LambLisp

01 Red Fox Alpha

Chapter 1

Real-time Lisp for Microprocessors

Figure 1.1 Eponymous Lamb

LambLisp by Frobenius Norm Copyright 2025 Frobenius Norm LLC.

LambLisp is a **real-time** implementation of the **Scheme** dialect of the **LISP** programming language. *Scheme* is governed by standards known as *Scheme R5RS* and *Scheme R7RS*. LambLisp is designed primarily to *Scheme R5RS*, with some features from *Scheme R7RS* and **Common Lisp** that aid in development of **real-time embedded control systems**.

LambLisp provides a robust and scalable Hardware Abstraction Layer (HAL). The LambLisp HAL provides Arduino-style hardware abstraction layer and many additional features focused on embedded controls, and enables convenient interoperability with existing device drivers or other C/C++ libraries.

LambLisp is now running on **Espressif ESP32** using the tools from **platformio**. The Espressif ESP32-S3 devkit is the base test platform.

A **ESP32 hardware demo** is available, built on the ESP32-based **Freenove 4WD Car Kit**, programmed as a simple autonomous vehicle. The vehicle demonstrates a variety of hardware control techniques, including I2C, PWM, analog & digital pin control, and time management for high throughput and low latency.

A **Linux** version of *LambLisp* is available for X86_64 (aka AMD64). Linux is not a real-time operating system, but it allows use of LambLisp's other advanced features, such as tail-recursion, adaptive incremental garbage collection, first-class hierarchical dictionaries and first-class macros, and simple integration with existing C/C++ libraries.

A **Linux** version is available for aarch64 (aka ARM64), with Nvidia CUDA bindings. It has been demonstrated working on the Nvidia Jetson Orin Nano development kit.

For more information:

- Read the LambLisp manual online at
  https://wawhite3.github.io/LambLispRT-v01-linux_x86_64/html/index.html
- Find the LambLisp manual in PDF at
  https://wawhite3.github.io/LambLispRT-v01-linux_x86_64/LambLisp.pdf
- Download the full LambLisp repositories from
  https://github.com/wawhite3/?tab=repositories&q=LambLispRT
- Visit the developer Frobenius Norm at https://frobeniusnorm.com.

Note

The code repository is renewed frequently, with history deleted. Always check back to see if a new repo is available, not just an update to the previous repo.

1.1 Why Lisp?

LISP is the original language of artificial intelligence, and is the second-oldest programming language still in use (after Fortran). It was designed for *symbolic reasoning* rather than computation. From the theoretical base beginning about 1960, *LISP* quickly grew into a practical language with several well-supported variants. *LISP* and LISP programs were always uppercase; the 8-bit byte had not yet been standardized and text was often packed into 6-bit chunks with no room for lower case.

Today, approaching the mid-21st century, *Lisp* is active in many variants: SKILL from Cadence Design, Auto↩ Desk Lisp, ChezScheme from Cisco, and others. Like *LambLisp*, these are often tailored to solve problems in a specific domain, and descend from either *Scheme* or the earlier *Common Lisp*. *LambLisp* is optimized for intelligent real-time control of physical processes.

The *LambLisp* bibliography contains a mini-history of automated symbolic reasoning since the Renaissance. It contains work beginning with Isaac Newton and Bertrand Russell, through to Alonzo Church, John McCarthy, and Edsger Dijkstra, as well as many less famous and more recent authors. In the 21st century development of automated symbolic reasoning continues, with recent publications on already well-trod topics as real-time garbage collection and macro expansion. New approaches to artificial intelligence have created renewed interest in systems (like *Lisp* and its descendents) that offer provable correctness and traceable results.

Any papers cited in the *LambLisp* documentation can be found in the bibliography at
https://github.com/wawhite3/LambLisp-Bibliography

Additionally, these books provide excellent exposure to many of the issues encountered when developing a *Lisp* programming system:

- Structure and Interpretation of Computer programs (Abelman and Sussman). Known as SICP, it was a standard Computer Science textbook for many years.
- Lisp in Small Pieces (Quiennec). Another excellent textbook, which drew lessons from SICP and examines more topics and more detail.
- The Art of Computer Programming (Knuth). Written in the pre-C era, TAOCP is an encyclopedia of algorithms that remain foundational to modern computer science.

Lisp invented, aggregated, or evolved all the important techniques used in programming languages today:

Some *Lisp* Innovations
stepwise code optimization
interactive computing
just-in-time compilation
on-the-fly code updates
macro expansion
polymorphism & "duck" typing
object systems
lexical and dynamic scoping
tail recursion
garbage collection
parameter passing modes
exception handling

Lisp offers tremendous scalability. At the *micro* end, *Lisp* makes an ideal *assembly language*, with a simple prefix notation that can transform directly into machine instructions. At the *macro* end, new functions can be defined and combined dynamically in a way not possible in C++ or Python. These new functions can be the results of substantial analysis performed elsewhere (call it **AI**), putting it at a scale billions of time larger than the assembly language application.

Lisp is **interactive** and *Lisp* programs can be modified on-the-fly, while they are running. Programs can evolve. In other languages, predefined parameters may be tuned during operation, but the code does not change. In *Lisp*, the code may continually evolve, and provides *Lisp* with the power of induction and of continuous improvement.

The concept of interactive computing applies not only to interaction with people. **Machine-to-machine interaction** is an inherent behavior of the interactive nature of *Lisp*. In *LambLisp*-based control systems new algorithms can be developed off-system, and then downloaded without interrupting the controlled process. Today, those off-system resources can be **AI assets** in the cloud, and *LambLisp* can be the target of **AI-generated code**.

An important part of the attraction of *Lisp* is the simple syntax, based on just a few rules that are compact and fast on any processor, even in 1960. This led to a variety of *Lisp* dialects; two of the most important are *Common Lisp* and *Scheme*. *Common Lisp* is in the "large Lisp" category, although there are many partially-compatible subset implementations.

Scheme is in the "medium-size Lisp" category, and was a purposeful attempt to distill the fundamental behaviors of *Common Lisp* into a simpler set.

In the lifetime of *Lisp*, many processors have come and gone. Today ARM-based processors have scaled to be competitive in many control applications. There is a wide variety of boards and modules available, and plenty of inexpensive parts running on the commonly available I2C bus. Today's 32-bit microprocessors present an ideal platform for *Lisp* real-time control.

1.2 Why *LambLisp*?

There are some features that make *LambLisp* special in the *Lisp* menagerie.

1.2.1 Real-time control

LambLisp has been optimized to solve **real-time control** problems, using a variety of techniques described elsewhere in this document. The primary breakthroughs in real-time *Lisp* were published in **Dijkstra 1976**, and elaborated in **Yuasa 1990**. These papers describe how to eliminate the long pause required for stop-the-world garbage collection.

LambLisp allows continuous interaction with supervisory controllers. In turn, this allows the use of off-system rescources, such as **AI systems**, to monitor embedded performance, tune existing parameters, and install new algorithms on-the-fly without loss of control.

LambLisp also adds many enhancements for real-time control, such as fast type-specific operations and Arduino-compatible API.

1.2.2 Widely recognized *Scheme* language specification

LambLisp substantially conforms to the **Scheme R5RS specification**. *Scheme* is widely taught and there is plenty of information online. There are also many desktop versions of *Scheme* available for study.

1.2.3 Reuse of existing C++/Arduino libraries

LambLisp's open API allows existing code to be easily incorporated into the *Lisp* environment. These libraries require only a thin dispatch layer and run at at the same full speed as any other *LambLisp* native function.

There are a few embedded micro-Lisps but they require new libraries for each supported device or feature. The same can be said of the micro-Pythons. The micro-Lisps also have evolved on the tiniest processors, and therefore provide a small *Lisp* core but no conformance to any particular specification.

LambLisp can incorporate existing C/C++ libraries unchanged, and has pre-built interfaces to Arduino-style I/O, including I2C (Wire) WiFi, PWM, and others.

1.2.4 Adaptive real-time garbage collection

Lisp behavior depends on a continuous supply of memory, which must be reclaimed when no longer in use, and then later reissued. The reclamation process is called **garbage collection** or just **GC**.

Until the 1990s, *Lisp* could not be applied to real-time control applications due to unpredictable lapses in control during *GC*. While *GC* was occurring, the controlled application was left unsupervised. This problem was solved by Taichi Yuasa, based on earlier work by Edsger Dijkstra. Yuasa provided a provably correct solution that divided *GC* into increments, and provided formulas for the key metrics in a correct implementation.

In *LambLisp* real-time guarantees are provided by Yuasa's *GC* formula. *LambLisp* continuously calculates the *GC* start threshold (Yuasa's M metric) and allocates additional memory when required (Yuasa's N metric).

LambLisp realizes additional performance gains with *GC* idle-time look-ahead. By setting a value for *idle threshold*, idle time can be applied to garbage collection; later during peak loads GC passes can be omitted altogether. This effectively pushes some workload out of the peak periods and into idle periods, reducing peak loop time.

These idle-time collections will be larger than the normal GC quantum. A small additional benefit is therefore obtained because GC loop overhead is spread over a larger collection than the normal GC quantum.

1.2.5 Open API for new *Lisp* primitives

LambLisp's open API allows new language features to be implemented in C++ and easily incorporated into the *Lisp* runtime environment. There is no special "foreign function interface" as in other *Lisps*; simply write a function with the correct signature, and assign it a value in the *LambLisp* runtime environment. This is how all the *Lamb↩ Lisp* native functions are implemented; any new functions added are not "foreign" and do not suffer a performance penalty. This design is also useful for writing and distributing proprietary code.

1.2.6 Incremental over-the-air updates without reboot

In C++-based systems, half of the persistent storage must be devoted to over-the-air updates. In this case, an entire new program image is downloaded before the existing image is marked obsolete. After, that, the storage for the old image is still required for future OTA updates, but otherwise is not used again. A reboot is required to start the new image.

With *LambLisp*, OTA updates can be incremental. By updating definitions in the execution environment, new *Lamb↩ Lisp* code runs as soon as it appears. The memory used by the obsolete code will be reclaimed. No reboot is required after an update.

1.2.7 First-class hierarchical dictionaries

A part of the success of Python has been its pervasive use of hash tables. Hash tables are also used extensively in *LambLisp*, supporting the representation of environments, dictionaries, and objects. *LambLisp* dictionaries are *hierarchical*, providing a parent/child relationship between hash tables. The internal execution environment of *LambLisp* is a dictionary.

1.2.8 Incremental, just-in-time compilation

Much of *LambLisp* behavior is implemented in the *runtime system*, a collection of compiled routines that perform the basic operations defined in the *Scheme* specification. The process of compiling begins with "expanding" those symbols whose values are functions. In this case, "expanding" means to replace the symbol with its value (a *procedure*) wherever appropriate. This can be done case-by-case; when a *LambLisp* program is fully transformed this way, execution becomes a series of linked C++ function calls. The downside to this is that debug information is lost.

1.2.9 Lexical scoping

Lexical scoping is what most programmers are already familiar with. C/C++ and Python programmers will discuss *local variables* and *global variables*; these are typical in lexical scoping. Some *Lisps* (but not *Scheme* or *LambLisp*) have *dynamic environments* and *special variables*, which do not follow the familiar lexical convention. In *Scheme* and *LambLisp* there is only *lexical scoping*.

1.2.10 Tail recursion and tail-calls

Tail calls allow many problems to be solved recursively without a growing stack, allowing compact, inductive code in place of iteration. *Continuation-passing style* is a programming idiom on which the next step in a computation is always an extra argument to every procedure. That "next step" is called a *continuation*. When a procedure has determined its result, instead of returning, it tail-calls the continuation. This is an elegant way to solve many problems by induction.

1.2.11 Procedures are first-class data types

Procedures are created as the result of evaluating a **lambda expression**. They get created, stored, and destroyed, just as any other value. They can be passed as arguments to other procedures. In LambLisp the **nlambda** and **macro** expressions are specialized *procedures*.

1.2.12 Runtime collaboration with C++ code

New *LambLisp* native functions can be written in C++, and then objects created by C++ constructors can be passed to *LambLisp* for processing when required. These objects can also (optionally) get automatically garbage collected when *LambLisp* has finished with them. One of the design goals of *LambLisp* is convenient scalability between C++ and *Lisp* layers.

1.2.13 Key feature summary

LambLisp Features
Substantial conformance to Scheme R5RS specification.
Additional features from Scheme R7RS and Common Lisp specifications.
Optimized for 32-bit microprocessor control applications (ARM, ESP32).
Available for Linux. (with limited hardware I/O)
High-level implementation of language primitives for compact high performance.
Compiler with just-in-time and incremental features for optimized size/speed.
Arduino-compatible interfaces for digital & analog I/O, I2C, SPI, WiFi, etc.
Easy addition of manufacturer-provided C/C++ hardware drivers.
Easy integration of application-specific C++ functions.
Leverage existing Arduino skill set, gcc-based tool chain, C runtime.
Adaptive incremental garbage collector for fast, uniform loop times.
Selectable run-time packages (math, ethernet etc) to control program size.
Virtual memory capability from remote or local storage.
Incremental code updates over-the-air without pausing control or rebooting.

1.3 *Lisp* in a nutshell

Elements of *Lisp*
Spaces are separators between expressions.
If an expression starts with `(`, then it is a **list** and requires a matching `)`.
Otherwise, the next group of characters, up to a space, is a **symbol**.
In both cases the expression is called a **symbolic expression**, or **S-expression**.
A *list* contains 0 or more S-expressions between its parenthesis, as in `() (foo) (1 (2 three) 4)`.
There is an **environment** that associates symbols with values.
There is a small foundational set of **procedures** that accept arguments and return new S-expressions.
An **evaluator** continuously reads *S-expressions*, evaluates them, and returns the result.
If the evaluator encounters a list, the first item in the list must be a *procedure* and it will be **applied** to the arguments provided.

Note

To **apply** a procedure to a list of arguments, the *formal parameters* of the *procedure* are paired with the actual arguments provided in the application. Then those pairs are added as bindings to the environment of the *lambda expression*. Finally, the body of the *procedure* is evaluated in that augmented environment.

In the *Lisp* world, any item not a list (or more accurately a *pair*) is called an **atom**. Numbers, strings, and vectors all are *atoms*. Perhaps surprisingly, the foundational *Lisp* does not require these, as they can be implemented using purely symbolic programming. There is an important distinction here between the *mathematical foundations of Lisp*, the *programming language called Lisp*, and *implementations of Lisp*.

Many areas of mathematics have only thrived after a notation was introduced to facilitate reasoning about it. Think of *sine*, *cosine*, the *square root* symbol, integral and differential calculus, matrix algebra, etc.

Likewise, *Lisp* established a small set of symbols and their behavior when encountered in a program. In the days of pen and paper, greek letters would have been used, as they were in the **lambda calculus** from which *Lisp* inherits its intellectual grounding and the **lambda** operator. In the teletype era, only the typewriter symbols were available.

Lisp was created to perform *symbolic reasoning*. *Lisp's* elements of reasoning about symbolic expressions:

Lisp symbolic expressions	Description of operation
symbols	A sequence of non-spaces, such as `x`, `2day`, and `next<>thursday`. Non-space characters may appear anywhere in a *symbol*.
`#f` and `#t`	`#f` represents **false**, while `#t` represents **true**. Anything not *false* is considered *true*.
`()` or `NIL` or `nil`	A singleton that is an *atom* and a *list*, but not a *pair*. Used as a sentinel to terminate lists.
lists	A set of parenthesis enclosing 0 or more *symbols* or *lists*, as in `(1-element-list)` and `(nested (nested () list) lists)`
environment	An association between symbols and values. Often implemented as a list of pairs: `((x 12) (2day 42))`.
`if`	Execute test and choose from alternate code paths.

Lisp symbolic expressions	Description of operation
`eq?`	Accepts 2 arguments and returns **false** if they are not the same S-expression.
`define`	Update or install new `(symbol value)` pair in the current environment.
`set!`	Update existing `(symbol value)` pair in the current environment.
`lambda`	Creates a **procedure**; i.e., a set of formal parameters, a body of *Lisp* code, and the environment in which is was defined.
`eval`	S-expression **evaluation**; if a list, the *procedure* at the front of the list is **applied** to the remaining arguments.
`apply`	Execute the body of a *lambda*, in its original environment enhanced with the arguments to `apply` paired to the procedure's formal parameters.
`quote`	Prevent evaluation of something that would normally be evaluated.
`macro`	A procedure that accepts its arguments unevaluated, i.e., as source code, and returns new source code to be executed in its place.

Some terms should be defined in the *Lisp* context; for example *operators*, *functions*, *procedures*, *macros* are related concepts, but not identical.

Important axiomatic behaviors
The word **operator** is used for any S-expression found in the first position of a list at evaluation time.
An *operator* must resolve to a *procedure* before it can be *applied*, i.e., the operator position may be occupied by a symbol or other expression.
The word **function** is ambiguous and may refer to a *lambda* expression or a *procedure*.
A **lambda expression** defines a function's parameters and body, such as `(lambda (p1 p2 p2) (do-something p1 p2 p3))`
The first argument of a *lambda expression* is the list of *formal parameters*, and the remaining arguments (the *body*) are S-expressions to be evaluated at runtime.
A **procedure** is a *pair* consisting of a *lambda expression* and an *environment* in which the procedure was created.
A **macro** is a procedure whose arguments are not evaluated before the code body is applied; afterwards the result of applying the code body is further evaluated.

Note

When a *lambda expression* is executed, it returns a *procedure*. A *procedure* records the formal arguments and the body of the *lambda expression*, and also the *environment* in which the *lambda expression* was evaluated.

This by itself is an incredibly powerful set of mathematics, but note that it has no numbers or strings. That is because, from the point of view of symbolic expressions, numbers and strings are just symbols. It is easy to write a function to add 2 digits when the digits are just `0` and `1`. From this point of view numbers are not fundamental or axiomatic. Instead they are a result of agreement on the meaning of symbols and their combination. The same can be said of strings. When the problem is presented in terms of symbols and their relationships, programs become amenable to proofs of correctness by familiar methods such as substitution and induction.

Adding numbers, strings, and vectors to *Lisp's* capabilities provides computing facilities for real-world problem types. Adding these types means adding representations for them, and adding operators that can manipulate them. *Lisp* follows the usual rules for representing numbers and strings, and so below we present a list of typical symbolic expressions:

```
"This is a string atom, below is a number atom"
1
```

```
"Below is a list of number atoms"
(1 2 3)
"Below is a list of mixed types, atoms and a nested list"
(42 "another string" (nested list))
"Below is an arthmetic list expression; it must be evaluated to produce the result 15"
(+ 1 2 (* 3 4))
"Below is a symbol atom; when evaluated the result is the symbol's value in the current environment"
sym
```

Numbers, strings, and symbols are *atoms*. When a number or string is encountered by the evaluator it is not further evaluated, or put another way, it evaluates to itself. When a symbol is encountered by the evaluator, the symbol is located in the current environment, and the result value associated with the symbol.

Note that the expression `(1 2 3)` is a valid S-expression (a list), but the evaluator will reject it because the first element of the list (`1`) is not a procedure. The evaluator must be able to *apply* the first element, which therefore must be a *procedure*.

Note

An important feature of *Lisp* is the equivalence of code and data. In some sense, everything is data, until it is submitted to the evaluator function `eval`, at which point it is executed as code.

For example, the job of the *Lisp reader* is to accept *Lisp* input from an implementation-defined *port*, and submit that input to the evaluator. Until it gets to the evaluator, it's just data.

Lisp was developed by discovery, rather than predefinition, and each successful experiment led to another language "standard feature". Eventually this led to overstuffing, as witnessed by Guy Steele's 1000-page tome *Common Lisp*. With a specification that size, many partially compatible subsets emerged, losing some of the *Common* in *Common Lisp*.

In reaction to the size of Common Lisp, an effort was made to extract the most useful and essential behaviors of Lisp into a simpler package, which became *Scheme*. Just as effectively as Guy Steele had recorded *Common Lisp* for posterity, he was also a key figure in factoring out the fundamental behaviors exhibited by a Lisp machine. Those basic behaviors, in combination, could be used to produce all the other more complex behavior. The result is the set of functions found in *Scheme*.

The *Scheme* variant of *Lisp* has many features familiar to programmers in other languages, such as *lexical scoping* and *duck typing*, as well as advanced features such as tail recursion, first-class procedures, and continuation-passing style.

1.4 Getting Started With Real-time *LambLisp*

LambLisp is a commercial product, and requires a license when used to generate revenue, to promote other products or services, or other commercial activity. Licensing allows for *LambLisp* customization, hardware application development assistance, post-deployment support, and a warranty.

LambLisp may be used for non-commercial purposes, but comes with no support and no warranty of any kind.

LambLisp is in ALPHA state, so these directions may be incomplete and contain errors.

Do not use **ALPHA** code on any control application where life or property may be endangered.

You should have some familiarity with embedded computing, cross compiling, and related issues. *LambLisp* is developed using some handy tools from *platformio* (pio), which handles most of the compiler-related activity formerly done through makefiles.

Note

Using *platformio* is not required, but convenient. There is a *LambLisp* release associated with each pio environment.

LambLisp is developed on Linux and there are prebuilt images available for x86_64 (AMD64) and aarch64 (ARM64). The x86 version has minimal support for hardware control, but has available the *LambLisp* open API so that new functionality can be added. The aarch64 version has a minimal working set of bindings for Nvidia CUDA.

For embedded use with ESP32, read on.

In examples it is generally assumed that the working directory is ∼/src/LambLispRT-vxx, where xx is the public version number.

- Purchase an Espressif ESP32 module. The ESP32-devkit-C is known to work, and has a second USB that can be used to access the onboard in-ciruit debugger. Connect both ESP32 USB ports to your development computer. The debugger port works with VScode, codelite, and other debugger GUIs.

  Additionally or alternatively, purchase and assemble the Freenove 4WD Car Kit based on ESP32. This will allow a thorough assessment of LambLisp's control capabilities, with direct digital & analog in/out, motor control over I2C, strip LED control, and many other software-based control features including timers and task queues. There is no in-circuit debugger on this Freenove ESP32 board.

  Note

  It is useful to set up both the ESP32 devkit and your own hardware project and be able to build each one on demand. When, as eventually happens, a difficult bug appears, the in-circuit debugger can sometimes save hours of labor. Arrange your hardware pin definitions and error handling so the the application continues to run on the ESP32 devkit even when the hardware is not present, and it will be easy to switch between dev kit and real hardware.

  There are also native Linux builds of *LambLisp* for X64_64 and aarch64. The aarch64 *LambLisp* is built on the Nvidia Orin Nano devkit and comes with bindings for Nvidia CUDA.

- Download the LambLisp repo from github to your working directory.

- To run on the ESP32 dev kist:

```
cd LambLispRT-vxx      #assuming this working directory

# Please check *platformio.ini* for all supported hardware configurations.
# Here are some examples.

#Use this to build for generic Linux.
#The Linux executable binary will be in build/native/program
#pio -e linux_x86_64

#Use this to build for the Espressif ESP32-S3 dev kit.
#The ESP32 binary will be in build/esp32-s3-devkitc-1/firmware.bin
pio -e esp32-s3-devkitc-1 run

#Use the following to upload the file system and the executable binary to the esp32.
pio -e esp32-s3-devkitc-1 run --target uploadfs
pio -e esp32-s3-devkitc-1 run --target upload
#
```

When LambLisp is started, it loads a file called `setup.scm` from the local file system. It must be a *Scheme* source code file, and must be located in the *current directory* where *LambLisp* is running. This is the **data** directory on embedded systems; on Linux it will be the "current directory" in which the LambLisp process started.

At a minimum, this file must contain a *Scheme* definition for the procedure `loop` (or your local equivalent). To transfer control to *LambLisp*, the C++ main program should run the *Lisp* `loop` procedure as follows:

- initialize a LambLisp instance instance, which will automatically load `setup.scm`, which should contain a definition for procedure `loop`.
- request the value `V` of the symbol `loop` in the *LambLisp* instance
- request that *LambLisp* evaluate the list (`V`), applying `V` as a procedure with no arguments.

An error will be raised if `V` is not a procedure accepting zero arguments.

The file name `setup.scm` is hard-coded. The symbol `loop` is an Arduino convention, and is not required. In fact, multiple *LambLisp* functions can be defined and called at different points in a `C++ loop()`.

1.5 What's in the repository?

LambLisp is delivered as a finished executable, but more importantly, as a set of library components that allow scalbility by adding more native functions to *LambLisp*. There is a separate repository for each supported architecture.

The root of the repository contains these files and directories:

File or Dir	Description
README.md	A mini-README
main.cpp	Sample C++ main program
LambLisp.bin	Pre-built application binary
LambLisp.pdf	Detailed documentation in PDF form
html	Detailed documentation in HTML form
src	Sample main.cpp and sample device code
data	Contains:↩
	- setup.scm
	- *LambLisp* features - object system, timers, etc
	- *LambLisp* I/O interfaces
lib	Contains:↩
	- liblamblisp.a - LambLispRT library archive
	- source code - selected examples for extensibility
platformio.ini	Sample configurations for ESP32 and Linux
w3_pio	Sample low-level board config files for platformio

There are also architecture-specific copies of the generically named LampLisp.bin and liblamplisp.a, which may assist those deploying multiple architectures.

1.6 Glossary

- abstract machine

The terms *abstract machine* and *virtual machine* are often used interchangeably, but there is a subtle difference. An *abstract machine* is a mathematics exercise; it is a set of elements, relations among the elements, and transformations between them. A *virtual machine* is an implementation of an abstract machine. When a VM runs, it behaves as defined by the abstract machine, although it runs on top of some other lower-level machine. That lower-level machine may itself be a software or hardware implementation of another abstract machine.

- association list, or alist

A list of pairs. The first part of each pair is treated as a key, and the second part is treated as a value. The operations *set* and *get* are provided, to install new pairs or to retrieve existing pairs. Operations to *set* might either replace an existing pair, or add it if not already existing, or simply add the new pair to the front of the list, shadowing any matching pairs later in the list. Alists require linear search and so are used where the lists are expected to be short, as they are in hash tables.

- atom

A data item. Most often a simple data item such as a number or a string. Compound atoms, such as vectors, have their own set of native operators to set & get the internal parts.

- AST, or Abstract Syntax Tree

A data structure created as output of analyzing source code. The AST is an internal representation of the source code elements and their relationships. The AST can be used to re-create the original source code it was created from (not counting white space and comments). Creating the AST is the "front half" of a compiler.

- *car* and *cdr*

Refers to the first and second data word in a Cell, respectively. Called *car* and *cdr* for historical reasons. In *Lamb↩ Lisp*, a Cell consists of 3 sequential memory *words* (1 tag + 2 data), where a single *word* is large enough to hold a pointer or an integer, and 2 consecutive *words* are large enough to hold a real number.

In the 3-word Cell, the first word is the *tag* word, the second word is the *car*, and the third word is the *cdr*.

The *tag* word contains a Cell type indicator, and flags representing the garbage collection state of each cell.

Integers are stored in the *car* word. Real numbers are stored in the *car* word, and may possibly extend into the *cdr* word, depending on the underlying microprocessor. Some complex atoms, such as vectors, store a count in the *car* and a pointer in the *cdr*. Symbols store a hash in the *car*, and a pointer to the symbol identifier in the *cdr*.

- Cell

A triplet of information: tag, first word (aka *car*), second word (aka *cdr*). In LambLisp, the three elements of the triplet are consecutive in memory, and are all the same size, large enough to hold a pointer or integer. The tag indicates how the first and second words are to be interpreted. When the tag indicates a Cell is a *pair*, the *car* and *cdr* both point to other Cells (or possibly the same Cell).

A *list* is formed when the *car* of a pair Cell points to an S-expression (which is an item in the list), and the *cdr* of that Cell points to another *pair* Cell. That *pair* Cell may also have a list item in its *car*. By convention, lists terminate with a special S-expression called *the empty list*, denoted '(), or NIL, or nil. NIL is considered a list (*the empty list*), but is not a pair, so it has no *car* or *cdr*.

Cells that are not pairs are called *atoms*. Atoms do not have any directly accesible internal structure, but may have operations that work on them, such as vector subscripting. Many atoms are simple integers, strings, or floating-point nunbers, but there are compound atoms such as vectors and bytevectors.

In LambLisp, there are several additional *pair* types that facilitate high performance in conjunction with the underlying LambLisp virtual machine. In standard Lisp or Scheme code, these would be considered atoms, but in LambLisp they are a bit of a hybrid, because *car*, *cdr*, *append*, and a few other list operations will work as expected.

- compiler, interpreter

Both are language translators; in addition, an interpreter is also the language executor. There is a spectrum between compilers and interpreters, but we can identify a key difference:

Translator type	Description
compiler	a software process that reads source code and produces new code to be executed later, possibly by a different machine.
interpreter	a software process that reads source code, performs some amount of processing, and executes the result immediately on the same machine.

In a compiler, and in any nontrivial interpreter, the sofware process first produces an Abstract Syntax Tree (AST). Creating the AST is the first half of the compilation process; in the second half, code is generated for later execution. An interpreter will create the AST and then traverse it, performing the operations as they are encountered.

In *Lisp*, code compiling is commonly done through the use of a *macro* facility. Macro expansion allows code to be transformed as it is read in, factoring out invariants, and producing new code with the invariants removed or calculated once. For example, replacing a symbol with its value "freezes" the symbol; subsequent reads of the symbol will be replaced with the frozen value whenever they are encountered. When that replacement is done at read time, the value is located and replaces the symbol in the executable program. The symbol-value lookup operation is therefore no longer required at run time. In this way symbols become replaced by their read-time values, effectively compiling a block of code into the execution stream.

This method of compilation, called *incremental compilation* or *just-in-time compilation*, blurs the line between compiler and interpreter.

- dictionary

A *dictionary* is a data structure containing `(key value)` pairs. Pairs in the dictionary can be retrieved by matching a target `key` with the keys of each pair. Dictionaries also allow creating, updating, or deleting `(key value)` pairs.

A simple dictionary might be a list of `(key value)` pairs, requiring linear search. This is appropriate for small dictionaries, or short-lived dictionaries, where the construction overhead or runtime overhead of a faster solution outweighs any potential speed improvement.

For larger or long-lived dictionaries, intelligent key creation and caching when combined with hash tables makes this a very fast data structure upon which to operate.

What's described above is a *flat dictionary*, having a single layer of `(key value)` pairs.

- environments & hierarchical dictionaries

Often called the "symbol table" in other programming languages, in LambLisp the environment is a *hierarchical dictionary*. The *flat dictionary* described above provides a single *dictionary frame* for a *hierarchical dictionary*. A *hierarchical dictionary* is simply a list of flat dictionaries. The frame at the *car* of the list is a *child frame*, while frames in the *cdr* are called *parent frames* or *ancestor frames*.

The *hierarchical dictionary* provides a high-performing foundation for the LambLisp execution environment. It also provides a simple and natural foundation for the LambLisp Object System. Inheritance (including multiple inheritance) and shadowing are inherent behaviors of the *hierarchical dictionary*.

- evaluation

Lisp operates by reading symbolic expressions, evaluating them, and returning the result of evaluation. Atoms need no further evaluation, and evauating them just returns the atom itself. To evaluate a *list*, the item at the front of the list, i.e., the *car* of the list, is assumed to be a function to be *applied* to the remaining items in the list.

- hash function

A function that takes a series of bytes as input, and produces a fixed-length number called a *hash*. When given a series of inputs, the hash function should give a series of outputs passing relevant test for randomness. It is not necessary that the series be truly random, only that the results be widely dispersed. For any given input, the output *hash* must always be the same. It can then be used as an index, replacing a search with an index lookup.

- hash table

A data structure used for efficiently storing `(key value)` pairs). Commonly implemented as a set of operations employing a vector and a *hash function*. To search for an item by its key:

```
- Obtain the hash of the key H, which may have been computed once and stored.
- Obtain the size of the hash table N.
- Compute the hash table index, which is (H % N), or (H & (N-1)) if N is power of 2.
- Search the (very short) alist at that index of the hash table.
```

- lambda calculus

A branch of mathematics developed during the 1930's, where the elements of interest are functions rather than data values.

- lambda, nlambda, macro

A **lambda** expression receives 2 arguments: a list of *formal parameters* and a list of *S-expressions* referred to as a *code body*. When executed, *lambda* returns a procedure, which is an encapsulation of the *formal parameters*, the *code body*, and the *environment* at the time of procedure creation.

An **nlambda** expression is the same as a *lambda*, except that at execution time its arguments are not evaluated before getting bound to the formal parameters.

Like *lambda* and *nlambda*, a **macro** definition requires a list of formal parameters and a code body. The behavior of a *macro* depends on whether it is encountered at *read time* or *evaluation time*.

When encountered at *read time* in the *operator position*, the macro arguments are bound to the macro's formal parameters unevaluated, the macro body is executed, the result replaces the original macro expression in the input stream.

When encountered at *evaluation time*, the macro body is executed as above, and the result is then evaluated.

- LISP

A symbolic programming language developed about 1960. LISP (always uppercase in those days) offered a platform for reasoning, while then-existing languages focused more on numeric applications. The objects of interest in *LISP* are called *symbolic expressions* or *S-expressions*.

- list

A **list** is data structure consisting of a backbone of Cells linked by their *cdrs*. The last Cell in the list points to a special NIL Cell to mark the end of the list. The *cars* of each Cell in the list backbone point to the item that is in the list at that position. Note that every item in the list requires 2 entities: the item itself, and a *pair* Cell that connects the item to the list backbone.

In *Lisp* source code, lists are represented between parenthesis, as in `(this is a list)`.

- NIL

All *Lisps* have some version of NIL, nil, '(), otherwise called "the empty list". This is used to mark the end of a list. In some *Lisps*, it may also be considered to be equivalent to "false", or equivalent to zero, but not in *Scheme* or *LambLisp*.

In LambLisp, NIL is implemented as a pointer to a special statically-allocated Cell. The space for the NIL Cell is a singleton, outside the blocks of Cells managed by the *LambLisp* garbage collector. This allows NIL to be constant at compile time, speeding up *LambLisp*.

There are a few other singleton Cells implemented this way, e.g., for undefined and void values.

- oblist

Lisp is a symbolic computing language and *symbol* is a first-class type. For best performance, LambLisp collects all the symbols encountered into an *interned symbol table*, for traditional reasons called the *obarray* or *oblist*. Symbol hashes are calulated on the symbol's identifier and then stored with the symbol in *oblist*. By this means symbols can be compared with a pointer comparison operation; 2 symbols are the same if they point to the same entry in the oblist. In LambLisp, the oblist is neither an array nor a list; instead it is a hash table, but the *oblist* name is retained.

- pair

A Cell whose purpose is to connect 2 other cells. In a typical *Lisp* tree structure, *pair* Cells will make up half of the Cells used, while the leaves of the tree will be atoms.

- prefix, infix, postfix

In typical algebra notation, to compose two functions *f* and *g* that each require a single parameter *x* we write:

```
f(g(x))
```

The functions are composed using *prefix* notation. Algebra and C/C++ use a mix of parenthesis, prefix, infix, and postfix notation like this:

```
--x + abs(x) + 2 + (x++ - y[z])
```

Lambda calculus uses prefix notation and makes frequent use of the greek symbol lambda (λ), which is not conducive to keyboard use. Instead *Lisp* spells out *lambda* when required.

Lisp (like lambda calculus) uses prefix notation, with parenthesis and symbols being the fundamental lexical elements. To compose functions *f* and *g*, we write:

```
(f (g x))
```

and to compare 2 elements *x* and *y*, there is the eq? operator:

```
(eq? x y)
```

To summarize prefix/infix/postfix, using C++ examples:

Notation		
Prefix	++x	f(x)
Infix	a $*$ b	a + b & c
Postfix	y++	y->foo

- symbolic expression

In its simplest implementations, a *Lisp* **symbol** is any sequence of characters not containing spaces or parenthesis. Some of these symbols sequences are defined by the language (`if`, `lambda`, and so on), along with a value for those symbols, but others are just symbols, to which a value may be attached. These are all symbolic expressions, or **S-expressions**, and combinations of S-expressions and balanced sets of parenthesis are also valid S-expressions.

The basic *Lisp* language can only test for equality of symbols, but real-world *Lisps* define useful data types like numbers and strings, along with operations on those.

The LambLisp implementation depends on a `Cell` structure to contain the details of an **atomic** S-expression (such an as integer), or the links between S-expressions (as in a list). When programming LambLisp at the C++ level, an *S-expression* is therefore a pointer to a Cell.

1.7 LambLisp Architecture Internals

1.7.1 The *dictionary* type

In computer science, a *dictionary* is a data structure and set of operations operations for managing `(key value)` pairs.

LambLisp function	Dictionary operations
dict-set!	install or replace a `(key value)` pair
dict-ref?	return a `(key value)` pair based on its key
dict-ref	return a value based on its key
dict-rm!	delete a `(key value)` pair

The name *dictionary* applies to the features and functions performed, and not to any particular implementation. A sequential search or a database query are both useful implementations of a dictionary for different purposes, with different performance expectations.

LambLisp uses several advanced techniques to ensure high performance dictionaries in embedded control.

Large or long-lived dictionaries are implemented as *hash tables*. When a dictionary is large or long-lived, great performance gains can be obtained by distributing the `(key value)` pairs throughout an array. To perform a lookup, compute a *hash value* for the key, and use that as an array index. The `(key value)` pair, if present, will be in the list of items at that index. The hash function, used to create a hash value from a `key`, produces a fixed-length number from the contents of the key. The output of the hash function must be the same whenever the same key is presented as input. The hash values produced should be widely distributed among the hash table indexes.

Small dictionaries are implemented as association lists, or **alists**. When a dictionary is small or short-lived, the one-time cost of creating a hash table is unjustified.

In hashed dictionaries, at each element of the hashed array there is a (very short) *alist*, so *alist lookup* is a foundational operation and should be as fast as possible.

LambLisp optimizes symbol lookup by computing hash values once when a symbol is created, and storing the hash value with the symbol. The most common symbol operation (test for equality of symbols) reduces to a fast pointer comparison when using a dictionary.

The dictionary type described above is a *flat dictionary*. That is, there is just one vector, and a `(key value)` pair is either present or not.

1.7.2 Hierarchical dictionaries

LambLisp implements *hierarchical dictionaries*. A LambLisp *hierarchical dictionary* consists of a list of *frames*, where each frame is a *flat dictionary*. Hereafter, any reference to *dictionaries* should be understood to refer to LambLisp *hierarchical dictionaries*. In cases where a *flat dictionary* is required, it will be called out specifically.

The fundamental operation in a dictionary is **lookup**. A *lookup* must be done to support every other dictionary operation: create, modify, delete etc. The *lookup* operation consists of providing a key K, and querying the dictionary for a `(key value)` pair having a key matching the key K provided.

When performing a dictionary lookup, each frame is checked in turn for a matching key. If a key is not present in the top frame of the dictionary, it may be present in succeeding (parent) frames. The *dictionary* is a first-class type in LambLisp, available to the Lisp application.

The dictionary type is a fairly general purpose data structure. Keys and values may be any type. Each frame may be an alist or vector, with the choice made on frame size and expected lifetime. Symbol keys are computed once and stored with the symbol, speeding up dictionary operations where the keys are symbols (as in environments and commonly as in object member identifiers).

The primary method of creating dictionaries is with the function `alist->dict`, which traverses an association list and returns a dictionary. The `(key value)` pairs of the alist are distributed into a hash table, which becomes a frame in the dictionary. It is also possible to specify a parent dictionary when creating a dictionary, or NIL if there is no parent (as in a *flat dictionary*). Because dictionaries are lists (of frames), they can be appended or examined with *car* and *cdr*, just as with other lists.

In addition to being available in the *Lisp* application, *hierarchical dictionaries* are central to several other aspects of LambLisp.

1.7.3 Execution environment is a *hierarchical dictionary*

The execution environment is a hierachical dictionary. In the *Scheme* specification, there are 2 environments mentioned, the *syntactic environment* and the *interaction environment*. The syntactic environment contains all those definitions that are found in the the *Scheme* specification. The other is called the *interaction environment*. This where application definitions are stored.

In LambLisp, the execution environment (which is a dictionary) is initialized with 2 frames. The base frame consists of the *syntactic environment* described in the *Scheme* spec, and the second (top) frame is the *interaction environment* also described there. Top-level definitions are installed the interaction environment.

When a function is called, its formal parameters are paired with the asscociated runtime values, and that set of pairs is added as a new frame on top of the current execution environment. This naturally and elegantly implements lexical scoping.

1.7.4 Objects are wrappers around *hierarchical dictionaries*

Hierachical dictionaries also form the basis for LampLisp's Object System (LOBS). Just as the dictionary presents an efficient model for the execution environment, is also provides an excellent base for an object-based appoach.

In the object view each frame in a dictionary represents an instance of a class, while the parent frames in the dictionary represent the parent object instance. This leads naturally to inheritance behavior, including multiple inheritance.

Ine the LambLisp Object System, there is no need to separate class definition from class instantiation. An association list is used to create a new dictionary, and the same alist can be used to produce new instances. In effect, the alist *is* the class definition.

This leads to some interesting freedoms as compared with the C++ or Python approach to class definition and instantiation.

LOBS features
Objects are created from an association list and (optionally) parent objects.
Objects inherit from their parent objects.
Multiple objects may inherit from the same parent object.
Objects may initialize from the same alist, but have different parent objects.
Objects may initialize from different alists, but may share a common parent object.
Objects may have new fields added/deleted any time, as any other dictionary.

To use LOBS as a C++-style object system, do the following:

1. Initialize all objects of the same type from the same association list.
2. Create and initialize new parent objects for each new child object.

To round out the object model, LambLisp provides `dict->obj`. You have correctly perceived that this will convert a dictionary into an "object". This provides a minimal object-style accessor for a dictionary, such that `(some-object 'foo)` will return the value of the field `foo` in the object `some-object`. and `(some-object 'foo 42)` will set the corresponding field on `some-object`.

The simple *set/get* object model suffices for many case. It is possible to add more complex behaviors by assigning procedues to dictionary values, and/or by enclosing dictionaries in a wrapper procedure.

1.7.5 Interpreter & compiler organization

Language implementations depend on *primitives*; these are the language elements that are assumed to exist when a programmer is programming in that language. For a high-level language, the *if-then-else* sequence is a common language primitive.

Low-level languages, such as assembly language, also have a set of primitives. Each primitive does less work when used, but by coding directly in assembly language optimization can be implemented that cannot at the higher level, and the result can be many times faster than a higher-level language. At some level, even a compiled high-level language is still interpreting a lower-level machine.

Many available Lisp implementations are based on underlying *abstract machines*. These mathematic constructs were developed in the 1990s as part of the emergence of the *Lisp machine industry*. Two of the most important abstract machines to emerge were the **SECD machine** and the **CEK machine**. *TinyScheme*, for example, uses an *SECD machine* as its execution target.

LambLisp's implements interpretation of Lisp at a high level; each of the functions in the *Scheme RxRS* specifications is implemented in C++. They all share the same C++ signature:

```
Sexpr_t any_function(Lamb &lamb, Sexpr_t sexpr, Sexpr_t env_exec)
```

The `lamb` parameter is the Lamb virtual machine instance which is to execute the operation. The parameters `sexpr` is the *symbolic expression* to be operated upon by the function, and `env_exec` is the environment in which to execute the function. The return type is a also a *symbolic expression*. Note that `env_exec` is known to be a dictionary, and that its keys are all symbols.

This organization has several implications.

First, the language primitives are coded directly in C++, rather than interpreted at run time through a lower-level bytecode system. This makes compilation of the language primitives easy; simply replace the symbol with its value at read time, using a macro facility.

Second, the *Scheme* specification describes only a small set of primitives, with much of the language being "derived expressions". These derived expressions are not language primitives, but instead they either get expanded once at read time, or interpreted at run time. Their performance depends entirely on the expanded versions.

In the highest-performing cases, macros expand into machine code, utilizing the CPU registers directly to carry out *Lisp* operations. That high performance comes with costs. In an embedded system, that type of compiler would be included onboard, taking up space even though used only occasionally. Alternatively, an offboard compiler could be used. That makes some features difficult to implement, such as on-the-fly incremental code updates.

The derived expressions in the *Scheme* specification form a "compiler" that transforms *Scheme* source code into expanded code that relies on relatively few low-level primitives. Some of those lower-level primitives, such as *if*, *define*, and *set!*, are fundamental to the application of *lambda calculus* to computing, while other (such as mathematical functions) are necessary to solve practical problems.

The simple compiler has a disadvantage: the code it generates is simpler to execute than the source, but not simple enough to be fast unless it is designed to produce assembly-level native code. For example, an expansion of `cond` results in many sequential `if` functions to be executed. In an optimizing compiler, a series of low-level test-and-jump sequences will be coded inline, one for each `if`. This will be very fast.

Short of that, executing `cond` requires a pass through the main loop for each test clause. A system that expanded `cond`, but required sequential execution of `if` (in the *Lisp* context), would relatively slower.

LambLisp has implemented nearly all the *Scheme* language elements as primitives, written in C++. This reduces the number of steps to be interpreted between source code intake, transformation in to an *abstract syntax tree* (AST), and its execution in C++. In the deign of *LambLisp*, a `cond` in *Lisp* source code correspond directly to a C++ function `cond(...)`. Encountering `cond` in an expression does not result in a chain of small bits of `if`

execution. Instead, the C++ `cond` is called, either after its symbol is resolved in interpreted mode, or called diretly through a pointer in compiled mode. All the clauses of the `cond` are executed in a loop at C++ speed, and not in a loop at the *Lisp*-language level or in a bytecode interpreter.

When compiling LambLisp, an immediate performance improvement is realized by replacing symbols in the input (define, if, let, etc) with their values. The downside is that debugging information is unavailable when exceptions occur, most importantly the symbols associated with functions.

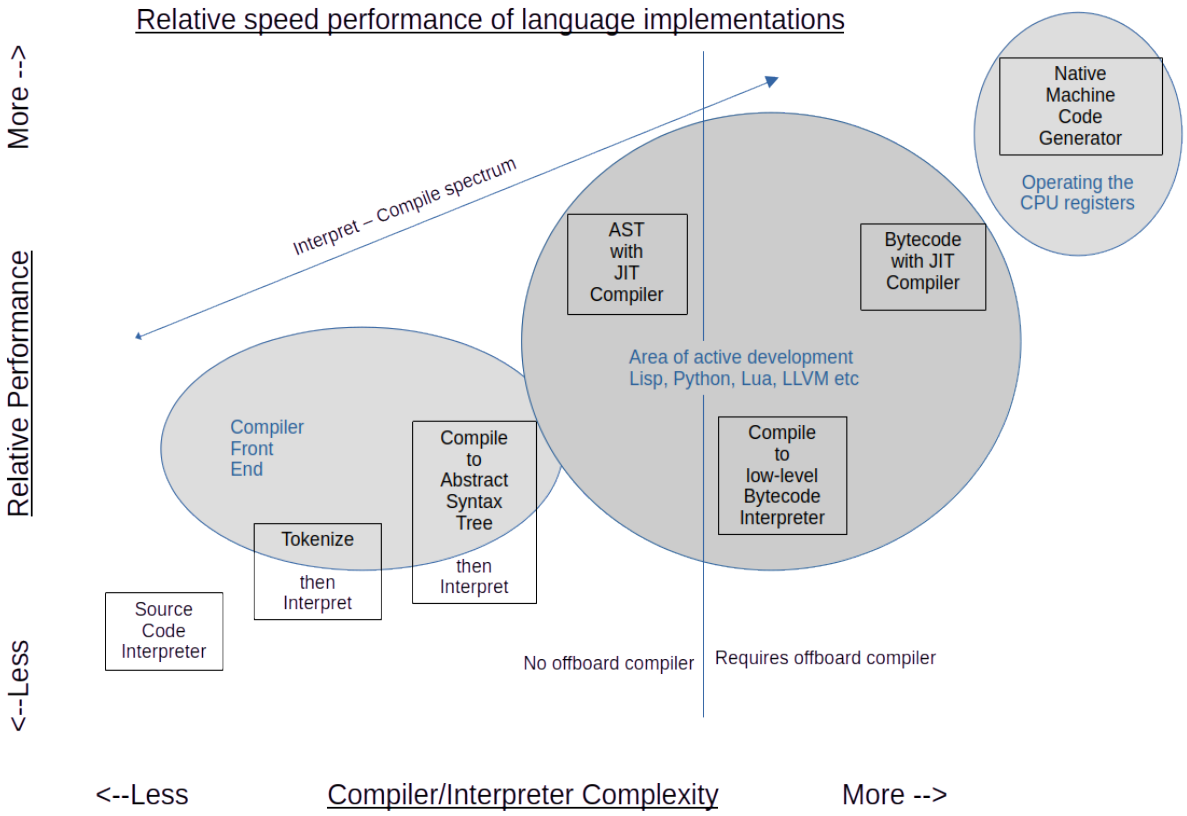

Figure 1.2 Performance Characteristics of Several Implementation Strategies

The compilation process begins by reading the source code and creating an *Abstract Syntax Tree* (AST). The AST is an internal representation of the program, with element of the source code being an element of the AST.

To represent the AST, *LambLisp* uses a "direct-connect" technique, in which high-level Lisp operations (implemented in C++) are executed through pointers directly embedded into Lisp objects. These objects correspond directly to high-level features in the language specification (define, lambda, etc). All of the LambLisp language primitives are implemented this way, and it is possible to add additional LambLisp functions implemented in C++. They are treated as any other first-class function.

As illustrated above and below, this architecture can provide high performance while remaining compact. The axis labeled "Complexity" is a proxy for "how much code transformation is performed?", and shows that there is not a simple choice of "compile or interpret", but a spectrum of solutions that begin with the simplest interpeter and ends with a compiler optimized for a particular CPU. Everything in between is "on the spectrum".

A compiler whose target is the hardware native machine registers will be fastest. The interesting thing to note is that bytecode interpreters can be outperformed by AST interpreters. The reason is that bytecode interpreters do only a small amount of work in the interpreter implementation language, for each pass through the loop while interpreting the implemented target language. A high-level AST interpreter does more work for each pass though the eval loop, as compared with a bytecode interpreter, and those larger chunks of work are done at C++ speed.

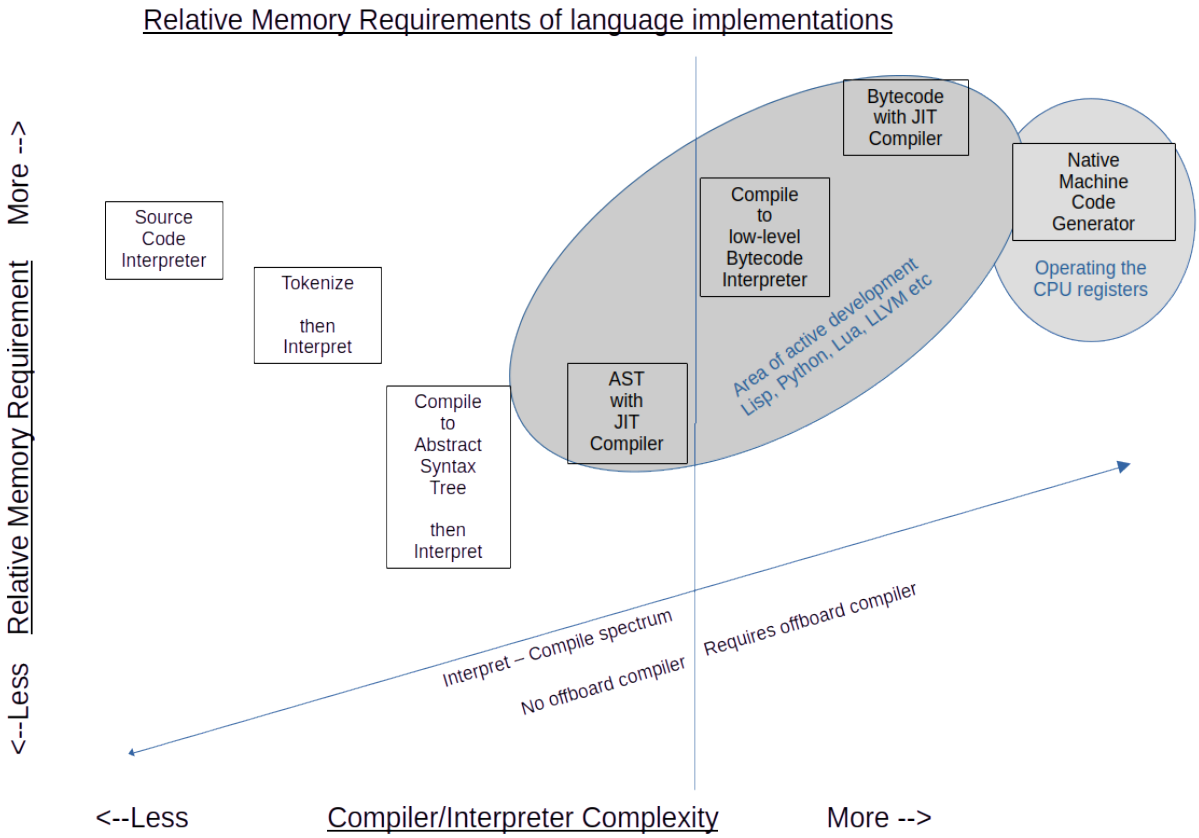

Figure 1.3 Memory Requirements of Several Implementation Strategies

Similarly, there is a relation between runtime memory requirements and position on the compile/interpret spectrum. Because individual bytecodes represent a small amount of computation, more of them are required to represent the original code. The AST representation is small, while retaining all the relationships embodied in the source.

Because *LambLisp*'s AST implementation is high-level and congruent to the features described in the specification, the "bytecode equivalents" used by *LambLisp* correspond exactly to specified features.

During compilation, the AST nodes representing language primitives implemented in C++ are replaced with AST nodes of the same size pointing directly to the C++ feature implementation. This compilation feature allows faster execution with no additional space overhead, because the C++ code reference is the same size as the original signal it replaces. The downside is that, should exceptions occur, the debugging iformation is more obscure, with human-readable code symbols replaced by addresses.

The compilation system is based on *nlambda* and *macro* primitives.

1.7.6 LambLisp Virtual Machine

1.7.6.1 Control Applications as Virtual Machines.

It will help us to think of the running application as a layers of virtual machines. In fact, there is a stack of virtual machines, starting at the bottom with single transistor, and finishing at the top with whatever runtime inputrs the system is designed to respond to.

Those inputs themselves arrive from another, external virtual machine, which has its own internal organization and its own set of inputs, computation, and outputs.

It is also useful to identify the boundary between the *hardware virtual machines* and *software virtual machines*. The illustration below shows a few layers representing the virtual machines involved.

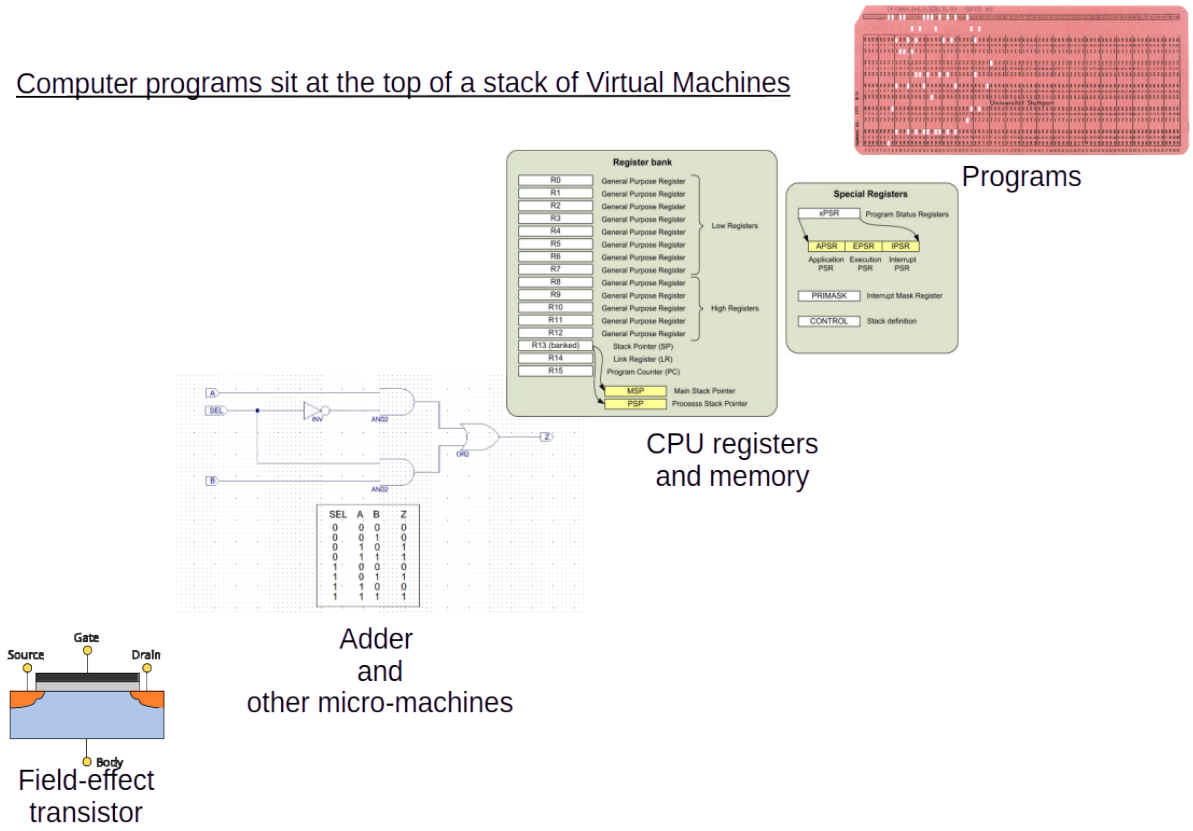

Figure 1.4 A CPU shown as a stack of Virtual Machines

The item labeled "Programs" above is its own virtual machine, consisting of multiple software layers. The layers below that are typically considered "hardware".

At the bottom is the very lowest virtual machine - a field effect transistor. These are operated such that the gate either allows or blocks electron flow through the substrate. When we ascribe meaning to the flow (either a 1 or a 0), the physical flows become meaningful symbols that people can use to communicate.

Transistors are assembled into several types of micro-machines that, when used together, become what we recognize as a "CPU". Symbols of additional complexity are created at every level: numbers, sums and products of numbers, comparison of numbers, interpreting numbers as instructions, etc.

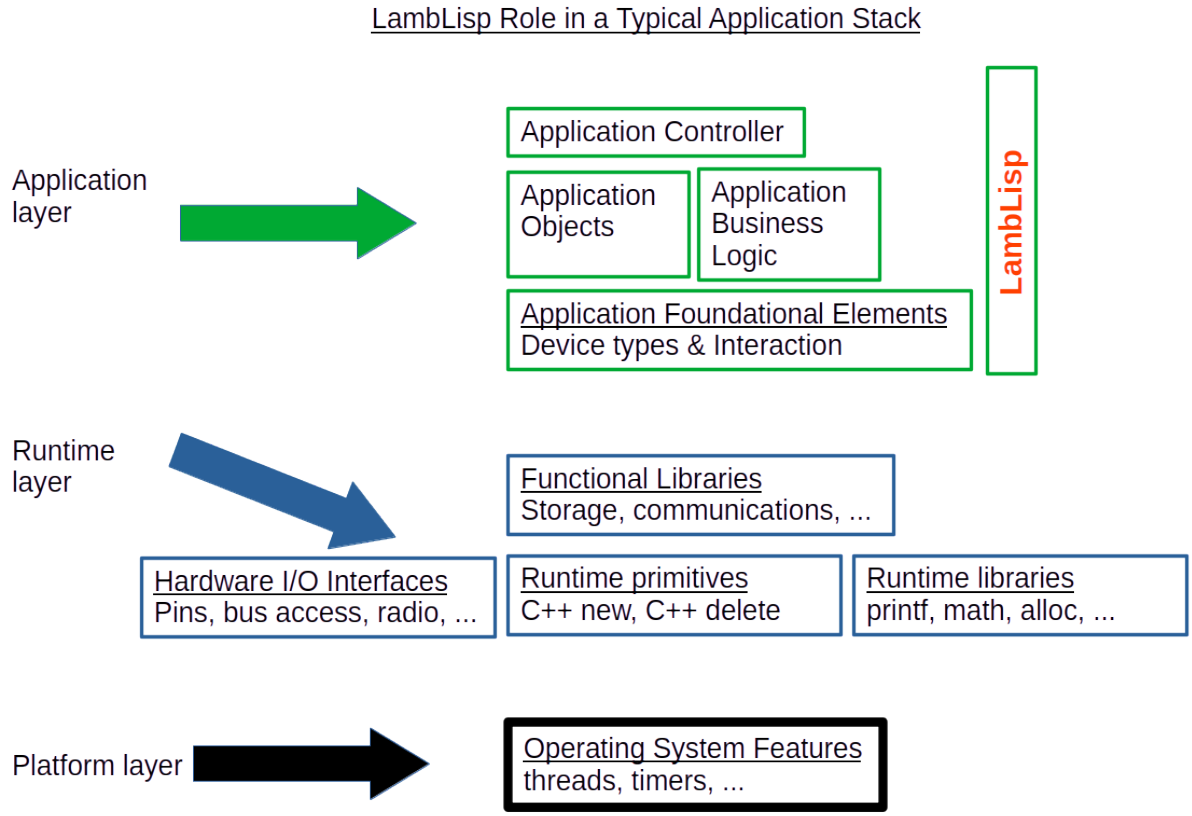

Figure 1.5 Typical Control Application Organization

The illustration above shows a typical control application breakdown. In this view, the code entities form a set with a *partial ordering* relationship, so any of the code entities in the diagram can rely on any of the other entities at an equal or lower level. They can also rely on any entity to the left.

In many loop-based control applications, there are a also few exceptional dependencies, whereby an entity may call another to its right or above. These often occur at system startup. For example, if there is one entity TIME that is responsible for knowing the time, it should not fail under any circumstances, even of not initialized and is unsure of the time. It must provide an answer of some kind.

Another example is when several different types of measurements are required to meet thresholds before the next process step can proceed. These measurements are often averaged over time, and a simultaneous value is required for all them to ensure process synchronization. Each software entity may have some work to do at synchronization time. Each may query the application control layer to obtain formation about the larger environment it is operating in, such as whether the current loop is the synchronization loop, and adapt its behavior as needed.

1.7.6.2 LambLisp Block Diagram

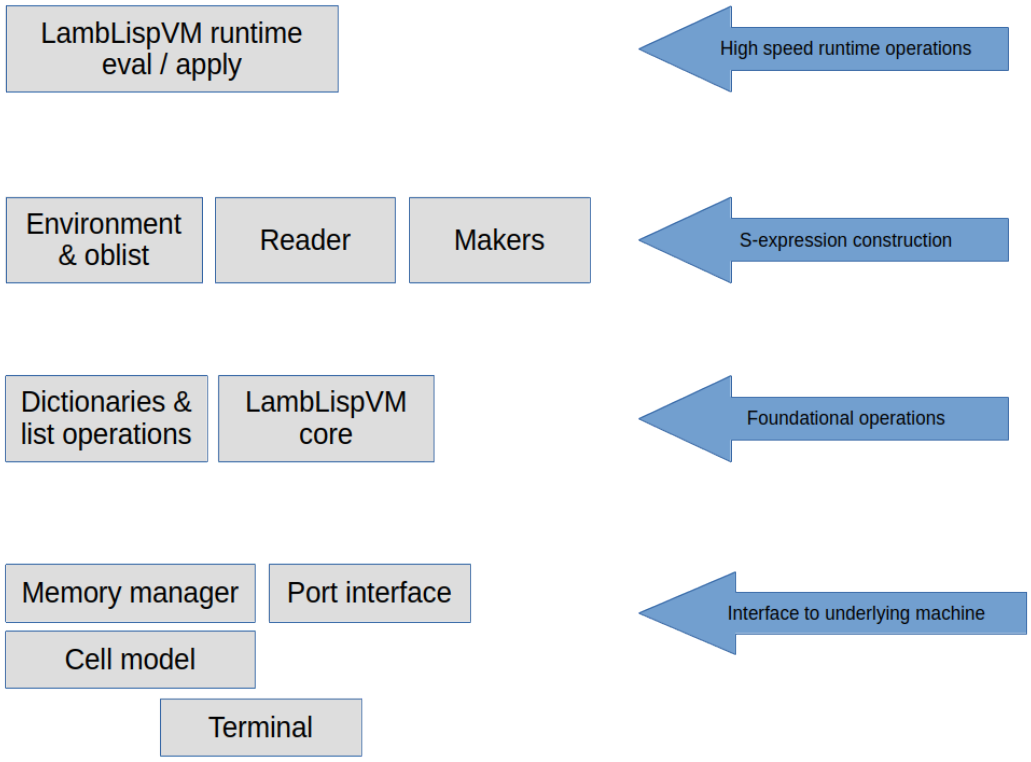

Figure 1.6 LambLisp Block Diagram

When source code is read in, the *reader* front end of the interpreter tokenizes the text, parses the the resulting code, and constructs an *abstract syntax tree (AST)*. The AST is the executable version of the code, an ordered collection containing all the parts and their relationships. Because the resulting executable is a series of direct links, the AST can be traversed rapidly without a lookup table, and because the tree end nodes are high-level functions (rather than low-level bytecodes), the result is a fast interpreter.

Once the interpreter is available, and correctly executes the AST, the macro facility of *Lisp* is used to transform high-level nodes in the AST into a collection of lower-level nodes that perform the same computation. In a simple *Lisp*, consisting only of *if, eq? define, lambda, set! and cons*, every correct program will be factored into those primitives.

When macros are expanded at *read time*, then the effect is one of incremental compilation. After each macro is expanded in to a series of pointers to native C++ code, the result is a compiled program consisting of a tree of native code pointers to be traversed. In addition to allowing for faster execution, this tree-structured architecture is what allows *Lisp* programs to be modified on-the-fly.

One clear example of the performance advantage is in the *Scheme cond* function, which acts as a sequential if-then-elseif. Using the minimal set of primitives, *cond* would expand into a series of *if* functions, each of which would need to be processed by the *Lisp* eval loop. In *LambLisp*, *cond* is a primitve implemented in C++, and does not require its own set of passes through the *eval loop* of *Lisp*. This provides improved performance.

1.7.7 Memory Management

1.7.7.1 Overview

Memory reuse was recognized as a challenge right from the early days of programmable computers. Most computational results are intermediate; once calculated, they are quickly combined with other results and no longer required. *Garbage collection* is the pejorative term for activities related to the identification and reclamation of unused memory. Every programming language must have a strategy for reusing the memory space that no longer holds useful results.

There are 4 main ways to allocate memory: 1) static allocation, 2) stack allocation, 3) heap allocation, and 4) "manually".

Static allocation is done once before the program starts or immediately after. The memory remains available for use by the application for the life of the program. If this memory is to be used for multiple purposes, the processes involved in reuse must be done "manually" by the running application.

Stack allocation is what C programmers experience as "local variables". When a function is called, a new block is allocated from the top of the execution stack. These blocks are interwoven with the return address of the calling functions. When a function returns, the local storage block is popped off and the calling function's context is now at top of stack. In embedded control systems, the available stack space is usually small, often just 4k or 8k bytes.

The memory not allocated statically or to the stack is referred to as the "heap". C and C++ programmers must manage this space manually, using the alloc()/free() or new/delete functions, and employ bespoke tracking mechanisms to distinguish used space from unused. In C++ the bespoke aspect can be somewhat mitigated by requiring exclusive use of constructors/destructors for memory allocation, but still it is a problem for which the programmer must provide a solution.

Many other languages allocates large blocks once from the heap, and automatically manage the interior allocation of those blocks according to the requirements of the language. Among these languages are Python, Java, JavaScript, Lua, and of course Lisp.

1.7.7.2 Cell Memory Model and List Structures

At the lowest level, LambLisp depends on a *cell* structure in 3 consecutive words of computer memory. There are several assumptions about the computer words:

- each word consists of multiple bytes, and the bytes are individually addressable
- each word is capable of holding an address
- each word is capable of holding a signed integer of useful size
- 2 consecutive words of memory may be used to hold a floating-point value.

Of the 3 words in a Cell, the first word is called the *tag* and contains information about the cell, such as the cell type and its garbage collection state. If the tag indicates that the cell is a pair type, then the Lisp *car* and *cdr* are represented in the following 2 words of the cell. If the tag indicates an atom (such as a symbol, integer, string etc), then the remaining parts of the Cell are used to encode the details of the atom.

Nearly all *Lisp* implementations use this *tagged* arrangement, sometimes with variations such as putting the tag, *car*, and *cdr* in separate parallel arrays, or storing some of the tag bits in unused bits of the *car* and *cdr* fields. These variations are generally not useful on today's microprocessors for 2 reasons: 1) Data structure fields are aligned on word boundaries by the C++ compiler. Defeating this feature results in reduced performance. 2) Separate, parallel arrays will reduce the efficiency of the hardware processor's onboard cache.

Instead of a barrier, these features provide a new opportunity for embedded real-time control with *LambLisp*.

First, *LambLisp* devotes an entire byte for type codes. At present only 5 of the 8 type bits are used. It has proven beneficial to performance to add new types that correspond to the objects handled by the *LambLisp* interpreter. For example, the tail recursion implementation results in many *thunks*, which are code expressions paired with an environment to run them in,. As it turns out, there are 2 kinds of thunks: a single S-expressions, or a list of S-expressions. Once freed from the imperative to reduce the number of types, adding just a few types provides immense leverage in the *LambLisp* Virtual Machine. Also, using the entire byte means no masking operation is required, speeding up the very common type-testing operation.

Second, *LambLisp* devotes the second byte to Cell state information. Garbage collection relies on a state-based assessment of each Cell, and cells have 5 GC states, requiring 3 bits. That leaves 5 bits unused, as well as 3 additional unused values for the GC state. These are available for debugging purposes.

Third, in word 0 of each Cell only bytes 0 and 1 are required for the tag. In LambLisp, the extra bytes in this word are used to store *immediate* types, such as short strings or bytevectors. Using these types for short strings or byte vectors eliminates the need for heap operations. These can be especially useful for small-packet interactions with onboard devices via SPI or I2C.

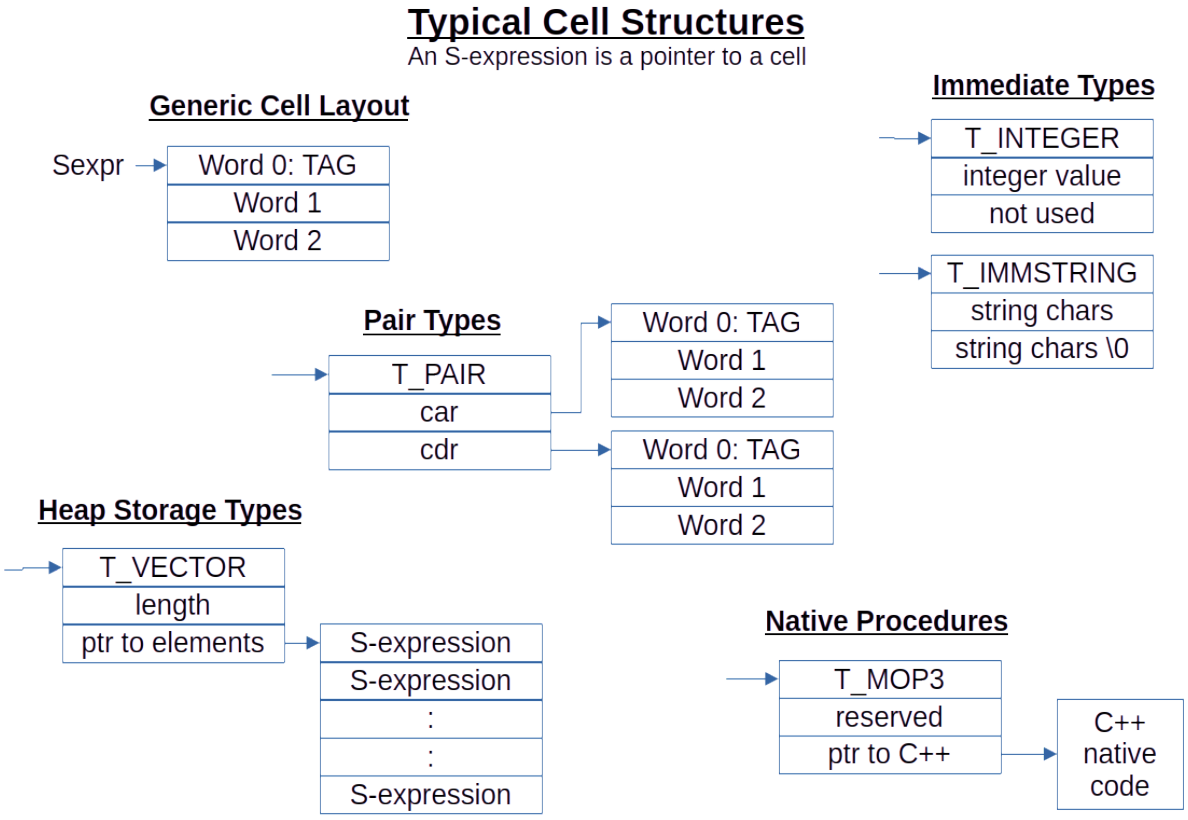

Figure 1.7 LambLisp VM Cell Structure

At the *Lisp* level, only the *car* and *cdr* are visible; the tag word is not directly available. However, the *Lisp* language provides *predicates* for identifying the type of an S-expression, either a pair (using the *pair?* predicate), or a type of atom (using predicates *integer?*, *real?*, etc). This pattern is encouraged in applications programs, where new data types may be created and predicates provide confirmation of an object's type.

The following illustrates an example of a list structure constructed from the cells decsribed above:

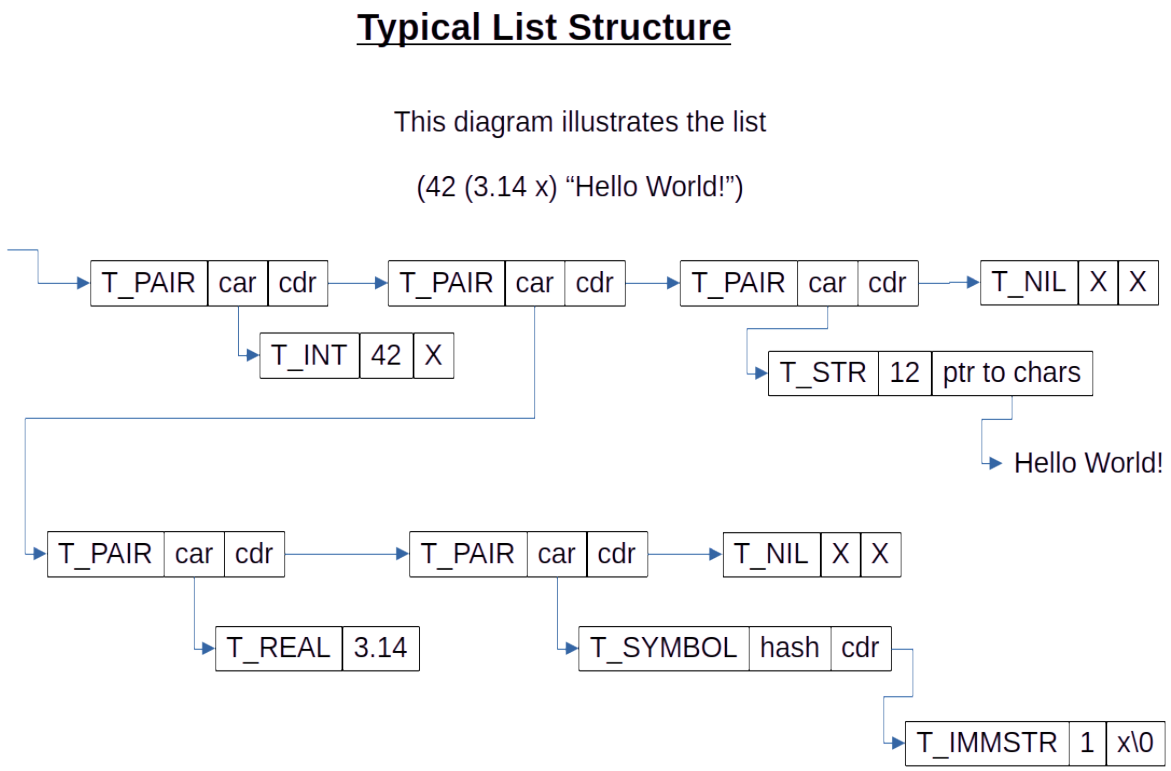

Figure 1.8 Typical List Structure

1.7.7.3 Garbage Collection

The literature on garbage collection is broad and deep, with several important behavioral features identified among the various possible algorithms, primarily:

Garbage collector distinguishing features
copying vs. not copying
compacting vs. not compacting
support for circular structures
need for auxiliary memory during GC
need for computational pause during GC
average throughput vs. peak pause time
concurrent access to shared memory

In a real-time control system, the discriminating factor for implementing a garbage collector is that it can perform its function incrementally. Time spent on GC is time spent not actively monitoring the controlled process. LambLisp implements a tunable control parameter (the *GC time quantum*), that limits the time spent on each GC increment, ensuring that the cell allocator will not become starved of free cells.

LambLisp's memory reuse strategy relies on these key papers:

Key GC Papers	
Knuth 1963	In "The Art of Programming", Algorithm 2.3.5b describes the fundamental GC stack-based marking strategy
Dijkstra 1976	Describes the "tricolor abstraction" and proves the correctness of incremental garbage collection
Yuasa 1990	Provides a method for determining optimum memory size and free reserve during incremental GC

LambLisp implements the stack-based idle-mark-sweep garbage collection cycle described by Knuth, the tri-color abstraction described by Dijkstra, and actively adapts GC performance based on analysis of the kind described by Yuasa.

Yuasa developed 2 parameters, M (GC start threshold), and N (minimum required cell population). The garbage collector begins in the *idle* state. *Marking* begins when the free cell reserve falls below the GC start threshold, and proceeds incrementally. When marking is complete, the *sweep* phase is begun and also proceeds incrementally. Each increment is less than or equal to the configured GC time quantum. Once sweeping has finished, the GC state is *idle* again until the next M threshold is reached.

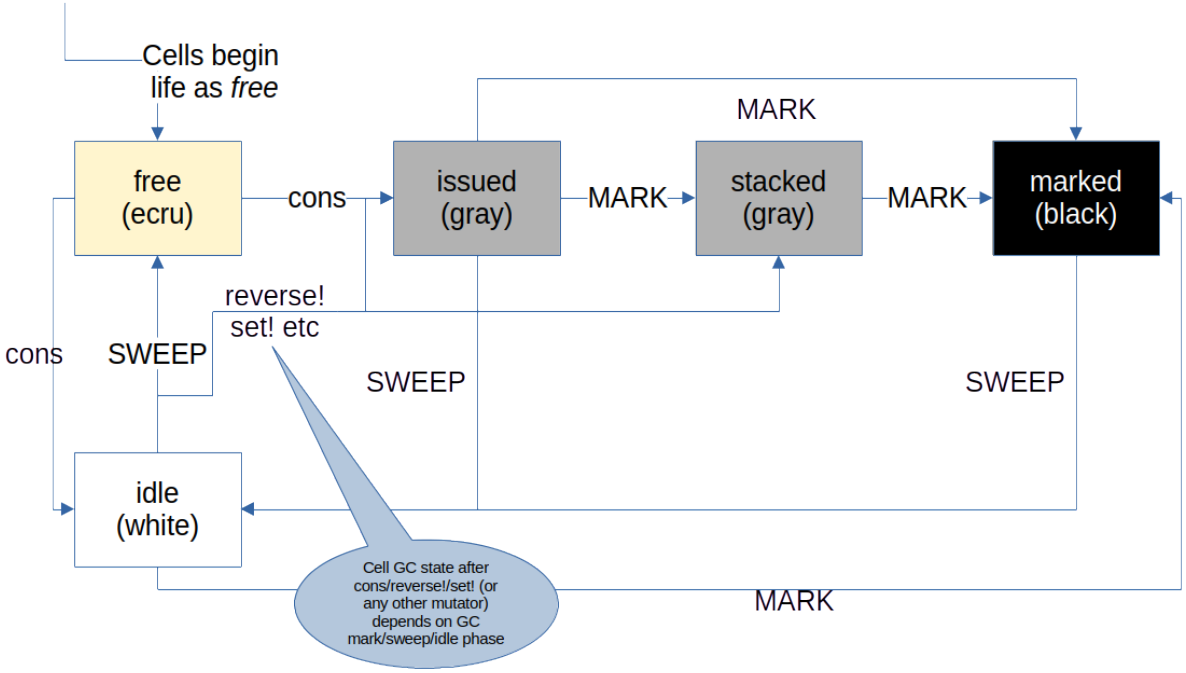

Figure 1.9 LambLisp VM Cell Life Cycle

The GC strategy followed by LambLisp is as follows:

- Each cell produced must be matched by an incremental amount of GC, and on the average, that GC must free up at least one cell per cons.
- LambLisp performs an incremental GC pass at `cons` time.
- LambLisp measures the time required for mark and sweep operations.
- LambLisp provides a tunable parameter, the *GC time quantum*, which is the number of microseconds allocated to garbage collection at `cons` time.
- The time quantum is converted into mark and sweep quanta, which are the number of cells that can be marked/swept in the configured time quantum. The numeric quanta have a minimum value of 2.
- The Yuasa parameters are recomputed at the end of every sweep. The parameter M (minimum free cell reserve) is continuously adjusted as required, and the parameter N (minimum required number of cells) is also checked; if N is greater than the available memory then more memory is requested from the system heap.

1.7.8 Scalability

Scalability is a matrix; one axis is simply "scale up or down". Another axis consists of discrete modes or techniques for scaling. LambLisp has several modes of scaling up and down.

1.7.8.1 Scaling up by adding new fundamental features

LambLisp has a clean, simple interface to C++ code. All the LambLisp native functions have the same signature. As part of the control application, additional native features can be added and will run at full C++ speed. These C++ additions may be useful in several circumstances:

- Implement a low-level hardware abstraction and make it available to Lisp.
- Provide a Lisp interface to a specialized computation library.
- Improve performance at key algorithmic bottlenecks.

For example, LambLisp implements the *digitalWrite* function this way:

```
Sexpr_t mop3_hal_digitalWrite(Lamb &lamb, Sexpr_t sexpr, Sexpr_t env_stack)
{
  Sexpr_t sx_pin = lamb.car(sexpr);         //Get the pin # in S-expression form
  Sexpr_t val    = lamb.cadr(sexpr);        //Get the value in S-expression form
  Int_t pin      = sx_pin->mustbe_Int_t();  //throw error if an INT not provided

  digitalWrite(pin, val != HASHF);          //in Scheme, any value "not false" is "true"
  return val;                               //return whatever value the pin was set to
}
```

To use this C++ function from within LambLisp to set pin 38 HIGH and then LOW:

```
(digitalWrite 38 #t)
(digitalWrite 38 #f)
```

1.7.8.2 Scaling down by removing unneeded functions

At build time, LambLisp allows fine-grained control of the system functions that are included in the application image.

For example, LambLisp supports all the math functions described in the *Scheme RxRS* specifications, and by default all are included in the build. Functions that are not needed can be commented out in the source code and will not be linked into the control application.

This same approach can be used for other groups of function, such as ports and strings. These groups of functions are completely defined in the spec to provide a full solution in their respective domains, but in practice each application uses only a subset of the available capability.

As another example, few embedded control applications will need the full set of case-independent string operators, and they are easy to add in Lisp later if needed, These functions are candidates for implementation in Lisp and loading at runtime, rather than including them in every binary application image.

This scalability feature provides a means to reduce the application size.

1.7.8.3 Scaling up or down by controlling memory allocation.

LambLisp has an adaptable real-time garbage collection implementation that can be scaled in several ways. At every *cons* operation, the GC has an opportunity to run, and may mark or sweep cells at that time.

GC parameter	Function
GC time quantum	Amount of time to spend on GC for each cons
Cell block size	# cells per block
Max cell blocks	maximum # of blocks to allocate
GC % threshold	Increase memory until threshold reached (or max)

The GC time quantum determines how many mark or sweep operations to do during each increment. A smaller number results in shorter GC interruptions, but lower total throughput.

The cell block size is not critical, but the minimum system setup needs 4k cells, and so 8k is a natural default for this parameter, allowing space for a useful *Lisp* application.

The maximum number of cell blocks can be chosen to tune the amount of time spent in GC. The amount of time spent marking depends on the number of cells used by the *LambLisp* program, and not the total number of cells available. Conversely, when sweeping, the entire cell population must be swept.

With more cells in the population, more cells are swept and reclaimed with each GC mark/sweep cycle, because the fixed cost of marking the used cells is amortized over the entire cell population. More memory results in less time spent in garbage collection, but with diminishing returns as more memory is added with constant program size.

1.7.8.4 LambLisp Adaptive Tuning

LambLisp implements several adaptive tuning mechanisms in the GC implementation.

First, *LambLisp* provides an incremental garbage collector, based on the work of Taichi Yuasa. This eliminates extended pauses for garbage collection. Yuasa began with a stack-based, mark & sweep, stop-the-world garbage collector, and transformed it into an incremental adaptation. In the incremental version, each time a new memory cell is issued, some quantum of garbage collection must also be done. The requirement is to keep the reclamation of memory ahead of the issuance, so for each memory cell issued, at least 1 must be reclaimed.

Yuasa determined several parameters required to optimize incremental garbage collection.

- The size of the *Lisp* program.
- How many marks or sweeps to perform at each new memory issuance.
- The minimal amount of free memory before GC begins.
- The size of memory required to support the ongoing issuance rate.

The amount of memory and the amount of free cell reserve is adjusted dynamically to preserve realtime behavior, based on a Yuasa analysis after every marking phase.

Another principal concern is the amount of CPU time devoted to garbage collection. In LambLisp, the percentage of time used in GC is one factor used to determine whether to expand the cell population. If the percentage time in GC is above the configured threshold, then an additional cell block is added. As long as the threshold is exceeded, new blocks will be added up to the configured maximum.

The Yuasa analysis determines how many cells total are required to maintain realtime production, and how many must be kept in free reserve to issue while GC collects some new free cells. The Yuasa parameters are recalculated during every garbage collection.

To allow for maximum throughput while maintaining low GC pause time, *LambLisp* provides a *target loop time* parameter. When each loop has finished, but before returning to the C++ main() that called it, the *LambLisp* virtual

machine will check if the target loop time has been reach yet. If not, then there are potentially some idle tasks that could be done, that might perhaps usefully fill the time.

The most important of these is garbage collection. If there is loop time left before the target time limit, and if the garbage collection process is active (marking or sweeping) and not idle, *LambLisp* will calculate a new, temporary GC time quantum based on the time remaining, and run a GC incremental pass with that time quantum. This tends to put much of the GC effort into "idle time", so the GC subsystem moves more quickly into its own *idle* phase, where no GC is done at all until the Yuasa M parameter is breached.

Interestingly, the printing of messages on *Serial* (or equivalent) also takes time, despite the UARTs and software buffering involved. Two messages printed during the same loop might easily run the loop time up to 10 milliseconds, regardless of what the loop time is without the messages. *LambLisp* provides a means to make the printing occur only if idle time is available. This is useful for informational but noncritical messages. This feature has the added advantage of not computing the message contents if time is not available; it is not just skipping printing, but the entire message evaluation.

The *idle time* feature is available to *LambLisp* applications, as well in as the underlying C++ interface.

1.7.8.5 Scaling by incremental compilation

Once constructed, ASTs are subject to further optimization for speed, sometimes at the expense of size. ASTs are compounds of native Lisp functions (implemented in C++), as well as those composed in Lisp. By replacing Lisp functions with *macros*, expressions can be expanded once into their final forms, replacing functions composed in Lisp with their equivalent in Lisp primitives. By doing that replacement selectively, it is possible to tune the application while expanding only those portions that improve performance.

1.7.8.6 Scaling by using remote memory

Because programs are ASTs, and ASTs are data structures, it is possible to store programs, delete them from memory, and retrieve them later. This approach provides a *virtual memory* capability, which is useful in many circumstances.

Virtual memory advantages
Startup code that is used once can afterward be purged and the space reclaimed.
Algorithms may be chosen and downloaded at runtime, depending on circumstance.
New algorithms may be introduced in the field, while the process is still under control.

1.7.9 Benefits of the LambLisp Scalable Architecture.

Benefit
The interpreter is compact, allowing more memory for the high-level control application.
LambLisp's direct-connect parse tree execution provides high performance.
The interpreter is the entire Lisp virtual machine runtime, already compiled from efficient C++. The Lisp runtime would anyway be included as the functional core of any compiled application.
No need for full over-the-air updates and reboot. Download new code into the storage system and it can execute on next loop().
Lisp code can be stored in file system, dynamically loaded & purged, reloaded etc, providing a virtual memory capability.
No off-board compiler/linker are required to run LambLisp code. Just download the source code to the device.
LambLisp's incremental, adaptive garbage collector recycles memory automatically and predictably, without worst-case pauses.
The LambLisp virtual machine, written in C++, is much smaller than an all-C++ application. This means there are fewer low-level features, fewer bugs, requiring less memory and fewer updates than a full C++ application.
Arduino compatibility, with easy addition of extensions.
No "foreign functions" are required as in other Lisp implementations. Functions written in C++ can be natively incorporated into the Lisp execution environment at full speed.
Provides a basis for macro-based incremental & just-in-time compilation

1.7.10 Advanced topics

1.7.10.1 lambda, nlambda, and macro

From the beginning, *Lisp* functions have been divided into **special functions** and other functions, which have no special designation. They are *special* primarily in that they do not interact with their parameters in the same way as non-special functions. The main differences between these *special functions* and the non-special variety are these:

- Normal lambda functions have their arguments evaluated before the function is applied.
- Special functions receive some or all of their arguments unevaluated.
- Special functions may also be special in many other ways that are function-specific.

While the notion of "function-specific features of special functions" may seem vague, it is a result of the evolutionary nature of early *Lisp*. It was the first computer programming language that understood itself, and that meta-understanding required and produced a lot of exploration.

The topic of *special functions* became inextricably bound up with the topic of **macros**. In all the many *Lisp* implementations over the years, macros have the largest variety of implementations of any other feature in the language.

A major shortcoming of *Scheme R5RS* is the inclusion of `syntax-rules` as the sole language element for a macro capability, Prior to *R5RS* `syntax-rules` was optional. It is not required to implement the rest of *Scheme*, when the simpler **nlambda** construct will do. There is no lower-level macro feature in the language itself that could support `syntax-rules`, even though there are several to choose from in previous *Lisps*. To use `syntax-rules` for creating macros first requires an implementation of another (non-Lisp) pattern-matching language. Therefore, every *Scheme* implementation needs to invent or borrow some other lower-level macro facility to support implementation of `syntax-rules`. This puts the cart before the horse, and goes against the original *Scheme* minimalist values.

As a result, there is no common best practice on the implementation of `syntax-rules`. And it gets worse:↩ `syntax-rules` has now been superceded by `syntax-case`, which has the same drawbacks, while presenting new backward-compatibility challenges.

In *Common Lisp*, a macro system is described that has its own set of quirks, such as not being able to use *apply* with a macro.

1.7.10.1.1 lambda

Recall that a **lambda expression** is a pair, with the *car* being the formal parameter list and the *cdr* being the code body. Recall that the result evaluating a `lambda` expression is a **procedure**. A *procedure* is a first-class type in *Scheme*, which was an innovation at the time that *Scheme* was first formulated. The details of the *procedure* type are left unspecified, and yet one must be returned from `lambda`. In *LambLisp*, a *procedure* is a kind of **pair**, with one part being the lambda expression and the other being the environment that the *procedure* was created in.

In an application of a *procedure* to a set of parameters, the parameters are each first individually evaluated, and set of results submitted to the *procedure* as its arguments.

1.7.10.1.2 nlambda

In *LambLisp*, the first step to implementation of *special forms* is **nlambda**, a concept inherited from **InterLisp**. The *nlambda* operator returns a value of type **nprocedure**. An *nprocedure* is also a kind of *pair*, with one part being a body of code, and the other an environment, the same structure as a *procedure* created with *lambda*.

When applying an *nprocedure* to arguments, the arguments are **not evaluated**. The *nlambda* feature, by itself, is enough to implement all of *Scheme*, except for `syntax-rules`. Furthermore, *nlambda* can be used as an essential part of a `syntax-rules` implementation.

Note

The key distinction between `lambda` and `nlambda` is that **nprocedure arguments are not evaluated**. This allows *Lisp* to operate on a body of source code, even as it evaluates the same body of source code.

1.7.10.1.3 macro

LambLisp provides the **macro** operator, which returns a procedure to be used as a **macro transformer**. A *macro transformer* is a procedure that receives its arguments unevaluated, and returns a value, just an *nlambda*. However, the value returned by a *macro transformer* has the additional property that it will be executed as *Scheme* code. Applying the *macro transformer* to a set of arguments is usually termed *expanding* the macro. When the expanded code runs, it is termed *executing the macro*.

Macro behavior is different when encountered at read time vs. evaluation time.

It usually desireable to define macros before first use, and expand them when encountered at read time. At read time, the value produced by the *transformer* replaces the original macro invocation in the input stream. In that case, the expansion happens once and the transformed (expanded) code is what runs at execution time.

If a macro is encountered at execution time, the same expansion process takes place, and then the result is executed immediately. If a symbol is read, and later defined as a macro, performance will be negatively impacted because the symbol will be macro-expanded each time it is encountered by the evaluator. When defined before being read, the symbol is expanded once and the expansion is what gets executed later.

In *LambLisp* macros are first-class types. That means they can be used as values, as dictionary keys, passed as arguments, etc., like any other data type. Being a variation on procedures, they can be *applied* to a list of arguments. And being implemented as a list type, they are garbage collected as with any other pair.

Note

lambda, **nlambda**, and **macro** all create subtypes of procedures. These provide clean implementations of early *Lisp* concepts like *special forms*, *funargs* etc.

1.7.10.2 Hash tables, environments, classes, dictionaries, and objects

Like Python, LambLisp uses *dictionaries* extensively for flexibility and speed. Dictionaries associate a key with a value, and provide a means for rapid lookup (*O(1)* in the jargon) of values in the dictionary given a key. In *Lisp*, dictionaries are commonly implemented with a hash table, which is a vector of size 2^n, each element of which stores an alist of (key . value) pairs.

It will be useful to review hash tables, to see how dictionaries are built using them.

1.7.10.2.1 Hash functions

A hash function examines a block of data and computes a fixed-length integer called a *hash* of the data. The hashes produced should be well-distributed. Although pseudo-random number generators produce good hashes (using the data as a seed), but adequate distribution can be provided by other, less computationally intensive hash functions. In *LambLisp*, hashes are computed on S-expressions. The resulting "random-ish" number will be the same each time the S-expession hash is computed, and that is what makes it useful.

The hash value is used as the index to a vector (modulo the vector size). At each element of the hash table will be an association list of (key . value) pairs that have hashed to that index.

When multiple items hash to the same index, it is referred to as a *collision*. Key runtime metrics are how much of the hash table is populated, and the length of the longest collision chain.

Collisions are common but usually not a problem because the chains are short. Hash tables are typically chosen to be 1-3 times the expected number of entries, so that the entries can be easily well-distributed by allowing for a significant number of empty hash table entries. This leads to short collision chains, mitigating the sequential performance of association lists used in the chain.

Hash table size is usually rounded up to the next power of 2. This allows use of the C++ bitwise-and (&) operator for extracting the hash index from the full hash value. That takes much less time than the C++ modulo (%) operator, which uses the underlying multiply instruction.

Because symbols hashes are used to look up variables values, symbol hashes are computed once when a symbol is created, and then stored with the rest of the symbol info. Symbols are stored in a hash table called *oblist*. Each symbol record includes a pointer to the print representation and a hash of the print representation. Therefter, testing for symbol equality is a simple pointer comparison, while lookups are speeded by using the pre-computed hash.

The *oblist* is a hash table, but it is not a *dictionary*. Instead, it is used as a set containing symbols, with operations to locate a symbol in the set, or to add a new symbol. Each symbol's hash is computed once and stored with the symbol.

Its purpose of *oblist* is to record the text of each symbol encountered, and convert the text to a unique number. That number is the address of the symbol in *oblist*. This allows fast pointer comparison for checking the equality of symbols.

1.7.10.2.2 Dictionaries and environments

Dictionaries build on hash tables by requiring each element of the hash table to be a list of `(key . value)` pairs. It is possible to check for existence of a key with `(dict-ref? dict key)`, to obtain the value associated with a key using `(dict-ref dict key)`, and set the value associated with a key by executing `(dict-set! dict key value)`.

Basic *dictionaries* are *flat*, consisting of a single hash table.

LambLisp adds an additional feature, a *hierarchical dictionary*. A LambLisp dictionary is a *list of frames*, and each *frame* is either an association list (best for small frames), or a vector of size 2^n, used as a hash table. When querying a dictionary, LambLisp will examine successive frames to find a matching key.

1.7.10.2.3 Environments

This dictionary implementation is used within LambLisp as the data structure holding the *environment* in a series of *environment frames*. To obtain the desired behavior, all the keys in an *environment dictionary* are symbols.

To look up the value associated with a symbol, each frame is searched in turn to find an entry with a matching key. This key match if addresses of the two compared keys are the same.

Recall that symbol hash values are precomputed and stored with the symbol's characters in *oblist*. This provides improved performance in hashed frames, and so any symbols in the 2 large initial frames (base and interaction frames) will benefit from this optimization. Those symbols include all the language primitives (*define*, *if*, etc.)

The execution environment for LambLisp is a dictionary in which all the keys are symbols. Each symbol has a hash value computed at the time of its creation, and that hash is used to reduce the search time to *O(1)*.

1.7.10.2.4 Objects

The flexibility of the dictionary is not limited to the execution environment. In lambLisp, *dictionaries* are first-class types, so you can create new ones, query them, and add or replace their bindings. You can also add a new frame on to an existing dictionary.

It is easy to see that this implementation is useful to represent objects in general, including inheritance from parent objects. Each object will have its local data in the top frame of its dictionery, and parent data in successive frames.

It is not required that dictionary keys be symbols. Any type of LambLisp object can be used as a key, and they all benefit from the intensive use of hash tables. The general rules for hashing are:

- The system hash is djb2.
- Symbol hashes are computed once, when the symbol is created.
- Hashes for immutable atoms (integer, real etc) are computed based on the atom value.
- All other hashes are done on the S-expression value (i.e., the address of a Cell).
- This effectively applies the *eq?* predicate for key matching.

If you are familiar with Python, you may notice the last described behavior is different. Python requires that dictionary keys be immutable types such as *tuples* instead of *lists*. Because LambLisp hashes on address, any type may be used as a dictionary key.

Dictionaries themselves may be used as keys or values in other dictionaries.

Note that LambLisp supports object-oriented programming without requiring predeclared classes. If fact, because objects are dictionaries, and dictionaries are lists of frames, it is possible to create child objects from parent objects on the fly by simply using the list operator *append*.

1.7.10.3 Adding native C++ code to LambLisp.

All of the LambLisp language primitives conform to the same function signature:

```
Sexpr_t f(Lamb &lamb, Sexpr_t sexpr, Sexpr_t env_exec);
```

Where:

Parameter	Description
Sexpr_t	Return value is a symbolic expression type, which is a pointer to a LambLisp memory cell.
lamb	An instance of a Lamb virtual machine; in C++ produced by `Lamb lamb = new Lamb;`
sexpr	The symbolic expression to be evaluated
env_exec	The environment in which the evaluation should take place.

To make the C++ function available in LambLisp, it must be bound into the execution environment. Because the environment is a hierarchical dictionary, the binding can occur in any frame of the dictionary. To bind function `cpp_func` to the LambLisp function `Lamb-func`, use the Lamb *bind_bang* function.

A few things to note about the *bind_bang* function:

- Like most other function in Scheme-based Lisps, a *bang* or exclamation point ! is used to highlight functions that cause mutations to existing data.
- The *bind_bang* function takes 4 arguments, 2 of which are environments (i.e., dictionaries). One is the execution environment. All LambLisp native C++ functions require this argument. The other is the target environment (which is a dictionary) that is to be modified.
- The full signature of *bind_bang* is:

```
void bind_bang(
                Sexpr_t env_to_be_modified,
                Sexpr_t key_to_be_bound,
                Sexpr_t value_to_be_bound,
                Sexpr_t env_exec
                );
```

Sample code for adding a C++ function to LambLisp:

```
//Define your native function somewhere, with the necessary signature.
Sexpr_t my_native_operator(Lamb &lamb, Sexpr_t sexpr, Sexpr_t env_exec);

Sexpr_t sym  = lamb.mk_symbol("Lamb-func", env_exec);   //Create a symbol in Lisp.
Sexpr_t proc = lamb.mk_mop3_proc(cpp_proc, env_exec);   //Create a native code Lisp function.

//Bind the symbol to the function, in this example binding to the top-level environment.
lamb->bind_bang(lamb.r5_interaction_environment(), sym, proc, enc_exec);
```

1.7.10.4 Interfacing to devices

LambLisp provides several examples demonstrating how to interface with a variety of devices. Here we review the different approaches used for different types of devices.

Low level input/output, such as digitalRead(), analogRead() and similar functions are represented directly in Lamb↩ Lisp, identical to their Arduino counterparts, if one discounts the shift in parenthesis. The Sonar example illustrates the techniques required.

The Sonar example also shows that some things are best done in C++; in this case busy-waiting for a pin to change state and measure its time in state with microsecond resolution. Although the system is blocked waiting for the pin, other approaches fail on the ESP32-Arduino. Using interrupts avoids blocking, but the interrupt latency is large and variable, leading to a useless measurement. Use of threads (pthreads) with a dedicated wait thread leads to the same problem.

It's interesting that the Sonar measurement falls in the same range as the loop time, and that is the source of the requirement for busy-waiting. If the loop time was much longer, the interrupt jitter would not matter. If the loop time was much shorter, we would loop and poll instead of waiting.

The Sonar example also exhibits the *singleton* pattern. There is only 1 Sonar, and it has C++ operations, so it makes sense to bundle it into a C++ class and to create a single instance of that class. The interface then becomes very thin, just a matter of marshalling the parameters in the *mop3* interfaces for each function required in Lisp.

I2C and SPI are common board-level communication protocols used to control embedded devices. SPI is point-to-point, while I2C is a command/response protocol with one command device on the bus at any time, and multiple reponse devices available to receive commands and report data.

I2C has been implemented and included in LambLisp. SPI has not been implemented yet, but the same principles apply and SPI could be added to LambLisp in the field.

Interfaces to I2C devices fall into 3 sizes: small, medium, and large.

A simple device like the PCF8574 can be operated directly Lisp over I2C with minimal overhead; this device accepts only I2C write and read commands, which are available directly in Lisp. See the file PCF8574 for the minimal example using I2C low-level operations.

Devices like Wire (aka I2C) and WiFi are in the medium category. These libraries depend on predefined *singletons* of a predefined class. The LambLisp implementation is similar to Sonar, but a bit simpler because the Wire/↩ WiFi class and instances are predefined. On the other hand, both Wire and WiFi have a lot more methods and parameters to deal with.

In the case of Wire and WiFi, C++ does a bit of work, checking and returning high-level errors rather than returning raw error codes to Lisp. Mostly though, the interface layer is concerned with marshalling parameters between C++ and Lisp.

The PCA9685 is a 16 channel PWN controller, originally advertised as an LED controller, but in fact it can be used anywhere a controllable square wave is needed. As it turns out, a variety of motors and linear actuators can be controlled this way.

The PCA9685 is simple in principle, but it allows multiple devices on the bus, and allows each of them to repond to a secondary address, which is programmable and may be shared between devices. This allows operating them in groups with a single command.

That is relevant because it makes the driver code much larger, more than most people would want to rework and take ownership. The PCA9685 LambLisp example shows how to incorporate a large driver without needing to undertand it in detail or modify it, by only interfacing the with behaviors required for your application.

1.7.11 Other Scheme Implementations

This project was started after a review of available options for run-time languages to augment C++ embedded applications, and the likely candidates were all on the Lisp/Scheme spectrum. Some of the existing implementation that were examined include: TinyScheme, ulisp, microlisp, picolisp, T, L, Pre-Scheme, LispBM, chicken, bigloo, racket, chibi and others. Plug those into your favorite search engine to find even more.

General objections to existing Lisp/Scheme implementations
Required off-board compilation.
Out of scale for a microprocessor.
Non-portable hardware tricks (e.g., using the bottom bits of addresses as flag bits).
Non-Scheme lisps (e.g., no lexical scoping or tail recursion).
Bad fit on the interpret<->compile spectrum (e.g, low-level interpeter or huge compiler).
Lisp-on-Python, Lisp-on-JavaScript, and other such proofs.
Garbage collection: Real-time applications must avoid pauses, so stop-the-world, reference counting and generational GC cannot be used.

1.8 LambLisp Compatibility Matrix

Feature status codes	
🟢	Supported
🟩	In Progress
🔴	Unsupported
⚪	Unspecified

***LambLisp* Compatibility Matrix**			
Embedded Systems Adaptations	Scheme R5RS	Scheme R7RS	LambLisp
Arduino-compatible API (Wire, WiFi, analog & digital I/O, . . .)	🔴	🔴	🟢
Incremental adaptive garbage collector	⚪	⚪	🟢
High performance optimized type hierarchy	⚪	⚪	🟢
Integrate native dictionary type	🔴	🔴	🟢
Object system	🔴	🔴	🟢
Incremental over-the-air updates	🔴	🔴	🟩
Specialized integer and float instructions	🔴	🔴	🟢
Specialized "Bitwise operators AND OR XOR"	🔴	🔴	🟢
Timers to support asynchronous operation	🔴	🔴	🟢
Common interface for all native procedures	⚪	⚪	🟢
Links with existing C++ hardware drivers	⚪	⚪	🟢
Logging facility	🔴	🔴	🟢
Comprehensive interface to operating system	⚪	⚪	🟢

***LambLisp* Compatibility Matrix**			
Scheme language features	Scheme R5RS	Scheme R7RS	LambLisp
Proper tail recursion, lexical scoping, “duck” typing, REPL	🟢	🟢	🟢
Datum labels	🔴	🟢	🟢
#u8 data type	🔴	🟢	🟢
Type predicates	Scheme R5RS	Scheme R7RS	LambLisp
boolean? char? number? symbol? pair? vector?	🟢	🟢	🟢
procedure? string? port? eof-object?	🟢	🟢	🟢
null? bytevector?	🔴	🟢	🟢
Symbols	Scheme R5RS	Scheme R7RS	LambLisp
symbol? symbol=? symbol->string string->symbol	🟢	🟢	🟢
Procedures	Scheme R5RS	Scheme R7RS	LambLisp
define lambda	🟢	🟢	🟢
nlambda	🔴	🔴	🟢
macro	🔴	🔴	🟢
Conditionals	Scheme R5RS	Scheme R7RS	LambLisp
if else cond case and or not	🟢	🟢	🟢
when unless	🔴	🟢	🟢
cond-expand case-lambda	🔴	🟢	🔴

***LambLisp* Compatibility Matrix**			
Assignments, Binding, and Syntax Definition	Scheme R5RS	Scheme R7RS	LambLisp
set! define let let∗ letrec =>	🟢	🟢	🟢
define-syntax let-syntax letrec-syntax	🟢	🟢	🟢
syntax-rules	🟢	🟢	🟢
syntax-error	🔴	🟢	🟢
let-values let∗values define-values	🔴	🟢	🟢
include	🔴	🟢	🟢
include-ci	🔴	🟢	🔴
nlambda	🟢	🔴	🔴
Evaluation and quotation	Scheme R5RS	Scheme R7RS	LambLisp
quote quasiquote unquote unquote-splicing	🟢	🟢	🟢
Reader macros ' ` , ,@	🟢	🟢	🟢
begin do “named let”	🟢	🟢	🟢
delay force	🟢	🟢	🟢
delay-force promise? make-promise	🔴	🟢	🟢
Dynamic bindings	Scheme R5RS	Scheme R7RS	LambLisp
make-parameter	🔴	🟢	🔴
parameterize	🔴	🟢	🔴
Libraries and Importing	Scheme R5RS	Scheme R7RS	LambLisp
import only except prefix rename define library	🔴	🟢	🔴
Records	Scheme R5RS	Scheme R7RS	LambLisp
define-record-type	🔴	🟢	🟢
Equivalence Predicates	Scheme R5RS	Scheme R7RS	LambLisp
eq? eqv? equal?	🟢	🟢	🟢

LambLisp Compatibility Matrix			
Numeric types	Scheme R5RS	Scheme R7RS	LambLisp
integer real	🟢	🟢	🟢
complex	🔴	🟢	🔴
rational	🔴	🟢	🟢
Numeric Operations	Scheme R5RS	Scheme R7RS	LambLisp
number? complex? real? rational? integer?	🟢	🟢	🟢
exact? inexact? exact-integer? finite? infinite?	🔴	🟢	🟢
nan? zero? positive? negative? odd? even?	🟢	🟢	🟢
abs max min + - * / < <= = >= >	🟢	🟢	🟢
quotient remainder modulo	🟢	🟢	🟢
floor ceiling truncate round	🔴	🟢	🟡
floor/ floor-quotient floor-remainder	🔴	🟢	🟡
truncate/ truncate-quotient truncate-remainder	🔴	🟢	🟡
numerator denominator gcd lcd	🟢	🟢	🟡
abs expt log square sqrt	🟢	🟢	🟢
sin cos tan asin acos atan	🔴	🟢	🟢
exact-integer-sqrt	🟢	🟢	🟡
real-part imag-part magnitude angle	🟢	🟢	🔴
make-rectangular make-polar	🟢	🟢	🔴
number->string string->number	🟢	🟢	🟡

***LambLisp* Compatibility Matrix**			
Pairs and Lists	Scheme R5RS	Scheme R7RS	LambLisp
pair? cons car cdr set-car! set-cdr! list?	🟢	🟢	🟢
null?	🔴	🟢	🟢
atom?	🔴	🔴	🟢
make-list list	🔴	🟢	🟢
caar .. cddr (all combinations of car & cdr)	🟢	🟢	🟢
append reverse list-tail list-ref list-set! list-copy!	🟢	🟢	🟢
reverse!	🔴	🔴	🟢
memq memv member assq assv assoc	🟢	🟢	🟢
vector->alist alist->vector	🔴	🔴	🟢

***LambLisp* Compatibility Matrix**			
Characters	Scheme R5RS	Scheme R7RS	LambLisp
char? char=? char<? char>? char<=? char>=?	🟢	🟢	🟢
char-alphabetic? char-numeric? char-whitespace?	🟢	🟢	🟢
char-uppercase? char-lowercase?	🟢	🟢	🟢
char->integer integer->char	🟢	🟢	🟢
Case-independent char-ci- functions	🟢	🟢	🔴
char-upcase char-downcase	🟢	🟢	🟢
char-foldcase	🔴	🟢	🔴
digit-value	🔴	🟢	🟢
Strings	Scheme R5RS	Scheme R7RS	LambLisp
string? make-string string string-length string-ref string-set!	🟢	🟢	🟢
string<? string <=? string=? string>=? string>?	🟢	🟢	🟢
Case-independent string-ci- functions	🟢	🟢	🔴
substring string-append string->list list->string	🟢	🟢	🟢
string-copy string-fill!	🟢	🟢	🟢
string-copy!	🔴	🟢	🟢
string-foldcase	🔴	🟢	🔴

LambLisp Compatibility Matrix			
Vectors	Scheme R5RS	Scheme R7RS	LambLisp
vector? make-vector vactor-length vector-ref vector-set!	🟢	🟢	🟢
vector-fill! vector->list list->vector	🟢	🟢	🟢
vector->string string->vector vector-append	🔴	🟢	🟢
vector->alist alist->vector	🔴	🔴	🟢
vector-copy	🔴	🟢	🟢
Bytevectors	Scheme R5RS	Scheme R7RS	LambLisp
bytevector? make-bytevector bytevector bytevector-length	🔴	🟢	🟢
bytevector-u8-ref bytevector-u8-set!	🔴	🟢	🟢
bytevector-copy bytevector-copy! bytevector-append	🔴	🟢	🟢
utf8->string string->utf8	🔴	🟢	🟢
Control Features	Scheme R5RS	Scheme R7RS	LambLisp
procedure? apply map for-each	🟢	🟢	🟢
string-map vector-map	🔴	🟢	🟢
string-for-each vector-for-each	🔴	🟢	🟢
force delay	🟢	🟢	🟢
call-with-current-continuation	🟢	🟢	🔴
values call-with-values	🟢	🟢	🔴
dynamic-wind	🟢	🟢	🔴
Environments and Evaluation	Scheme R5RS	Scheme R7RS	LambLisp
environment	🟢	🟢	🟢
eval scheme-report-environment	🟢	🟢	🟢
null-environment interaction-environment	🟢	🟢	🟢

***LambLisp* Compatibility Matrix**			
Ports	Scheme R5RS	Scheme R7RS	LambLisp
call-with-input-file call-with-output-file	🟢	🟢	🔴
call-with-port	🔴	🟢	🔴
port? input-port? output-port?	🟢	🟢	🟢
textual-port? binary-port?	🔴	🟢	🟢
input-port-open? output-port-open?	🔴	🟢	🟢
current-input-port current-output-port close-port	🟢	🟢	🟢
current-error-port	🔴	🟢	🟢
with-input-from-file with-output-to-file	🟢	🟢	🟢
open-input-file open-output-file	🟢	🟢	🟢
open-input-binary-file open-output-binary-file	🔴	🟢	🟢
close-input-port close-output-port	🔴	🟢	🟢
open-input-string	🔴	🟢	🟢
open-output-string get-output-string	🔴	🟢	🟢
open-input-bytevector	🔴	🟢	🟢
open-output-bytevector	🔴	🟢	🟢

LambLisp Compatibility Matrix			
Input and Output	Scheme R5RS	Scheme R7RS	LambLisp
read read-char peek-char eof-object? char-ready?	🟢	🟢	🟢
read-string read-line	🔴	🟢	🟢
read-u8 peek-u8 u8-ready? read-bytevector read-bytevector!	🔴	🟢	🟢
write display newline	🟢	🟢	🟢
write-shared write-simple	🔴	🟢	🔴
write-char-string write-u8 write-bytevector flush-output-port	🔴	🟢	🟢
System Interface	Scheme R5RS	Scheme R7RS	LambLisp
load	🟢	🟢	🟢
file-exists? delete-file	🔴	🔴	🟢
command-line	🔴	🟢	🔴
exit emergency-exit	🔴	🟢	🔴
get-environment-variables	🔴	🟢	🔴
current-second current-jiffy jiffies-per-second	🟢	🟢	🟢
features	🔴	🟢	🔴
Exception handling	Scheme R5RS	Scheme R7RS	LambLisp
guard raise	🔴	🟢	🟢
with-exception-handler	🔴	🟢	🟢
raise-continuable	🔴	🟢	🟢
error error-object? error-object-message error-object-irritants	🔴	🟢	🟢
read-error? file-error?	🔴	🟢	🟢

1.9 LambLisp Frequently Asked Questions

1.9.1 What is a real-time control system?

A control system senses the physical world and operates tools to influence the physical world. Today the term applies mainly to software-based controls, but in the past pneumatic and relay-based control systems were used.

A real-time control system offers guarantees that outputs will be updated within a certain time period (the "deadline") after inputs changing.

LambLisp provides "loop-based control", in which the control software has a time slice in which it can operate inputs and outputs in any order, and then returns to its caller. The time slice is not a fixed length, and the control software takes all the time needed to complete its slice. The "loop time" is the total time between the start of one time slice and the start of the next, so it includes system overhead, and also includes time used by any other software components that also get time slices.

It is easy to see that the loop time is closely related to the real-time guarantee that can be provided. Metrics such as maximum and average loop time are important in loop-based control.

There are other types of control systems. Some impose a strict order on the input-calculate-output cycle, with all inputs read at the beginning of the time slice, and all outputs written at the end. Others are asynchronous, with different sections of code running independently, and perhaps in parallel, in response to input changes. Some asynchronous control systems are simulated on top of a loop-based foundation layer.

The real-time guarantees may be described as "hard" or "soft", but these terms have no universally accepted definition.

A hard guarantee is sometimes used to describe applications with negative consequences for missing a deadline. This may be true in a financial application, for example.

However, a more strict definition of "hard guarantee" would be that the system has failed when a deadline is missed, and therefore it should signal the failure to a supervisory system, stop normal operation, and await some external stimulus. This is sometimes appropriate in physical systems, where a missed deadline indicates that the process is no longer under close control and requires a fallback strategy.

With the strict definition, if the system continues to operate, even in a sub-optimal mode or with additional missed deadlines, then it is "soft" real time.

Even with these definitions, the difference between a "fallback strategy" and a "sub-optimal mode" leaves a grey area. A fallback strategy will focus on safety and prevention of further losses. Operating in a sub-optimal mode provides continued economic value while the root cause is addressed.

Soft real-time solutions also commonly have a timeout threshold, after which they will declare failure; this further blurs the difference between hard and soft solutions.

Soft real-time applications vastly outnumber hard real-time. That is partly because these problems can be factored to reduce or eliminate the "hard" aspect. Some examples of factoring a hard RT problem: airbag chip, UART, other types of buffering (perhaps interrupt-driven), being provably "fast enough" by a known margin, using interlocks to pace the controlled process and therefore reduce dependency on timing.

A constrained system does not create a hard problem out of a soft problem. The processor must have enough capacity to solve the problem, else it is a system design defect, not a real-time problem. The real-time problem exists whether or not you have a solution for it.

Also, specifications other than the deadline do not create hard problems out of soft ones. A promise to deliver 10 millisecond loop times is not a "hard" guarantee, unless the system should declare failure and stop normal operation when the 10 ms is exceeded.

Industrial control systems rarely rely on timing to ensure that their different parts are in the correct position for the next operation. Eventually something will age, perform outside the assumed time envelope, and destruction results.

To assure correct operation, one or more sensors will be in place to confirm correct positioning of parts and equipment, and these sensors require an additional loop to fulfill an operating cycle. A faster control loop will result in faster throughput on the controlled system, up to its physical limits, but each step is controlled by its latest inputs, and not by timing expectations.

1.9.2 What are the advantages of *LambLisp* in combination with C/C++ on micro-controllers?

The Lisp language itself provides features not available in C/C++, such as interactive programming, dynamic "duck" typing, and dynamic loading and purging of code. While C++ and other Fortran descendants excel at rapid calculation, Lisp was the original language of AI, developed for solving problems that require reasoning. The definitive book, Common Lisp by Guy Steele, grew to 1000 pages. Lisp, including its Scheme dialect, is used by Cisco, Cadence, and Autodesk in their flagship products.

LambLisp is an implementation focused on real-time control applications while adhering to the well-known and popular Scheme R5RS specification (about 50 pages). The Scheme dialect of Lisp prescribes features familiar to C++ and Python programmers, such as lexical scoping, as well as advanced features such as tail recursion.

LambLisp adheres to the Scheme R5RS standard, and then adds specializations for real time control, including a few helpful features from Scheme R7RS. There are built-in interfaces to Arduino-style I/O, and a simple way to add additional external code, with plenty of examples.

LambLisp allows convenient scalability between C++ and Lisp. It is not required to choose one or the other; they may be used together, each for their advantages. It is easy to call back and forth, even multiple times within one loop.

All the top-level Scheme primitives are written in C++ using the same simple API used to interact with hardware drivers. This API is available to external developers.

1.9.3 What makes LambLisp different from small Lisps, such as Picobit and uLisp, or big Lisps, such as racket, ChezScheme, and chicken?

The limitation on complexity of embedded Lisp derives from limited memory and address space. The recent generation of inexpensive 32-bit ARM is thousands of times more capable than a PIC. Such peripherals as a file system, WiFi, or Bluetooth are not available on the tiny processors.

A Lisp implementation should look very different for a small platform vs. a larger one. Indeed that is what happened. Picobit requires offboard processing and uses a bytecode compiler, and then the bytecode is interpreted at runtime.

In contrast, LambLisp is completely onboard, and does not internally loop over bytecode. Instead it builds an abstract syntax tree and executes that. The nodes on the syntax tree may be interpreted or compiled. The nodes are high-level syntactic elements (not bytecode) specified in Scheme R5RS, such as if, define, set!. This architecture results in a very short interpret/compile stack, a fast transition to the underlying C++ implementation, and fast control loops.

While LambLisp is compact, it is not a "mini Lisp". LambLisp is not trying to occupy the tiniest cpus. Instead, it takes advantage of the increasing processing power to put a more powerful standards-conforming Lisp into recent-generation micro-controller cpus.

Likewise, LambLisp is not a "big Lisp", and is not a platform for general computer science investigations. The focus is on high-performance control.

There are other Lisps and Schemes, but LambLisp was designed with these constraints and differences from other implementations:

- Substantially conform to some recognized and popular specification
- Small enough to fit in a micro-controller (so no ChezScheme and other *Big Lisps*)
- Big enough to take full advantage of the latest 32-bit generation (ruling out Picobit and other tiny implementations)
- Must offer real-time guarantees (no stop-the-world GC, no DSW pointer-inversion algorithm)
- No bytecode interpreter (slow)
- Vectors allocated from heap, not consecutive cells (slow)
- Bytevector support
- No off-board processing required
- Easy reuse of existing Arduino-style hardware abstraction layer and existing hardware drivers

1.9.4 Why not use embedded Python derivatives?

Python derivatives may work for a subset of embedded control applications, but will not thrive in that solution space. The garbage collector is tuned for throughput, without real-time guarantees. A python derivative called Serpent comes closest, with a real incremental garbage collector, but it is not Python, requires off-board processing, has no generalized interface to hardware, and otherwise does not fill the same solution space as LambLisp

Python syntax is a runtime burden for an interpreter. It is the result of a series of graduate school homework assignments, done in Scheme, and the homework answers were required to look very different from Scheme. Interpreting Python syntax requires solutions to all that homework, creating a greater challenge (relative to Lisp) when trying to completely embed a real-time Python programming system. One of the great strengths of Lisp is that the language already interprets itself.

Part of Python's strength is the many interfaces to external libraries, but the mini-pythons cannot take full advantage.

On the other hand, there is a lot to learn from Python, and lessons are incorporated into LambLisp wherever they apply. For example, *dictionaries* are a first-class type in LambLisp, as they are in Python.

LambLisp dictionaries are more powerful than those in Python. They are stacked into a child/parent arrangement, supporting their use directly by the application. They can also be used as the data store for an object wrapper. Also, LambLisp dictionary keys need not be immutable.

LambLisp dictionaries are used internally as execution environments for LambLisp itself.

1.9.5 How do I link existing libraries to *LambLisp*?

The interface to external libraries is the same as that used for *LambLisp* top-level functions, so external C++ code just needs a *LambLisp* veneer to become available to the *Lisp* application.

Every *LambLisp* native operator is written in C++ and has the following signature:

```
Sexpr_t mop3_func(Lamb &lamb, Sexpr_t sexpr, Sexpr_t env_exec)
```

Note

There are just 2 data types required for C++/LambLisp interaction: symbolic expressions of type `Sexpr_t`, and a LambLisp virtual machine of type `Lamb`.

By convention, functions with this signature are referred to as **mop3** type, because they are the fundamental *machine operations* in *LambLisp*, and they accept 3 parameters. LambLisp *mop3* language primitives all have the same signature:

- a reference to a `Lamb` LambLisp virtual machine
- an S-expression of type `Sexpr_t` to evaluate
- an environment of type `Sexpr_t` to evaluate within; this is known to be a *LambLisp hierarchical dictionary*.

On the ESP32-S3 devkit module there is an LED accessed via the C++ `neopixel` function:
This examples shows how to attach the `neopixel` function to *LambLisp*.

The native `neopixel` signature is this:

```
void neopixel(int pin, int red, int green, int blue);
```

To make this available in *LambLisp*, first create the C++ *mop3* function that you want to appear in *LambLisp*.

```
Sexpr_t mop3_neopixel(Lamb &lamb, Sexpr_t sexpr, Sexpr_t env_exec)
{
   int pin   = lamb.car(sexpr)->as_Int_t();
   int red   = lamb.cadr(sexpr)->as_Int_t();
   int green = lamb.caddr(sexpr)->as_Int_t();
   int blue  = lamb.cadddr(sexpr)->as_Int_t();
   neopixel(pin, red, green, blue);
   return OBJ_VOID;
}
```

Then, create a new variable (a [name, value] pair) in the environment:

```
Lamb *lamb = new Lamb;
lamb->setup();

//make a new symbol, or obtain the existing symbol
Sexpr_t sym = lamb->mk_symbol("neopixel", env_exec);

//preserve the symbol from GC while the procedure is under construction
lamb->gc_root_push(sym);

//create the native ("mop3") procedure, which may trigger GC
Sexpr_t proc  = lamb->mk_Mop3_procst_t(mop3_neopixel, env_exec);

//unpreserve the symbol from GC
lamb->gc_root_pop();

//add the variable into the target environment (often the same as the execution environment).
lamb->bind_bang(env_target, sym, proc, env_exec);
```

To operate the LED from *Lisp*:

```
(neopixel 38 128 0 0)  ;pin 38, half red, no green, no blue
```

Note

Garbage collection can happen anytime a new cell is created. Items not in the current environment, or saved on the GC protect stack, are subject to collection. If a native function consumes several cells, each must be pushed onto the GC protect stack before the next is created. All must then be popped off before returning the function result.

1.9.6 What are the currently implemented features?

LambLisp v01 Red Fox Alpha is the inaugural release. It is ALPHA - do not use it on applications that put persons or property at risk.

1.9.6.1 Scheme R5RS coverage

LambLisp is designed to substantially cover the Scheme R5RS specification. All the major language elements (define, set!, if, cond, case, let ,...) have been implemented. There are a few areas that are unfilled, such as the many versions of the "floor" function, but every element of the spec has been stubbed in at least. It passes the chibi tests for the functions that have been implemented (with a couple of bugs).

1.9.6.2 Adaptive Real-Time Garbage Collector

The adaptive real-time garbage collector is implemented and thoroughly tested. It automatically tunes the GC start threshold based on the analysis published by Taichi Yuasa in 1990.

It also implements idle-time collection, time-shifting GC activity out of the program execution phase and into the end of short loops. By collecting ahead, later GC passes during program execution can be skipped entirely. Idle-time collection also processes many more cells than the normal GC increment, reducing the loop overhead associated with the incremental GC.

The GC will also request more memory from the system if the GC load is above a programmed percentage, or if the Yuasa analysis indicates the need for more cells to maintain real-time cell production.

The optimized Yuasa parameters may be stored as initial values for later sessions, which will optimize the allocated cell block size.

1.9.6.3 Tail recursion and Continuation-Passing Style

Tail recursion is supported, and so continuation-passing style can be used in the LambLisp application. For best performance tail recursion has been implemented directly in C++ using the "trampoline" technique, and not through the use of a bytecode virtual machine with its own linked stack.

1.9.6.4 What is still being developed?

The macro system is still under development, so there is no *syntax-rules* yet. Instead *nlambda* is used for syntax elements. Borrowed from *InterLisp*, *nlambda* is similar to *lambda*, but at run time arguments to *nlambda* are **not** evaluated before the procedure body is executed. This has proved sufficient for implementing the *Scheme* language, except for *syntax-rules*. There is a *macro* facility, that will run (aka *expand*) at read time or execution time, and the resulting expansion will execute at execution time. In *LambLisp*, *macro expressions* and *nlambda expressions* are both implemented as variants of *lambda expressions*.

Work on the macro system will continue, and it's not clear that *syntax-rules* is the best way to do that. It was optional in Scheme R4RS, then mandatory in R5RS, and is rumored to be optional again in a hypothetical future R8RS. Something called *syntax-case* has arisen in the meantime.

Given the multi-decade approval process for Scheme specifications, it's reasonable to explore macro techniques that have been used successfully in other Lisps, and look for advantages in the embedded control context.

The quasiquote function is buggy.

1.9.6.5 What is not supported?

As a side effect of the efficient C++ tail recursion, it is not convenient to capture the *current continuation*. Related behavior *tail recursion* is supported, and therefore also *continuation-passing style*; it is just the capture of the *current continuation* that is not supported.

A continuation is essentially a complete process context, containing the execution state (CPU registers, CPU flags, stack, and perhaps main memory) of a thread with the expectation of restoring it later. In C++ the continuation contains (among other things) a complete contiguous stack history. In code specialized for *Lisp Machines*, the return stack is linked (not contiguous). This provides a convenient way to capture the process context (i.e., the *continuation*), for later restoration.

That kind of stack horsemanship is burdensome with the contiguous C++ stack and provides limited value in the embedded control context. not least because there are alternatives (such as actual *threads*). Therefore there is no `call-with-current-continuation` function.

The built-in funtion `syntax-rules` is not implemented. It is not possible to implement `syntax-rules` without having already implemented another macro capability to support it. Therefore, *LambLisp* focused first on `nlambda`. `nlambda` operates like `lambda` except that its arguments are not evaluated before the function body is applied. This is sufficient to implement an otherwise-conformant *Scheme* interpreter.

While `syntax-rules` can surely be implemented using `nlambda/macro` as a base, it is not a priority at present.

Functions that are easy to implement directly in Scheme, such as the case-independent string functions, are omitted temporarily. Ultimately they will be loadable application code and not part of the LambLisp virtual machine.

Rational numbers are not supported, but may be added if demand requires.

1.9.6.6 Why is it called LambLisp instead of Scheme?

During the 1960s, a great many of the desired and desirable behaviors of programming languages were still being worked out. While Fortran was used heavily for numeric computation, *lexical scoping* (which C programmers experience as "local variables") arose in the Algol language, and later "message passing" systems (primarily *SmallTalk*) resulted in the "tail recursion" technique and interactive computing.

LISP provided a significant abstraction of the computation process. By operating primarily on addresses rather than values, *Lisp* provided a method for creating new and arbitrary data structures. A great deal of research and discussion was expended on functions and their abstract properties (fexprs, property lists, nlambdas and the rest). With these tools and techniques, *Lisp* provided a facility for reasoning, not just calculating.

The experimental nature of *Lisp* let to a large number of overlapping solutions to overlapping problems. This led to the development of **Common Lisp**, a 1000-page compendium of *Lisp* functions that had proven useful.

At the same time, there was a realization that "more and bigger" *Lisp* was not necessarily a good thing, producing many incompatible partial implementations. An alternative mindset emerged, that a small set of language primitives would be easier to standardize. The key to success is to be sure that the set of primitives is sufficient to create the kinds of more complex functions found in *Common Lisp*.

These and other ideas (especially *tail recursion* and *continuation-passing style*) came together in *Scheme*, which was designed in the 1970s to integrate the theoretical advances and practical understanding that had occurred over the previous 15 years.

Scheme went through several iterations and the usual technical committee sausage-making process, resulting in two main specifications today, known as R5RS and R7RS.

There has always been tension between *Scheme* minimalists, who prefer to specify a small set of language primitives, and utilitarians, who prefer a language that is useful for real-world problems.

This had led to the sadly comical state where the R7RS specification had to be divided into R7RS-small and R7↩ RS-large, with R7RS-small already being significantly larger than the R5RS specification.

There are few *Scheme* implementations that are completely conformant to R5RS or R7RS. Both specs allow plenty of leeway for alterative behaviors, for example in the implementation of macros, or the results of invalid computations.

For some *Scheme* enthusiasts, the function *call-with-current-continuation* is the *sine qua non* of *Scheme* implementations, so that argues against labeling this system *Scheme*. The latest thinking seems to be that this feature is actually not as great in practice as in theory, and may be removed (or optionalized) in some future R8RS specification.

Likewise, *LambLisp* has a simple macro system based on the *nlambda* feature of InterLisp, while the RxRS specs call for a macro system based on the language primitive *syntax-rules*. This seems to be in flux with *syntax-case* proposed as an improved alternative to *syntax-rules*.

Because *nlambda* is sufficient for *LambLisp* implementation, *syntax-rules* is parked. Macros are implemented for use in just-in-time compilation.

The differences in support for *call-with-current-continuation* and *syntax-rules* argue against calling this language *Scheme*.

1.10 Acknowledgements

I reviewed many *Lisp* and *Scheme* implementations, but the one I studied in detail was *TinyScheme 1.42*, writen in 5000+ lines of high-quality circa-1992 C code. It was too big for the processor I was working on at the time (in 2020). Ultimately I returned to the embedded Lisp problem, but made different design choices than the TinyScheme authors, especially on the compile-vs-interpret spectrum and garbage collection techniques. The study was hugely informative nonetheless.

Chez Scheme is the ultimate large implementation. It is the work of a lifetime, primarily of one man, and a standard for performance and completeness in the Scheme world. During the development of *LambLisp*, *Chez Scheme* was used to provide benchmark behavior in cases where the spec is not clear or offers a choice of outcomes.

I have provided an extensive bibliography that also informed my efforts. If the reader would like to better understand Lisp design, begin your journey there. No doubt you will discover a different destination.

Some images above require:

By VectorVoyagerPNG version: user:rogerb - Own work CC BY-SA 3.0, https://commons.wikimedia.↩ org/w/index.php?curid=127589172 Harke CC BY-SA 3.0 https://creativecommons.org/↩ licenses/by-sa/3.0, via Wikimedia Commons

Bill White 2025

Chapter 2

Class Documentation

2.1 Cell Class Reference

Public Types

Cell type enumeration

Note the following properties of this enumeration:
NIL is a singleton, in the middle of the enumeration.
Types <= NIL are atoms.
Types >= NIL are lists.
Types > NIL are pairs.
Types <= T_PORT_HEAP are heap-allocated objects, requiring specialized finalizing but not specialized marking.
Types <= T_ANY_HEAP_SVEC are heap allocated vectors requiring specialized marking in addition to finalizing.
For Scheme purposes, there is one `pair` type, but there are several other specialized `pair` types that facilitate execution.
These additional `pair` types are atoms, but list operations such as `car`, `cdr`, and `append` can operate on them.

- enum {
  T_SVEC_HEAP =0 , T_SVEC2N_HEAP , T_ANY_HEAP_SVEC = T_SVEC2N_HEAP , T_SYM_HEAP ,
  T_BVEC_HEAP , T_STR_HEAP , T_CPP_HEAP , T_PORT_HEAP ,
  T_NEEDS_FINALIZING = T_PORT_HEAP , T_BVEC_EXT , T_STR_EXT , T_BVEC_IMM ,
  T_STR_IMM , T_GENSYM , T_BOOL , T_CHAR ,
  T_INT , T_REAL , T_RATIONAL , T_MOP3_PROC ,
  T_MOP3_NPROC , T_VOID , T_UNDEF , T_NIL ,
  T_PAIR , T_SVEC_IMM , T_PROC , T_NPROC ,
  T_MACRO , T_DICT , T_THUNK_SEXPR , T_THUNK_BODY ,
  T_ERROR , Ntypes }

Public Member Functions

- Sexpr_t mk_error (const char *fmt,...) CHECKPRINTF_pos2

- Sexpr_t mk_error (Sexpr_t irritants, const char *fmt,...) CHECKPRINTF_pos3

Static Cell constructors, used only for "well-known" atomic cells (nilL, true, false etc).

In general, it is best not to do anything complex at static construction time. There is no guarantee that dependencies will be ready. In particular, use of the terminal may cause the system to crash. Most applications should leave any construction activities to more purposeful code at runtime, using only cells obtained through cons.

- **Cell** (Word_t typ, Word_t w1, Word_t w2)
- **Cell** (Word_t typ, Int_t w1, Sexpr_t w2)
- **Cell** (Word_t typ, Sexpr_t w1, Sexpr_t w2)

Low-level cell field manipulation.

Note that the type field occupies all of byte 0, although not all bits are required. This speeds up the very common type-checking operation, because no mask is required to extract the type bits.

The flags are in byte 1, and a mask operation is required to access the individual flag fields.

- void zero ()
- Int_t type (void)
- void type (Int_t t)
- Int_t flags (void)
- void flags_set (Int_t f)
- void flags_clr (Int_t f)
- void rplaca (Word_t p)
- void rplacd (Word_t p)

Cell value extractors, dependent on Cell type.

- Ptr_t get_car_addr ()
- Word_t get_car (void)
- Word_t get_cdr (void)
- Bool_t as_Bool_t ()
- Char_t as_Char_t ()
- Int_t as_Int_t ()
- Real_t as_Real_t ()
- Ptr_t as_Ptr_t ()
- Charst_t as_Charst_t ()
- Bytest_t as_Bytest_t ()
- CharVec_t as_CharVec_t ()
- ByteVec_t as_ByteVec_t ()
- Portst_t as_Portst_t ()
- Int_t as_numerator ()
- Int_t as_denominator ()

Cell hash value

Hashing is used extensively throughout LambLisp. Each Cell has a hash value, calculated as follows:

- *If the Cell is a symbol, the Cell's hash value is the hash value of the symbol.*
- *Otherwise, the address of the Cell is hashed, and that is the result.*

- Int_t hash_sexpr (void)
- Int_t hash_contents (void)
- Int_t hash (void)

Cell setters converting the C types into S-expressions.

- Sexpr_t set (Int_t typ, Word_t w1, Word_t w2)
- Sexpr_t set (Int_t typ, Int_t a, Sexpr_t b)

- Sexpr_t set (Int_t typ, Sexpr_t a, Sexpr_t b)
- Sexpr_t set (Bool_t b)
- Sexpr_t set (Char_t c)
- Sexpr_t set (Int_t i)
- Sexpr_t set (Real_t r)
- Sexpr_t set (Port_t &p)
- Sexpr_t set (Int_t typ, Int_t a, Charst_t b)
- Sexpr_t set (Int_t typ, Int_t a, Bytest_t b)
- Sexpr_t set (Int_t typ, Int_t a, CharVec_t b)
- Sexpr_t set (Int_t typ, Int_t a, ByteVec_t b)

Accessors for use when the type is known.

If the cell type is already known, then these accessors can be used to efficiently access the cell contents.

- Sexpr_t **prechecked_anypair_get_car** ()
- Sexpr_t **prechecked_anypair_get_cdr** ()
- Int_t **prechecked_sym_heap_get_hash** ()
- Charst_t **prechecked_sym_heap_get_chars** ()
- void **prechecked_sym_heap_get_info** (Int_t &hsh, Charst_t &chars)
- CharVec_t prechecked_str_heap_get_chars ()
- CharVec_t prechecked_str_ext_get_chars ()
- CharVec_t prechecked_str_imm_get_chars ()
- void **prechecked_bvec_imm_set_length** (Int_t l)
- Charst_t **prechecked_gensym_get_chars** ()
- void **prechecked_gensym_get_info** (Int_t &hsh, Charst_t &chars)
- Sexpr_t **prechecked_error_get_irritants** ()
- Sexpr_t **prechecked_error_get_str** ()
- Charst_t **prechecked_error_get_chars** ()

Accessors for use when the cell type is unverified.

*The **any** and **mustbe** accessors will perform type checking and throw an error if an improper access is attempted. The **coerce** operators will perform C coercion on its operand if possible, otherwise throw an error.*

- Bool_t mustbe_Bool_t ()
- Char_t mustbe_Char_t ()
- Int_t mustbe_Int_t ()
- Real_t mustbe_Real_t ()
- Sexpr_t mustbe_any_str_t ()
- Sexpr_t mustbe_cppobj_t ()
- CPPDeleterPtr prechecked_cppobj_get_deleter ()
- Ptr_t prechecked_cppobj_get_ptr ()
- CPPDeleterPtr **any_cppobj_get_deleter** ()
- Ptr_t **any_cppobj_get_ptr** ()
- void **any_cppobj_get_info** (CPPDeleterPtr &d, Ptr_t &p)
- Real_t **coerce_Real_t** ()
- Real_t **coerce_Int_t** ()
- Sexpr_t **anypair_get_car** ()
- Sexpr_t **anypair_get_cdr** ()
- Sexpr_t **error_get_str** ()
- Sexpr_t **error_get_irritants** ()
- Charst_t **error_get_chars** ()
- Int_t **any_sym_get_hash** ()
- Charst_t **any_sym_get_chars** ()
- void **any_sym_get_info** (Int_t &hsh, Charst_t &chars)

Operations on strings

*LambLisp supports several subtypes of **strings**. At the time of writing, there are heap-allocated strings (read-write), strings whose characters outside of LambLisp's managed memory, (aka external or EXT strings), and short **immediate** strings that are contained completely within a single cell. Additional subtypes (such as load-on-demand strings) may be added in future.*

There are functions of the form any_xxx() that can be used with any subtype, and there are type-specific functions of the form any_xxx_yyy() for use where the type is already known, with yyy being a code hint.

- CharVec_t any_str_get_chars ()
- Int_t **any_str_get_length** ()
- void **any_str_get_info** (Int_t &len, CharVec_t &chars)

Operations on vectors and sub-types of vectors

Within the LambLisp virtual machine, a Lisp vector is referred to as svec. This is an array of S-expressions having a fixed dimension. There are also bytevectors; these are an array of bytes, also of fixed dimension.

As with strings, there are several subtypes of S-expression vectors. There is a heap-allocated vector, which may be of any size. There is a second type of heap-allocated vector, that is always sized to be a power of 2. These are provided to support efficient hash tables. There are immediate vectors, which may be of 0, 1, or 2 elements. The 2-element immediate vector can also be used as a hash table. This can reduce search time by half without requiring any heap allocation, at the cost of 1 extra cell allocation.

Bytevectors are also diverse, having heap, external, and immediate variants. Heap bytevectors are allocated, obviously, on the system heap, and the heap space is freed when the bytevector is garbage-collected.

External bytevectors operate on bytes provided externally to LambLisp's memory manager. This space may heve been dynamically allocated from outside LambLisp, or may be located in read-only memory. When a C++ object os injected into LambLisp, it can optionally be provided with a garbage collector callback; in that case the external object can be automatically garbage collected when no it's longer used in the Lisp program.

Immediate bytevectors are contained within a LambLisp Cell. The maximum size is limited by the word size of the underlying platform.

Within the LambLisp virtual machine, there are generic functions of the form any_xvec_xxx() that can operate on any xvec (svec or bvec) subtype, as well as type-specific functions of the form xvec_yyy_xxx(), where yyy is a code hint for the cell storage type (heap, immediate, ROM).

- void **any_svec_get_info** (Int_t &Nelems, Sexpr_t *&elems)
- Sexpr_t * **any_svec_get_elems** ()
- Int_t **any_bvec_get_length** ()
- ByteVec_t **any_bvec_get_elems** ()
- void **any_bvec_get_info** (Int_t &Nelems, ByteVec_t &elems)

Cell conversions to printable representation

These functions convert a Cell (or parts of a Cell) to a printable representation of the S-expression contents of the Cell. Because environments are often included in the descendants of the Cell being printed, the depth of environment recursiveness is limited.

- String cell_name (void)
- Charst_t type_name (Int_t typ)
- Charst_t **type_name** (void)
- Charst_t gcstate_name (void)
- String dump ()
- String **str** (Sexpr_t sx, Bool_t as_write_or_display, Int_t env_depth, Int_t max_depth)
- String **str** (Bool_t as_write_or_display, Int_t env_depth, Int_t max_depth)
- String **str** (Bool_t as_write_or_display, Int_t env_depth)
- String **str** (Bool_t as_write_or_display)
- String **str** (Int_t env_depth)
- String **str** (void)

Static Public Attributes

Cell flags, including gc states for multi-pass incremental gc, tail marker, and spares.

- static const int F_GC01 = 0x01
- static const int F_GC02 = 0x02
- static const int F_GC04 = 0x04
- static const int F_TAIL = 0x08
- static const int F_0x10 = 0x10
- static const int F_0x20 = 0x20
- static const int F_0x40 = 0x40
- static const int F_0x80 = 0x80
- static const int GC_STATE_MASK = F_GC01 | F_GC02 | F_GC04

Flag testing & setting for garbage collection and tail recursion.

The garbage collection algorithm is based on the *tricolor abstraction* described in *Dijkstra 1978*. The original set of 3 colors was enlarged to 4 with *Kung and Song 1977*, with their correctness proof of the incremental GC algorithm. In LambLisp, is is useful to have a fifth state, and to think of the state or color as a stage in the Cell life cycle. When GC is in progress, Cells may advance in their life cycle, but may also be moved back to an earlier as the result of an assignment. This enumeration allows some tests to be combined in an inequality rather than a sequence of equality tests or s C `switch`.

LambLisp∗ uses a **trampoline technique** to implement tail recursion. The **tail** of a series of expressions is the last one evaluated; this is the expression that will return the value of the series.

In the *trampoline*, instead of evaluating the last expression in the series and returning the value (as in C/C++), the expression is returned unevaluated, with the **tail flag** set. The evaluator checks every result to see if it is a *tail* that needs additional evaluation, or is a final result to be returned.

This removes the need for an additional stack frame during recursion.

- enum {
  gcst_idle , **gcst_issued** , **gcst_stacked** , **gcst_marked** ,
  gcst_free , **Ngcstates** }
- Int_t gc_state (void)
- Sexpr_t gc_state (Int_t st)
- Int_t tail_state (void)
- Sexpr_t tail_state_set (void)
- Sexpr_t tail_state_clr (void)

Cell type testing

The cell type enumeration has been designed to group together cell types according to their most common operations.

The main groups are those needing special sweep and finalizing during garbage collection, those that are treated as pairs, and simple atoms. There is a subgroup of those needing special sweep, that also need specialised marking during the garbage collection mark phase.

There are some types that have optimized subtypes (such as **immediate** types), and they differ in their processing at GC time. Type-testing functions below have `is_any_x()` predicates that group all subtypes together (e.g., `is_any_str_atom()` returns **true** for any type of string (heap, immediate, external). There are also functions further down that will return the contents of complex cells, and throw an error if called with incorrect type. Therefore it is often not necessary to use the predicates before accessing the atom internals with `any_x_get_info()`.

Note also: it is sometimes faster to check the types directly, rather than check the cell feature table. Preliminary testing indicates more than 2 type tests should use the table instead.

- typedef struct {
  Int_t **typ**
  bool **is_any_pair**
  bool **is_any_svec**
  bool **is_any_svec2n**
  bool **is_any_str**
  bool **is_any_sym**
  bool **is_any_bvec**
  const char ∗ **type_name**
  } **CellFeatures**

- static const CellFeatures **features** [Ntypes]
- void **init_static_data** ()
- Bool_t is_atom (void)
- Bool_t is_pair (void)
- Bool_t is_any_pair (void)
- Bool_t is_any_svec_atom ()
- Bool_t is_any_svec2n_atom ()
- Bool_t is_any_str_atom (void)
- Bool_t is_any_sym_atom (void)
- Bool_t is_any_bvec_atom (void)

2.1.1 Detailed Description

The Cell class is the foundational class for the LambLisp Virtual Machine.

Nearly all the Cell methods are *inline* for performance. A Cell has only getters and setters; there are no other side effects. Cell fields are changed only through explicit requests, and not as the result of any other mutations. This means that Cells do not participate in garbage collection, which is managed from outside the Cell class. Indeed, there is no concept within the Cell that there might be a plurality of them; only the behavior of a single Cell is defined.

A Cell may be one of several types. Historically, the number of types varied with the variant of Lisp. In LambLisp, there are Cell types that correspond directly to the types described in the *Scheme RxRS* specifications (integers, procedures etc), and there are additional types that implement underlying behavior to support the higher-level language (thunks, dictionaries).

For example, LambLisp supports several types of *strings* internally, depending on whether the character are stored on the heap, in externally-provided memory, or immediately within the cell (for fast operations on short strings). Each of these *string* types is compatible with the string type in the *Scheme RxRS* specifications. Likewise, bytevectors have external and immediate variants.

LambLisp also supports specialized *pair* types, such as the LambLisp *dictionary*. These act as *Lisp* pairs for purposes of allocation and garbage collection, but their specialized operators are executed by the underlying LambLisp virtual machine and therefore operate at C++ speed.

The Cell type enumeration is purposefully ordered in such a way as to allow efficient integer comparisons instead of a C switch in most cases, enhancing runtime performance as well as garbage collection.

Cell enumeration characteristics:
Cells requiring submarking
Cells requiring finalizing
Simple atoms: immediate cell types like bool char int real etc
External (non-GC) objects and pointers to native C++ code
Cells that point to C++-allocated objects, paired with optional GC finalizers.
"well-known singleton atoms" such as NIL, undefined, and void types.
Pair - the Cell type that responds to the Lisp *pair?* predicate.
Other pairs - specialized pair types

With this ordering of Cell types, the solution to many common cases during expression evaluation can be obtained with an inequality, rather than a C++ *switch* statement. This provides a significant performance boost, because the most common cases are checked first, and the total number of cases is reduced.

Cell optimized type tests
(Types < "simple atoms") need specialized GC marking and heap reclamation.
(Types <= "cells requiring finalizing") need specialized GC heap reclamation.
(Types >= T_NIL) are lists.
(Types > T_NIL) are pairs (but not only *Lisp* pairs, extended pairs too).
(Types != T_PAIR) are atoms, including numbers, strings, vectors, and singletons such as NIL.
(Types > T_PAIR) are extended pairs, used for specialized lists known to LambVM, but receiving regular *pair* GC processing.

There are cases where the ordering does not help so much. For example the LambLisp type system allows for several types of strings. Because the type system is ordered according to GC requirements, these are not adjacent in the enumeration and less amenable to inequality tests. For these cases, there is a cell feature matrix available to be queried by type and feature. This allows the determination to be made with an array lookup, which is faster than several sequential tests or a C *switch*, putting a permanent cap on the cost of a type check. It also allows other details such as `is-immediate?` to be implemented inexpensively.

The garbage collector states are ordered for similar reasons. See the garbage collector chapter for details on the Cell life cycle.

2.1.2 Member Enumeration Documentation

2.1.2.1 anonymous enum

`anonymous enum`

Enumerator

T_SVEC_HEAP	(Int_t Sexpr_t[]) A vector of Sexprs is a (len Sexpr_t*) pair.
T_SVEC2N_HEAP	(Int_t Sexpr_t[]) Same as vector, but length must be a power of 2; useful for hash tables.
T_ANY_HEAP_SVEC	Any types less than or equal to this are heap-stored vectors of S-expressions. They need specialized marking at GC time.
T_SYM_HEAP	(Int_t Charst_t) Symbol is a (hash char*) pair.
T_BVEC_HEAP	(Int_t Bytest_t) Bytevector is a (len byte*) pair.
T_STR_HEAP	(reserved Charst_t) The reserved field may be used in future to store the string length (not possible with immediate strings, so some details to be worked).
T_CPP_HEAP	(ptr-to-cpp-deleter ptr-to-cpp-object) Deleter is a function `void f(void *cpp↩ _obj)` of the appropriate type T of the C++ object, and performes `delete (T *) cpp_obj;`
T_PORT_HEAP	(reserved Ptr_t) car is unused, cdr is ptr to underlying C++ port instance
T_NEEDS_FINALIZING	Any types less than or equal to this have data stored in the heap, and need specialized finalizing at GC time.
T_BVEC_EXT	(Int_t Bytest_t) Same as T_BYTEVEC but externally allocated; the byte array is not freed at GC time.
T_STR_EXT	(reserved Charst_t) Same as T_STR but externally allocated; the character array is not freed at GC time.

T_BVEC_IMM	Special format: type and flag bytes as usual, byte 2 is vector length, remaining bytes are vector elements.
T_STR_IMM	Special format: type and flag bytes as usual, remaining bytes are 0-terminated string embedded in the cell.
T_GENSYM	Runtime symbol generation with no heap operations.
T_BOOL	(Bool_t reserved) Boolean atom
T_CHAR	(Char_t reserved) Character atom
T_INT	(Int_t reserved) Integer atom
T_REAL	(Special format: Real_t uses double word) Real number atom
T_RATIONAL	(Int_t Int_t) The fields are numerator and denominator of a rational number.
T_MOP3_PROC	(reserved ∗Mop3st_t) pointer to native function - args are evaluated before calling
T_MOP3_NPROC	(reserved ∗Mop3st_t) pointer to native macro processor - args are not evaluated before calling
T_VOID	(don't care) VOID has its own type and a singleton instance OBJ_VOID
T_UNDEF	(ERROR ERROR) UNDEF has its own type and a singleton instance OBJ_UNDEF
T_NIL	(ERROR ERROR) NIL has its own type and a singleton instance. • The NIL singleton is both a list and an atom, but not a pair. • Types <= T_NIL are atoms, types >= T_NIL are lists, types > T_NIL are pairs, but only T_PAIR responds to the Scheme *pair?* predicate.
T_PAIR	(Sexpr_t Sexpr_t) Normal untyped cons cell. All C++ types < T_PAIR are atoms. • Pair types. Types > NIL and types >= T_PAIR are pairs, but only T_PAIR responds to Scheme (pair?). • All pair types can be garbage collected without type-specific GC marking or finalizing.
T_SVEC_IMM	(Sexpr_t Sexpr_t) Vector of 0, 1 or 2 elements.
T_PROC	(Sexpr_t Sexpr_t) Procedure pair; car is lambda (formals + body containing free variables), cdr is environment.
T_NPROC	(Sexpr_t Sexpr_t) Non-evaluating procedure pair; car is nlambda (formals + body containing free variables), cdr is environment.
T_MACRO	(Sexpr_t Sexpr_t) Macro; car is transformer (proc of 1 arg) and cdr is env.
T_DICT	(Sexpr_t Sexpr_t) Dictionary pair; car is local frame, cdr is list of parent frames, used for environments & object instances.
T_THUNK_SEXPR	(Sexpr_t Sexpr_t) S-expression thunk pair; car is sexpr, cdr is environment with all variable bindings.
T_THUNK_BODY	(Sexpr_t Sexpr_t) Code body thunk pair; car is body (i.e., list of sexprs), cdr is environment with all variable bindings.
T_ERROR	(Sexpr_t Sexpr_t) An error cell; car is a T_STRING, cdr is a pointer to *irritants*.
Ntypes	Number of LambLisp virtual machine types.

2.1.3 Member Function Documentation

2.1.3.1 zero()

```
void Cell::zero ()  [inline]
```

Set all cell bits to zero.

2.1.3.2 type() [1/2]

```
Int_t Cell::type (
            void )  [inline]
```

Return the type of the cell as a small integer.

2.1.3.3 type() [2/2]

```
void Cell::type (
            Int_t t)  [inline]
```

Set the type of this cell.

2.1.3.4 flags()

```
Int_t Cell::flags (
            void )  [inline]
```

Return the entire set of cell flags.

2.1.3.5 flags_set()

```
void Cell::flags_set (
            Int_t f)  [inline]
```

Set the selected flags.

2.1.3.6 flags_clr()

```
void Cell::flags_clr (
            Int_t f)  [inline]
```

Clear the selected flags.

2.1.3.7 rplaca()

```
void Cell::rplaca (
            Word_t p)  [inline]
```

Replace the cell car field. Called rplaca for historical reasons, and to distinguish it from set-car!, which must respect the GC flags.

2.1.3.8 rplacd()

```
void Cell::rplacd (
            Word_t p)  [inline]
```

Replace the cell cdr field. Called rplacd for historical reasons, and to distinguish it from set-cdr!, which must respect the GC flags.

2.1.3.9 is_atom()

```
Bool_t Cell::is_atom (
            void )  [inline]
```

Return true if the cell is an atom.

2.1.3.10 is_pair()

```
Bool_t Cell::is_pair (
            void )  [inline]
```

Return true if the cell is a cons pair.

2.1.3.11 is_any_pair()

```
Bool_t Cell::is_any_pair (
            void )  [inline]
```

Return true if the cell is any pair type.

2.1.3.12 is_any_svec_atom()

```
Bool_t Cell::is_any_svec_atom ()  [inline]
```

Return true if the cell is any kind of Sexpr_t vector.

2.1.3.13 is_any_svec2n_atom()

```
Bool_t Cell::is_any_svec2n_atom ()  [inline]
```

Return true if the cell is Sexpr_t vector of size 2^n.

2.1.3.14 is_any_str_atom()

```
Bool_t Cell::is_any_str_atom (
            void )  [inline]
```

Return true if the cell is any kind of string.

2.1.3.15 is_any_sym_atom()

```
Bool_t Cell::is_any_sym_atom (
            void )  [inline]
```

Return true if the cell is any kind of symbol.

2.1.3.16 is_any_bvec_atom()

```
Bool_t Cell::is_any_bvec_atom (
            void )  [inline]
```

Return true if the cell is any kind of bytevector.

2.1.3.17 gc_state() [1/2]

```
Int_t Cell::gc_state (
            void )  [inline]
```

Return the garbage collection stat eof this cell.

2.1.3.18 gc_state() [2/2]

```
Sexpr_t Cell::gc_state (
            Int_t st)  [inline]
```

Set the garbage collection state of this cell, and return the cell.

2.1.3.19 tail_state()

```
Int_t Cell::tail_state (
            void )  [inline]
```

Return the taill state of this cell.

2.1.3.20 tail_state_set()

```
Sexpr_t Cell::tail_state_set (
            void )  [inline]
```

Set the tail state flag on this cell, and return the cell.

2.1.3.21 tail_state_clr()

```
Sexpr_t Cell::tail_state_clr (
            void )  [inline]
```

Clear the tail state flag on this cell, and return the cell.

2.1.3.22 get_car_addr()

```
Ptr_t Cell::get_car_addr ()  [inline]
```

Return the address of the cell car.

2.1.3.23 get_car()

```
Word_t Cell::get_car (
            void )  [inline]
```

Return the value of the cell car.

2.1.3.24 get_cdr()

```
Word_t Cell::get_cdr (
            void )  [inline]
```

Return the value of the cell cdr.

2.1.3.25 as_Bool_t()

```
Bool_t Cell::as_Bool_t ()  [inline]
```

Return the value of this cell as a boolean.

2.1.3.26 as_Char_t()

```
Char_t Cell::as_Char_t ()  [inline]
```

Return the value of this cell as a character.

2.1.3.27 as_Int_t()

```
Int_t Cell::as_Int_t ()  [inline]
```

Return the value of this cell as an integer.

2.1.3.28 as_Real_t()

```
Real_t Cell::as_Real_t ()  [inline]
```

Return the value of this cell as a real number.

2.1.3.29 as_Ptr_t()

```
Ptr_t Cell::as_Ptr_t ()  [inline]
```

Return the value of this cell as a generic pointer (i.e., void∗).

2.1.3.30 as_Charst_t()

```
Charst_t Cell::as_Charst_t ()  [inline]
```

Return the value of this cell as a pointer to a zero-terminated character array.

2.1.3.31 as_Bytest_t()

```
Bytest_t Cell::as_Bytest_t ()  [inline]
```

Return the value of this cell as a pointer to an array of bytes.

2.1.3.32 as_CharVec_t()

```
CharVec_t Cell::as_CharVec_t ()  [inline]
```

Return the value of this cell as a pointer to a zero-terminated character array.

2.1.3.33 as_ByteVec_t()

```
ByteVec_t Cell::as_ByteVec_t ()  [inline]
```

Return the value of this cell as a pointer to an array of bytes.

2.1.3.34 as_Portst_t()

```
Portst_t Cell::as_Portst_t ()  [inline]
```

Return the value of this cell as a pointer to an instance of the system underlying "port" implementation.

2.1.3.35 as_numerator()

```
Int_t Cell::as_numerator ()  [inline]
```

Return the value of this cell as the numerator of a rational number.

2.1.3.36 as_denominator()

```
Int_t Cell::as_denominator ()  [inline]
```

Return the value of this cell as the denominator of a rational number.

2.1.3.37 hash_sexpr()

```
Int_t Cell::hash_sexpr (
            void )
```

For symbols, the hash of its characters; for numbers, the hash of the number; otherwise the hash of the S-expression itself (i.e., the address of a cell).

2.1.3.38 hash_contents()

```
Int_t Cell::hash_contents (
            void )
```

For symbols, the hash of its characters; for numbers, strings, vectors, and bytevectors, the hash of the contents; otherwise the hash of the S-expression itself.

2.1.3.39 hash()

```
Int_t Cell::hash (
            void )  [inline]
```

Return the hash value of this cell.

2.1.3.40 set() [1/12]

```
Sexpr_t Cell::set (
            Int_t typ,
            Word_t w1,
            Word_t w2)  [inline]
```

This is the lowest-level generic "set" function.

2.1.3.41 set() [2/12]

```
Sexpr_t Cell::set (
            Int_t typ,
            Int_t a,
            Sexpr_t b)  [inline]
```

This is a convenience function for common cases.

2.1.3.42 set() [3/12]

```
Sexpr_t Cell::set (
            Int_t typ,
            Sexpr_t a,
            Sexpr_t b)  [inline]
```

This is a convenience function for common cases.

2.1.3.43 set() [4/12]

```
Sexpr_t Cell::set (
            Bool_t b)  [inline]
```

Set cell as boolean.

2.1.3.44 set() [5/12]

```
Sexpr_t Cell::set (
            Char_t c)  [inline]
```

Set cell as character.

2.1.3.45 set() [6/12]

```
Sexpr_t Cell::set (
            Int_t i)  [inline]
```

Set cell as integer.

2.1.3.46 set() [7/12]

```
Sexpr_t Cell::set (
            Real_t r)  [inline]
```

Set cell as real.

2.1.3.47 set() [8/12]

```
Sexpr_t Cell::set (
            Port_t & p)  [inline]
```

Set cell as port.

2.1.3.48 set() [9/12]

```
Sexpr_t Cell::set (
            Int_t typ,
            Int_t a,
            Charst_t b)  [inline]
```

Set cell as a immutable string.

2.1.3.49 set() [10/12]

```
Sexpr_t Cell::set (
            Int_t typ,
            Int_t a,
            Bytest_t b)  [inline]
```

Set cell as a immutable bytevector.

2.1.3.50 set() [11/12]

```
Sexpr_t Cell::set (
            Int_t typ,
            Int_t a,
            CharVec_t b)  [inline]
```

Set cell as a mutable string.

2.1.3.51 set() [12/12]

```
Sexpr_t Cell::set (
            Int_t typ,
            Int_t a,
            ByteVec_t b)  [inline]
```

Set cell as a mutable bytevector.

2.1.3.52 mk_error() [1/2]

```
Sexpr_t Cell::mk_error (
            const char * fmt,
             ...)
```

Fills and returns the single Cell-level T_ERROR object.

2.1.3.53 mk_error() [2/2]

```
Sexpr_t Cell::mk_error (
            Sexpr_t irritants,
            const char * fmt,
             ...)
```

Fills and returns the single Cell-level T_ERROR object.

2.1.3.54 prechecked_str_heap_get_chars()

```
CharVec_t Cell::prechecked_str_heap_get_chars ()  [inline]
```

Return a pointer to the character array in the heap.

2.1.3.55 prechecked_str_ext_get_chars()

```
CharVec_t Cell::prechecked_str_ext_get_chars ()  [inline]
```

Return a pointer to the character array located outside the heap.

2.1.3.56 prechecked_str_imm_get_chars()

```
CharVec_t Cell::prechecked_str_imm_get_chars ()  [inline]
```

Return a pointer to the character array embedded in this cell.

2.1.3.57 mustbe_Bool_t()

```
Bool_t Cell::mustbe_Bool_t ()  [inline]
```

Return the value of this cell as a boolean.

2.1.3.58 mustbe_Char_t()

```
Char_t Cell::mustbe_Char_t ()  [inline]
```

Return the value of this cell as a character.

2.1.3.59 mustbe_Int_t()

```
Int_t Cell::mustbe_Int_t ()  [inline]
```

Return the value of this cell as an integer.

2.1.3.60 mustbe_Real_t()

```
Real_t Cell::mustbe_Real_t ()  [inline]
```

Return the value of this cell as a real number.

2.1.3.61 mustbe_any_str_t()

```
Sexpr_t Cell::mustbe_any_str_t ()  [inline]
```

Return this cell if it is a string.

2.1.3.62 mustbe_cppobj_t()

```
Sexpr_t Cell::mustbe_cppobj_t ()  [inline]
```

Return this cell if it is a CPP object.

2.1.3.63 prechecked_cppobj_get_deleter()

```
CPPDeleterPtr Cell::prechecked_cppobj_get_deleter ()  [inline]
```

Return the function to be called at garbage collection time to recycle the C++ object.

2.1.3.64 prechecked_cppobj_get_ptr()

```
Ptr_t Cell::prechecked_cppobj_get_ptr ()  [inline]
```

Return a pointer to a C++ object obtained earlier.

2.1.3.65 any_str_get_chars()

```
CharVec_t Cell::any_str_get_chars ()  [inline]
```

If the cell is any kind of string, return a pointer to the zero-terminated character array.

2.1.3.66 cell_name()

```
String Cell::cell_name (
            void )
```

A convenience feature to produce a string name for cells which are "well known" like NIL. Otherwise the name is the hex representation of the cell address.

2.1.3.67 type_name()

```
Charst_t Cell::type_name (
            Int_t typ)
```

Return a pointer to the C string corresponding to the cell type.

2.1.3.68 gcstate_name()

```
Charst_t Cell::gcstate_name (
            void )
```

Return a pointer to a C string corresponding to the given GC state.

2.1.3.69 dump()

```
String Cell::dump ()
```

Return a printable representation of the Cell internals.

2.1.4 Member Data Documentation

2.1.4.1 F_GC01

```
const int Cell::F_GC01 = 0x01  [static]
```

gc flag

2.1.4.2 F_GC02

```
const int Cell::F_GC02 = 0x02  [static]
```

gc flag

2.1.4.3 F_GC04

```
const int Cell::F_GC04 = 0x04  [static]
```

gc flag

2.1.4.4 F_TAIL

```
const int Cell::F_TAIL = 0x08  [static]
```

trampoline tail marker

2.1.4.5 F_0x10

```
const int Cell::F_0x10 = 0x10  [static]
```

spare

2.1.4.6 F_0x20

```
const int Cell::F_0x20 = 0x20  [static]
```

spare

2.1.4.7 F_0x40

```
const int Cell::F_0x40 = 0x40  [static]
```

spare

2.1.4.8 F_0x80

```
const int Cell::F_0x80 = 0x80  [static]
```

spare

2.1.4.9 GC_STATE_MASK

```
const int Cell::GC_STATE_MASK = F_GC01 | F_GC02 | F_GC04  [static]
```

Mask for obtaining garbage collection state bits from Cell flag byte.

2.2 Lamb Class Reference

Public Types

- typedef Sexpr_t(∗ Mop3st_t) (Lamb &lamb, Sexpr_t sexpr, Sexpr_t env_exec)

Public Member Functions

- Sexpr_t load (Charst_t name, Sexpr_t env_exec, Int_t verbosity=0)

Arduino-compatible loop-based control interface.

- Sexpr_t setup (void)
- Sexpr_t loop (void)
- void end (void)

A few foundational functions for embedded debugging.

- void log (const char ∗fmt,...) CHECKPRINTF_pos2
- void printf (const char ∗fmt,...) CHECKPRINTF_pos2
- bool debug (void)
- void debug (bool onoff)

Information about the current build.

- bool build_isDebug ()
- unsigned long build_version ()
- unsigned long build_UTC ()
- unsigned long build_pushUTC ()
- const char ∗ build_buildRelease ()
- const char ∗ build_buildDate ()
- const char ∗ build_pushDate ()

Cell constructors

Garbage collection may happen at any time during the execution of the Cell constructors. The cell constructors that accept S-expression arguments will protect those arguments from GC during the operation, using the gc root push/pop functions.

- void expand ()

- Sexpr_t tcons (Int_t typ, Word_t a, Word_t b, Sexpr_t env_exec)
- Sexpr_t tcons (Int_t typ, Sexpr_t a, Sexpr_t b, Sexpr_t env_exec)
- Sexpr_t cons (Sexpr_t a, Sexpr_t b, Sexpr_t env_exec)
- Sexpr_t gensym (Sexpr_t env_exec)
- void gc_root_push (Sexpr_t p)
- void gc_root_pop (Int_t n=1)

A subset of the car/cdr accessors for list cells. This is the subset used within LambLisp.

- Sexpr_t **car** (Sexpr_t l)
- Sexpr_t **cdr** (Sexpr_t l)
- Sexpr_t **caar** (Sexpr_t l)
- Sexpr_t **cadr** (Sexpr_t l)
- Sexpr_t **cdar** (Sexpr_t l)
- Sexpr_t **cddr** (Sexpr_t l)
- Sexpr_t **caddr** (Sexpr_t l)
- Sexpr_t **cdddr** (Sexpr_t l)
- Sexpr_t **cadddr** (Sexpr_t l)
- Sexpr_t **cddddr** (Sexpr_t l)

The "bang" functions are the "mutators" of GC literature.

- void set_car_bang (Sexpr_t c, Sexpr_t val)
- void set_cdr_bang (Sexpr_t c, Sexpr_t val)
- void vector_set_bang (Sexpr_t vec, Int_t k, Sexpr_t val)
- Sexpr_t reverse_bang (Sexpr_t l)

Equivalence tests and sequential search

- Sexpr_t eq_q (Sexpr_t obj1, Sexpr_t obj2)
- Sexpr_t eqv_q (Sexpr_t obj1, Sexpr_t obj2)
- Sexpr_t equal_q (Sexpr_t obj1, Sexpr_t obj2)
- Sexpr_t assq (Sexpr_t obj, Sexpr_t alist)

Interned symbols

There are just a few operations on symbols:

- *Remember a new symbol.*
- *Check if symbol has already been seen.*
- *Compare 2 symbols for equality.*
- *At evaluation time, lookup the symbol in the current environment.*

For best performance, symbols in LambLisp are interned. This means a single copy of the symbol's characters are stored in a data structure containing the set of all interned symbols. Once interned, symbols can be tested for equality by address rather than character-by-character.

Traditionally, the data structure was called oblist if an association list was used, or called obarray if an array was used. Arrays allow for a hash table implementation, which greatly reduces search time, and can be further optimized if the array size is a power of 2 (2^n).

LambLisp uses the term oblist for the interned symbol table, implemented using a 2^n hash table. Symbol hashes are computed only once and are stored with the symbol, speeding runtime lookups.

- Sexpr_t **oblist_query** (Sexpr_t oblist, const char *identifier, Bool_t force, Sexpr_t env_exec)
- Sexpr_t **oblist_test** (Sexpr_t oblist, const char *identifier, Sexpr_t env_exec)
- Sexpr_t **oblist_intern** (Sexpr_t oblist, const char *identifier, Sexpr_t env_exec)
- Sexpr_t **oblist_analyze** (Sexpr_t oblist, Int_t verbosity, Sexpr_t env_exec)

Hierarchical Dictionary type

LambLisp's high-performance hierarchical dictionary implementation is used internally to represent the runtime environment. Dictionaries are also directly usable in the Lisp applications, and the dictionary data type is the basis for the LambLisp Object System (LOBS).

- Sexpr_t dict_new (Int_t framesize, Sexpr_t env_exec)
- Sexpr_t dict_new (Sexpr_t env_exec)
- Sexpr_t **dict_add_empty_frame** (Sexpr_t dict, Int_t framesize, Sexpr_t env_exec)
- Sexpr_t dict_add_empty_frame (Sexpr_t dict, Sexpr_t env_exec)
- Sexpr_t dict_add_keyval_frame (Sexpr_t dict, Sexpr_t keys, Sexpr_t vals, Sexpr_t env_exec)
- Sexpr_t dict_add_alist_frame (Sexpr_t dict, Sexpr_t alist, Sexpr_t env_exec)
- void dict_bind_bang (Sexpr_t dict, Sexpr_t key, Sexpr_t value, Sexpr_t env_exec)
- void dict_rebind_bang (Sexpr_t dict, Sexpr_t key, Sexpr_t value, Sexpr_t env_exec)
- void dict_bind_alist_bang (Sexpr_t dict, Sexpr_t alist, Sexpr_t env_exec)
- void dict_rebind_alist_bang (Sexpr_t dict, Sexpr_t alist, Sexpr_t env_exec)
- Sexpr_t dict_ref_q (Sexpr_t dict, Sexpr_t key)
- Sexpr_t dict_ref (Sexpr_t dict, Sexpr_t key)
- Sexpr_t dict_keys (Sexpr_t dict, Sexpr_t env_exec)
- Sexpr_t dict_values (Sexpr_t dict, Sexpr_t env_exec)
- Sexpr_t dict_to_alist (Sexpr_t dict, Sexpr_t env_exec)
- Sexpr_t dict_to_2list (Sexpr_t dict, Sexpr_t env_exec)
- Sexpr_t alist_to_dict (Sexpr_t alist, Sexpr_t env_exec)
- Sexpr_t twolist_to_dict (Sexpr_t twolist, Sexpr_t env_exec)
- Sexpr_t dict_analyze (Sexpr_t dict, Int_t verbosity=0)

Reading and writing

- Sexpr_t **read** (LL_Port &src, Sexpr_t env_exec)
- Sexpr_t **write_or_display** (Sexpr_t sexpr, Bool_t do_write)
- Sexpr_t **write_simple** (Sexpr_t sexpr)
- String **sprintf** (Charst_t fmt, Sexpr_t sexpr, Sexpr_t env_exec)
- Sexpr_t **printf** (Sexpr_t args, LL_Port &outp)
- Sexpr_t **printf** (Sexpr_t args)

Evaluation and function application

- Sexpr_t eval (Sexpr_t sexpr, Sexpr_t env_exec)
- Sexpr_t eval_list (Sexpr_t args, Sexpr_t env_exec)
- Sexpr_t apply_proc_partial (Sexpr_t proc, Sexpr_t sexpr, Sexpr_t env_exec)
- Sexpr_t map_proc (Sexpr_t proc, Sexpr_t lists, Sexpr_t env_exec)

Querying underlying system features defined by Scheme.

- Sexpr_t **r5_base_environment** ()
- Sexpr_t **r5_interaction_environment** ()
- Sexpr_t **lamb_oblist** ()
- Sexpr_t **current_input_port** ()
- Sexpr_t **current_output_port** ()
- Sexpr_t **current_error_port** ()

Useful list processing used internally by the LambLisp virtual machine.

- Sexpr_t **append** (Sexpr_t sexpr, Sexpr_t env_exec)
- Sexpr_t append2 (Sexpr_t lis, Sexpr_t obj, Sexpr_t env_exec)
- Sexpr_t **list_copy** (Sexpr_t sexpr, Sexpr_t env_exec)
- Sexpr_t list_analyze (Sexpr_t sexpr, Sexpr_t env_exec)
- Sexpr_t **list_to_vector** (Sexpr_t l, Sexpr_t env_exec)
- Sexpr_t **vector_to_list** (Sexpr_t v, Sexpr_t env_exec)
- Sexpr_t **vector_copy** (Sexpr_t from, Sexpr_t env_exec)

Sparse vectors

The sparse vector representation is a sorted association list, in which the car of each pair is the vector index and the cdr is the vector value at that index.

- Sexpr_t **vector_to_sparsevec** (Sexpr_t vec, Sexpr_t skip, Sexpr_t env_exec)
- Sexpr_t **sparsevec_to_vector** (Sexpr_t alist, Sexpr_t fill, Sexpr_t env_exec)

Makers for error types

These functions will sprintf a message into a buffer, and return an error object containing the message. They differ in that one (mk_error) creates a new error object, and the other (mk_syserror) avoids use of the memory manager by reusing the system error singleton.

- Sexpr_t mk_error (Sexpr_t env_exec, Sexpr_t irritants, const char ∗fmt,...) CHECKPRINTF_pos4
- Sexpr_t mk_error (Sexpr_t env_exec, const char ∗fmt,...) CHECKPRINTF_pos3
- Sexpr_t mk_syserror (const char ∗fmt,...) CHECKPRINTF_pos2
- Sexpr_t mk_syserror (Sexpr_t irritants, const char ∗fmt,...) CHECKPRINTF_pos3

Makers for simple compact types with embedded values, without external storage or dependencies

These types all have their values embedded in the Cell. Note that other types may also hold immediate values, but are "less simple" because they are C++ subtypes that are variations on the Lisp-declared types.

- Sexpr_t **mk_bool** (Bool_t b, Sexpr_t env_exec)
- Sexpr_t **mk_character** (Char_t ch, Sexpr_t env_exec)
- Sexpr_t **mk_integer** (Int_t i, Sexpr_t env_exec)
- Sexpr_t **mk_real** (Real_t r, Sexpr_t env_exec)
- Sexpr_t **mk_number** (Charst_t str, Sexpr_t env_exec)
- Sexpr_t **mk_sharp_const** (Charst_t name, Sexpr_t env_exec)

Makers for heap storage types

In principle, these types require space from the system heap. At GC time, they may also require specialized marking, and also specialized sweeping.

*As an optimization, LambLisp also implements **immediate** types. When the size required for the object is small, the internal bytes of the Cell can be used to hold the contents, avoiding heap operations. Using immediate types in this way has been common practice for integers and real numbers. LambLisp makes some additional use of immediate types. Immediate types are a LambLisp internal optimization, and they are all subtypes of the main Lisp type.*

The types that employ the immediate optimization are:

- *strings, providing up to 9 content bytes plus trailing 0 (32 bit word) or 22 content bytes (64 bit).*
- *bytevectors a count plus up to 9 or 22 content bytes.*
- *vectors of length 0, 1 or 2.*

*There are also **EXTERNAL** versions of those data types. These data types can be used to accesss data that should not be garbage collected, such as data contained in the application image, or C++ objects provided by another application.*

A variant of the EXTERNAL types is also provided, so that objects created in C++ can be passed to Lamb↩ Lisp, along with an optional deleter function. In C++, the deleter is best implemented as a one-line C++ lambda function accepting 1 argument (a void∗ pointer to a C++ object). The deleter function casts the void∗ pointer to match the C++ object type, and applies the C++ delete or delete[] operators to free the object space and any dependent resources.

By passing the C++ objects to LambLisp, they can be automatically garbage collected when LambLisp has finished with them.

- Sexpr_t **mk_string** (Sexpr_t env_exec, const char ∗fmt,...) CHECKPRINTF_pos3
- Sexpr_t **mk_string** (Int_t k, Charst_t src, Sexpr_t env_exec)
- Sexpr_t **mk_string** (String s, Sexpr_t env_exec)

- Sexpr_t **mk_symbol_or_number** (Charst_t str, Sexpr_t env_exec)
- Sexpr_t **mk_symbol** (Charst_t str, Sexpr_t env_exec)
- Sexpr_t mk_bytevector (Int_t k, Sexpr_t env_exec)
- Sexpr_t mk_bytevector (Int_t k, Bytest_t src, Sexpr_t env_exec)
- Sexpr_t mk_bytevector (Int_t k, Int_t fill, Sexpr_t env_exec)
- Sexpr_t mk_bytevector_ext (Int_t k, Bytest_t ext, Sexpr_t env_exec)
- Sexpr_t mk_intvector (Int_t k, Sexpr_t env_exec)
- Sexpr_t mk_realvector (Int_t k, Sexpr_t env_exec)
- Sexpr_t **mk_intvector** (Int_t k, Int_t fill, Sexpr_t env_exec)
- Sexpr_t **mk_realvector** (Int_t k, Real_t fill, Sexpr_t env_exec)
- Sexpr_t **mk_vector** (Int_t len, Sexpr_t fill, Sexpr_t env_exec)
- Sexpr_t **mk_hashtbl** (Int_t len, Sexpr_t fill, Sexpr_t env_exec)
- Sexpr_t **mk_serial_port** (Sexpr_t env_exec)
- Sexpr_t **mk_input_file_port** (Charst_t name, Sexpr_t env_exec)
- Sexpr_t **mk_output_file_port** (Charst_t name, Sexpr_t env_exec)
- Sexpr_t **mk_input_string_port** (Charst_t inp, Sexpr_t env_exec)
- Sexpr_t **mk_output_string_port** (Sexpr_t env_exec)

Makers for interface to native procedures and C++ objects.

- Sexpr_t **mk_Mop3_procst_t** (Mop3st_t f, Sexpr_t env_exec)
- Sexpr_t **mk_Mop3_nprocst_t** (Mop3st_t f, Sexpr_t env_exec)
- Sexpr_t **mk_cppobj** (void ∗obj, CPPDeleterPtr deleter, Sexpr_t env_exec)

Makers for pair types.

*These **pair** types all have the same structure for construction and garbage collection purposes, but are not Scheme pair types. LambLisp∗'s scalable type system presents opportunities for efficiency when implementing common operations on lists. These are executed by the LambLisp virtual machine at C++ speed, rather than at the slower speed of the Lisp evaluation loop.*

- *procedures ... the evaluator must be able to identify these directly.*
- *dictionaries ... often containing cycles when used as environments, so a depth limit is imposed when traversing.*
- *thunks ... used in combination with the trampoline technique to implement tail recursion.*

Note that there is no need for mk_pair(), it is just cons().

- Sexpr_t **mk_macro** (Sexpr_t nlam, Sexpr_t env_nlam, Sexpr_t env_exec)
- Sexpr_t **mk_procedure** (Sexpr_t formals, Sexpr_t body, Sexpr_t env_proc, Sexpr_t env_exec)
- Sexpr_t **mk_nprocedure** (Sexpr_t formals, Sexpr_t body, Sexpr_t env_nproc, Sexpr_t env_exec)
- Sexpr_t **mk_thunk_sexpr** (Sexpr_t sexpr, Sexpr_t env_thunk, Sexpr_t env_exec)
- Sexpr_t **mk_thunk_body** (Sexpr_t body, Sexpr_t env_thunk, Sexpr_t env_exec)
- Sexpr_t **mk_dict** (Sexpr_t frame, Sexpr_t base, Sexpr_t env_exec)
- Sexpr_t **mk_dict** (Int_t framesize, Sexpr_t env_exec)

Macro support

These macro primitives behave as described in Steele's *Common Lisp*.

- Sexpr_t **macroexpand** (Sexpr_t form, Sexpr_t env_exec)
- Sexpr_t **macroexpand1** (Sexpr_t form, Sexpr_t env_exec)
- Sexpr_t **macroexpand** (Sexpr_t proc, Sexpr_t args, Sexpr_t env_exec)
- Sexpr_t **macroexpand1** (Sexpr_t proc, Sexpr_t args, Sexpr_t env_exec)

2.2.1 Detailed Description

The Lamb class represents a single Lisp virtual machine. To closely match the Arduino-style of loop-based control, LambLisp provides begin(), loop(), and end().

To initialize the LambLisp VM, begin() should be called once before any other call. Generally LambLisp loop() will be called one time for every main loop(), but it is not necessary. If end() is called, all LambLisp resources are freed; Lamb::setup() can be called again to restart that LambLisp VM.

The LambLisp VM functions are grouped this way:

- Logging functions
- Build and version informational functions.
- Cell constructors, getters & setters
- Base data structures for the interpreter: alist, hash tables, and environments, along with their getters & setters.
- Printers
- Partial evaluation and application
- Port access from *Lisp*
- Vector, list, and dictionary utilities, used internally and also available in *Lisp*.
- GC interface functions to protect critical sections.
- Bindings
- Makers

2.2.2 Member Typedef Documentation

2.2.2.1 Mop3st_t

```
typedef Sexpr_t(* Lamb::Mop3st_t) (Lamb &lamb, Sexpr_t sexpr, Sexpr_t env_exec)
```

This is a pointer to a C++ native function that interacts directly with the S-expression in the given environment. Every LambLisp native function shares this signature. New external functions that conform to this signature can be used from within LambLisp and will run at full speed.

2.2.3 Member Function Documentation

2.2.3.1 setup()

```
Sexpr_t Lamb::setup (
            void )
```

Run once after base platform has started. In particular, *Serial* should be initialized before calling setup().

2.2.3.2 loop()

```
Sexpr_t Lamb::loop (
            void )
```

Run often to maintain control. The main responsibility of C++ Lamb::loop() is to call the LambLisp function of the same name (loop).

2.2.3.3 end()

```
void Lamb::end (
            void )
```

Release resources used by the Lamb virtual machine.

2.2.3.4 log()

```
void Lamb::log (
            const char * fmt,
             ...)
```

C-level printf feature with a limit on the length of strings produced. Takes care of log prompt.

2.2.3.5 printf()

```
void Lamb::printf (
            const char * fmt,
             ...)
```

C-level printf feature with a limit on the length of strings produced.

2.2.3.6 debug() [1/2]

```
bool Lamb::debug (
            void )
```

Return the state of the Lamb internal debug flag.

2.2.3.7 debug() [2/2]

```
void Lamb::debug (
            bool onoff)
```

Set the state of the Lamb debug flag.

2.2.3.8 build_isDebug()

```
bool Lamb::build_isDebug ()
```

Return true if this build is not checked in. This is unrelated to the runtime debug flag.

2.2.3.9 build_version()

```
unsigned long Lamb::build_version ()
```

Return the build version as a long integer (implemented as a UTC time stamp).

2.2.3.10 build_UTC()

```
unsigned long Lamb::build_UTC ()
```

Return the UTC time of this build.

2.2.3.11 build_pushUTC()

```
unsigned long Lamb::build_pushUTC ()
```

Return the UTC time this repo was last pushed.

2.2.3.12 build_buildRelease()

```
const char * Lamb::build_buildRelease ()
```

Return a pointer to a character array containing the release description.

2.2.3.13 build_buildDate()

```
const char * Lamb::build_buildDate ()
```

Return a pointer to a character array containing the build date.

2.2.3.14 build_pushDate()

```
const char * Lamb::build_pushDate ()
```

Return a pointer to a character array containing the date this repo was last pushed.

2.2.3.15 expand()

```
void Lamb::expand ()
```

Add another block of cells to the population.

2.2.3.16 tcons() [1/2]

```
Sexpr_t Lamb::tcons (
            Int_t typ,
            Word_t a,
            Word_t b,
            Sexpr_t env_exec)
```

Generic cell constructor with no GC protection for its arguments.

2.2.3.17 tcons() [2/2]

```
Sexpr_t Lamb::tcons (
            Int_t typ,
            Sexpr_t a,
            Sexpr_t b,
            Sexpr_t env_exec)
```

Constructor for any pair type; protects both S-expression arguments.

2.2.3.18 cons()

```
Sexpr_t Lamb::cons (
            Sexpr_t a,
            Sexpr_t b,
            Sexpr_t env_exec)  [inline]
```

Cell constructor for normal pairs.

2.2.3.19 gensym()

```
Sexpr_t Lamb::gensym (
            Sexpr_t env_exec)
```

Produce a new unique symbol.

2.2.3.20 gc_root_push()

```
void Lamb::gc_root_push (
            Sexpr_t p)
```

Preserve the cell given from GC, until popped.

2.2.3.21 gc_root_pop()

```
void Lamb::gc_root_pop (
            Int_t n = 1)
```

Release the preserved cells for normal GC processing.

2.2.3.22 set_car_bang()

```
void Lamb::set_car_bang (
            Sexpr_t c,
            Sexpr_t val)
```

Replace the car field in the cell with *val*. GC flags will be maintained as required.

2.2.3.23 set_cdr_bang()

```
void Lamb::set_cdr_bang (
            Sexpr_t c,
            Sexpr_t val)
```

Replace the cdr field in the cell with *val*. GC flags will be maintained as required.

2.2.3.24 vector_set_bang()

```
void Lamb::vector_set_bang (
            Sexpr_t vec,
            Int_t k,
            Sexpr_t val)
```

Replace the specified vector element with *val*. GC flags will be maintained as required.

2.2.3.25 reverse_bang()

```
Sexpr_t Lamb::reverse_bang (
            Sexpr_t l)
```

Reverse the list in-place and return the new list head (the former list tail).

2.2.3.26 eq_q()

```
Sexpr_t Lamb::eq_q (
            Sexpr_t obj1,
            Sexpr_t obj2)
```

Return true if 2 cells are the same cell, or are atoms with the same value.

2.2.3.27 eqv_q()

```
Sexpr_t Lamb::eqv_q (
            Sexpr_t obj1,
            Sexpr_t obj2)
```

Return true if 1 cells are the same cell, or are atoms with the same value.

2.2.3.28 equal_q()

```
Sexpr_t Lamb::equal_q (
            Sexpr_t obj1,
            Sexpr_t obj2)
```

Returns true when obj1 and obj2 are eqv?, and also all their descendants.

2.2.3.29 assq()

```
Sexpr_t Lamb::assq (
            Sexpr_t obj,
            Sexpr_t alist)
```

Search an association list for a matching key, and return the `(key . value)` pair, or **false** if not found.

2.2.3.30 dict_new() [1/2]

```
Sexpr_t Lamb::dict_new (
            Int_t framesize,
            Sexpr_t env_exec)  [inline]
```

Return a new empty dictionary with the given top frame size.

2.2.3.31 dict_new() [2/2]

```
Sexpr_t Lamb::dict_new (
            Sexpr_t env_exec)  [inline]
```

Return a new empty dictionary with alist top frame.

2.2.3.32 dict_add_empty_frame()

```
Sexpr_t Lamb::dict_add_empty_frame (
            Sexpr_t dict,
            Sexpr_t env_exec)  [inline]
```

Returns a new dictionary with an empty top frame, and an existing dictionary as parent.

2.2.3.33 dict_add_keyval_frame()

```
Sexpr_t Lamb::dict_add_keyval_frame (
            Sexpr_t dict,
            Sexpr_t keys,
            Sexpr_t vals,
            Sexpr_t env_exec)
```

Returns a new dictionary with a new top frame containing the keys bound to the values.

2.2.3.34 dict_add_alist_frame()

```
Sexpr_t Lamb::dict_add_alist_frame (
             Sexpr_t dict,
             Sexpr_t alist,
             Sexpr_t env_exec)  [inline]
```

Returns a new dictionary with the alist bindings added in a new frame on top of the base dictionary.

2.2.3.35 dict_bind_bang()

```
void Lamb::dict_bind_bang (
             Sexpr_t dict,
             Sexpr_t key,
             Sexpr_t value,
             Sexpr_t env_exec)
```

Modify the target dictionary; `value` is re-assigned to `key`if found, else created in the top frame.

2.2.3.36 dict_rebind_bang()

```
void Lamb::dict_rebind_bang (
             Sexpr_t dict,
             Sexpr_t key,
             Sexpr_t value,
             Sexpr_t env_exec)
```

Modify the target dictionary; `value` is assigned to `key` wherever first found in env, else error if not found.

2.2.3.37 dict_bind_alist_bang()

```
void Lamb::dict_bind_alist_bang (
             Sexpr_t dict,
             Sexpr_t alist,
             Sexpr_t env_exec)
```

Modify the target dictionary; binding keys in the alist to their corresponding values.

2.2.3.38 dict_rebind_alist_bang()

```
void Lamb::dict_rebind_alist_bang (
             Sexpr_t dict,
             Sexpr_t alist,
             Sexpr_t env_exec)
```

Modify the target dictionary; rebinding keys in the alist to their corresponding values.

2.2.3.39 dict_ref_q()

```
Sexpr_t Lamb::dict_ref_q (
            Sexpr_t dict,
            Sexpr_t key)
```

Returns (key value) pair, or **false** if key is unbound.

2.2.3.40 dict_ref()

```
Sexpr_t Lamb::dict_ref (
            Sexpr_t dict,
            Sexpr_t key)
```

Returns the value associated with the key in the given dictionary; throws error if key is unbound.

2.2.3.41 dict_keys()

```
Sexpr_t Lamb::dict_keys (
            Sexpr_t dict,
            Sexpr_t env_exec)
```

Note that keys and values are guaranteed to return in the corresponding order in dict_keys and dict_values, duplicate keys may occur.

Return a list of all the keys in this dictionary. If the dictionary has parents, keys may appear multiple times.

2.2.3.42 dict_values()

```
Sexpr_t Lamb::dict_values (
            Sexpr_t dict,
            Sexpr_t env_exec)
```

Return a list of all the values in this dictionary. If the dictionary has parents, values may appear multiple times for each key.

2.2.3.43 dict_to_alist()

```
Sexpr_t Lamb::dict_to_alist (
            Sexpr_t dict,
            Sexpr_t env_exec)
```

Convert the dictionary to an alist, retaining only the top-level `(key . value)` pairs.

2.2.3.44 dict_to_2list()

```
Sexpr_t Lamb::dict_to_2list (
            Sexpr_t dict,
            Sexpr_t env_exec)
```

Convert the dictionary to a list of 2-element lists, retaining only the top-level `(key value)` sublists.

2.2.3.45 alist_to_dict()

```
Sexpr_t Lamb::alist_to_dict (
            Sexpr_t alist,
            Sexpr_t env_exec)
```

Convert an alist into a dictionary. The resulting dictionary has no parent.

2.2.3.46 twolist_to_dict()

```
Sexpr_t Lamb::twolist_to_dict (
            Sexpr_t twolist,
            Sexpr_t env_exec)
```

Convert a list of 2-element lists into a dictionary. The resulting dictionary has no parent.

2.2.3.47 dict_analyze()

```
Sexpr_t Lamb::dict_analyze (
            Sexpr_t dict,
            Int_t verbosity = 0)
```

Internal integrity check.

2.2.3.48 eval()

```
Sexpr_t Lamb::eval (
            Sexpr_t sexpr,
            Sexpr_t env_exec)
```

Evaluate the S-expression in the environment provided.

2.2.3.49 eval_list()

```
Sexpr_t Lamb::eval_list (
            Sexpr_t args,
            Sexpr_t env_exec)
```

Evaluate the list of S-expressions in the environment provided, and return a lkist of results.

2.2.3.50 apply_proc_partial()

```
Sexpr_t Lamb::apply_proc_partial (
            Sexpr_t proc,
            Sexpr_t sexpr,
            Sexpr_t env_exec)
```

Evaluate the function arguments and then apply the function to them. The result may be a tail requiring further evaluation.

2.2.3.51 map_proc()

```
Sexpr_t Lamb::map_proc (
            Sexpr_t proc,
            Sexpr_t lists,
            Sexpr_t env_exec)
```

Apply the procecure to the lists per R5RS.

2.2.3.52 load()

```
Sexpr_t Lamb::load (
            Charst_t name,
            Sexpr_t env_exec,
            Int_t verbosity = 0)
```

Load and evaluate S-expressions from the named file.

2.2.3.53 append2()

```
Sexpr_t Lamb::append2 (
            Sexpr_t lis,
            Sexpr_t obj,
            Sexpr_t env_exec)
```

Destructively attach *obj* to the end of *lis*, which must be a proper list.

2.2.3.54 list_analyze()

```
Sexpr_t Lamb::list_analyze (
            Sexpr_t sexpr,
            Sexpr_t env_exec)
```

Traverse the list and return either false (if circular), the list length (if a proper list), ###.

2.2.3.55 mk_error() [1/2]

```
Sexpr_t Lamb::mk_error (
            Sexpr_t env_exec,
            Sexpr_t irritants,
            const char * fmt,
             ...)
```

Fill a new error object with the information given, and return it.

2.2.3.56 mk_error() [2/2]

```
Sexpr_t Lamb::mk_error (
            Sexpr_t env_exec,
            const char * fmt,
             ...)
```

Fill a new error object with the information given, and return it. The *irritants* field is left NIL.

2.2.3.57 mk_syserror() [1/2]

```
Sexpr_t Lamb::mk_syserror (
            const char * fmt,
             ...)
```

Fill the system error object with the info given, and return it.

2.2.3.58 mk_syserror() [2/2]

```
Sexpr_t Lamb::mk_syserror (
            Sexpr_t irritants,
            const char * fmt,
             ...)
```

Fill the system error object with the info given, and return it.

2.2.3.59 mk_bytevector() [1/3]

```
Sexpr_t Lamb::mk_bytevector (
            Int_t k,
            Sexpr_t env_exec)
```

Simplest heap allocation with no initialization.

2.2.3.60 mk_bytevector() [2/3]

```
Sexpr_t Lamb::mk_bytevector (
            Int_t k,
            Bytest_t src,
            Sexpr_t env_exec)
```

Heap allocation with initialization.

2.2.3.61 mk_bytevector() [3/3]

```
Sexpr_t Lamb::mk_bytevector (
            Int_t k,
            Int_t fill,
            Sexpr_t env_exec)
```

Heap allocation with initialization.

2.2.3.62 mk_bytevector_ext()

```
Sexpr_t Lamb::mk_bytevector_ext (
            Int_t k,
            Bytest_t ext,
            Sexpr_t env_exec)  [inline]
```

Injection of externally allocated memory, which will not be freed at GC time.

2.2.3.63 mk_intvector()

```
Sexpr_t Lamb::mk_intvector (
            Int_t k,
            Sexpr_t env_exec)
```

Note that integer vectors and real vectors are just bytevectors underneath.

Allocates a bytevector from the heap to ensure proper alignment (no IMM type).

2.2.3.64 mk_realvector()

```
Sexpr_t Lamb::mk_realvector (
            Int_t k,
            Sexpr_t env_exec)
```

Allocates a bytevector from the heap to ensure proper alignment (no IMM type).

2.3 LambStdioClass Class Reference

Public Member Functions

- int **setTxBufferSize** (int n)
- int **setRxBufferSize** (int n)
- void **begin** (void)
- void **begin** (unsigned long baudrate)
- void **end** ()
- int **available** (void)
- int **availableForWrite** (void)
- int **read** (void)
- int **write** (uint8_t c)
- int **write** (char c)
- void **flush** (void)
- int **read** (byte ∗buf, int max)
- int **read** (char ∗s, int max)
- int **write** (const char ∗s, int n)
- int **write** (const byte ∗b, size_t n)
- int **write** (const char ∗s)
- **operator bool** ()

2.3.1 Detailed Description

A wrapper around the underlying stdin/stdout implementation. On an embedded system, this class will use the primary serial in/out (`Serial` on Arduino-compatibles). On Linux, this class will set the terminal to byte-at-a-time mode (called "non-CANONICAL" mode).

2.4 LL_File Class Reference

Public Member Functions

- bool **isOpen** ()
- int **read** (void)
- int **write** (byte b)
- int **seek** (unsigned long target, int whence=SEEK_SET)
- int **tell** ()
- int **size** ()
- int **close** ()
- int **read** (byte ∗b, int n)
- int **read** (char ∗s, int n)
- int **write** (const byte ∗b, int n)
- int **write** (const char ∗s, int n)
- int **peek** ()

Public Attributes

- File_Native **_theFile**
- String **_path**
- String **_mode**

2.4.1 Detailed Description

The *file* type is ultimately provided by the underlying operating system, not by LambLisp. This class elides the differences between file types on different platforms, providing a POSIX-like interface.

Note that there is no *open* operstion on files. A file is opened by the *file system* and then a *file* is returned. After a *file* is closed, the same file object cannot be opened again; instead a new file must be requested from the *file system*.

2.5 LL_File_System Class Reference

Public Member Functions

- LL_File ∗ **open** (const char ∗path, const char ∗mode)
- int **rm** (const char ∗path)
- int **mv** (const char ∗from, const char ∗to)

2.5.1 Detailed Description

The "file system" type elides the differences between different underlying platforms. For example, it will deal with the leading '/' required by SPIFFS. This minimal file system interface is platform-independent.

Index