A Short Introduction to the Basics of

JudoShiai Lisp

JudoShiai Lisp is a small subset of real Lisp implementations. It is intended to be used by the SVG functionality. Reader is expected to have some knowledge about Lisp to understand this paper.

Introduction

There are many ways to express the same thing, like competitor's points. For example, if competitor has lost you may not want to display "0" but text "LOSS". Or if there is hikiwake you may want to show text "DRAW". JudoShiai is quite flexible, but there are limits what can be done. JudoShiai Lisp introduces some extra help.

Example: You want to show text "LOSS" if competitor has lost. Now you can do that by writing to SVG file text

```
%m23p1-1==0'LOSS'
```

Here

%m	Match related text
23	Match number
p	Points
1	1 = white, 2 = blue competitor
-1==0'LOSS'	Print LOSS if points are 0

What if that is not enough? How to print DRAW in case of hikiwake and still print LOSS? Perhaps you want to have text IPPON flashing red? That is possible by using Lisp. The SVG text above is replaced by %, a Lisp expression, and match identification:

```
%(pts 1)m23
```

Lisp code in the example is

```
(pts 1)
```

m23 sets global Lisp variables to values related to match #23.

You have defined function `pts` that can write anything on the SVG sheet.

Lisp files are in the same directory as SVG files. JudoShiai first reads all the SVG files and then tries to find files with suffix `.lisp`. One file should be enough; unnecessary extra files use memory and slow down operations.

JudoShiai Lisp

JudoShiai Lisp is a subset of complete implementations. Next we go through the basic usage and features that should be enough to write simple code.

Lisp expressions are called symbolic expressions or s-expressions. The s-expressions are composed of three valid objects, atoms, lists and strings. Any s-expression is a valid program. Example code writes sum of three numbers:

```
(write (+ 7 9 11))
```

Lisp uses prefix notation. In the above program the + symbol works as the function name for the process of summation of the numbers. Calculation 1 + 2*3 in Lisp:

```
(+ 1 (* 2 3))
```

To write text and other expressions:

```
(write "Square of " x " is " (* x x))
```

Basic Building Blocks

Lisp programs are made up of three basic building blocks:

- An **atom** is a number (an integer or a double precision floating point) or string of contiguous characters. It includes numbers and special characters.
  - Hello, *xyz#, 12345, 3.14159265
- A **list** is a sequence of atoms and/or other lists enclosed in parentheses.
  - (a (b c) 5)
- A **string** is a group of characters enclosed in double quotation marks.
  - "Hello world!"

Comments

The semicolon symbol (;) is used for indicating a comment line.

```
(write "Hello World") ; greet the world
;; Top level comment starts with two semicolons by convention.
```

Lists

Lists are single linked lists. Lists are constructed as a chain of a simple record structure named **cons** linked together. A cons contains two components called the **car** and the **cdr**.

The **cons** function takes two arguments and returns a new cons cell containing the two values. These values can be references to any kind of object. If the second value is not nil, or another cons cell, then the values are printed as a dotted pair enclosed by parentheses.

The two values in a cons cell are called the **car** and the **cdr**. The **car** (or **first**) function is used to access the first value and the **cdr** (or **rest**) function is used to access the second value.

```
(write (cons 1 2))                               ; writes (1 . 2)
(write (cons ‘a ‘b))                             ; writes (a . b)
(write (cons 1 ()))                              ; writes (1)
(write (cons 1 (cons 2 nil)))                    ; writes (1 2)
(write (cons 1 (cons 2 (cons 3 nil))))           ; writes (1 2 3)
(write (first (cons 1 (cons 2 (cons 3 nil))))) ; writes 1
(write (rest (cons 1 (cons 2 (cons 3 nil)))))  ; writes (2 3)
```

Use of Single Quotation Mark

Lisp evaluates everything including the function arguments and list members. At times, we need to take atoms or lists literally and don't want them evaluated or treated as function calls. To do this, we need to precede the atom or the list with a single quotation mark.

```
(write (+ 1 2)) ; this will print 3
(write '(+ 1 2)) ; this will print (+ 1 2)
```

Defining a Macro

A named macro is defined using defmacro. Syntax for defining a macro is:

```
(defmacro macro-name (parameter-list) body-form)
```

The following example writes the first member of a list added by 10:

```
(defmacro print-1st (expr . rest)
    (+ (car expr) 10))
(write (print-1st 1 2 3)) ; writes 11
```

```
(defmacro print-2nd (expr . rest)
    (+ (car rest) 10))
(write (print-2nd 1 2 3)) ; writes 12
```

In the previous example `expr` will have value 1 and rest is a list `(2 3)`. Thus `(car rest)` has value 2.

Global Variables

Global variables have permanent values throughout the Lisp system and remain in effect until a new value is specified. Global variables are generally declared using the define construct.

```
(define a 7)
(write a) ; writes 7
```

Change the value using setq:

```
(setq a 42)
```

Global variables defined by JudoShiai

You use `define` to create global variables. However, there are some predefined read-only objects that you can use after their values are set. To set match related variables call your Lisp function from an SVG page like this:

```
%(pts 1 “some text”)m23
```

m23 sets the following global predefined read-only variables for the match #23 (note: JudoShiai variable names start with %) before calling function `pts`:

%category	Number of the category. This has not much use except in SQL command.
%number	Number of the match.
%comp_1	Internal index of the white competitor.
%comp_2	Blue competitor. Indexes have not much use except in SQL commands.
%score_1	White competitor’s scores (ippons, waza-aris, shidos)
%score_2	Scores for the blue competitor.
%points_1	Value of the win for the white (ippon = 10 points).
%points_2	Blue competitor’s winning points.
%match_time	Match time in seconds.
%comment	Match is forced to be next or preparing or delayed.
%tatami	Default contest area for the match.
%group	Group number category belongs to.
%flags	Misc. flags.
%forcedtatami	Contest area where fight has been moved manually.
%forcednumber	Number of the fight, if moved manually.
%date	Date information of the match as seconds since the Epoch.
%legend	Legend for the match.
%roundnum	Round number of the match (first round, repechage, etc).

To set competitor related variables call your Lisp function like this:

```
%(print-name)m23-1
```

This will first set white competitor’s name and other data to global variables and then call your Lisp function (print-name). The following competitor’s variables are set:

%first	First name.
%last	Last name.
%club	Club.
%country	Country.
%comment	Comment text.
%compid	Competitor’s ID.
%coachid	Competitor’s coach ID.
%compix	Internal index. Used with SQL commands.
%birthyear	Year of birth.
%grade	Grade as an integer. Default: 0 = unknown, 1 = 6. kyu, 2 = 5. kyu, … 7 =

	1. dan, 8 = 2. dan, ...
%regcategory	Registered category.
%category	Real category.
%weight	Weight in grams.
%seeding	Seeding 1-8.
%clubseeding	Clubseeding.
%gender	Gender. Male = 1, female = 2.

For the blue competitor the Lisp call would be:

```
%(print-name)m23-2
```

Lisp call is the one surrounded by parenthesis. You can have any name and arguments.

Other read-only variables:

%pages	Number of pages to generate from an SVG file.
%page	Current SVG file page. Pages are counted from 0 onwards.

Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. The operations allowed on data could be categorized as:

- Arithmetic Operations
- Comparison Operations
- Logical Operations
- Bitwise Operations

Arithmetic Operations

The following table shows all the supported arithmetic operators. Assume variable A holds 10 and variable B holds 20 then:

Operator	**Example**
+	(+ A B) will give 30
-	(- A B) will give -10
*	(* A B) will give 200
/	(/ B A) will give 2
mod	(mod B A) will give 0

Comparison Operations

Following table shows all the supported relational operators that compares between numbers. Assume variable A holds 10 and variable B holds 20, then:

Operator	**Description**	**Example**
=	Checks if the values of the operands are all equal or not, if yes then condition becomes true.	(= A B) is not true.
/=	Checks if the values of the operands are all different or not, if values are not equal then condition becomes true.	(/= A B) is true.
>	Checks if A is greater than B.	(> A B) is not true.
<	Checks if A is less than B.	(< A B) is true.
>=	Checks if A is greater than or equal to B.	(>= A B) is not true.
<=	Checks if A is less than or equal to B.	(<= A B) is true.

Logical Operations on Boolean Values

Lisp provides three logical operators: and, or, and not that operates on Boolean values. Assume A has value nil and B has value 5, then:

Operator	**Description**	**Example**
and	It takes any number of arguments. The arguments are evaluated left to right. If all arguments evaluate to non-nil, then the value of the last argument is returned. Otherwise nil is returned.	(and A B) will return nil.
or	It takes any number of arguments. The arguments are evaluated left to right until one evaluates to non-nil, in such case the argument value is returned, otherwise it returns nil.	(or A B) will return 5.
not	It takes one argument and returns t if the argument evaluates to nil.	(not A) will return t.

Bitwise Operations on Numbers

Bitwise operators work on bits and perform bit-by-bit operation. Assume variable A holds 60 and variable B holds 13, then:

Operator	**Description**	**Example**
logand	This returns the bit-wise logical AND of its arguments.	(logand A B)) will give 12
logior	This returns the bit-wise logical INCLUSIVE OR of its arguments.	(logior A B) will give 61
logxor	This returns the bit-wise logical EXCLUSIVE OR of its arguments.	(logxor A B) will give 49
lognot	This inverts the bits.	(lognot A) will give -61

Decision Making

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

The **if** construct has various forms. In simplest form it is followed by a test clause, a test action and some other consequent action(s). If the test clause evaluates to true, then the test action is executed; otherwise, the consequent clause is evaluated.

```
 (if (= p 0) (write "LOSS")         ; write LOSS if p = 0
    (if (= p 7) (write 5)           ; else if p = 7 write 5
      (if (= p 11) (write "DRAW")   ; else if p = 11 write DRAW
        (write p)                   ; else write p
        (+ v 1))))                  ; and increment v
```

Loops

There may be a situation, when you need to execute a block of code numbers of times. A loop statement allows us to execute a statement or group of statements multiple times.

The **while** loop executes while a condition evaluates to true.

```
(setq i 0)
(while (< i 10)
    (write "Value: " i "\n")
    (+ i 1))
```

Functions

A function is a group of statements that together perform a task. The **defun** is used for defining functions. The defun needs three arguments:

- Name of the function
- Parameters of the function

- Body of the function

```
(defun name (parameter-list) body)
```

Example code calculates Fibonacci numbers:

```
(defun fib (n)
  (if (< n 2) n
      (+ (fib (- n 1)) (fib (- n 2)))))

(write (fib 6)) ; writes 8
```

Note recursion. The body of the function may consist of any number of Lisp expressions. The value of the last expression in the body is returned as the value of the function.

lambda is the symbol for an anonymous function, a function without a name. Every time you use an anonymous function, you need to include its whole body.

Although functions are usually defined with `defun` and given names at the same time, it is sometimes convenient to use an explicit lambda expression. Anonymous functions are valid wherever function names are. They are often assigned as variable values, or as arguments to functions. In the example squares for numbers 1 – 4 are printed:

```
(define x 1)
(while (< x 5)
  (print ((lambda (a) (* a a)) x) "\n")
    (setq x (+ x 1)))
```

Note function `print`. Function `write` is used to write to SVG file but destination of `print` is standard output. Thus you can use it to print debug information to the console that will not be visible in the SVG file. To enable console printing in Windows start judoshiai from command line with optional argument `–console` (in Linux console is printed automatically):

```
cd C:\path\to\JudoShiai\bin
judoshiai -console
```

Supported features

Next table lists available features:

Lisp code	***Explanation***
Symbols	
()	Constant nil.
t	Constant true.
(define *sym e*)	Defines *sym* to be a global variable with initial value *e*.
Value assignment	
(setq *sym e*)	Evaluates *e* and makes it the value of the symbol *sym*; the value of *e* is returned. Cannot be used with JudoShiai predefined read-only variables starting with %.
(copy *sym*)	Returns copy of a number or a string.

Output	
(write e_1 e_2 ...)	Prints a readable representation of *e*'s on SVG page.
(print e_1 e_2...)	Prints a readable representation of *e*'s to standard output. Useful for debug printing.
(println e_1 e_2 ...)	Prints a readable representation of *e*'s to standard output. Ends line with a new line character.
(fopen)	Opens a file for writing whose name will be asked (file select dialog).
(fopen *e*)	Opens a file for writing whose name is *e*.
(fwrite e_1 e_2 ...)	Writes a readable representation of *e*'s to a file that was opened.
(fclose)	Closes the file that was previously opened.
Lists	
(car *lst*) (first *lst*)	Returns the first element of the list *lst* (nil if *lst* is empty)
(cdr *lst*) (rest *lst*)	Returns a list containing all except the first element of the list *lst* (nil if the length of *lst* is less than 2)
(cons *e lst*)	Creates a list with *e* as the first element and *lst* as the rest.
(setcar *lst e*)	Replaces car of *lst* with *e*.
Testing for Equality	
(eq e_1 e_2)	Returns T if and only if 1 and 2 are the same object.
Arithmetic	
(+ n_1 n_2 … n_k)	Returns n_1 + n_2 … + n_k.
(- n_1 n_2 … n_k)	Returns n_1 - n_2 … - n_k.
(* n_1 n_2 … n_k)	Returns n_1 * n_2 … * n_k.
(/ n_1 n_2)	Returns n_1 / n_2.
(- *n*)	Returns -*n*.
(mod *number divisor*)	Returns *number* modulo *divisor*.
(ceiling *n*)	Returns the smallest integer that is not smaller than the value of *n*. For example, (ceiling 0.5) is 1.0, and (ceiling -0.5) is 0.0.
(floor *n*)	Returns the largest integer that is not larger than the value of *n*. For example, (floor 0.5) is 0.0, and (floor -0.5) is -1.0.
(sqrt *n*)	Returns the principal square root of the value of *n*.
(exp *n*)	Returns e^n
(expt *base n*)	Returns $base^n$
(log *n*)	Returns the logarithm base e of *n*.
(cos *n*)	Returns the cosine of *n* (given in radians).

(sin *n*)	Returns the sine of *n* (given in radians).
(tan *n*)	Returns the tangent of *n* (given in radians).
(acos *n*)	Returns the arc cosine (in radians) of *n*.
(asin *n*)	Returns the arc sine (in radians) of *n*.
(atan *n*)	Returns the arc tangent (in radians) of *n*.
(float *n*)	Returns the value of *n* as a floating point number.
(truncate *n*)	Returns the integer having the largest absolute value that is not larger than the absolute value of *n*. Thus truncate truncates toward zero.
(round *n*)	Returns the integer closest to the value of *n*.
Bitwise Arithmetic	
(logand n_1 n_2 … n_k)	Returns bitwise and of arguments.
(logior n_1 n_2 … n_k)	Returns bitwise inclusive or of arguments.
(logxor n_1 n_2 … n_k)	Returns bitwise exclusive or of arguments.
(lognot *n*)	Returns n bits inverted.
Comparisons on Numbers	
(= n_1 n_2)	Returns T if $n_1 = n_2$; otherwise returns nil.
(/= n_1 n_2)	Returns T if $n_1 <> n_2$; otherwise returns nil.
(< n_1 n_2)	Returns T if $n_1 < n_2$; otherwise returns nil.
(<= n_1 n_2)	Returns T if $n_1 <= n_2$; otherwise returns nil.
(> n_1 n_2)	Returns T if $n_1 > n_2$; otherwise returns nil.
(>= n_1 n_2)	Returns T if $n_1 >= n_2$; otherwise returns nil.
Logical Functions	
(and e_1 e_2 … e_k)	Evaluates each argument sequentially. If and reaches an argument that returns nil, it returns nil without evaluating any more arguments. If it reaches the last argument, it returns that argument's value.
(or e_1 e_2 … e_k)	Evaluates each argument sequentially. If or reaches an argument that is not nil, it returns the value of that argument without evaluating any more arguments. If it reaches the last argument, it returns that argument's value.
(not *e*)	Returns T if the value of *e* is nil; otherwise returns nil.
Conditional Constructs	
(if *test* e_1 e_2 … e_n)	Evaluates *test* and, if not nil, evaluates e_1. Otherwise, e_2... e_n are evaluated. e_2... e_n can be omitted.
(while *test* e_1 e_2 … e_n)	Evaluate e_1 … e_n while test is true.
Function Definition	
(defun *name* (a_1 … a_k) e_1 … e_n)	Defines a named function with arguments a_1 … a_k and body e_1 … e_n.

(lambda (a_1 … a_k) e_1 … e_n)	Defines local functions with arguments a_1 … a_k and body e_1 … e_n.
(defmacro name (a_1 … a_k) e_1 … e_n)	
(macroexpand *e*)	
Evaluation-Related	
(quote *e*)	quote simply returns its argument without evaluating it. This allows any Lisp object to be written as a constant value in a program. (quote x) can be abbreviated as 'x .
Miscellaneous	
(gensym)	gensym creates a temporary symbol.
JudoShiai Specific Functions	
(get-data-by-ix *index*)	Sets predefined global competitor or category specific variables. *Index* is the internal ID of the competitor or category.
(get-next-match *tatami n*)	Sets predefined global match specific variables for the *n*th match on mat *tatami*.
(round-name *roundnum*)	Returns printable name for round number.
(set-pages *n*)	Sets number of pages to generate from an SVG file.
(sql *function request*)	Make a SQL request. *function* will be called for each resulting table line. List of table line items will be delivered as arguments.
(hexcolor *r g b*)	Returns a string atom whose value is #RRGGBB. *R*, *g*, and *b* are floating point numbers 0.0 – 1.0. Example: (hexcolor 0.5 0 1.0) returns “#7f00ff”. The string describes a color in SVG compatible format.

Example

In the beginning we had a problem how to print competitor’s points. Requirements:

- Write nothing if the match has not been fought.
- If match ends up hikiwake (draw) write DRAW.
- Write LOSS if competitor has no points.
- We want to write 5 for waza-ari win. Database marks that with number 7.
- Otherwise write the database points value.

We have to modify the SVG file. Write text `%(pts 1)m<n>` if you want to print the points of the white competitor and `%(pts 2)m<n>` for the blue competitor (<n> is the match number).

Make a file GBR.Lisp. Start with function pts:

```
(defun pts (who)                          ; line 1
  (if (or (> %points_1 0) (> %points_2 0))  ; line 2
    (if (= %points_1 %points_2)           ; line 3
      (write "DRAW")                      ; line 4
      (if (= who 1)                       ; line 5
        (print-pts %points_1)             ; line 6
        (print-pts %points_2)))))         ; line 7
```

Explanation line by line:

1. Define function `pts`. It has one argument, `who`. If `who` is 1 it means white competitor, 2 means blue competitor.

2. Check if the match has been fought. Either white or blue must have points.
`(> %points_1 0)` is true if white has more points than 0. `Or` function is true if blue or white have points.

3. If previous test was true check if the points are the same. This means hikiwake or draw.

4. This line is executed if the points are the same. Write DRAW on the SVG page.

5. Otherwise, check if `who` is 1.

6. If `who` is 1 (= white) call function `print-pts` with the white points as its argument. Note: function name can have letter '-', it doesn't mean subtraction.

7. If who was not white call function `print-pts` with the blue points as its argument.

Finally implement function print-`pts`:

```
(defun print-pts(p)                 ; line 1
  (if (= p 0) (write "LOSS")        ; line 2
    (if (= p 7) (write 5)           ; line 3
      (if (= p 11) (write "DRAW")   ; line 4
        (write p)))))               ; line 5
```

Explanation:

1. Function `print-pts` has one argument `p`, the points to print.

2. If `p` is 0 print LOSS.

3. Otherwise, if `p` is 7 print 5. In this case a national rule says that waza-ari scores 5 points.

4. Otherwise, if `p` is 11 print DRAW. In principle draw was detected in the previous function, but let's have it here too for completion.

5. Otherwise print points as is.

Points in the database can have the following values:

1: Win by shido

2: Win in golden score when that always scores 1 point

3: Win by koka (not used)

5: Win by yuko (not used)

7: Win by waza-ari

10: Win by ippon

11: Hikiwake

Real points and how they are shown depend on the national rules.

Converting old SVG files

It is easy to convert old SVG files to use Lisp calls. The following Linux Bash shell script does the trick (Windows lovers: you are on your own):

```
#!/bin/bash
#
# Run this in svg directory
#
# Conversion example (match #23):
#   %m23p1-1==0'LOSS' is converted to %(pts 1)m23

for file in [0-9]*.svg
do
    echo "File: $file"
    sed -i "s/%m\([0-9]*\)[-]*p\([12]\)-1==[^<]*/%(pts \2)m\1/" $file
done
```

SVG + LISP scripts

You can create and print pages generated from SVG files that contain embedded Lisp code. Usually Lisp code is divided in two parts:

- Simple code in the SVG file, that calls the more complicated functions.
- Code in a Lisp file that contains most of the code.

When you select an SVG file to run JudoShiai will first read all the Lisp files in the same directory **once**. This is important to avoid running and defining the same things all over again. JudoShiai is delivered with example code in directory `svg-lisp`. You can find for example a Lisp file `common.lisp` and SVG file `print-matches.svg`.

In JudoShiai select *Tournament → SQL Dialog → Run Script*. Instead of running a Basic script select an SVG file.

SVG file can have Lisp code. Whenever JudoShiai finds text like `%(`*`lisp code`*`)` the lisp code will be executed. Lisp code writes whatever you like and the output will replace the original `%(…)`.

However, this way you cannot draw graphics since the output from the Lisp code will be surrounded by `<text...>` and `</text>` tags. Solution is to have one (and only one) function that is called in the end of the SVG file. Text has an exclamation mark between the “%” and “(“:

`%!(`*`lisp code to the end`*`)`

Multiple pages

SVG has no standard for multiple pages. However, there is a trick to do the job, but it is restricted to copying the same page with different additions. Most probably this is what you want in the first place!

There are two predefined variables

`%pages` is the number of pages. It is set using the function `(set-pages `*`n`*`)` in the SVG file.

`%page` is the current page to show or print. It is set internally, but you can use its value to select what to print or draw. Pages are counted from 0 onwards.

Example: You want to have a list on three pages. In SVG file you should have texts

- `%(set-pages 3)`
- `%!(print-my-list)`

The first text can be anywhere, it will not be visible. The second one is executed in the very end. Lisp file will have functions:

```
(defun print-items (start)           ; prints 10 items from start
  ; your lisp code here
)

(defun print-my-list ()
  (if (= %page 0) (print-items 1)    ; first page has items 1...10
    (if (= %page 1) (print-items 11) ; second page has items 11...20
      (print-items 21))))            ; third page has items 21...30
```

SQL commands

Example: Print members of category `cat-name`:

```
(defun print-competitor (name)
  (write (first name) ", "             ; write last name
         (first (rest name)) "\n"))     ; write first name

(defun print-category (cat-name)
  (sql print-competitor "select last,first from competitors where category='"
   cat-name "'"))
```

Good practices

There are a few things to consider. For example how to split the code between a Lisp file and a SVG file?

In practice you cannot write long Lisp code in a SVG file since the editors like Inkscape will split the code to separate instances and break it. Thus it is better to have one common Lisp file and several SVG files in the same directory. Lisp file will be read only once, but the code in SVG files will be executed every time the page is generated.

Example:

`common.lisp` contains a definition:

```
(defun my-function (arg1 arg2)
    … code …
```

SVG file will have a text line

```
%(my-function 3 7)
```

If the definition were in the SVG file, too, everything would work for a while, but every time you generated the page the definition would be saved in the memory in addition to the previous definitions. Since the definitions don't replace each other JudoShiai would run out of memory or at least become slow.

Sometimes you may have a SVG file that you don't use often. It doesn't make sense to always load its definitions to the JudoShiai Lisp system. There is a way to have temporary definitions. JudoShiai has an example file `svg-lisp/hanoi.svg`. It solves the classical puzzle Towers of Hanoi. There are two ways to construct the code:

Option1, static function definitions:

Divide code in two parts. Definitions go to `common.lisp`:

```
(defun dohanoi (n to from u)
  (if (= n 0) ()
    (dohanoi (- n 1) u from to)
    (write "move " from " > " to "\n")
    (dohanoi (- n 1) to u from))))

(defun hanoi (n)
  (dohanoi n 3 1 2))
```

SVG file contains a call to function `hanoi`:

```
%(hanoi 3)
```

His works fine.

Option 2, temporary function definitions:

If you do not want to make the common Lisp file unnecessary big just because of some temporary trials you can put everything in the SVG file. To avoid saving a new definition every time you render a SVG page you can encapsulate the code inside a *lambda* function:

```
((lambda (n)
   (defun dohanoi (n to from u)
     (if (= n 0) ()
       (dohanoi (- n 1) u from to)
       (write "move " from " > " to "\n")
       (dohanoi (- n 1) to u from)))
   (dohanoi n 3 1 2)) 3)
```

Lambda defines a function temporarily. To clarify let's hide the function body:

```
((lambda (n) . . . ) 3)
```

Lambda defines an anonymous function that here takes one argument *n*. The argument will be number 3. Now this will do the same as

```
(hanoi 3)
```

earlier. Another difference is that in the option1 there were two functions defined, but now `dohanoi` is defined inside the lambda function. Thus it will exist only the lifetime of the lambda function and the definitions don't pile up.