Z3S5 Lisp User Manual

by Erich Rast

2022-8-26 18:30

for Z3S5 Lisp Version “2.3.10+6cddbd5”

1 Introduction

Z3S5 Lisp can be used as a standalone interpreter or as an extension language embedded into Go programs. It is a traditional Lisp-1 dialect, where the suffix 1 means that symbols hold one value and this value might either represent functions and closures or data. This is in contrast to Lisp-2 language like CommonLisp in which symbols may hold functions in a separate slot. Scheme dialects are also Lisp-1 and Z3S5 Lisp shares many similarities with Scheme while also having features of traditional Lisp systems.

1.1 Invoking Lisp

1.1.1 The Standalone Interpreter

In the directory `cmd/z3` there is an example standalone version `z3.go` that you can build on your system using `go build z3.go`. The interpreter is started in a terminal using `./z3`. The `z3` interpreter is fairly limited. It starts a read-eval-reply loop until it is quit with the command `(exit [n])` where the optional number `n` is an integer for the Unix return code of the program. It reads one line of input from the command line and returns the result of evaluating it. Better editing capabilities and parenthesis matching are planned for the future.

When an interpreter starts either in a standalone executable or when the `interp.Boot()` function is called in a Go program, then it first loads the standard prelude and help files in directory `embed`. These are embedded into the executable and so the directory is not needed to run the interpreter. After this start sequence, the interpreter checks whether there is a file named `init.lisp` in the executable directory – that is, the `z3` directory for standalone or the directory of the program that includes Z3S5 Lisp as a package. If there is such a file, then it is loaded and executed.

The `z3` interpreter also has a number of command-line options. If option `-l <filename>` is provided, then Z3S5 Lisp sets `*interactive-session*` to `nil` (which usually suppresses the start banner), loads and executes the specified file, and returns to the shell afterwards. The flag `-e` does the same but executes the expressions directly provided as string on the command-line. Special characters in the provided expressions have to be suitably escaped not to be interpreted by the shell, of course, and how to do this depends on the type of shell used to run the interpreter.

If in addition to the `-e` or `-l` flags the `-i` option is provided, then `*interactive-session*` is set to true right from the start, the file is loaded and executed, and then an interactive session is started

as if the interpreter had been started without any command-line options. By using the `-s` flag, `*interactive-session*` can be set to `nil` regardless of how the interpreter is started, and therefore printing a start banner is suppressed (and anything else that requires `*interactive-session*` to be non-nil.)

Example 1: `./z3 -e "(out 'hello-world)(nl)"-i -s` starts the interpreter, does not print a start banner since `-s` is specified, prints *hello-world* and new line, and then enters an interactive session since `-i` is specified.

Example 2: `./z3 -l /home/user/tests/test.lisp -s` starts the interpreter, loads and executes the file in directory `/home/tests/test.lisp` (an absolute Unix path) while suppressing the start-banner with the `-s` option, and returns to the shell.

Example 3: `./z3 -i -l my_prelude.lisp` launches the interpreter and loads the file `my_prelude.lisp` in the same directory as the interpreter and then starts an interactive session

See `./z3 -h` for a list of all command-line options.

1.1.2 Using the Interpreter in Go

See `cmd/z3/z3.go` for an example of how to use Z3S5 Lisp in Go. It is best to include the package with `z3 "github.com/rasteric/z3s5-lisp"` so `z3` can be used as the package shortcut, since this has to be called often when writing extensions. The following snippet from `z3.go` imports the Lisp interpreter, boots the standard prelude, and runs an interactive read-eval-reply loop:

```
import (
    "fmt"
    "os"

    z3 "github.com/rasteric/z3s5-lisp"
)

func main() {
    interp, err := z3.NewInterp(z3.NewBasicRuntime(z3.FullPermissions))
    if err != nil {
        fmt.Fprintf(os.Stderr, "Z3S5 Lisp failed to start: %v\n", err)
        os.Exit(1)
    }
    err = interp.Boot()
    if err != nil {
        fmt.Fprintf(os.Stderr, "Z3S5 Lisp failed to boot the standard
           prelude: %v\n", err)
        os.Exit(2)
    }
    interp.SafeEval(&z3.Cell{Car: z3.NewSym("protect-toplevel-symbols")
        , Cdr: z3.Nil}, z3.Nil)
```

```
20      interp.Run(nil)
21  }
```

What deserves explanation is the line starting with `interp.SafeEval`. This creates a list with the symbol `protect-toplevel-symbols` as car and an empty cdr and executes it within an empty environment. So it runs the equivalent of `(protect-toplevel-symbols)`, which is interpreted as a function call that applies `protect` to all toplevel symbols defined thus far. It is not put in the standard prelude because a common way to extend Z3S5 Lisp is to simply redefine some of its primitives. Once a symbol is protected, attempts of redefining or mutating it result in a security violation error. As long as the `Permissions` given to `NewBasicRuntime` have `AllowUnprotect` set to true, you may use `unprotect` to remove this safeguard again and redefine existing functions. Symbols are protected in the `z3` interpreter because it is easy to mess up the system with dynamic redefinitions. For example, you could redefine `(setq + -)` and + is suddenly interpreted as subtraction! However, redefining toplevel symbols is a very important feature. By using `unbind` on certain functions or setting them to something else like `(setq func (lambda ()(error "not allowed to use func!")))` it is possible to create fine-grained access control to functions even when they are intrinsic or defined in the standard prelude. It is not possible to access an intrinsic function after it has been defined away in this manner. After you added all your toplevel symbols and removed or redefined the ones you do not like, you may thus protect all symbols and use `set-permissions` to revoke the privilege `AllowUnprotect`. This fixes the base vocabulary and any attempt to redefine it in the future will result in a security violation, since permissions can only be changed from less secure to more secure and never vice versa.

See the Chapter Extending Z3S5 Lisp for information about how to define your own functions in Z3S5 Lisp.

1.2 Data Types

Basic data types of Z3S5 Lisp are:

- Bools: `t` and `nil`. The empty list `nil` is interpreted as false and any non-nil value is true; the symbol `t` is predefined as non-nil and usually used for true.
- Symbols: `abracadabra`, `foo`, `bar`, `...hel%lo*_`. These have few restrictions, as the last example illustrates.
- Integers: `3`, `1`, `-28`. These are bignums with functions such as `+`, `-`, `sub`, `div`,...
- Floats: `3.14`, `1.92829` with a large range of functions with prefix `fl.`. Get a list with `(dump 'fl.)`
- UTF-8 Strings: `"This is a test."` with functions such as `str+`, `str-empty?`, `str-forall?`, `str-join`,... Get an overview with `(dump 'str)`.

- Lists: `'(a b c d e f)` with functions such as `car`, `cdr`, `mapcar`, `1st`, `2nd`, `member`, `filter` ,...
- Arrays: `#(a b c d e f)` with functions such as `array-ref`, `array-set`, `array-len`, ...
- Dictionaries: `(dict)` with functions such as `get`, `set`, `dict->alist`, `dict->array`, `dict->values`, `dict-map`,...
- Blobs: These contain binary data with functions like `peek`, `poke`, `blob->base64`,...
- Boxed values with functions like `valid?`. Boxed values are used to embed foreign types like Go structures into the runtime system which cannot be automatically garbage collected. They typically require manual destruction to free memory. See the source code for `lisp_decimal.go` for an example of such an embedding.
- Futures: Futures encapsulate the result of a future computation and are returned by the special form `(future ...)` whose body evaluates to a future. The result of a future computation may be obtained with `(force <future>)`.
- Tasks: Tasks are heavyweight concurrency constructs similar to threads in other programming languages. See `(dump 'task)` for a list of functions.

Strings, lists, and arrays are also sequences testable with the `seq?` predicate. These support convenience successor functions like `map` and `foreach`. Check the `init.lisp` preamble in directory `embed/` for how these work.

Numbers are generally one bignum type for which `num?` is true and they will extend and shrink in their representation as necessary. However, numbers larger than 64 bit only support basic arithmetics and floating point arithmetics is limited to values that fit into a Float64. There is also a built-in decimal arithmetics extension with prefix `dec` for correct accounting and banking arithmetics and rounding – see `(dump 'dec.)` for a list of available decimal arithmetics functions and `lisp_decimal.go` for implementation details.

2 Writing Programs

The `z3` interpreter loads and executes any file `init.lisp` in the same directory as the executable at startup. It is also possible to use the command-line option `-l <filename>` to load and execute it. This sets `*interactive-session*` to `nil`, executes the contents of the file and returns to the shell. This behavior can be overridden by specifying the `-i` flag on the command-line, which sets '*interactive-session* to true and ensures that an interactive session is started after the initial files are loaded.

2.1 Finding Functions and Online Help

Built-in help on functions can be obtained with `(help func)` where the function name is not quoted. For example, `(help help)` returns the help entry for the `help` function. A list of bound toplevel symbols can be obtained with `(dump [sym])` where the optional symbol `sym` must be a quoted prefix. For example, `(dump)` returns all toplevel bindings and `(dump 'str)` returns a list of all bound symbols starting with "str". By convention, internal helper functions without help entry are prefixed with an underscore and omitted by `dump`. To get a list of all bindings, including those starting with an underscore, use `(dump-bindings)`.

2.2 Peculiarities of Z3S5 Lisp

- Iterators over sequences use the order `(iterator sequence function)`, not sometimes one and sometimes the other. Exception: `memq` and `member` ask whether an element is a member of a list and have order `(member element list)`.
- There is no meaningful `eq`, equality is generally tested with `equal?`.
- Predicate names usually end in a question mark like in Scheme dialects, with a few exceptions like = for numeric equality. Example: `equal?`. There is no p-suffix like in other Lisps.
- `dict` data structures are multi-threading safe.
- There is support for futures and concurrent tasks.

2.3 Basic Control Flow and Examples

When Z3S5 Lisp encounters an unquoted list, it attempts to interpret the first element of the list as a function call. Thus, if the symbol is bound to a function or macro, it calls the function with the remainder of the list as argument:

```
1 > (+ 10 20 30 40 50)
2 150
```

Symbols evaluate to their values and need to be quoted otherwise. They can be defined with `setq`:

```
1 > hello
2 EvalError: void variable: hello
3     hello
4 > (setq hello 'world)
5 world
6 > hello
7 world
```

To define a function, the macro `defun` can be used as in other Lisp dialects. The syntax is the traditional one, not in Scheme dialects:

```
> (defun fib (n)
    (if (or (= n 0) (= n 1))
        1
        (+ (fib (- n 1))
           (fib (- n 2)))))
fib
> (fib 10)
89
> (fib 30)
1346269
```

`(defun foo (bar)...)` is really just a macro shortcut for `(setq foo (lambda (bar)...))` like in most other Lisp dialects. Macros are expanded before a program is executed, which unfortunately makes error reporting in Z3S5 Lisp sometimes a bit obtuse. The system neither keeps track of source locations nor of original definitions and will present the expanded macros when an error occurs. This will not change in the future, so better get used to it! Changing this and turning Z3S5 Lisp into a real, full-fledged Lisp would be a gigantic task and not worth the effort, as there are already far more capable Lisp systems like `CommonLisp` and `Racket` out there.

The above function demonstrates recursion, a feature that is often used in Lisp. Z3S5 Lisp eliminates tail recursion. The example also illustrates the use of = for numeric equality, + and − functions on (potential) bignums and the macro `(if <condition> <then-clause> <else-clause>)`. The more general `cond` construct is also available and in fact the primitive operation:

```
>  (defun day (n)
     (cond
       ((= n 0) 'sunday)
       ((= n 1) 'monday)
       ((= n 2) 'tuesday)
       (t 'late-in-the-week)))
day
> (day 1)
monday
> (day 0)
sunday
> (day -1)
late-in-the-week
> (day 5)
late-in-the-week
```

Looks like some programmer was lazy there.

In Lisp languages it is common to iterate over lists and other data structures either using recursive functions or by using some of the standard iteration constructs such as `map`, `foreach`, `mapcar`, and

so on. Nesting calls of macros or functions like `map` and `filter` to achieve the desired data transformation is very common functional programming style and often desirable for performance. Here is an example:

```
> (map '(1 2 3 4 5 6) add1)
(2 3 4 5 6 7)
```

Local variables are bound with `let` or with `letrec` in Z3S5 Lisp. The latter needs to be used whenever a variable in the form depends on the other variable, which is not possible with `let` since it makes the bindings only available in the body. Here is the definition of `apropos` from the preamble:

```
(defun apropos (arg)
  (let ((info (get *help* arg nil)))
    (if info
      (cadr (assoc 'see info))
      nil)))
```

The variable `info` is bound to the result of `(get *help* arg nil)`, which evaluates to default `nil` if there is no help entry for `arg` in the global dictionary `*help*`. That avoids calling `(get *help* arg nil)` twice, once for the check to nil and once in the first **`if`** branch. But it can often be better to avoid uses of `let` and re-use accessors; it depends a bit on clarity and intent, and whether the initial computation is costly or not. Here is an example where `let` would not do and instead `letrec` has to be used:

```
> (letrec ((is-even? (lambda (n)
                       (or (= n 0)
                           (is-odd? (sub1 n)))))
           (is-odd? (lambda (n)
                      (and (not (= n 0))
                           (is-even? (sub1 n))))))
    (is-odd? 11))
t
```

The reason this doesn't work with `let` is that `is-even?` is not bound in the definition of `is-odd?` and vice versa when `let` is used, whereas `letrec` makes sure that these bindings are mutually available. Although the current implementation always uses `letrec` under the hood, it is best to use `let` whenever it is possible since at least in theory it could be implemented more efficiently. Notice that `setq` can be used to mutate local bindings, of course, and is not just intended for toplevel symbols. Although beginners should avoid it, sometimes mutating variables with `setq` can make definitions simpler at the cost of some elegance.

3 Advanced Topics

3.1 Error Handling

There are no continuations and there is no fancy stuff like dynamic wind. Instead, there are primitives like `push-error-handler` and `pop-error-handler` and macros such as `with-final` and `with-error-handler`.

3.2 Debugging

Not only are error messages fairly rudimentary, they also use the expanded macro definitions and are therefore hard to read. Local bindings are implemented with lambda-terms and displayed as such. This makes debugging a challenge, as it should be. Z3S5 programmers have the habit of writing bug-free code from the start and thereby avoid debugging entirely. But you could write your own trace or stepper functions for enhanced debugging features by redefining all toplevel symbols appropriately. Then again, you could also just write your own Lisp with better debugging capabilities. The choice is up to you!

3.3 Concurrency

Dicts use Go's `sync.Map` under the hood and are therefore concurrency-safe. The global symbol table is also cuncurrency-safe. This means that dicts can be accessed safely from futures and tasks and can even be used for synchronization purposes, as the (admittedly horrible) current implementation of tasks in `embed/init.lisp` illustrates. Generally, futures should be used and can be spawned in large quantities without much of a performance penalty. Tasks need some work to become efficient and there are plans to include a more direct interface to Go's goroutines with cancelable contexts in the future.

3.4 File Access

For obvious reasons, not all embedded interpreters should provide full file access. Therefore, this option needs to be enabled with build tag `fileio`, or otherwise none of the filesystem-related functions will be available. The `z3` Makefile in `cmd/z3` enables this option by default for the standalone executable.

3.5 Images

Z3S5 Lisp supports the writing and reading of Lisp images called "zimages". To save a zimage, use `(save-zimage <version> <info> <entry-point> <file>)`, where the `<version>` must be a semver string and `<info>` a list. The entry point is a procedure that is executed after the image has been loaded and can be `nil` for not executing anything. The filename `<file>` by convention ends in `.zimage`. To load such an image, overwriting the current lisp system, use `(load-zimage <file>)`.

Important limitations need to be kept in mind when working with zimages. Symbols will get unprotected when loading a zimage and therefore the interpreter must have the permission to unprotect symbols. See `declare-unprotected`, `protect`, `unprotect`, and **`protected`**? for more information. Symbols may also be declared volatile with `declare-`**`volatile`** and some of the predefined global variables such as `*tasks*` are already declared volatile. A volatile toplevel symbol is neither written to a zimage nor is a symbol marked volatile in the running system overwritten when it is present in a zimage and the image is loaded. This can be a rare source of incompatibility; there is currently no way to force the loading of a symbol in the image when it is declared volatile in the running system and providing such a mechanism would introduce new problems. You should declare as volatile any symbol that cannot be overwritten because it contains data essential to the running system, and at the same time be aware of the possibility that a global symbol might be declared volatile in the future and therefore not be loaded. Use prefixes and sanity checks where necessary to avoid problems.

Finally, not all values can be externalized. For example, ports, tasks, futures, and mutexes cannot be externalized. If a symbol bound to such a value is declared volatile, then it is neither written nor loaded. If it is *not* volatile, however, then the value `nil` is written and loaded for it. For most applications this is desirable. The procedure in your entry point can check for nil and make sure to re-introduce the desired state of such values after loading an image if this is necessary for the proper functioning of the system. In general, it is desirable to use initialization functions and not rely on global variables when dealing with images.

3.6 Database Access

The build tags `db` and `fts5` enable a database module with Sqlite3 support. See `(dump 'db.)` for available functions and consult the reference and help system for more information on them. The tag `fts5` is required if `db` is used, since the key-value module `kvdb` makes use of the fts5 text indexing features of Sqlite. So the tags always have to be combined.

Take a look at `(dump 'kvdb)` for more information about the key-value database and look up `(help remember)` for information on the remember system. Basically, you can use `(remember key value)` to remember a value, `(recall key)=> value` to recall it, and `(forget key)` to forget it. However, this requires `(init-remember)` to be executed first once. In the `z3` executable this is loaded in the

local `init.lisp` file that also prints the start banner, but you might want to disable it if you don't use remember because it slows down startup.

3.7 Object-oriented Programming

The extension for object-oriented programming is embedded into the init file and adds the symbol `oop` in `*reflect*`. It provides a simple object system with multiple inheritance and more lightweight structure. Both of them are based on arrays whose first element is a symbol starting with `%`, so it is best not to use arrays starting with such symbols for other purposes.

3.7.1 Classes, Objects, Methods

Classes should be created with the `defclass` macro; there are other ways of creating them but this is most convenient. When a class is created, it's name, a possibly empty list of superclasses, and if necessary some properties can be declared:

```
(defclass named nil (name "<unknown>"))
```

This defines a class called `named` with no superclasses and a `name` property with default value `"<unknown>"`. Methods that subclasses shall inherit must be defined *before* the respective subclass is defined. So let's add a convenience method to retrieve the name:

```
(defmethod named-name (this) (prop this 'name))
```

This method takes the instance of the class as first argument ("this") and retrieves the instance's property `name`. The name in the first unquoted argument must be composed out of a valid class name and the method name. The following definitions illustrate inheritance:

```
(defclass point (named) (x 0) (y 0))
(defmethod point-move (this delta-x delta-y)
  (setprop this 'x (+ (prop this 'x) delta-x))
  (setprop this 'y (+ (prop this 'y) delta-y)))
```

This defines a named point with methods `point-name` and `point-move`, where the former is inherited from the superclass `named`. To make instances, use **`new`** as follows:

```
(setq a (new point (x 10) (y 20) (name "A")))
(point-name a)
==> "A"
(prop a 'x)
(setprop a 'x 99)
(prop a 'x)
==> 99
```

```
(point-move a 1 20)
(prop a 'x)
==> 100
(prop a 'y)
==> 40
```

As you can see in the example, the "this" argument must be named in the definition of `point-move` but when calling the method does not need to be taken into account; when the method is called, the first argument is always the object it is called upon. If no property value is specified in a **`new`** call, then a property gets its default value. If no default value has been specified during class definition, then the value is nil. Aside from the direct names bound by `defmethod`, it is also possible to use the `method` function to call methods, and properties can be get and set with `prop` and `setprop` respectively. There are a few more helper functions such as `object?` and **`class`**`?` to test for objects and classes, as well as `isa?` for checking whether an object is a subclass of a given class. Check out the "Object-oriented Programming" section of the Reference Manual for a complete list.

One thing to bear in mind when using this very simple OOP extension is that everything is defined dynamically at runtime and order matters. If you instantiate a class before all of its methods are defined, the resulting object will not take into account any future changes or definitions of the class, it will just be derived from the current state of the class. Likewise, when a subclass is defined, the methods of the superclass need to be defined already or else they will not become part of the subclass. If you look at the implementation, you can see why: It simply copies symbols and their closure from dictionaries.

3.7.2 Structures

Structures are more lightweight array-based representations of named fields without inheritance. They are similar to association lists and the corresponding macros allow the getting and setting of values based on a name in the struct (which is just an assessor to the array):

```
(defstruct point (x 0) (y 0))
```

This is similar to the previous example but no inheritance is possible. Instances of a struct are arays called "records" and created with `make` and `make*`:

```
(setq a (make point '((x 20)(y 100))))
(setq b (make* point (x 10) (y 10)))
(point-x a)
==> 20
(point-x! a 30)
(point-x a)
==> 30
```

That's about it. These macros basically just provide conveniently named getter and setter functions for

arrays. Consult the Reference Manual for a complete list of all structure-related functions.

3.8 Language Stability

At this stage, built-in commands may still change. The good news is that any change introduced in Z3S5 Lisp needs to be tracked and checked in Z3S5 Machine, and so no larger changes are planned. However, no guarantees can be made and you should vendor the repository or even fork it if you want to make sure no breaking changes occur. Once the language is stable, it will be marked on the home page.

3.9 Known Bugs

Since this language has been ported from a larger system, there may be known bugs not in the issue tracker at Github. A known issue of this version is a problem with the recursive externalization of Dict containing Dict.

3.10 Roadmap

A library system, a robust key-value database with fulltext search, some basic OOP, and a simple persistence layer will likely be ported from Z3S5 Machine to this embedded Lisp in the future.

4 Extending Z3S5 Lisp

The system is extended by calling `interp.Def`. As an example, consider the following function from `lisp_base.go`:

```
1     // (str->chars s) => array of int convert a UTF-8 string into an
          array of runes
2     interp.Def("str->chars", 1, func(a []any) any {
3         runes := []rune(norm.NFC.String(a[0].(string)))
4         arr := make([]any, len(runes), len(runes))
5         for i := range runes {
6             arr[i] = goarith.AsNumber(runes[i])
7         }
8         return arr
9     })
```

The `Def` function takes the function symbol as string, the number *n* of arguments, and a function that takes an array of `any` and returns a value `any` (aka `interface{}`). The function may explicitly check

the type of the arguments for correctness but doesn't need to. It is normal for functions to panic and even deliberately throw an error. These are caught by the intepreter and displayed to the user. So how much you check depends primarily on what errors you want to provide. By convenience, custom errors thrown in function definitions start with the name of the function such as `panic(error.New("foobar: the foobar function has failed"))` for a function `foobar`. If a function returns no value, then the definition should return `Void`, a special Lisp value that is not printed in the read-eval-print loop.

Care must be taken with numbers. Pure Go numbers will not do and may lead to bizzare and unexpected runtime behavior. Every number needs to be converted using `goarith.AsNumber` and other conversion functions from `z3s5-lisp` and the package `github.com/nukata/goarith`. Check the source code of some of the implementation files for examples. Again, this is really important: Always convert Go numbers to bigint numbers with `goarith.AsNumber`!

There are a few helper functions such as `ExpectInts` that can be used for checking arguments. Other useful conversion functions are `AsBool`, `ToLispBool`, `ArrayToList`, `ListToArray`, etc.

Notice that lists are structures composed by `&Cell{Car: a, Cdr: b}`, where `Cdr` might be another Cell and the final Cdr is Nil, just like in any Lisp. If you construct these without the member names `Car` and `Cdr` Go will complain about this and refuse to compile, even though in this case it is perfectly fine to use anonymous access. If you want to create a lot of lists by hand in your function definitions without creating runtime performance penalties by using functions like `ArrayToList`, then it might make sense to switch off this behavior of the `go vet` command by running `go vet -composites=false` instead. Of course, this will also disable such checks for other parts of your application where they might be useful. The only other way is to make the list construction fully explicit, as in: `&z3.Cell{z3.NewSym("hello"), &z3.Cell{z3.NewSym("world"), z3.Nil}}` which yields the list `(hello world)`.

Custom data structures: Since there is currently no way to modify the printer for custom structures directly in the Lisp system, it is best to put them into a box using `interp.DefBoxed`, which takes a symbol for a boxed value and creates a number of auxiliary functions. See `lisp_decimal.go` for an example of how to use boxed values.

5 License

Z3S5 Lisp was written by RAST Erich and is based on Nukata Lisp by SUZUKI Hisao. It is licensed under the MIT License that allows free use and modification as long as the copyright notices remain. Please read the LICENSE file for more information.