for self.

free ((*self* OBJECT)) :   Method
      Default method. Deallocate storage for *self*.

free-hash-table-values ((*self* ABSTRACT-HASH-TABLE)) :   Method
      Call free on each value in the hash table *self*.

get-calendar-date ((*date* CALENDAR-DATE) (*timezone* FLOAT)) :   Method
        INTEGER INTEGER INTEGER KEYWORD
      Returns multiple values of year, month, day and day of week for *date* in *timezone*.
      timezone is the number of hours added to UTC to get local time. It is in the range
      -12.0 to +14.0 where UTC is zone 0.0

get-global-value ((*self* SURROGATE)) : OBJECT   Function
      Return the (possibly-wrapped) value of the global variable for the surrogate *self*.

get-quoted-tree ((*tree-name* STRING) (*modulename* STRING)) : CONS   Function
      Return the quoted tree with name *tree-name*.

get-slot ((*self* STANDARD-OBJECT) (*slot-name* SYMBOL)) : SLOT   Function
      Return the slot named *slot-name* on the class representing the type of *self*.

get-stella-class ((*class-name* TYPE) (*error?* BOOLEAN)) : CLASS   Method
      Return a class with name *class-name*. If none exists, break if *error?*, else return `null`.

get-stella-class ((*class-name* SYMBOL) (*error?* BOOLEAN)) : CLASS   Method
      Return a class with name *class-name*. If non exists, break if *error?*, else return `null`.

get-stella-class ((*class-name* STRING) (*error?* BOOLEAN)) : CLASS   Method
      Return a class with name *class-name*. If none exists, break if *error?*, else return `null`.

get-time ((*date* CALENDAR-DATE) (*timezone* FLOAT)) : INTEGER INTEGER Method
        INTEGER INTEGER
      Returns multiple values of hours, minutes, seconds, milliseconds for the calendar date
      date in *timezone*. *timezone* is the number of hours added to UTC to get local time.
      It is in the range -12.0 to +14.0 where UTC is zone 0.0

global-variable-type-spec ((*global* GLOBAL-VARIABLE)) : TYPE-SPEC   Function
      Return the type spec for the global variable *global*.

hash-code ((*self* OBJECT)) : INTEGER   Method
      Return a hash code for *self*. Two objects that are `eql?` are guaranteed to generate the
      same hash code. Two objects that are not `eql?` do not necessarily generate different
      hash codes.

hash-string ((*string* STRING) (*seedCode* INTEGER)) *:* INTEGER   Function
    Generate a hash-code for *string* and return it. Two strings that are equal but not
    eq will generate the same code. The hash-code is based on *seedCode* which usually
    will be 0. However, *seedCode* can also be used to supply the result of a previous
    hash operation to achieve hashing on sequences of strings without actually having to
    concatenate them.

home-module ((*self* OBJECT)) *:* MODULE   Method
    Return the home module of *self*.

if-output-language ((*language* KEYWORD) (*thenForm* OBJECT)   Macro
       (*elseForm* OBJECT)) *:* OBJECT
    Expand to *thenForm* if the current translator output language equals *language*. Oth-
    erwise, expand to *elseForm*. This can be used to conditionally translate Stella code.

if-stella-feature ((*feature* KEYWORD) (*thenForm* OBJECT)   Macro
       (*elseForm* OBJECT)) *:* OBJECT
    Expand to *thenForm* if *feature* is a currently enabled STELLA environment feature.
    Otherwise, expand to *elseForm*. This can be used to conditionally translate Stella
    code.

ignore (&body (*variables* CONS)) *:* OBJECT   Macro
    Ignore unused *variables* with NoOp `setq` statements.

incrementally-translate ((*tree* OBJECT)) *:* OBJECT   Function
    Translate a single Stella expression *tree* and return the result. For C++ and Java print
    the translation to standard output and return NIL instead.

inform (&body (*body* CONS)) *:* OBJECT   Macro
    Print informative message, placing non-string arguments in quotes, and terminating
    with a newline.

initial-value ((*self* CLASS)) *:* OBJECT   Method
    Return an initial value for the class *self*.

initial-value ((*self* STORAGE-SLOT)) *:* OBJECT   Method
    Return an initial value for *self*, or `null`. The initial value can be defined by the slot
    itself, inherited from an equivalent slot, or inherit from the :initial-value option for
    the class representing the type of *self*.

initially ((*self* STORAGE-SLOT)) *:* OBJECT   Method
    Defines the value of a slot before it has been assigned a value.

insert ((*self* SET) (*value* (LIKE (ANY-VALUE SELF)))) *:*   Method
    Add *value* to the set *self*. First checks for duplicate.

interpret-command-line-arguments ((*count* INTEGER)   Function
      (*arguments* (ARRAY () OF STRING))) :
    Interpret any STELLA-relevant command line *arguments*.

isa? ((*object* OBJECT) (*type* TYPE)) : BOOLEAN   Function
    Return `true` iff *object* is an instance of the class named *type*.

java-translate-system ((*systemName* STRING)) :   Function
    Translate the system *systemName* to Java.

jptrans ((*statement* OBJECT)) :   Command
    Translate *statement* to C++ and print the result.

lispify ((*thing* UNKNOWN)) : LISP-CODE   Function
    Convert a Stella *thing* as much as possible into a Common-Lisp analogue. The cur-
    rently supported *thing* types are CONS, LIST, KEY-VALUE-LIST, ITERATOR,
    SYMBOL, KEYWORD, and all wrapped and unwrapped literal types. BOOLEANs
    are translated into Lisp's CL:T and CL:NIL logic. Unsupported types are left un-
    changed.

lispify-boolean ((*thing* UNKNOWN)) : LISP-CODE   Function
    Lispify *thing* which is assumed to be a (possibly wrapped) Stella boolean.

listify ((*self* CONS)) : (LIST OF (LIKE (ANY-VALUE SELF)))   Method
    Return a list of elements in *self*.

listify ((*self* LIST)) : (LIST OF (LIKE (ANY-VALUE SELF)))   Method
    Return *self*.

listify ((*self* KEY-VALUE-LIST)) : (LIST OF (LIKE (ANY-VALUE SELF)))   Method
    Return a list of key-value pairs in *self*.

listify ((*self* VECTOR)) : (LIST OF (LIKE (ANY-VALUE SELF)))   Method
    Return a list of elements in *self*.

listify ((*self* ITERATOR)) : (LIST OF (LIKE (ANY-VALUE SELF)))   Method
    Return a list of elements generated by *self*.

load-configuration-file ((*file* FILE-NAME)) : CONFIGURATION-TABLE   Function
    Read a configuration *file* and return its content as a configuration table. Also enter
    each property read into the global system configuration table. Assumes Java-style
    property file syntax. Each property name is represented as a wrapped string and
    each value as a wrapped string/integer/float or boolean.

load-file ((*file* STRING)) : Command
    Read STELLA commands from *file* and evaluate them. The file should begin with
an `in-module` declaration that specifies the module within which all remaining com-
mands are to be evaluated The remaining commands are evaluated one-by-one, ap-
plying the function `evaluate` to each of them.

load-system ((*systemName* STRING) (*language* KEYWORD) Function
    &rest (*options* OBJECT)) : BOOLEAN
    Natively *language*-compile out-of-date translated files of system *systemName* and then
load them into the running system (this is only supported/possible for Lisp at the
moment). Return true if at least one file was compiled. The following keyword/value
options are recognized:

    `:force-recompilation?` (default false): if true, files will be compiled whether or not
their compilations are up-to-date.

    `:startup?` (default true): if true, the system startup function will be called once all
files have been loaded.

lookup-class ((*name* SYMBOL)) : CLASS Method
    Return a class with name *name*. Scan all visible surrogates looking for one that has
a class defined for it.

lookup-class ((*name* STRING)) : CLASS Method
    Return a class with name *name*. Scan all visible surrogates looking for one that has
a class defined for it.

lookup-command ((*name* SYMBOL)) : METHOD-SLOT Function
    If *name* names an evaluable command return its associated command object; other-
wise, return `null`. Currently, commands are not polymorphic, i.e., they can only be
implemented by functions.

lookup-configuration-property ((*property* STRING) Function
    (*defaultValue* WRAPPER) (*configuration* CONFIGURATION-TABLE)) : WRAPPER
    Lookup *property* in *configuration* and return its value. Use the global system configu-
ration table if *configuration* is NULL. Return *defaultValue* if *property* is not defined.

lookup-demon ((*name* STRING)) : DEMON Function
    Return the demon named *name*.

lookup-function ((*functionSymbol* SYMBOL)) : FUNCTION Function
    Return the function defined for *functionSymbol*, if it exists.

lookup-function-by-name ((*name* STRING)) : FUNCTION Function
    Return a function with name *name* visible from the current module. Scan all visible
symbols looking for one that has a function defined for it.

lookup-global-variable ((*self* SURROGATE)) : GLOBAL-VARIABLE   Method
    Return a global variable with name *self*.

lookup-global-variable ((*self* GENERALIZED-SYMBOL)) :   Method
    GLOBAL-VARIABLE
    Return a global variable with name *self*.

lookup-global-variable ((*self* STRING)) : GLOBAL-VARIABLE   Method
    Return a global variable with name *self*.

lookup-local-slot ((*class* CLASS) (*slot-name* SYMBOL)) : SLOT   Function
    Lookup a local slot with *slot-name* on *class*.

lookup-macro ((*name* SYMBOL)) : METHOD-SLOT   Function
    If *name* has a macro definition, return the method object holding its expander function.

lookup-slot ((*class* CLASS) (*slot-name* SYMBOL)) : SLOT   Function
    Return a slot owned by the class *class* with name *slot-name*. Multiply inherited slots are disambiguated by a left-to-right class precedence order for classes with multiple parents (similar to CLOS).

lptrans ((*statement* OBJECT)) :   Command
    Translate *statement* to Common-Lisp and print the result.

make-matching-name ((*original* STRING) &rest (*options* OBJECT)) :   Function
    STRING
    Keyword options: :break-on-cap one of :YES :NO :CLEVER default is :CLEVER :break-on-number one of :YES :NO :CLEVER default is :CLEVER :break-on-separators string default is "-_ " :remove-prefix string :remove-suffix string

    :case one of :UPCASE :TitleCase :titleCaseX :downcase :Capitalize :preserve default is :TitleCase :separator string default is "" :add-prefix string :add-suffix string

    MAKE-MATCHING-NAME returns a matching name (a string) for the input name (a string). A matching name is constructed by breaking the input into `words` and then applying appropriate transforms. The arguments are used as follows: *original* is the input name. It is a string. :break-on-cap is a keyword controlling whether changes in capitalization is used to indicate word boundaries. If :YES, then all capitalization changes delineate words. If :CLEVER, then unbroken runs of capitalized letters are treated as acronyms and remain grouped. If :NO or NULL, there is no breaking of words based on capitalization. :break-on-number is a flag controlling whether encountering a number indicates a word boundary. If :YES, then each run of numbers is treated as a word separate from surrounding words. If :CLEVER, then an attempt is made to recognize ordinal numbers (ie, 101st) and treat them as separate words. If :NO or NULL, there is no breaking of words when numbers are encountered. :break-on-separators A string of characters which constitute word delimiters in the input word. This is used to determine how to break the name into individual words.

Defaults are space, - and _. :remove-prefix Specifies a prefix or suffix that is stripped from the input :remove-suffix name before any other processing. This allows the removal of any naming convention dictated prefixes or suffixes. :add-prefix Specifies a prefix or suffix that is added to the output name :add-suffix after all other processing. This allows the addition of any naming convention dictated prefixes or suffixes. :case The case of the resulting name. This is applied to the name before adding prefixes or suffixes. The two title case options differ only in how the first word of the name is treated. :TitleCase capitalizes the first letter of the first word and also the first letter of all other words. :TitleCaseX does not capitalizes the first letter of the first word but capitalizes the first letter of all subsequent words. :preserve results in no change in case. :separator This is a string specifying the word separator to use in the returned name. An empty string (the default) means that the resulting words are concatenated without any separation. This normally only makes sense when using one of the title case values for the case keyword.

make-matching-name-full ((*originalname* STRING)   Function
       (*breakoncap* KEYWORD) (*breakonnumber* KEYWORD)
       (*breakonseparators* STRING) (*removeprefix* STRING) (*removesuffix* STRING)
       (*addprefix* STRING) (*addsuffix* STRING) (*outputcase* KEYWORD)
       (*outputseparator* STRING)) : STRING
Non-keyword version of `make-matching-name`, which will probably be easier to use when called from non-Lisp languages.

make-system ((*systemName* STRING) (*language* KEYWORD)   Command
       &rest (*options* OBJECT)) : BOOLEAN
Translate all out-of-date files of system *systemName* into *language* and then compile and load them (the latter is only possible for Lisp right now). The following keyword/value *options* are recognized:

`:two-pass?`: if true, all files will be scanned twice, once to load the signatures of objects defined in them, and once to actually translate the definitions. Otherwise, the translator will make one pass in the case that the system is already loaded (and is being remade), and two passes otherwise.

`:development-settings?` (default false): if true translation will favor safe, readable and debuggable code over efficiency (according to the value of `:development-settings` on the system definition). If false, efficiency will be favored instead (according to the value of `:production-settings` on the system definition).

`:production-settings?` (default true): inverse to `:development-settings?`.

`:force-translation?` (default false): if true, files will be translated whether or not their translations are up-to-date.

`:force-recompilation?` (default false): if true, translated files will be recompiled whether or not their compilations are up-to-date (only supported in Lisp right now).

`:load-system?` (default true): if true, compiled files will be loaded into the current STELLA image (only supported in Lisp right now).

`:startup?` (default true): if true, the system startup function will be called once all files have been loaded.

member? ((*self* SET) (*object* (LIKE (ANY-VALUE SELF))))) *:* BOOLEAN   Method
    Return `true` iff *object* is a member of the set *self*. Uses an `eql?` test.

member? ((*self* COLLECTION) (*object* OBJECT)) *:* BOOLEAN   Method
    Return true iff *object* is a member of the collection *self*.

member? ((*self* SEQUENCE) (*value* OBJECT)) *:* BOOLEAN   Method
    Return TRUE if *value* is a member of the sequence *self*.

memoize ((*inputArgs* CONS) &body (*body* CONS)) *:* OBJECT   Macro
    Compute the value of an expression and memoize it relative to the values of *inputArgs*.
inputArgs should characterize the complete set of values upon which the computation
of the result depended. Calls to `memoize` should be of the form

    (memoize (<arg>+) {:<option> <value>}* <expression>)

    and have the status of an expression. The following options are supported:

    :timestamps A single or list of keywords specifying the names of timestamps which
when bumped should invalidate all entries currently memoized in this table. :name
Names the memoization table so it can be shared by other memoization sites. By
default, a gensymed name is used. CAUTION: IT IS ASSUMED THAT ALL EN-
TRIES IN A MEMOZATION TABLE DEPEND ON THE SAME NUMBER OF
ARGUMENTS!! :max-values The maximum number of values to be memoized. Only
the `:max-values` most recently used values will be kept in the memoization table,
older values will be discarded and recomputed if needed. Without a `:max-values`
specification, the memoization table will grow indefinitely.

    PERFORMANCE NOTES: For most efficient lookup, input arguments that vary the
most should be listed first. Also, arguments of type STANDARD-OBJECT (and all
its subtypes) can be memoized more efficiently than arguments of type OBJECT or
(wrapped) literals (with the exception of BOOLEANs).

multiple-parents? ((*class* CLASS)) *:* BOOLEAN   Method
    Return `true` if *class* has more than one direct superclass.

multiple-parents? ((*module* MODULE)) *:* BOOLEAN   Method
    Return TRUE if *module* has more than one parent.

name-to-string ((*name* OBJECT)) *:* STRING   Function
    Return the string represented by *name*. Return `null` if *name* is undefined or does
not represent a string.

no-duplicates? ((*self* COLLECTION)) *:* BOOLEAN   Method
    Return `true` if the collection *self* forbids duplicate values.

non-empty? ((*self* SET)) *:* BOOLEAN   Method
    Return `true` if the set *self* has at least one member.

nth ((*self* NATIVE-VECTOR) (*position* INTEGER)) : (LIKE (ANY-VALUE   Method
        SELF))
    Return the element in *self* at *position*.

object-hash-code ((*self* OBJECT)) : INTEGER   Function
    Return a hash code for *self*. Two objects that are `eq?` are guaranteed to generate the
same hash code. Two objects that are not `eq?` do not necessarily generate different
hash codes. Similar to `hash-code` but always hashes on the address of *self* even if it
is a wrapper.

only-if ((*test* OBJECT) (*expression* OBJECT)) : OBJECT   Macro
    If *test* is TRUE, return the result of evaluating *expression*.

open-network-stream ((*host* STRING) (*port* INTEGER)) :   Function
        INPUT-STREAM OUTPUT-STREAM
    Open a TCP/IP network stream to *host* at *port* and return the result as an in-
put/output stream pair.

ordered? ((*self* COLLECTION)) : BOOLEAN   Method
    Return `true` if the collection *self* is ordered.

parameters ((*self* CLASS)) : (LIST OF SYMBOL)   Method
    Returns the list of parameters names of *self*.

pick-hash-table-size-prime ((*minSize* INTEGER)) : INTEGER   Function
    Return a hash table prime of at least *minSize*.

primary-type ((*self* OBJECT)) : TYPE   Method
    Returns the primary type of *self*. Gets defined automatically for every non-abstract
subclass of OBJECT.

primitive? ((*self* RELATION)) : BOOLEAN   Method
    Return `true` if *self* is not a defined relation.

print (&body (*body* CONS)) : OBJECT   Macro
    Print arguments to the standard output stream.

print-exception-context ((*e* NATIVE-EXCEPTION)   Function
        (*stream* OUTPUT-STREAM)) :
    Prints a system dependent information about the context of the specified exception.
For example, in Java it prints a stack trace. In Lisp, it is vendor dependent.

print-recycle-lists () :   Function
    Print the current state of all recycle lists.

print-stella-features () :   Command
>   Print the list of enabled and disabled STELLA features.

print-unbound-surrogates (&rest (*args* OBJECT)) :   Command
>   Print all unbound surrogates visible from the module named by the first argument (a symbol or string). Look at all modules if no module name or `null` was supplied. If the second argument is `true`, only consider surrogates interned in the specified module.

print-undefined-methods ((*module* MODULE) (*local?* BOOLEAN)) :   Function
>   Print all declared but not yet defined functions and methods in *module*. If *local?* is true, do not consider any parent modules of *module*. If *module* is NULL, look at all modules in the system. This is handy to pinpoint forward declarations that haven't been followed up by actual definitions.

print-undefined-super-classes ((*class* NAME)) :   Command
>   Print all undefined or bad (indirect) super classes of *class*.

private-class-methods ((*class* CLASS)) : (ITERATOR OF   Function
>   METHOD-SLOT)
>   Iterate over all private methods attached to *class*.

private-class-storage-slots ((*class* CLASS)) : (ITERATOR OF   Function
>   STORAGE-SLOT)
>   Iterate over all private storage-slots attached to *class*.

private? ((*self* RELATION)) : BOOLEAN   Method
>   Return `true` if *self* is not public.

ptrans ((*statement* OBJECT)) :   Command
>   Translate *statement* to Common-Lisp and print the result.

public-class-methods ((*class* CLASS)) : (ITERATOR OF METHOD-SLOT)   Function
>   Iterate over all private methods attached to *class*.

public-class-storage-slots ((*class* CLASS)) : (ITERATOR OF   Function
>   STORAGE-SLOT)
>   Iterate over all public storage-slots attached to *class*.

public-slots ((*self* CLASS)) : (ITERATOR OF SLOT)   Method
>   Return an iterator over public slots of *self*.

public-slots ((*self* OBJECT)) : (ITERATOR OF SLOT)   Method
>   Return an iterator over public slots of *self*.

public? ((*self* SLOT)) : BOOLEAN   Method
>   True if *self* or one it its ancestors is marked public.

pushf ((*place* CONS) (*value* OBJECT)) : OBJECT   Macro
      Push *value* onto the cons list *place*.

reader ((*self* STORAGE-SLOT)) : SYMBOL   Method
      Name of a method called to read the value of the slot *self*.

remove ((*self* SET) (*value* (LIKE (ANY-VALUE SELF)))) : SET   Method
      Remove all entries in *self* that match *value*.

remove-duplicates ((*self* COLLECTION)) : (LIKE SELF)   Method
      Return *self* with duplicates removed. Preserves the original order of the remaining
      members.

required-slots ((*self* CLASS)) : (LIST OF SYMBOL)   Method
      Returns a list of names of required slots for *self*.

required? ((*self* STORAGE-SLOT)) : BOOLEAN   Method
      True if a value must be assigned to this slot at creation time.

reset-stella-features () :   Command
      Reset STELLA features to their default settings.

reverse-interval ((*lowerbound* INTEGER) (*upperbound* INTEGER)) :   Function
        REVERSE-INTEGER-INTERVAL-ITERATOR
      Create a reverse interval object.

run-hooks ((*hooklist* HOOK-LIST) (*argument* OBJECT)) :   Function
      Run all hook functions in *hooklist*, applying each one to *argument*.

running-as-lisp? () : BOOLEAN   Function
      Return true if the executable code is a Common Lisp application.

safe-lookup-slot ((*class* CLASS) (*slot-name* SYMBOL)) : SLOT   Function
      Alias for `lookup-slot`. Kept for backwards compatibility.

safety ((*level* INTEGER-WRAPPER) (*test* OBJECT) &body (*body* CONS)) :   Macro
        OBJECT
      Signal warning message, placing non-string arguments in quotes.

search-for-object ((*self* OBJECT) (*typeref* OBJECT)) : OBJECT   Function
      If *self* is a string or a symbol, search for an object named *self* of type `type`. Otherwise,
      if *self* is an object, return it.

seed-random-number-generator () :   Function
      Seeds the random number generator with the current time.

sequence ((*collectiontype* TYPE) &rest (*values* OBJECT)) : (SEQUENCE   Function
        OF OBJECT)
        Return a sequence containing *values*, in order.

set-call-log-break-point ((*count* INTEGER)) :   Command
        Set a call log break point to *count*. Execution will be interrupted right at the entry
        of the *count*th logged function call.

set-configuration-property ((*property* STRING) (*value* WRAPPER)   Function
        (*configuration* CONFIGURATION-TABLE)) : WRAPPER
        Set *property* in *configuration* to *value* and return it. Use the global system configu-
        ration table if *configuration* is NULL.

set-global-value ((*self* SURROGATE) (*value* OBJECT)) : OBJECT   Function
        Set the value of the global variable for the surrogate *self* to *value*.

set-optimization-levels ((*safety* INTEGER) (*debug* INTEGER)   Function
        (*speed* INTEGER) (*space* INTEGER)) :
        Set optimization levels for the qualities *safety*, *debug*, *speed*, and *space*.

set-stella-feature (&rest (*features* KEYWORD)) :   Command
        Enable all listed STELLA *features*.

set-translator-output-language ((*new-language* KEYWORD)) :   Command
        KEYWORD
        Set output language to *new-language*. Return previous language.

setq? ((*variable* SYMBOL) (*expression* CONS)) : OBJECT   Macro
        Assign *variable* the result of evaluating *expression*, and return TRUE if *expression* is
        not NULL else return FALSE.

shadowed-symbol? ((*symbol* GENERALIZED-SYMBOL)) : BOOLEAN   Function
        Return `true` if *symbol* is shadowed in its home module.

signal ((*type* SYMBOL) &body (*body* CONS)) : OBJECT   Macro
        Signal error message, placing non-string arguments in quotes.

signal-read-error (&body (*body* CONS)) : OBJECT   Macro
        Specialized version of `signal` that throws a READ-EXCEPTION.

start-function-call-logging ((*fileName* STRING)) :   Command
        Start function call logging to *fileName*.

stella-collection? ((*self* OBJECT)) : BOOLEAN   Function
        Return `true` if *self* is a native collection.

stella-object? ((*self* OBJECT)) : BOOLEAN   Function
> Return true if *self* is a member of the STELLA class `OBJECT`.

stella-version-string () : STRING   Function
> Return a string identifying the current version of STELLA.

stellafy ((*thing* LISP-CODE) (*targetType* TYPE)) : OBJECT   Function
> Partial inverse to `lispify`. Convert the Lisp object *thing* into a Stella analogue of
> type *targetType*. Note: See also `stellify`. it is similar, but guesses *targetType* on
> its own, and makes somewhat different translations.

stellify ((*self* OBJECT)) : OBJECT   Function
> Convert a Lisp object into a STELLA object.

stop-function-call-logging () :   Command
> Stop function call logging and close the current log file.

subclass-of? ((*subClass* CLASS) (*superClass* CLASS)) : BOOLEAN   Function
> Return `true` if *subClass* is a subclass of *superClass*.

subtype-of? ((*sub-type* TYPE) (*super-type* TYPE)) : BOOLEAN   Function
> Return `true` iff the class named *sub-type* is a subclass of the class named *super-type*.

super-classes ((*self* CLASS)) : (ITERATOR OF CLASS)   Method
> Returns an iterator that generates all super classes of *self*. Non-reflexive.

sweep ((*self* OBJECT)) :   Method
> Default method. Sweep up all *self*-type objects.

system-default-value ((*self* STORAGE-SLOT)) : OBJECT   Method
> Return a default value expression, or if *self* has dynamic storage, an initial value
> expression.

system-default-value ((*self* SLOT)) : OBJECT   Method
> Return a default value expression, or if *self* has dynamic storage, an initial value
> expression.

system-loaded? ((*name* STRING)) : BOOLEAN   Function
> Return `true` if system *name* has been loaded.

terminate-program () :   Function
> Terminate and exit the program with normal exit code.

toggle-output-language () : KEYWORD   Function
> Switch between Common Lisp and C++ as output languages.

translate-system ((*systemName* STRING) (*outputLanguage* KEYWORD)   Function
&rest (*options* OBJECT)) : BOOLEAN
Translate all of the STELLA source files in system *systemName* into *outputLanguage*.
The following keyword/value *options* are recognized:

`:two-pass?` (default false): if true, all files will be scanned twice, once to load the
signatures of objects defined in them, and once to actually translate the definitions.

`:force-translation?` (default false): if true, files will be translated whether or not
their translations are up-to-date.

`:development-settings?` (default false): if true translation will favor safe, read-
able and debuggable code over efficiency (according to the value of `:development-`
`settings` on the system definition). If false, efficiency will be favored instead (ac-
cording to the value of `:production-settings` on the system definition).

`:production-settings?` (default true): inverse to `:development-settings?`.

translate-to-common-lisp? () : BOOLEAN   Function
Return `true` if current output language is Common-Lisp.

translate-to-cpp? () : BOOLEAN   Function
Return `true` if current output language is C++

translate-to-java? () : BOOLEAN   Function
Return `true` if current output language is Java

try-to-evaluate ((*tree* OBJECT)) : OBJECT   Function
Variant of `evaluate` that only evaluates *tree* if it represents an evaluable expression.
If it does not, *tree* is returned unmodified. This can be used to implement commands
with mixed argument evaluation strategies.

two-argument-least-common-superclass ((*class1* CLASS)   Function
(*class2* CLASS)) : CLASS
Return the most specific class that is a superclass of both *class1* and *class2*. If there
is more than one, arbitrarily pick one. If there is none, return `null`.

two-argument-least-common-supertype ((*type1* TYPE-SPEC)   Function
(*type2* TYPE-SPEC)) : TYPE-SPEC
Return the most specific type that is a supertype of both *type1* and *type2*. If there
is more than one, arbitrarily pick one. If there is none, return @VOID. If one or both
types are parametric, also try to generalize parameter types if necessary.

type ((*self* SLOT)) : TYPE   Method
The type of a storage slot is its base type.

type-specifier ((*self* SLOT)) : TYPE-SPEC   Method
If *self* has a complex type return its type specifier, otherwise, return `type` of *self*.

type-to-symbol ((*type* TYPE)) *:* SYMBOL   Function
  Convert *type* into a symbol with the same name and module.

type-to-wrapped-type ((*self* TYPE)) *:* TYPE   Method
  Return the wrapped type for the type *self*, or *self* if it is not a bare literal type.

unbound-surrogates ((*module* MODULE) (*local?* BOOLEAN)) *:*   Function
  (ITERATOR OF SURROGATE)
  Iterate over all unbound surrogates visible from *module*. Look at all modules if
  module is `null`. If *local?*, only consider surrogates interned in *module*.

unescape-html-string ((*input* STRING)) *:* STRING   Function
  Replaces HTML escape sequences such as & with their associated characters.

unset-stella-feature (&rest (*features* KEYWORD)) *:*   Command
  Disable all listed STELLA *features*.

unstringify-stella-source ((*source* STRING) (*module* MODULE)) *:*   Function
  OBJECT
  Unstringify a STELLA *source* string relative to *module*, or `*MODULE*` if no module is
  specified. This function allocates transient objects as opposed to `unstringify-in-`
  `module` or the regular `unstringify`.

unwrap-boolean ((*wrapper* BOOLEAN-WRAPPER)) *:* BOOLEAN   Function
  Unwrap *wrapper* and return its values as a regular BOOLEAN. Map NULL onto
  FALSE.

unwrap-function-code ((*wrapper* FUNCTION-CODE-WRAPPER)) *:*   Function
  FUNCTION-CODE
  Unwrap *wrapper* and return the result. Return NULL if *wrapper* is NULL.

unwrap-method-code ((*wrapper* METHOD-CODE-WRAPPER)) *:*   Function
  METHOD-CODE
  Unwrap *wrapper* and return the result. Return NULL if *wrapper* is NULL.

warn (&body (*body* CONS)) *:* OBJECT   Macro
  Signal warning message, placing non-string arguments in quotes.

with-permanent-objects (&body (*body* CONS)) *:* OBJECT   Macro
  Allocate `permanent` (as opposed to `transient`) objects within the scope of this dec-
  laration.

with-system-definition ((*systemnameexpression* OBJECT)   Macro
  &body (*body* CONS)) *:* OBJECT
  Set *currentSystemDefinition* to the system definition named `system`. Set *cur-
  rentSystemDefinitionSubdirectory* to match. Execute *body* within that scope.

with-transient-objects (&body (*body* CONS)) : OBJECT   Macro
    Allocate `transient` (as opposed to `permanent`) objects within the scope of this dec-
    laration. CAUTION: The default assumption is the allocation of permanent objects.
    The scope of `with-transient-objects` should be as small as possible, and the user
    has to make sure that code that wasn't explicitly written to account for transient
    objects will continue to work correctly.

wrap-boolean ((*value* BOOLEAN)) : BOOLEAN-WRAPPER   Function
    Return a literal object whose value is the BOOLEAN *value*.

wrap-function-code ((*value* FUNCTION-CODE)) :   Function
        FUNCTION-CODE-WRAPPER
    Return a literal object whose value is the FUNCTION-CODE *value*.

wrap-method-code ((*value* METHOD-CODE)) :   Function
        METHOD-CODE-WRAPPER
    Return a literal object whose value is the METHOD-CODE *value*.

wrapped-type-to-type ((*self* TYPE)) : TYPE   Function
    Return the unwrapped type for the wrapped type *self*, or *self* if it is not a wrapped
    type.

wrapper-value-type ((*self* WRAPPER)) : TYPE   Function
    Return the type of the value stored in the wrapper *self*.

writer ((*self* STORAGE-SLOT)) : SYMBOL   Method
    Name of a method called to write the value of the slot *self*.

yield-define-stella-class ((*class* CLASS)) : CONS   Function
    Return a cons tree that (when evaluated) constructs a Stella class object.

Function Index

Variable Index

Concept Index

(Index is nonexistent)

Table of Contents