1. Project Overview

1.1 Introduction

LISP is a formal mathematical language. It is therefore possible to give a concise yet complete description of it. [1] Lisp is a functional programming language with imperative features. By functional we mean that the overall style of the language is organized primarily around expressions and functions rather than statements and subroutines. Every Lisp expression returns some value. Every Lisp procedure is syntactically a function; when called, it returns some data object as its value. By imperative we mean that some Lisp expressions and procedures have side effects, such as storing into variables or array positions. Thus Lisp procedures are not always functions in the "pure" sense of logicians, but in practice they are frequently referred to as "functions" anyway, even those that may have side effects, in order to emphasize that a computed result is always returned.

Let us see how computation is performed in lisp:

All computation in Lisp is done by means of function calls. A function call is specified as a list consisting of the function name followed by the arguments of the function; this is sometimes referred to as “Cambridge Polish” notation.

Example: (quotes are not displayed)

- (+ 2 2) gives us “4”.
- (+ 3 (* 4 5)) gives us “23”.
- (print ‘(hi mom)) gives us “hi mom”.

In most cases, the arguments of the function are evaluated before the function is called. For example, in evaluating **(+ 3 (* 4 5))** the argument **(* 4 5)** is evaluated to give **20**, and the function **+** is then applied to the arguments **3** and **20**.

A function always returns a value; some functions also have side effects, such as printing.

Originally specified in 1958 Lisp is the second oldest high level programming language in widespread use today, older than FORTRAN by one year.

Below are examples of Lisp and FORTRAN difference in syntax:

Fortran	**Lisp**
SQRT(2.0)	(SQRT 2.0)
X = 2	(SETQ X 2)
X = 2 + 2	(SETQ X (+ 2 2))
PRINT 1, X	(PRINT X)
IF (X .GT. Y) Y = 3	(COND ((> X Y) (SETQ Y 3)))
GO TO 10	(GO 10)

Application of Lisp:

Below are some of the applications of Lisp:

- Artificial Intelligence
- Symbolic Algebraic Manipulation
- Natural Language Understanding
- Machine Translation
- Formal Logical Reasoning
- Expert Systems:
    - Diagnosis
    - Identification
    - Design
- Automatic Programming
- Robotics
- Perception (Vision, Speech Understanding)

Today, the most widely known general-purpose Lisp dialects are **Common Lisp** and **Scheme**.

Advantages of Lisp:

Below are some of the advantages of Lisp:

- Lisp is
  - a general-purpose programming language and an AI language
  - interactive
- Lisp programs are
  - easy to test (interactive)
  - easy to maintain (depending on programming style)
  - portable across hardware/OS platforms and implementations (there is a standard for the language and the library functions)
- Lisp provides
  - clear syntax, carefully designed semantics
  - several data types: numbers, strings, arrays, lists, characters, symbols, structures, streams etc.
  - runtime typing: the programmer need not bother about type declarations, but he gets notified on type violations.
  - many generic functions: 88 arithmetic functions for all kinds of numbers (integers, ratios, floating point numbers, complex numbers), 44 search/filter/sort functions for lists, arrays and strings
  - automatic memory management (garbage collection)
  - packaging of programs into modules
  - an object system, generic functions with powerful method combination (Common Lisp was the first ANSI standard object oriented programming language. DSB)
  - macros: every programmer can make his own language extensions

1.2Objective and Scope of the project

The objective of the project is to design a GUI (Graphical User Interface) based application for android devices (Mobile). The applications works exactly the same way it works on a desktop computer. It can be useful for students who want to learn Lisp programming, but couldn't afford a desktop computer.

List of functions that are implemented as follows:

Addition	Subtraction	Multiplication	Division
Less-than	Greater-Than	Less-Than or Equal-To	Greater-Than or Equal-To
If-Else	Setq	When	Unless
Defun	Print	First	Rest
Loop	Eq	User-Defined Functions	

This project is useful for two reasons:

- It provides a GUI for students as well as teachers to learn and teach Lisp programming language.
- The application has documentation for each of these functions listed above along with sample codes. Users can read them can learn Lisp programming language entirely on his own.

Features of the application:

- Sample code: The application provides sample codes of various functions that are implemented.
- Documentation: The application provides information about the functions that are implemented.
- Open/Save code: The application provides the user to save his code, and also open it at a later time.
- Last Activity: Can be used to get the last code typed.
- Interpreter: The main application.

1.3Problem Definition

- To develop a system to enhance the way of processing.
- To develop application that can be easily used, irrespective of the system or hardware.
- The Graphical User Interface (GUI) should be attractive and user friendly that will helps to overcome the complication of user those are not aware about system.
- The system has been developed in a continuous way so that new function definitions can be added for later improvement.
- To develop an application that uses hardware and software that is easily available.
- To develop an application that reduces the cost and is efficient.
- To develop an application that can be easily used by the users.

1.4 Hardware and Software Specification

<u>Hardware Required:</u>

- Android mobile device.

<u>Software Required:</u>

- Need to download the LISP Interpreter application from google play store (available on every android mobile device).
- Must have Android version 2.3.3 or above.

2. Literature Survey

2.1 Introduction to LISP Programming Language

2.1.1Brief History of LISP

Lisp is the second-oldest programming language that is still in widespread use (the oldest is FORTRAN). Lisp was developed by John McCarthy in the late 1950 as a tool for proofs of program correctness; see McCarthy, “Recursive Functions of Symbolic Expressions and their computation by machine”, Communications of the ACM, April

For many years, Lisp was considered an esoteric language that was mainly used by academic. In the early 1970’s work on high-powered personal workstations specialized for use with Lisp was begun at M.I.T. and at Xerox Palo Alto Research Center; this research resulted in the Lisp machines that are commercially available today.

Lisp’s simple and uniform syntax, interpreter, and ability to generate new data structures at runtime have encouraged the development of powerful programming environments that are the best available for any language. Some companies are buying Lisp machines and using Lisp primarily for the programming environment, even when their applications do not use AI techniques.

The following are the key ideas of LISP computation adapted from McCarthy (1978) [2]:

1. Computers with arbitrary symbolic expressions,
2. Creates lists through associations of primitives called atoms,
3. Links list at different levels,
4. Controls structure by creating functions from simpler functions, and
5. Uses an evaluation function as an interpreter.

2.1.2 The Extensible Language

Not long ago, if you asked what Lisp was for, many people would have answered“for artificial intelligence.” In fact, the association between Lisp andAI is just anaccident of history. Lisp was invented by John McCarthy, who also invented theterm “artificial intelligence.” His students and colleagues wrote their programs inLisp, and so it began to be spoken of as anAI language. This line was taken upand repeated so often during the briefAI boom in the 1980s that it became almostan institution.[3]

Fortunately, word has begun to spread thatAI is not what Lisp is all about.Recent advances in hardware and software have made Lisp commercially viable:it is now used in Gnu Emacs, the best Unix text-editor; Autocad, the industry standarddesktopCAD program; and Interleaf, a leading high-end publishing program.The way Lisp is used in these programs has nothing whatever to do withAI.If Lisp is not the language ofAI, what is it? Instead of judging Lisp by thecompany it keeps, let’s look at the language itself. What can you do in Lisp thatyou can’t do in other languages? One of the most distinctive qualities of Lisp isthe way it can be tailored to suit the program being written in it. [3]

Lisp itself is aLisp program, and Lisp programs can be expressed as lists, which are Lisp datastructures. Together, these two principles mean that any user can add operators toLisp which are indistinguishable from the ones that come built-in.

2.1.3What Made Lisp Different

Lisp embodied nine new ideas:

1. Conditionals: A conditional is an if-then-else construct. We take these for granted now. They were invented by McCarthy in the course of developing Lisp. (FORTRAN at that time only had a conditional goto, closely based on the branch instruction in the underlying hardware.).

2. Function type: In Lisp, functions are first class objects-- they're a data type just like integers, strings, etc, and have a literal representation, can be stored in variables, can be passed as arguments, and so on.

3. Recursion: Recursion existed as a mathematical concept before Lisp of course, but Lisp was the first programming language to support it. (It's arguably implicit in making functions first class objects.)

4. A new concept of variables: In Lisp, all variables are effectively pointers. Values are what have types, not variables, and assigning or binding variables means copying pointers, not what they point to.

5. Garbage-collection.

6. Programs composed of expressions: Lisp programs are trees of expressions, each of which returns a value. (In some Lisps expressions can return multiple values.) This is in contrast to FORTRAN and most succeeding languages, which distinguish between expressions and statements.

It was natural to have this distinction in FORTRAN because (not surprisingly in a language where the input format was punched cards) the language was line-oriented. You could not nest statements. And so while you needed expressions for math to work, there was no point in making anything else returns a value, because there could not be anything waiting for it.

This limitation went away with the arrival of block-structured languages, but by then it was too late. The distinction between expressions and statements was entrenched. It spread from FORTRAN into Algol and thence to both their descendants.

When a language is made entirely of expressions, you can compose expressions however you want. You can say either (using Arc syntax)

(If foo (= x 1) (= x 2))

or

(= x (if foo 1 2))

7. A symbol type: Symbols differ from strings in that you can test equality by comparing a pointer.

8. A notation for code using trees of symbols.

9. The whole language always available: There is no real distinction between read-time, compile-time, and runtime. You can compile or run code while reading, read or run code while compiling, and read or compile code at runtime.

Running code at read-time lets users reprogram Lisp's syntax; running code at compile-time is the basis of macros; compiling at runtime is the basis of Lisp's use as an extension language in programs like Emacs; and reading at runtime enables programs to communicate using s-expressions, an idea recently reinvented as XML.

2.1.4S-expression, the Syntax of LISP

Introduction to Symbol Expressions

The syntactic elements of the Lisp programming language are *symbolic expressions*, also known as *s-expressions*. Both programs and data are represented as s-expressions: an s-expression may be either an *atom* or a *list*.

Lisp atoms are the basic syntactic units of the language and include both numbers and symbols. Symbolic atoms are composed of letters, numbers, and the non-alphanumeric characters.

Examples of Lisp atoms include:

```
3.1416
100
hyphenated-name
*some-global*
nil
```

A *list* is a sequence of either atoms or other lists separated by blanks and enclosed in parentheses.

Examples of lists include:

```
(1 2 3 4)
(georgekatejamesjoyce)
(a (b c) (d (e f)))
( )
```

Note that lists may be elements of lists. This nesting may be arbitrarily deep and allows us to create symbol structures of any desired form and complexity. The empty list, “()”, plays a special role in the construction and manipulation of Lisp data structures and is given the special name nil. nil is the only s-expression that is considered to be both an atom and a list. Lists are extremely flexible tools for constructing representational structures.

For example, we can use lists to represent expressions in the predicate calculus:

```
(on block-1 table)
(likes bill X)
(and (likes georgekate) (likes bill merry))
```

We use this syntax to represent predicate calculus expressions in the unification algorithm of this chapter. The next two examples suggest ways in which lists may be used to implement the data structures needed in a database application.

```
((2467 (lovelaceada) programmer)
 (3592 (babbagecharles) computer-designer))
((key-1 value-1) (key-2 value-2) (key-3 value-3))
```

An important feature of Lisp is its use of Lisp syntax to representprograms as well as data.

For example,

```
(* 7 9)
(– (+ 3 4) 7)
```

The listsmay be interpreted as arithmetic expressions in a prefix notation. This isexactly how Lisp treats these expressions, with (* 7 9) representing theproduct of 7 and 9. When Lisp is invoked, the user enters an interactivedialogue with the Lisp interpreter. The interpreter prints a prompt, in ourexamples “>”, reads the user input, attempts to evaluate that input, and, ifsuccessful, prints the result.

For example:

```
> (* 7 9)
63
>
```

Here, the user enters (* 7 9) and the Lisp interpreter responds with 63, i.e., the *value* associated with that expression. Lisp then prints another prompt and waits for more user input. This cycle is known as the *read-eval-print* loop and is the heart of the Lisp interpreter.

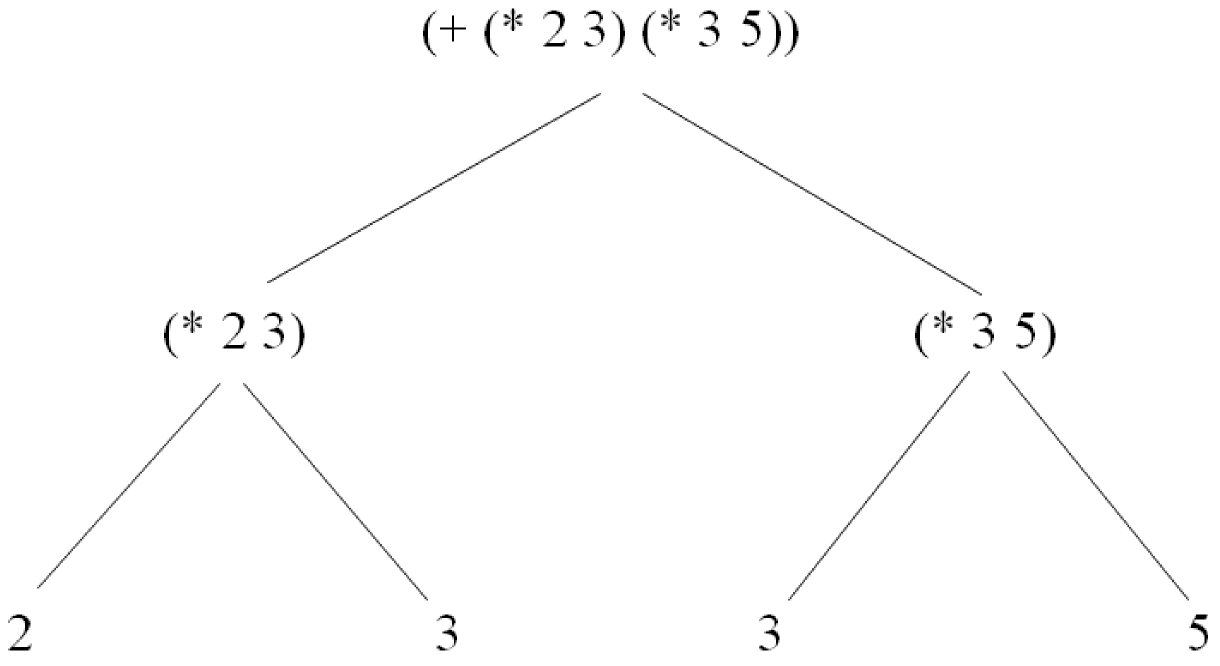
When given a list, the Lisp evaluator attempts to interpret the first element of the list as the name of a function and the remaining elements as its arguments. Thus, the s-expression (f x y) is equivalent to the more traditional looking mathematical function notation f(x,y). The value printed by Lisp is the result of *applying* the function to its arguments. Lisp expressions that may be meaningfully evaluated are called *forms*. If the user enters an expression that may not be correctly evaluated, Lisp prints an error message and allows the user to trace and correct the problem.

A sample Lisp session appears below:

```
> (+ 14 5)
19
> (+ 1 2 3 4)
10
> (– (+ 3 4) 7)
0
> (* (+ 2 5) (– 7 (/ 21 7)))
28
> (= (+ 2 3) 5)
t
> (> (* 5 6) (+ 4 5))
t
> (a b c)
Error: invalid function: a
```

Several of the examples above have arguments that are themselves lists, for example the expression (– (+ 3 4) 7). This indicates the composition of functions, in this case “subtract 7 from the *result* of adding 3 to 4”. The word “result” is emphasized here to indicate that the functionis not passed the s-expression “(+ 3 4)” as an argument but rather the result of *evaluating* that expression.

In evaluating a function, Lisp first evaluates its arguments and then applies the function indicated by the first element of the expression to the results of these evaluations. If the arguments are themselves function expressions, Lisp applies this rule recursively to their evaluation. Thus, Lisp allows nested function calls of arbitrary depth. It is important to remember that, by default, Lisp evaluates everything. Lisp uses the convention thatnumbers always evaluate to themselves. If, for example, 5 is typed into the Lisp interpreter, Lisp will respond with 5. Symbols, such as x, may have a value *bound* to them. If a symbol is bound, the binding is returned when the symbol is evaluated (one way in which symbols become bound is in a function call; see Section 13.2). If a symbol is unbound, it is an error to evaluate that symbol.



For example, in evaluating the expression (+ (* 2 3) (* 3 5)), Lisp first evaluates the arguments, (* 2 3) and (* 3 5). In evaluating (* 2 3), Lisp evaluates the arguments 2 and 3, which return their respective arithmetic values; these values are multiplied to yield 6. Similarly, (* 3 5) evaluates to 15. These results are then passed to the top-level addition, which is evaluated, returning 21.

2.1.5 Major Dialects

In computing a dialect is a particular version of a programming language. It is a variation or extension of the language that does not change its intrinsic nature. As it is common for one language to have several dialects, it can become quite difficult for an inexperienced programmer to find the right documentation.

The two major dialects of lisp are:

1. Common Lisp
2. Scheme
3. Clojure

Common Lisp:-

Common Lisp is a new dialect of Lisp, a successor to MacLisp, influenced strongly by Zetalisp and to some extent by Scheme and Interlisp. [4] Common Lisp is a general-purpose, multi-paradigm programming language. It supports a combination of procedural, functional, and object-oriented programming paradigms.As a dynamic programming language, it facilitates evolutionary and incremental software development, with iterative compilation into efficient run-time programs. This incremental development is often done interactively without interrupting the running application.Common Lisp provides some backwards compatibility to Maclisp and to John McCarthy's original Lisp. This allows older Lisp software to be ported to Common Lisp. [1]

Scheme:

Scheme is a general-purpose computer programming language. It is a high-level language, supporting operations on structured data such as strings, lists, and vectors, as well as operations on more traditional data such as numbers and characters. Scheme is a fairly simple language to learn, since it is based on a handful of syntactic forms and semantic concepts and since the interactive nature of most implementations encourages experimentation. Scheme programs are highly portable across versions of the same Scheme implementation on different machines, because machine dependencies are almost completely hidden from the programmer. Scheme evolved from the Lisp language and is considered to be a dialect of Lisp. Scheme inherited from Lisp the treatment of values as first-class objects, several important data types, including symbols and lists, and the representation of programs as objects, among other things. Lexical scoping and block structure are features taken from Algol 60. Scheme was the first Lisp dialect to adopt lexical scoping and block structure, first-class procedures, the treatment of tail calls as jumps, continuations, and lexically scoped syntactic extensions. [12]

While the two languages are similar, Common Lisp includes more specialized constructs, while Scheme includes more general-purpose building blocks out of which such constructs (and others) may be built.

Clojure:

Clojure is a dialect of the Lisp programming language created by Rich Hickey. Clojure is a general-purpose programming language with an emphasis on functional programming. It runs on the Java Virtual Machine, Common Language Runtime, and JavaScript engines. Like other Lisps, Clojure treats code as data and has a macro system. Clojure's focus on programming with immutable values and explicit progression-of-time constructs are intended to facilitate the development of more robust programs, particularly multithreaded ones. [6] Clojure is a compiled language, as it compiles directly to JVM bytecode, yet remains completely dynamic. Every feature supported by Clojure is supported at runtime. Clojure provides access to Java frameworks and libraries, with optional type hints and type inference, so that calls to Java can avoid reflection and enable fast primitive operations.

2.2 Working of Lisp Interpreter

As you might already know, the application takes in an s-expression (where the first element is an operator or function name and the remaining elements are treated as argument). Example (+ 2 2), where the + is the operator and 2, 2 are the arguments. (Already explained above will not go in detail).

So when the application starts, the first thing you will see is the text area where you will type in the Lisp code. Suppose you type in say (* (+ 4 5) 10).

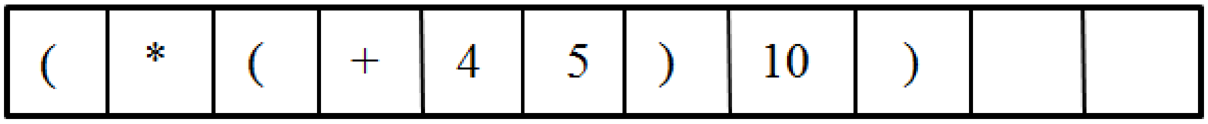


Fig 1: LISP code for Interpreting.

We explain the working of the application by parsing through the above cell diagram. There are nine elements for the input Lisp code we have provided consisting of 2 open brackets, 2 closing brackets, 2 operators, and 3 digits.

2.2.1 Parser

Now this code is passed through a parser (a parser is a program, usually part of a compiler, that receives input in the form of a sequential source program instructions, interactive online commands, or some other defined interface and breaks them up into parts (for example, the nouns (objects), verbs (methods), and their attributes or options) that can then be managed by other programming (for example, other components in a compiler). [6]

The first thing the parser does is to break the Lisp code into atoms and parenthesis, from the above figure we seen four parenthesis (opening and closing brackets) and five atoms (operators and digits). The parsing is done using a for loop.

We store this data in an ArrayList instead of an array because an ArrayList allows easy manipulation of data. We can add, retrieve and remove the elements from the ArrayList by using the methods add(), get() and remove(). Two ArrayList have been created one for storing the code named codeDataAL and the other for storing the expression retrieved from the code named NcodeDataAL (explained later on).

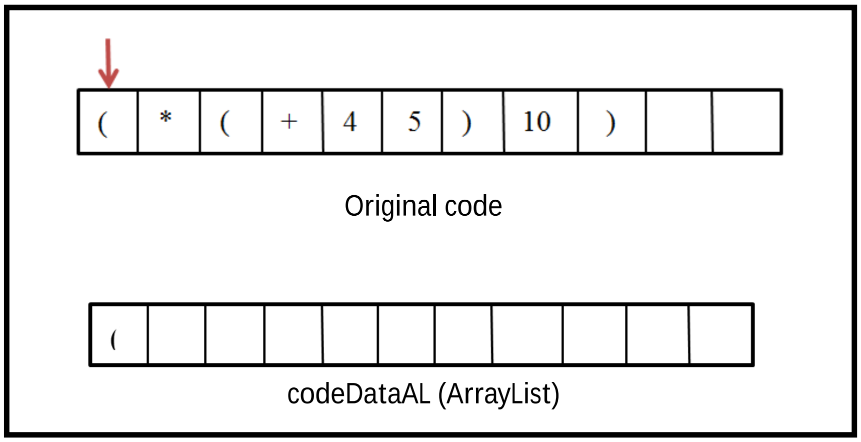


Fig 2: Code to be entered and the arraylist (Cell 0)

As you can see, the parser starts from first cell (red downward arrow) in figure 1 that is cell 0. The cell 0 contains the value '('open bracket. This value will be stored in the codeDataAL using the add() method. It will also check if there is a token/atom before the open bracket (will be explained later).

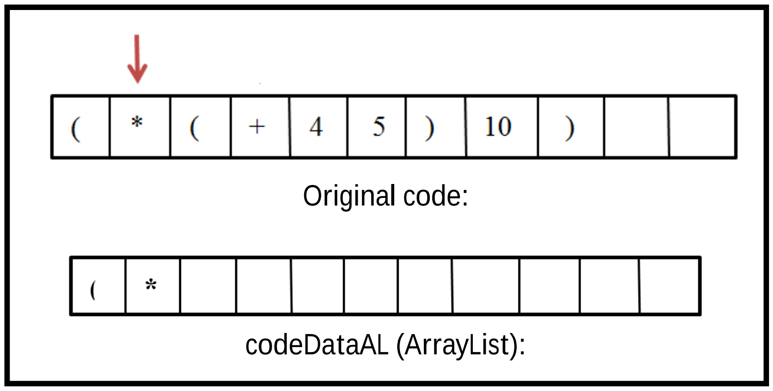


Fig 3: Code to be entered and the arraylist (Cell 1)

Once the open bracket is placed in the arraylist, the value from the next cell is placed in the arraylist. The cell 1 contains the value * (multiplication operator) which is now placed in the arraylist codeDataAL using the add() method.

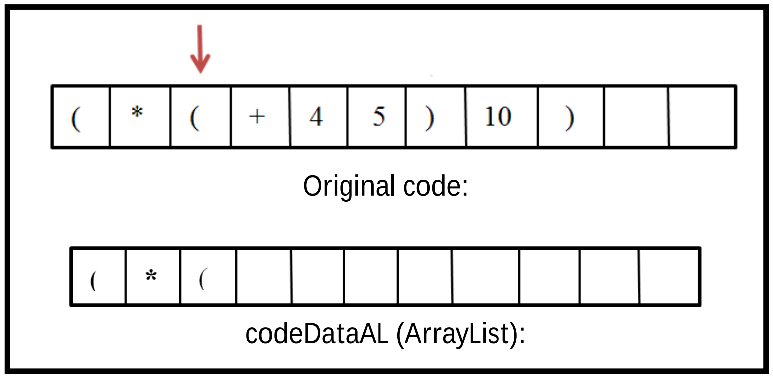


Fig 4: Code to be entered and the arraylist (Cell 2)

In Figure 4, here the open bracket is added to the arraylist.

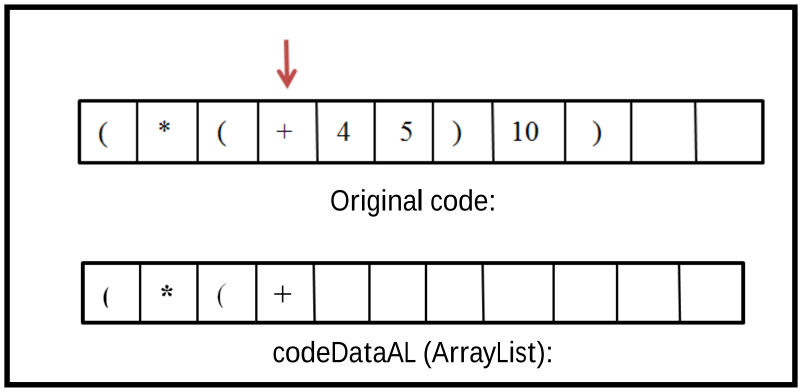


Fig 5: Code to be entered and the arraylist (Cell 3)

Here the operator + is added to the arraylist.

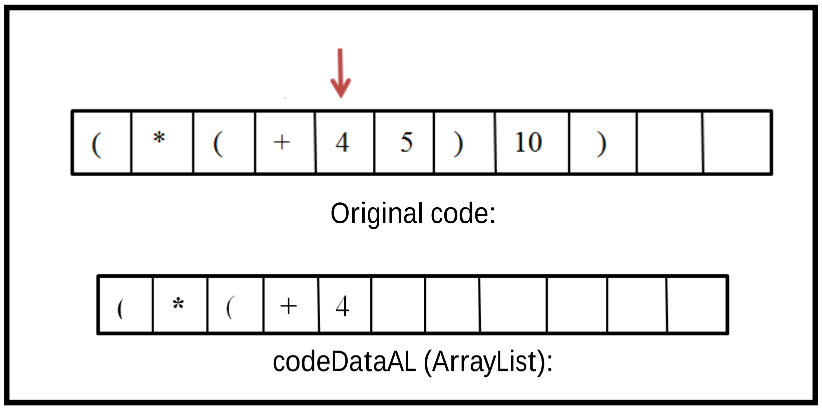


Fig 6: Code to be entered and the arraylist (Cell 4)

Here the digit 4 is added to the arraylist.

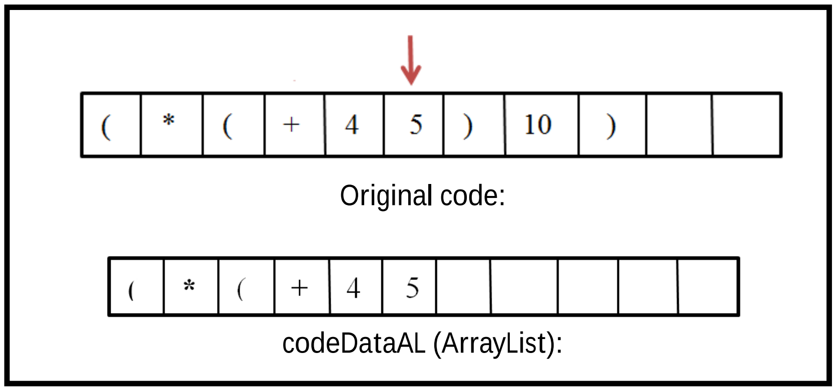


Fig 7: Code to be entered and the arraylist (Cell 5)

Here the digit 5 is added to the arraylist.

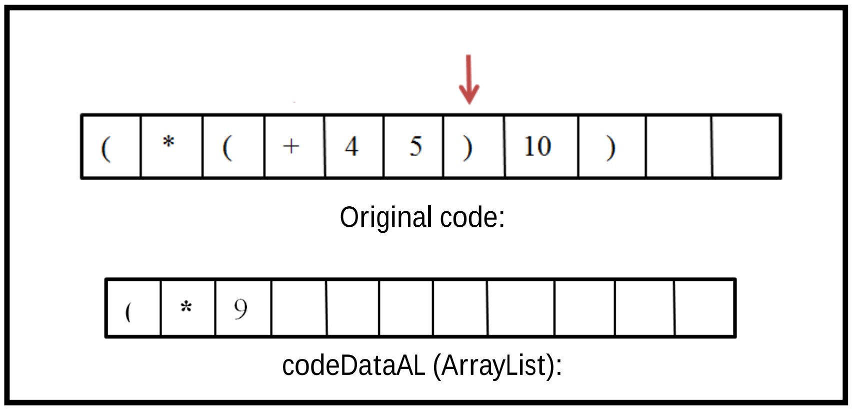


Fig 8: Code to be entered and the arraylist (Cell 6)

Once it reaches a close bracket, the parser now finds the last open bracket it then takes the atoms/tokens in between these brackets send it for interpreting. The atoms/tokens in between the brackets are termed as an expression. The result from the interpreting is stored in the arraylist at a position where the open bracket would have been placed.

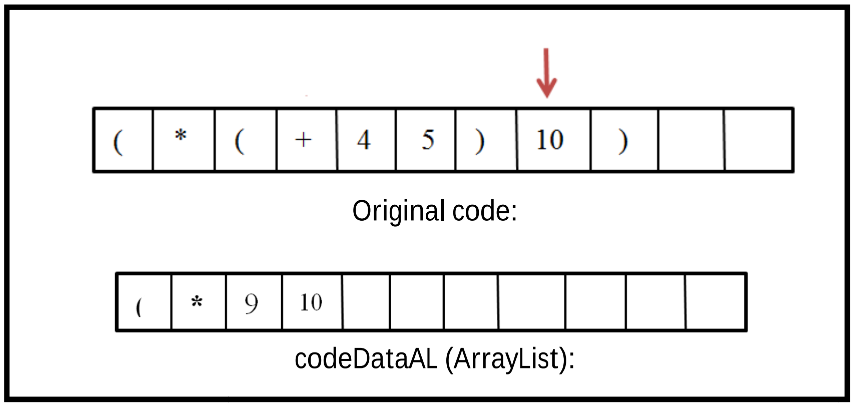


Fig 9: Code to be entered and the arraylist (Cell 7)

One question that may arise is. How multi character token is extracted? Since a loop is used for parsing and to be precise a for-loop is used, the interpreter stores the character in a StringBuilder (used to create mutable i.e. modified string) variable one character at a time. It stops when it reaches a whitespace character or an open/close bracket.

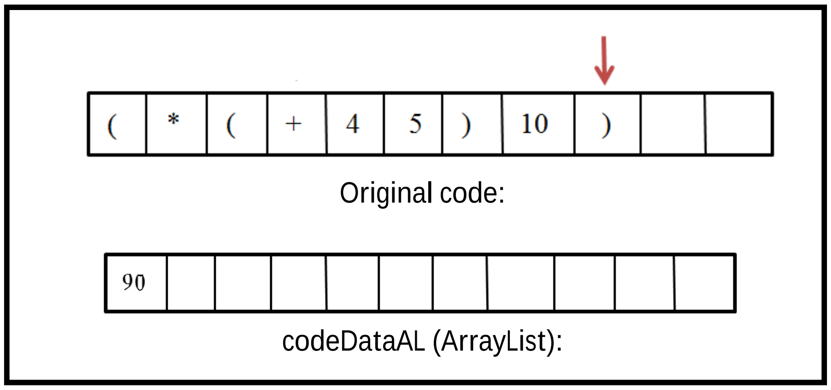


Fig 10: Code to be entered and the arraylist (Cell 8)

Similarly once it reaches the close bracket, the parser backtracks to find the last location of the open bracket. It then takes the tokens in between the open/ close brackets and sends it for interpreting. The result is then placed in the arraylist at location where the open bracket was supposed to be added.

As we can see figure there is no next token for it to parse which means the code to be executed has reached its end. The result that was placed in the arraylist is now displayed.

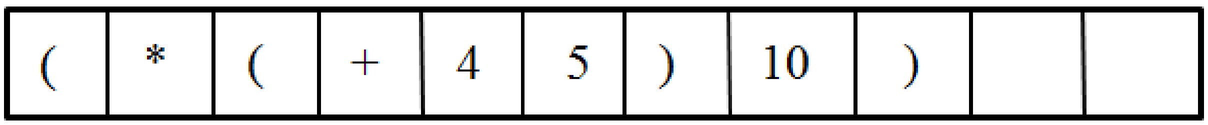
2.2.2 Executing the Expression

Once the LISP code is parsed, the output of the parser is one or more s-expressions. This s-expression usually has the form of (operator/ function arg1 arg2 … argn), which are then executed to get the necessary output.

Execution of the s-expression is performed in the following way:

1. We split the expression into tokens; these tokens may include operators or functions and arguments either one or more.
2. The next thing the interpreter does is that is takes the operator or function and checks it with already defined inbuilt rules.(Example suppose your LISP code is say (+ 2 3) now the operator here is '+'. This operator is then matched with the pre-defined rule say sum/add the arguments.
3. The result is then sent back to the parser where it is stored and the next expression if any is allowed through.

Let us take the previous example that is:



(	*	(	+	4	5	)	10	)		

Fig 11: LISP code for Interpreting.

From the above figure we see two s-expressions that are:

1. (+ 4 5)
2. The second one is hard to find but if you remove the first expression from the above figure, you will get the second expression that is (* n 10). We have used n because the first expression is still not executed,where n is (+ 4 5).

3. Fabrication and Details

3.1 User Interface

3.1.1 Main Window

- **Open File:**

This will allow the user to open a previously saved file on the editor. To save user time, the user can type in the name of the file and the file with that name is displayed. The user can then select that file.

- **Save File:**

This allows the user to save his code that he has typed for later use, user has to give a name for the file to be saved and click the save button. The file gets saved in his sdcardLISPApp folder.

- **Settings:**
  - **Font Size**

    Suppose the user feels like the font size of the app is too small, the application provides the user to change the font size form 14(default size) to 18.

  - **Line Numbering**

    Line numbering can be toggled either on or off, depending on the user preference.

- **Samples:**

Contains sample codes of functions that are implemented in this application.

- **Documentation**

Contains documentation for the functions implemented in this application.

- **Last Activity**

Allows the user to access the last code he/she typed.

- **Clear**

The user can delete the code in a single instance.

- **Execute**

Used for executing the typed code.

3.2 Lisp Interpreter Implemented

Implementation of Lisp Interpreter is tested with following examples:

1. Arithmetic Operations

a. Addition

(+ 4 5)

Add two or more numbers.

b. Subtraction

(- 10 2)

Subtract two or more numbers.

c. Multiplication

(* 5 5)

Multiply two or more numbers.

d. Division

(/ 10 2)

Divide two or more numbers.

2. Comparison Operations

a. Equal

(= 5 5)

Checks if the values of the operands are all equal or not, if yes then condition becomes true.

b. Greater-Than

(> 4 5)

Check if the values of the operands are monotonically decreasing.

c. Less-Than

(< 4 5)

Check if the values of the operands are monotonically increasing.

d. Greater-Than-Equal-To

(>= 5 6)

Check if the value of any left operand is greater than or equal to the value of next right operand, if yes then condition becomes true.

e. Less-Than-Equal-To

(<= 6 8)

Check if the value of any left operand is less than or equal to the value of its right operand, if yes then condition becomes true.

3. Conditional Statement

a. If-else

(if t 5 6)

The first argument of if determines whether the second or third argument will be executed.

b. <u>when</u>

(when t 3)

An “if” statement which lacks either a then or an else clause can be written using the when special form.

c. <u>unless</u>

(unless t 3)

An “if” statement which lacks either a then or an else clause can be written using the unless special form.

4. List Manipulation function

a. Car / first

(car ‘(1 2 3 4))

CAR function returns the first element in a list.

b. Cdr / rest

(cdr ‘(1 2 3 4))

CDR [pronounced "could-er"] is the complement of CAR in that the result of CDR is the "rest" of the list.

5. Output function

a. Print

(print ‘Hello)

Some functions can cause output. The simplest one is print, which prints its argument and then returns it.

6. **Assignment**
   a. Setq

(setqlst ‘(1 2 3 4))

Allow the value of a variable to be altered.

7. **User defined functions**
   a. defun

(defunfn (n1 n2)

    (+ n1 n2)

)

Use it to define your own function for later use.

b. fn

(fn 4 5)

Syntax for calling the above function.

8. Loop

a. Loop-when

```
(setqlst '(1 2 3 4 5 6))

(loop

    (when (eq (first lst) 6)(return))

    (print 'Hello)

    (setqlst (rest lst))

)
```

The Loop macro is different than most Lisp expressions in having a complex internal syntax that is more similar to programming languages like C or Pascal. So you need to read Loop expressions with half of your brain in Lisp mode, and the other half in Pascal mode.

Think of Loop expressions as having four parts: expressions that set up variables that will be iterated, expressions that conditionally terminate the iteration, expressions that do something on each of the iteration, and expressions that do something right before the Loop exits. In addition, Loop expressions can return a value. It is very rare to use all of these parts in a given Loop expression, but you can combine them in many ways.

4. Testing

Testing is an important phase in the development life cycle of the product this is the phase where the errors from all phases are detected. During the testing, the program to be tested is executed with a set of test cases and the output of the program for the test cases are evaluated to determine whether the program is performing as expected. Errors are found and corrected by using the following testing steps and corrections are recorded for future references. Thus, a series of testing is performed on the system before it is ready for implementation. During testing, the system is used experimentally to ensure that the software does not fail, i.e., that it will run according to its specifications and in the way users expect it to.

Special test data is input for processing (test plan) and the results are examined to locate unexpected results. A limited number of users or a team may also be allowed to use the system so analysts can see whether they try to use it in unexpected ways. There is merit or advantage in this approach; cause it is hard to see owns mistakes and a new eye can discover obvious errors much faster than the person who has read, re-read the code many times.

The Testing phase is done from the perspective of the system provider. Because it is nearly impossible to duplicate every possible customer's environment and because systems are released wit yet to be discovered errors, the customer plays an important, through reluctant, role in testing.

Tests Performed are as follows:

1. Unit Testing.
2. Integration Testing.
3. White Box Testing.
4. Black Box Testing.

1. Unit Testing

The primary goal of unit testing is to take the smallest piece of testable software in the application, (this is typically a method or a class, depending on scale) isolate it from the remainder of the code, and determine whether it behaves exactly as you expect. Each unit is tested separately before integrating them into modules to test the interfaces between modules. Unit testing has proven its value in that a large percentage of defects are identified during its use.

Test Case Number:	1
Name of the test:	Save file
Item being tested:	SaveFileActivity.java
Sample Input:	LISP code and File name
Expected Output:	Saved file on device
Actual Output:	File is saved on sd card
Remarks:	Pass

Test Case Number:	2
Name of the test:	Open file
Item being tested:	OpenFileActivity.java
Sample Input:	File name to be opened
Expected Output:	Open file from sd card and display code on screen
Actual Output:	Display of code on screen
Remarks:	Pass

2. Integration Testing

Integration testing is a logical extension of unit testing. In its simplest form, two units that have already been tested are combined into a component and the interface between them is tested. A component, in this sense, refers to an integrated aggregate of more than one unit. In a realistic scenario, many units are combined into components, which are in turn aggregated into even larger parts of the program. The idea is to test combinations of pieces and eventually expand the process to test your modules with those of other groups. Eventually all the modules making up a process are tested together. Beyond that, if the program is composed of more than one process, they should be tested in pairs rather than all at once.

Test Case Number:	1
Name of the test:	Interpreter
Item being tested:	Entire Lisp Interpreter
Sample Input:	LISP code
Expected Output:	Extraction of s-expression from code and interpreting it.
Actual Output:	Interpreted code result
Remarks:	Pass

3. White Box Testing

White Box Testing (also known as Clear Box Testing, Open Box Testing, Glass Box Testing, Transparent Box Testing, Code-Based Testing or Structural Testing) is a software testing method in which the internal structure/ design/ implementation of the item being tested is known to the tester. The tester chooses inputs to exercise paths through the code and determines the appropriate outputs. Programming know-how and the implementation knowledge is essential. White box testing is testing beyond the user interface and into the nitty-gritty of a system. This method is named so because the software program, in the eyes of the tester, is like a white/ transparent box; inside which one clearly sees.

Test Case Number:	1
Name of the test:	Check the working of any one function of LISP
Item being tested:	Function addition
Sample Input:	(+ 4 5)
Expected Output:	The code should pass through appropriate classes returning us the result
Actual Output:	9
Remarks:	Pass

4. Black Box Testing

Black Box Testing, also known as Behavioral Testing, is a software testing method in which the internal structure/ design/ implementation of the item being tested is not known to the tester. These tests can be functional or non-functional, though usually functional. This method is named so because the software program, in the eyes of the tester, is like a black box; inside which one cannot see. This method attempts to find errors in the following categories:

- Incorrect or missing functions
- Interface errors
- Errors in data structures or external database access
- Behavior or performance errors
- Initialization and termination errors

Test Case Number:	1
Name of the test:	Find Error
Item being tested:	Interpreter
Sample Input:	Function which is not implemented
Expected Output:	The input must be accepted and the processing should take place
Actual Output:	Error no such function
Remarks:	Pass

5. Future Enhancement

LISP is a vast language; there are still many more functions to implement in our application. The function done so far does not even scratch the surface. The interpreter is open enough to handle these new functions quite easily, requiring minor changes and all.

Some of the areas where the application can be further improved are as follows:

- Implementing new functions. The application has been made in such a way that, new functions of LISP can be implemented quite easily.
- Improved User Interface (like options to change themes and so on).
- Automatic text indentation.
- The application can be further developed with proper code validation; currently the application validates only alphabets and numbers.

6. Conclusion

The Lisp Interpreter has been implemented for android mobile devices. Application and its implementation have been discussed in the thesis. The thesis provides a complete insight into the actual working and practical implementation of various Lisp functions. This work will help in better understanding of Lisp programming language and can also be used as a tutorial.

UML Diagrams

Class Diagram

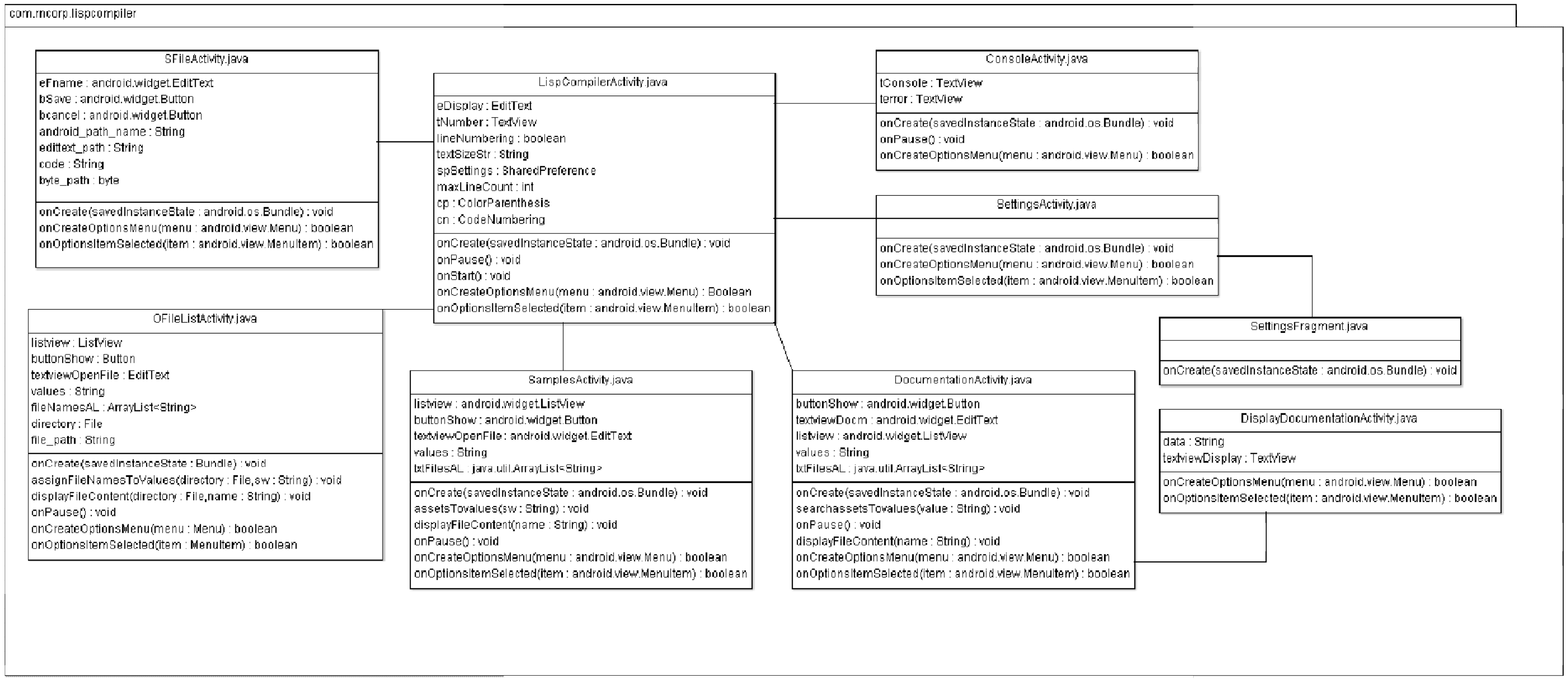
The class diagram is the main building block of object oriented modeling.It is used both for general **conceptual modeling** of the systematics of the application, and for **detailed modeling** translating the models into programming code.Class diagrams can also be used for data modeling. The classes in a class diagram represent both the main objects, interactions in the application and the classes to be programmed.

Classes are represented with boxes which contain three parts:

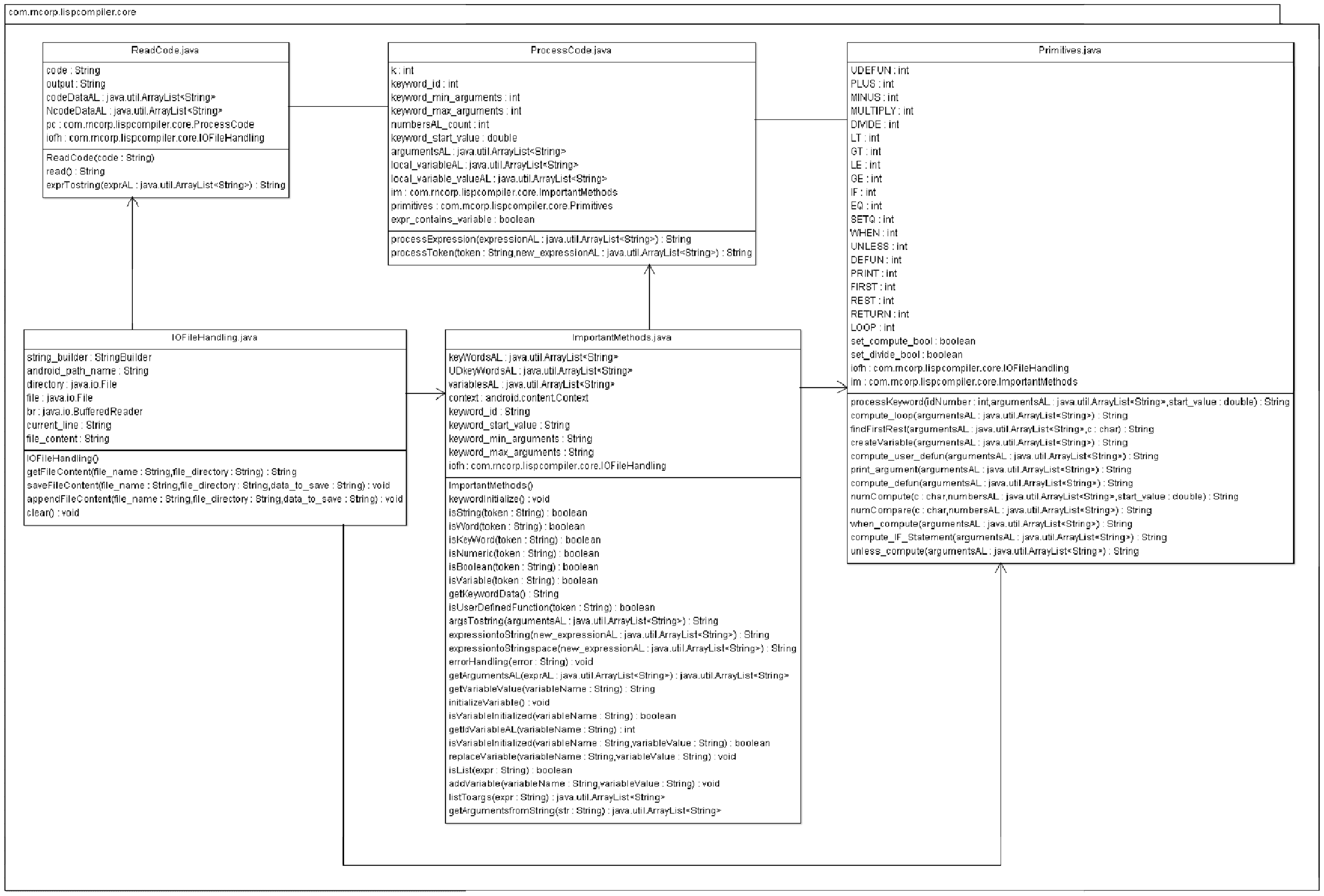
- The top part contains the name of the class. It is printed in bold and centered, and the first letter is capitalized.
- The middle part contains the attributes of the class. They are left-aligned and the first letter is lowercase.
- The bottom part contains the methods the class can execute. They are also left-aligned and the first letter is lowercase.

In the design of a system, a number of classes are identified and grouped together in a class diagram which helps to determine the static relations between those objects. With detailed modeling, the classes of the conceptual design are often split into a number of subclasses.In order to further describe the behavior of systems, these class diagrams can be complemented by a state diagram or UML state machine.

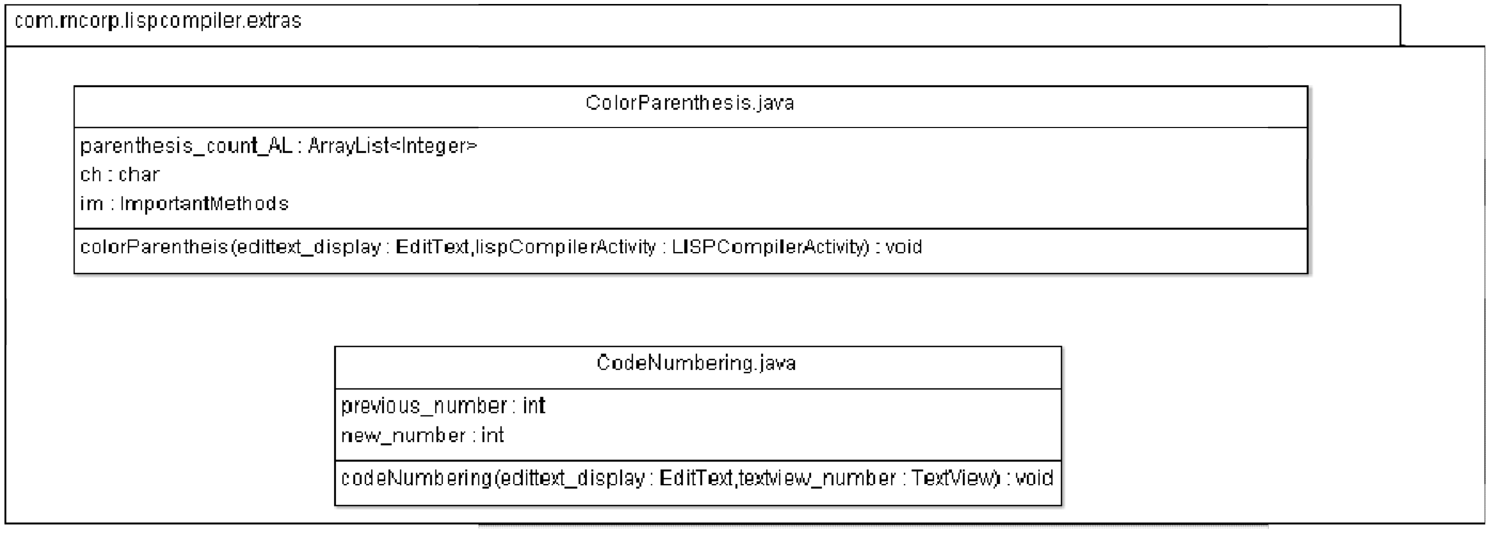
Class Diagram for the pakage com.rncorp.lispcompiler


com.rncorp.lispcompiler
SFileActivity.java
eFname : android.widget.EditText
bSave : android.widget.Button
bcancel : android.widget.Button
android_path_name : String
edittext_path : String
code : String
byte_path : byte
onCreate(savedInstanceState : android.os.Bundle) : void
onCreateOptionsMenu(menu : android.view.Menu) : boolean
onOptionsItemSelected(item : android.view.MenuItem) : boolean
LispCompilerActivity.java
eDisplay : EditText
tNumber : TextView
lineNumbering : boolean
textSizeStr : String
spSettings : SharedPreference
maxLineCount : int
cp : ColorParenthesis
cn : CodeNumbering
onCreate(savedInstanceState : android.os.Bundle) : void
onPause() : void
onStart() : void
onCreateOptionsMenu(menu : android.view.Menu) : Boolean
onOptionsItemSelected(item : android.view.MenuItem) : boolean
ConsoleActivity.java
tConsole : TextView
terror : TextView
onCreate(savedInstanceState : android.os.Bundle) : void
onPause() : void
onCreateOptionsMenu(menu : android.view.Menu) : boolean
SettingsActivity.java
onCreate(savedInstanceState : android.os.Bundle) : void
onCreateOptionsMenu(menu : android.view.Menu) : boolean
onOptionsItemSelected(item : android.view.MenuItem) : boolean
SettingsFragment.java
onCreate(savedInstanceState : android.os.Bundle) : void
OFileListActivity.java
listview : ListView
buttonShow : Button
textviewOpenFile : EditText
values : String
fileNamesAL : ArrayList<String>
directory : File
file_path : String
onCreate(savedInstanceState : Bundle) : void
assignFileNamesToValues(directory : File,sw : String) : void
displayFileContent(directory : File,name : String) : void
onPause() : void
onCreateOptionsMenu(menu : Menu) : boolean
onOptionsItemSelected(item : MenuItem) : boolean
SamplesActivity.java
listview : android.widget.ListView
buttonShow : android.widget.Button
textviewOpenFile : android.widget.EditText
values : String
txtFilesAL : java.util.ArrayList<String>
onCreate(savedInstanceState : android.os.Bundle) : void
assetsTovalues(sw : String) : void
displayFileContent(name : String) : void
onPause() : void
onCreateOptionsMenu(menu : android.view.Menu) : boolean
onOptionsItemSelected(item : android.view.MenuItem) : boolean
DocumentationActivity.java
buttonShow : android.widget.Button
textviewDocm : android.widget.EditText
listview : android.widget.ListView
values : String
txtFilesAL : java.util.ArrayList<String>
onCreate(savedInstanceState : android.os.Bundle) : void
searchassetsTovalues(value : String) : void
onPause() : void
displayFileContent(name : String) : void
onCreateOptionsMenu(menu : android.view.Menu) : boolean
onOptionsItemSelected(item : android.view.MenuItem) : boolean
DisplayDocumentationActivity.java
data : String
textviewDisplay : TextView
onCreateOptionsMenu(menu : android.view.Menu) : boolean
onOptionsItemSelected(item : android.view.MenuItem) : boolean

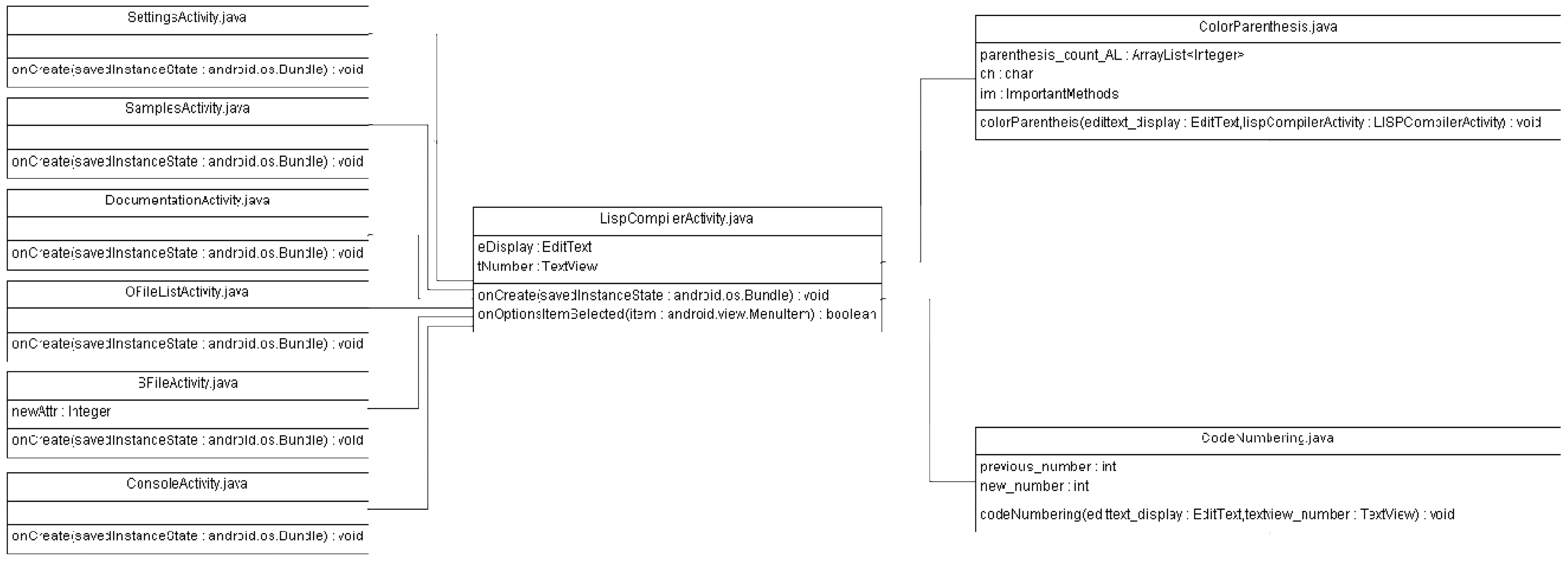
Class Diagram for the pakage com.rncorp.lispcompiler.core


com.rncorp.lispcompiler.core
ReadCode.java
code : String
output : String
codeDataAL : java.util.ArrayList<String>
NcodeDataAL : java.util.ArrayList<String>
pc : com.rncorp.lispcompiler.core.ProcessCode
iofh : com.rncorp.lispcompiler.core.IOFileHandling
ReadCode(code : String)
read() : String
exprTostring(exprAL : java.util.ArrayList<String>) : String
ProcessCode.java
k : int
keyword_id : int
keyword_min_arguments : int
keyword_max_arguments : int
numbersAL_count : int
keyword_start_value : double
argumentsAL : java.util.ArrayList<String>
local_variableAL : java.util.ArrayList<String>
local_variable_valueAL : java.util.ArrayList<String>
im : com.rncorp.lispcompiler.core.ImportantMethods
primitives : com.rncorp.lispcompiler.core.Primitives
expr_contains_variable : boolean
processExpression(expressionAL : java.util.ArrayList<String>) : String
processToken(token : String,new_expressionAL : java.util.ArrayList<String>) : String
Primitives.java
UDEFUN : int
PLUS : int
MINUS : int
MULTIPLY : int
DIVIDE : int
LT : int
GT : int
LE : int
GE : int
IF : int
EQ : int
SETQ : int
WHEN : int
UNLESS : int
DEFUN : int
PRINT : int
FIRST : int
REST : int
RETURN : int
LOOP : int
set_compute_bool : boolean
set_divide_bool : boolean
iofh : com.rncorp.lispcompiler.core.IOFileHandling
im : com.rncorp.lispcompiler.core.ImportantMethods
processKeyword(idNumber : int,argumentsAL : java.util.ArrayList<String>,start_value : double) : String
compute_loop(argumentsAL : java.util.ArrayList<String>) : String
findFirstRest(argumentsAL : java.util.ArrayList<String>,c : char) : String
createVariable(argumentsAL : java.util.ArrayList<String>) : String
compute_user_defun(argumentsAL : java.util.ArrayList<String>) : String
print_argument(argumentsAL : java.util.ArrayList<String>) : String
compute_defun(argumentsAL : java.util.ArrayList<String>) : String
numCompute(c : char,numbersAL : java.util.ArrayList<String>,start_value : double) : String
numCompare(c : char,numbersAL : java.util.ArrayList<String>) : String
when_compute(argumentsAL : java.util.ArrayList<String>) : String
compute_IF_Statement(argumentsAL : java.util.ArrayList<String>) : String
unless_compute(argumentsAL : java.util.ArrayList<String>) : String
IOFileHandling.java
string_builder : StringBuilder
android_path_name : String
directory : java.io.File
file : java.io.File
br : java.io.BufferedReader
current_line : String
file_content : String
IOFileHandling()
getFileContent(file_name : String,file_directory : String) : String
saveFileContent(file_name : String,file_directory : String,data_to_save : String) : void
appendFileContent(file_name : String,file_directory : String,data_to_save : String) : void
clear() : void
ImportantMethods.java
keyWordsAL : java.util.ArrayList<String>
UDkeyWordsAL : java.util.ArrayList<String>
variablesAL : java.util.ArrayList<String>
context : android.content.Context
keyword_id : String
keyword_start_value : String
keyword_min_arguments : String
keyword_max_arguments : String
iofh : com.rncorp.lispcompiler.core.IOFileHandling
ImportantMethods()
keywordInitialize() : void
isString(token : String) : boolean
isWord(token : String) : boolean
isKeyWord(token : String) : boolean
isNumeric(token : String) : boolean
isBoolean(token : String) : boolean
isVariable(token : String) : boolean
getKeywordData() : String
isUserDefinedFunction(token : String) : boolean
argsTostring(argumentsAL : java.util.ArrayList<String>) : String
expressiontoString(new_expressionAL : java.util.ArrayList<String>) : String
expressiontoStringspace(new_expressionAL : java.util.ArrayList<String>) : String
errorHandling(error : String) : void
getArgumentsAL(exprAL : java.util.ArrayList<String>) : java.util.ArrayList<String>
getVariableValue(variableName : String) : String
initializeVariable() : void
isVariableInitialized(variableName : String) : boolean
getIdVariableAL(variableName : String) : int
isVariableInitialized(variableName : String,variableValue : String) : boolean
replaceVariable(variableName : String,variableValue : String) : void
isList(expr : String) : boolean
addVariable(variableName : String,variableValue : String) : void
listToargs(expr : String) : java.util.ArrayList<String>
getArgumentsfromString(str : String) : java.util.ArrayList<String>

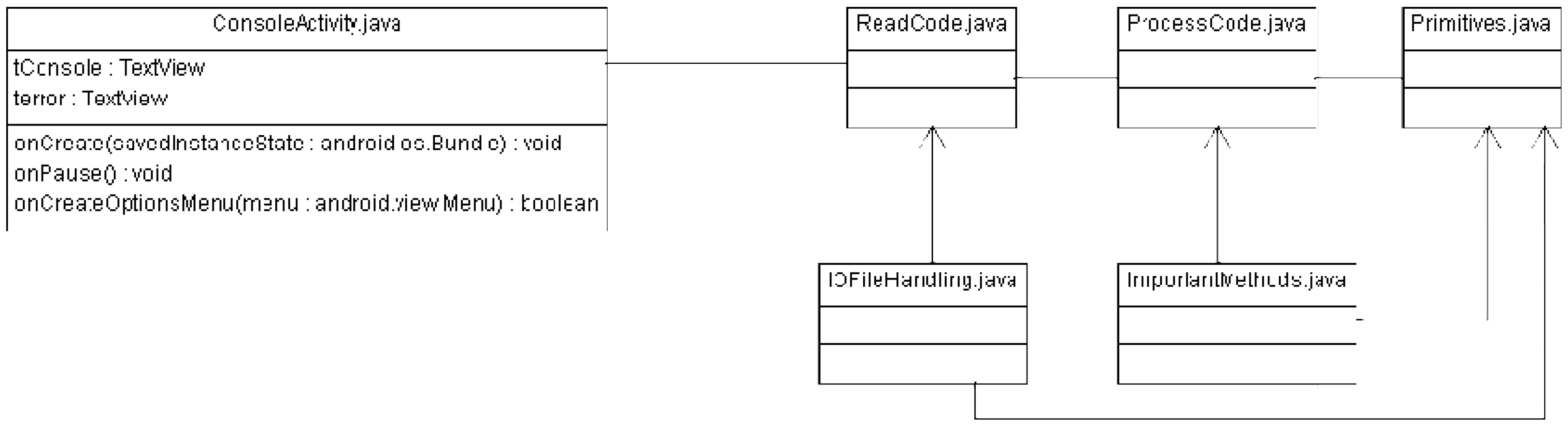
Class Diagram for the pakage com.rncorp.lispcompiler.extras


com.rncorp.lispcompiler.extras
ColorParenthesis.java
parenthesis_count_AL : ArrayList<Integer>
ch : char
im : ImportantMethods
colorParentheis(edittext_display : EditText,lispCompilerActivity : LISPCompilerActivity) : void
CodeNumbering.java
previous_number : int
new_number : int
codeNumbering(edittext_display : EditText,textview_number : TextView) : void

Class Diagram for the class LispCompilerActivity


SettingsActivity.java
onCreate(savedInstanceState : android.os.Bundle) : void
SamplesActivity.java
onCreate(savedInstanceState : android.os.Bundle) : void
DocumentationActivity.java
onCreate(savedInstanceState : android.os.Bundle) : void
OFileListActivity.java
onCreate(savedInstanceState : android.os.Bundle) : void
SFileActivity.java
newAttr : Integer
onCreate(savedInstanceState : android.os.Bundle) : void
ConsoleActivity.java
onCreate(savedInstanceState : android.os.Bundle) : void
LispCompilerActivity.java
eDisplay : EditText
tNumber : TextView
onCreate(savedInstanceState : android.os.Bundle) : void
onOptionsItemSelected(item : android.view.MenuItem) : boolean
ColorParenthesis.java
parenthesis_count_AL : ArrayList<Integer>
ch : char
im : ImportantMethods
colorParentheis(edittext_display : EditText,lispCompilerActivity : LISPCompilerActivity) : void
CodeNumbering.java
previous_number : int
new_number : int
codeNumbering(edittext_display : EditText,textview_number : TextView) : void

Class Diagram for the class ConsoleActivity

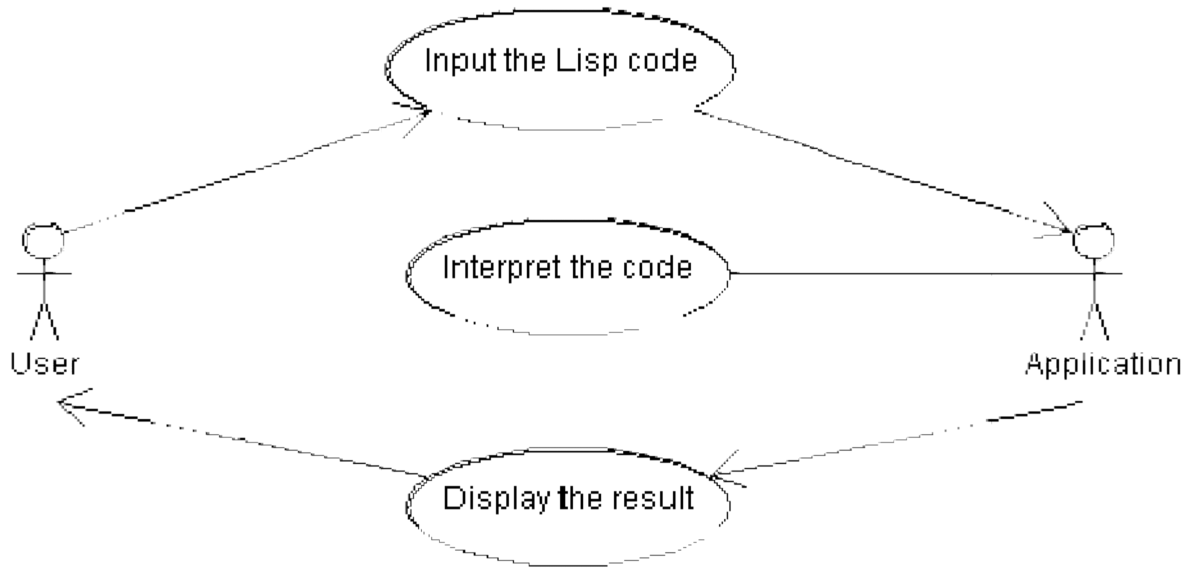


Use Case Diagram

UML use case diagrams overview the usage requirements for a system.They are useful for presentations to management and/or project stakeholders, but for actual development you will find that use cases provide significantly more value because they describe "the meat" of the actual requirements.

Use case diagrams depict:

- **Use cases:** A use case describes a sequence of actions that provide something of measurable value to an actor and is drawn as a horizontal ellipse.
- **Actors:** An actor is a person, organization, or external system that plays a role in one or more interactions with your system. Actors are drawn as stick figures.
- **Associations:** Associations between actors and use cases are indicated in use case diagrams by solid lines.An association exists whenever an actor is involved with an interaction described by a use case.Associations are modeled as lines connecting use cases and actors to one another, with an optional arrowhead on one end of the line.The arrowhead is often used to indicating the direction of the initial invocation of the relationship or to indicate the primary actor within the use case.The arrowheads are typically confused with data flow and as a result I avoid their use.


Input the Lisp code
Interpret the code
Display the result
User
Application

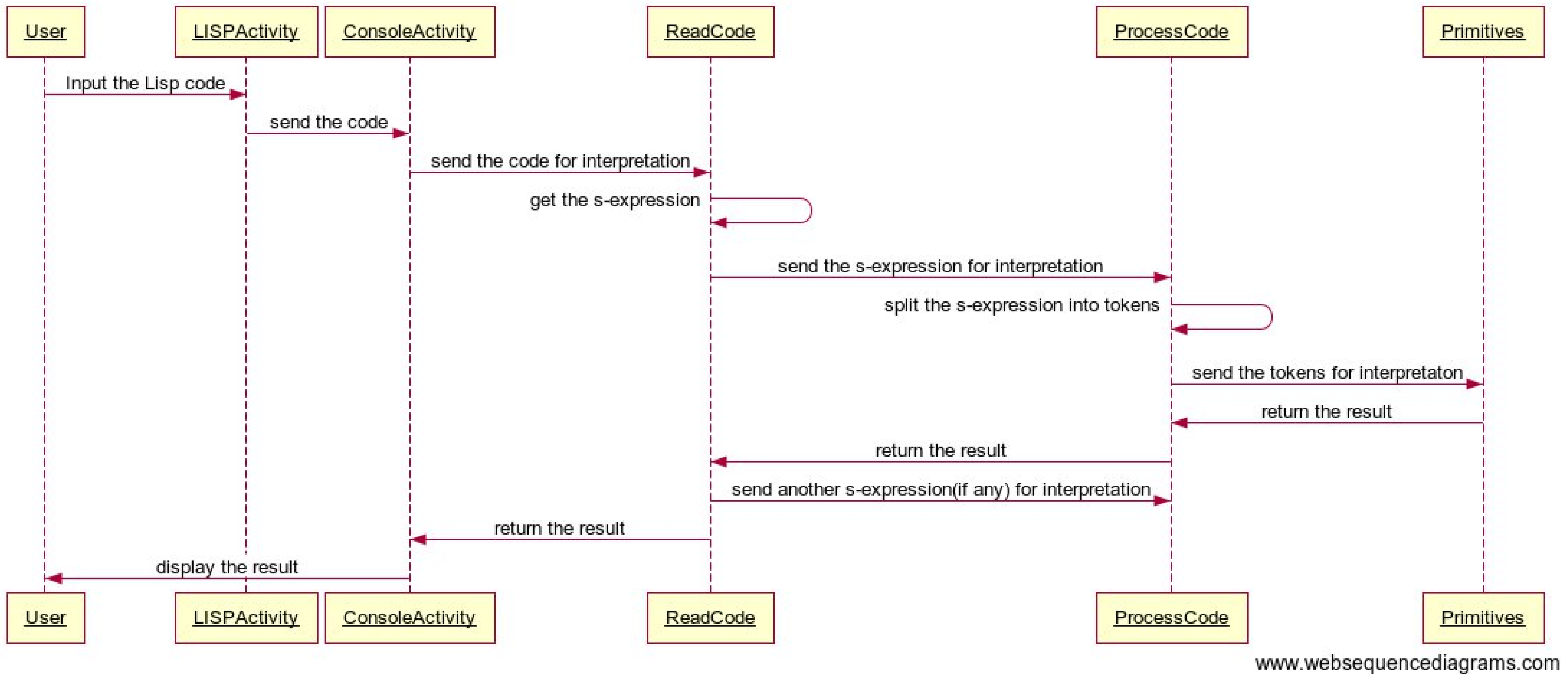
Sequence Diagram

UML sequence diagrams model the flow of logic within your system in a visual manner, enabling you both to document and validate your logic, and are commonly used for both analysis and design purposes.Sequence diagrams are the most popular UML artifact for dynamic modeling, which focuses on identifying the behavior within your system. Other dynamic modeling techniques include activity diagramming, communication diagramming, timing diagramming, and interaction overview diagramming.Sequence diagrams, along with class diagrams and physical data models are in my opinion the most important design-level models for modern business application development.

Sequence diagrams are typically used to model:

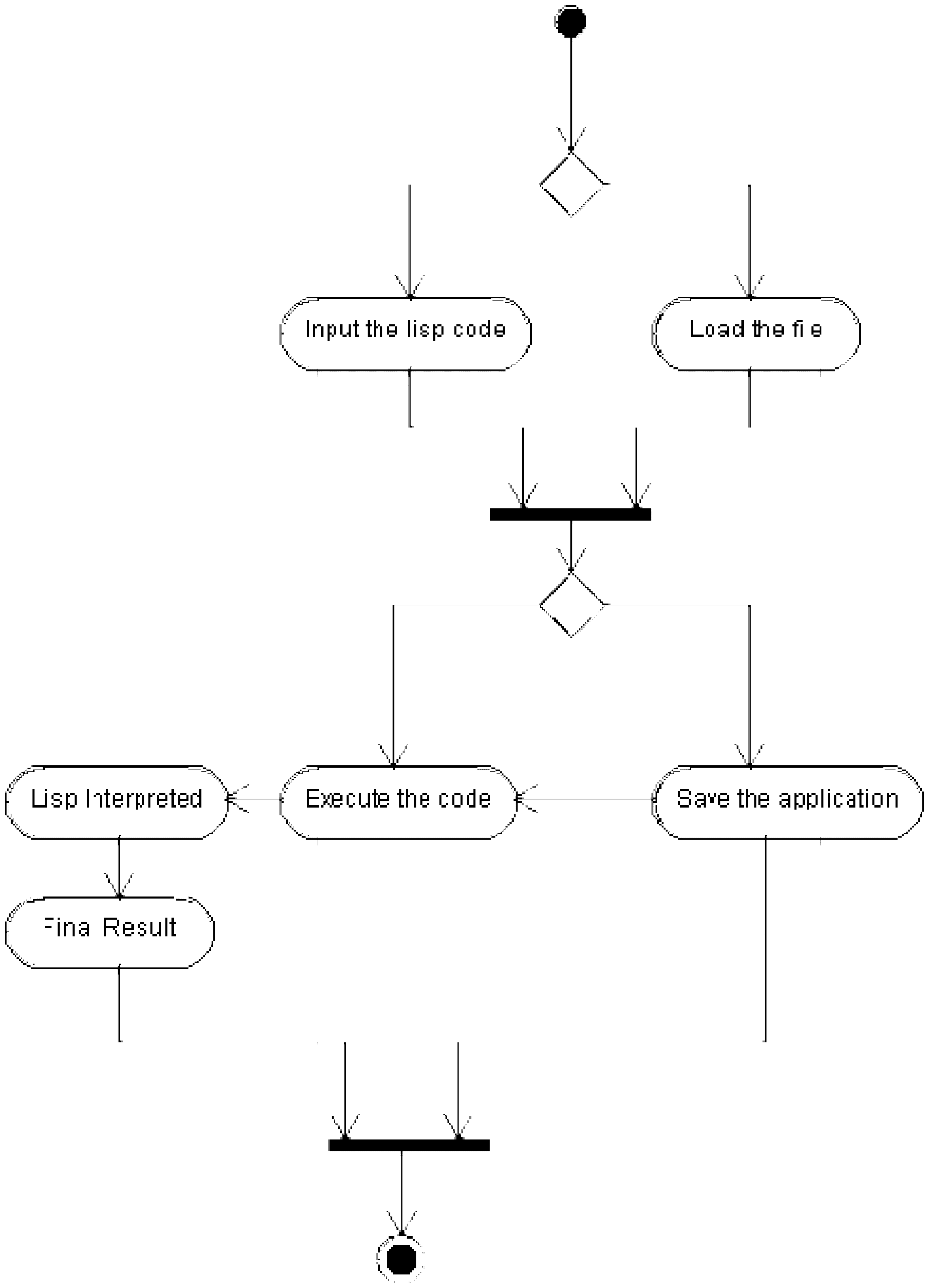
- **Usage scenarios:** A usage scenario is a description of a potential way your system is used.The logic of a usage scenario may be part of a use case, perhaps an alternate course.It may also be one entire pass through a use case, such as the logic described by the basic course of action or a portion of the basic course of action, plus one or more alternate scenarios.The logic of a usage scenario may also be a pass through the logic contained in several use cases. For example, a student enrolls in the university, and then immediately enrolls in three seminars.
- **The logic of methods:** Sequence diagrams can be used to explore the logic of a complex operation, function, or procedure. One way to think of sequence diagrams, particularly highly detailed diagrams, is as visual object code.
- **The logic of services:** A service is effectively a high-level method, often one that can be invoked by a wide variety of clients. This includes web-services as well as business transactions implemented by a variety of technologies such as CICS/COBOL or CORBA-compliant object request brokers (ORBs).

Sequence Diagram for LISP code interpretation.



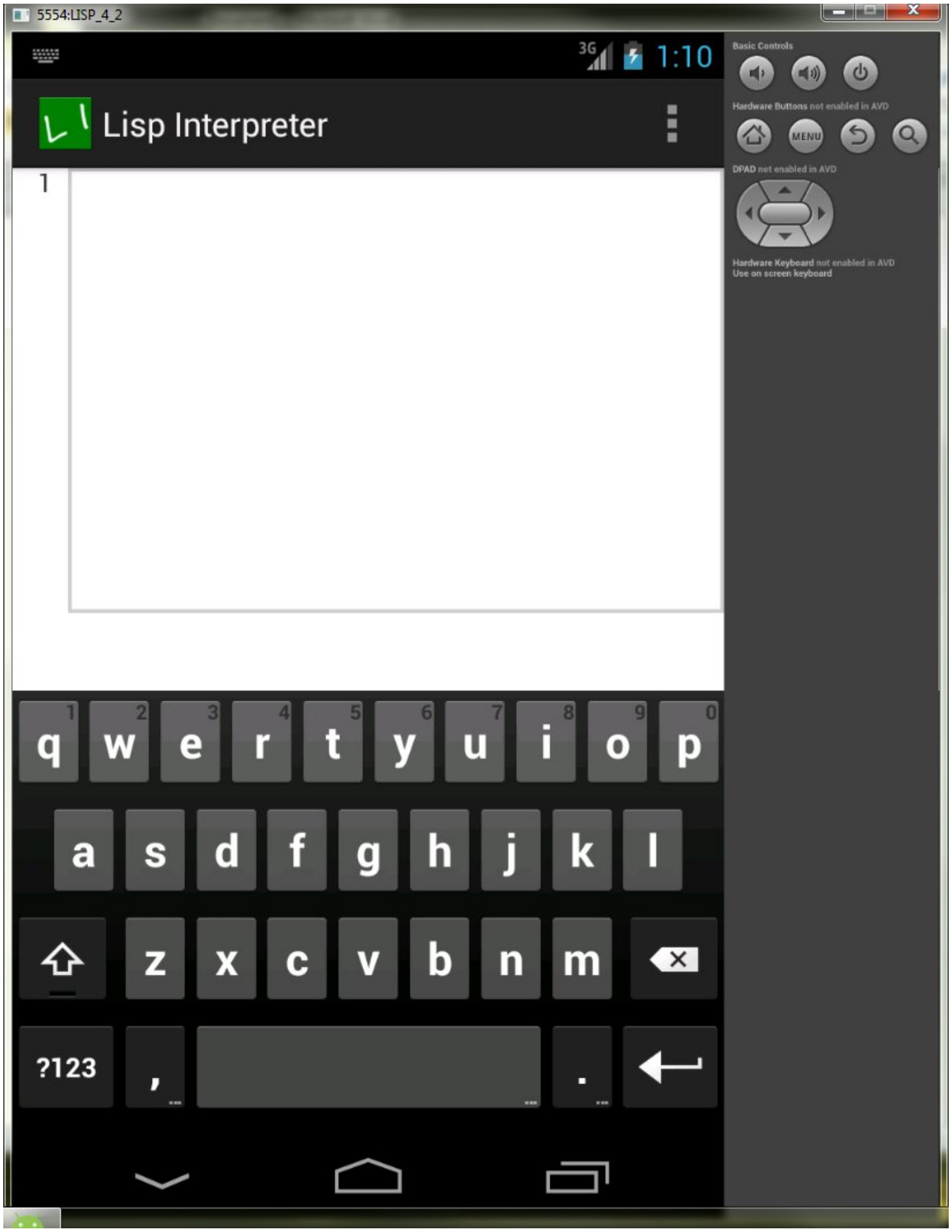
Activity Diagram

Activity diagrams are a loosely defined diagram technique for showing workflows ofstepwise activities and actions, with support for choice, iteration and concurrency. In theUnified Modeling Language, activity diagrams can be used to describe the business andoperational step-by-step workflows of components in a system. An activity diagram showsthe overall flow of control. In SysML the activity diagram has been extended to indicateflows among steps that convey physical element (e.g., gasoline) or energy (e.g., torque,pressure). Additional changes allow the diagram to better support continuous behaviors andcontinuous data flows. In UML 1.x, an activity diagram is a variation of the UML Statediagram in which the "states" represent activities, and the transitions represent the completionof those activities.

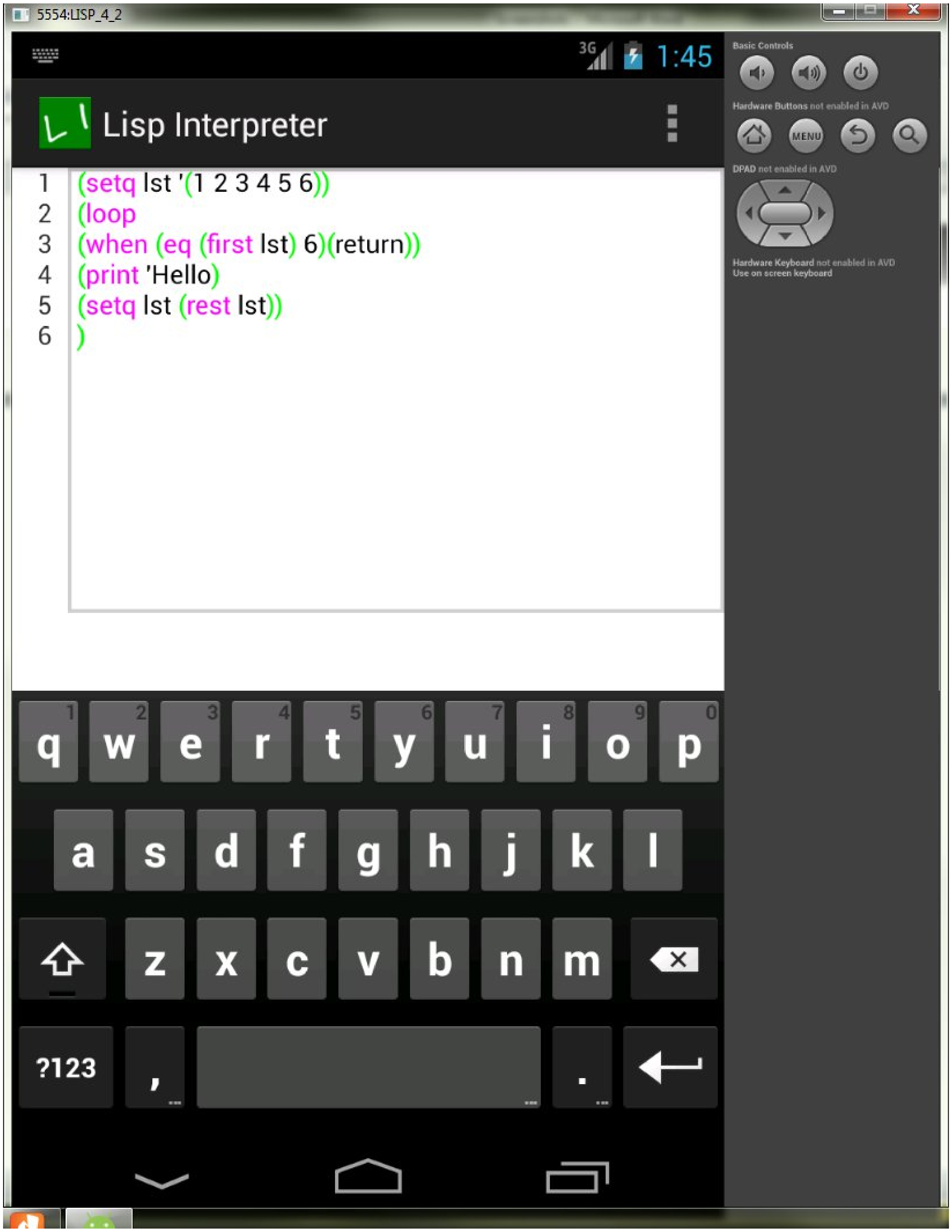

Input the lisp code
Load the fi e
Lisp Interpreted
Execute the code
Save the application
Fina Result

Snapshots

Lisp Interpreter home screen

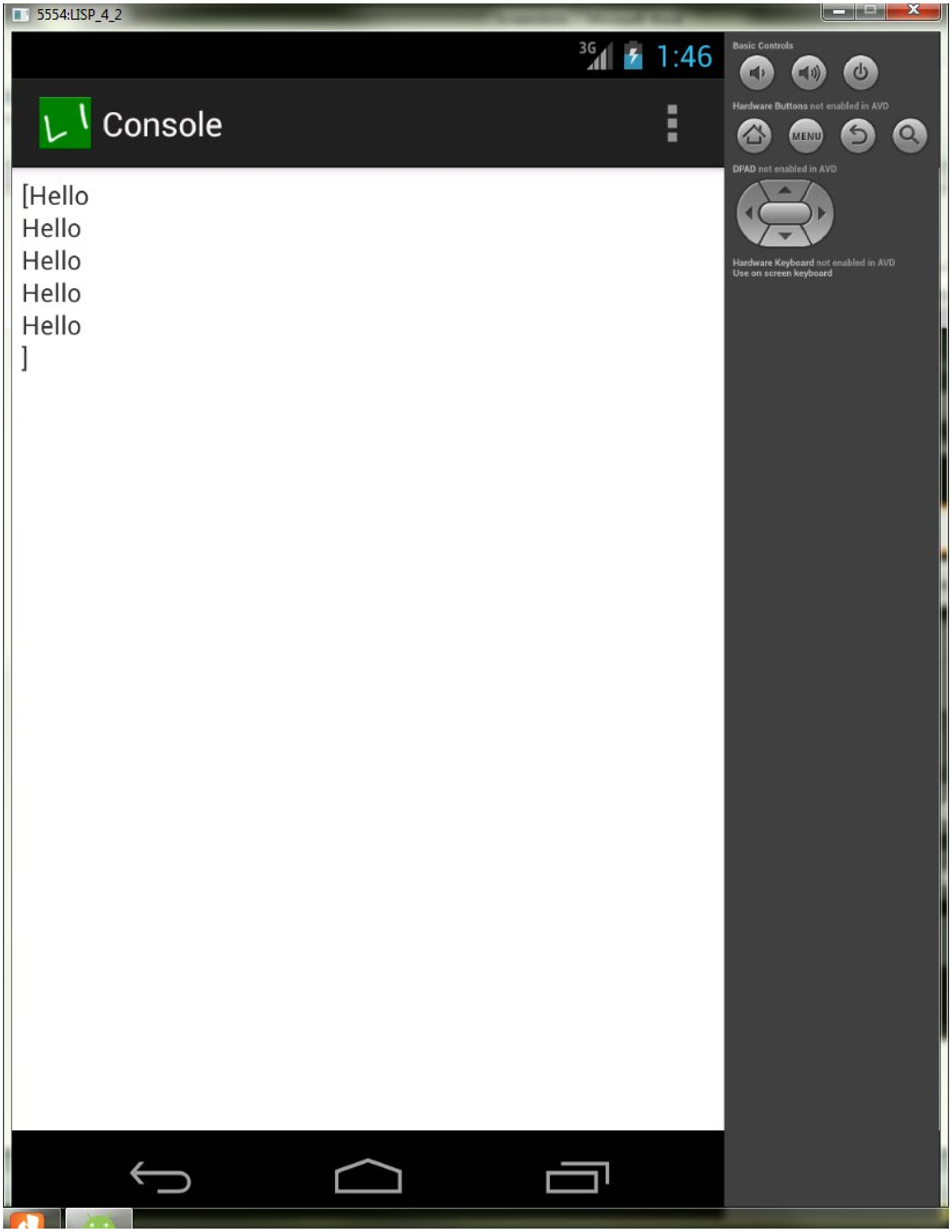


Lisp Interpreter home screen with code



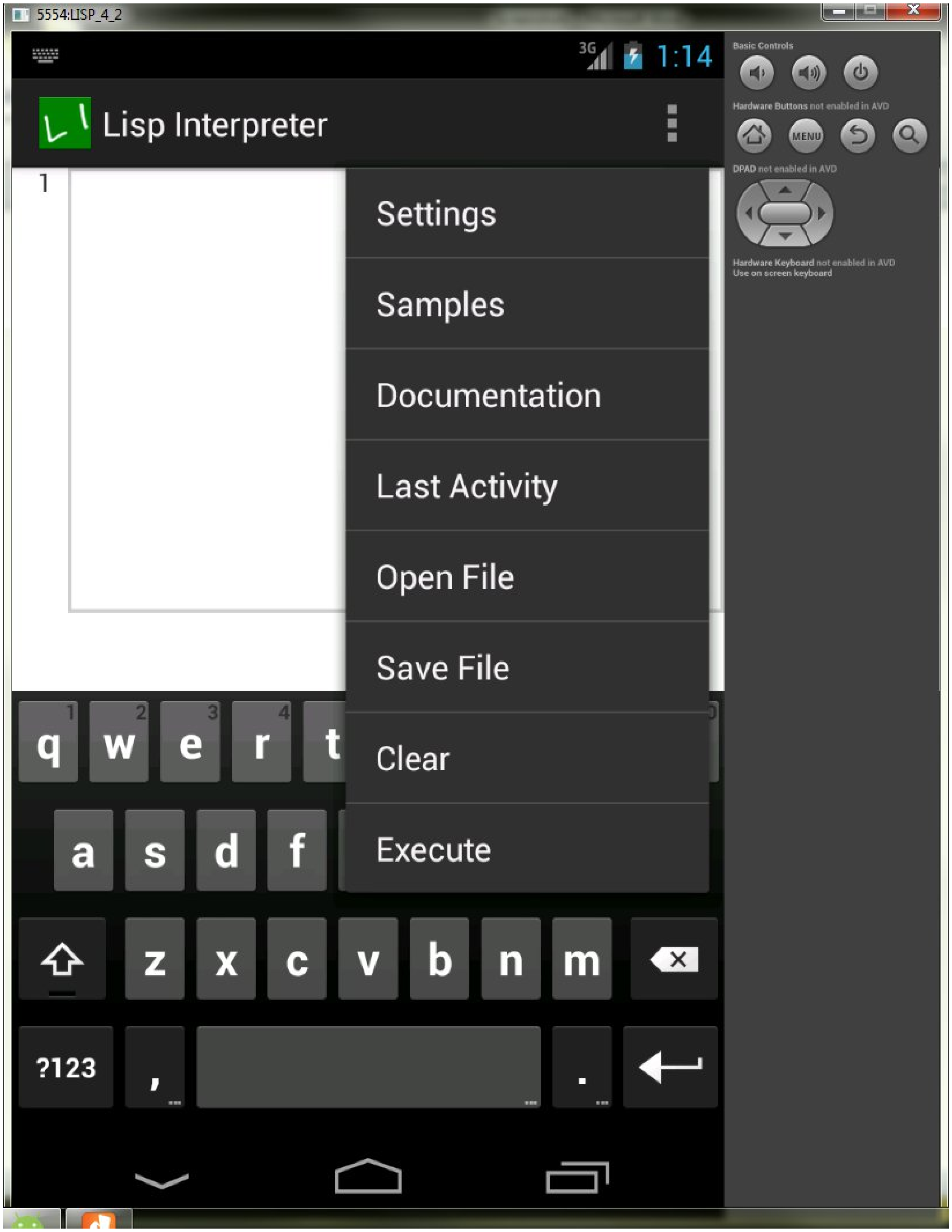
```
(setq lst '(1 2 3 4 5 6))
(loop
(when (eq (first lst) 6)(return))
(print 'Hello)
(setq lst (rest lst))
)
```

Lisp Interpreter console screen

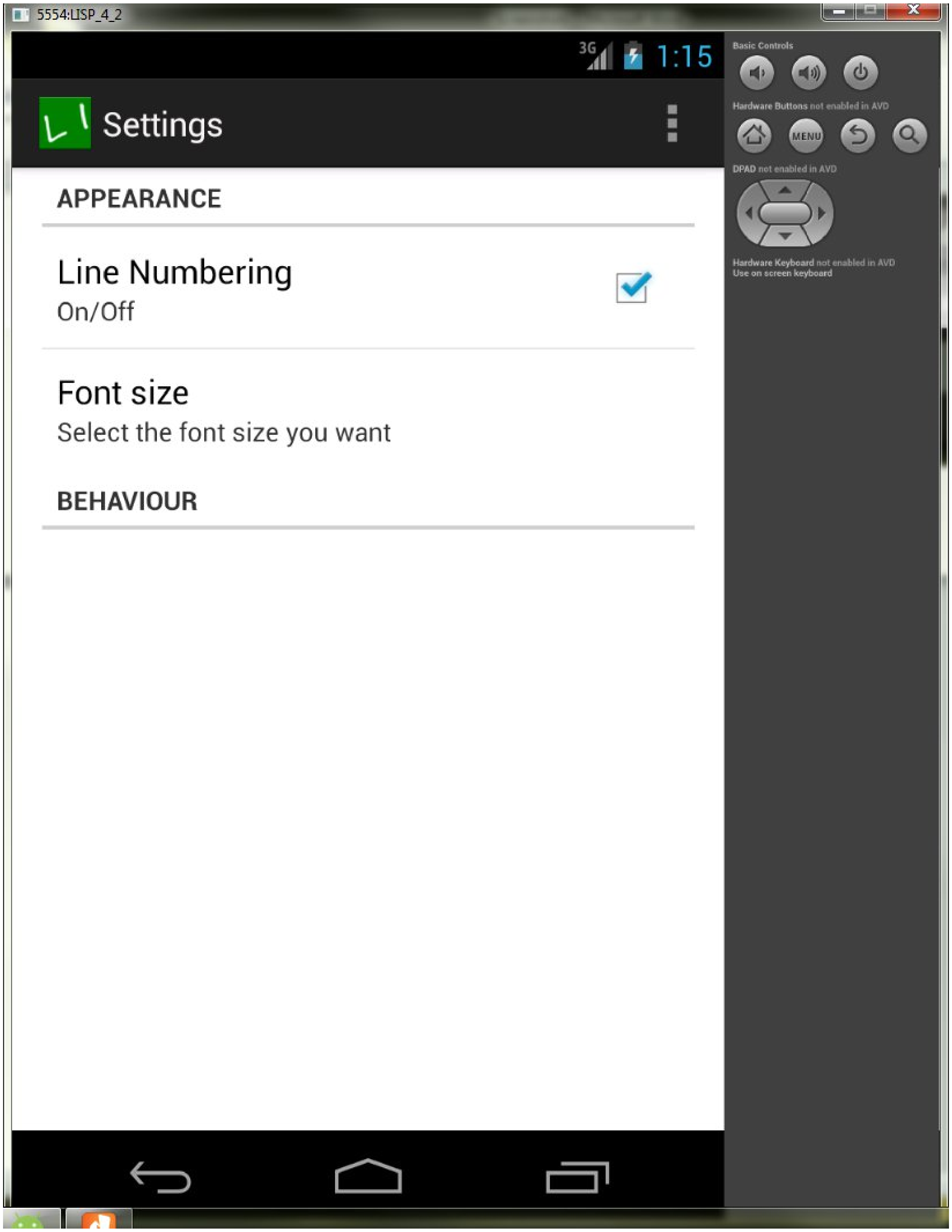


```
[Hello
Hello
Hello
Hello
Hello
]
```

Lisp Interpreter menu screen



Lisp Interpreter settings screen



Settings

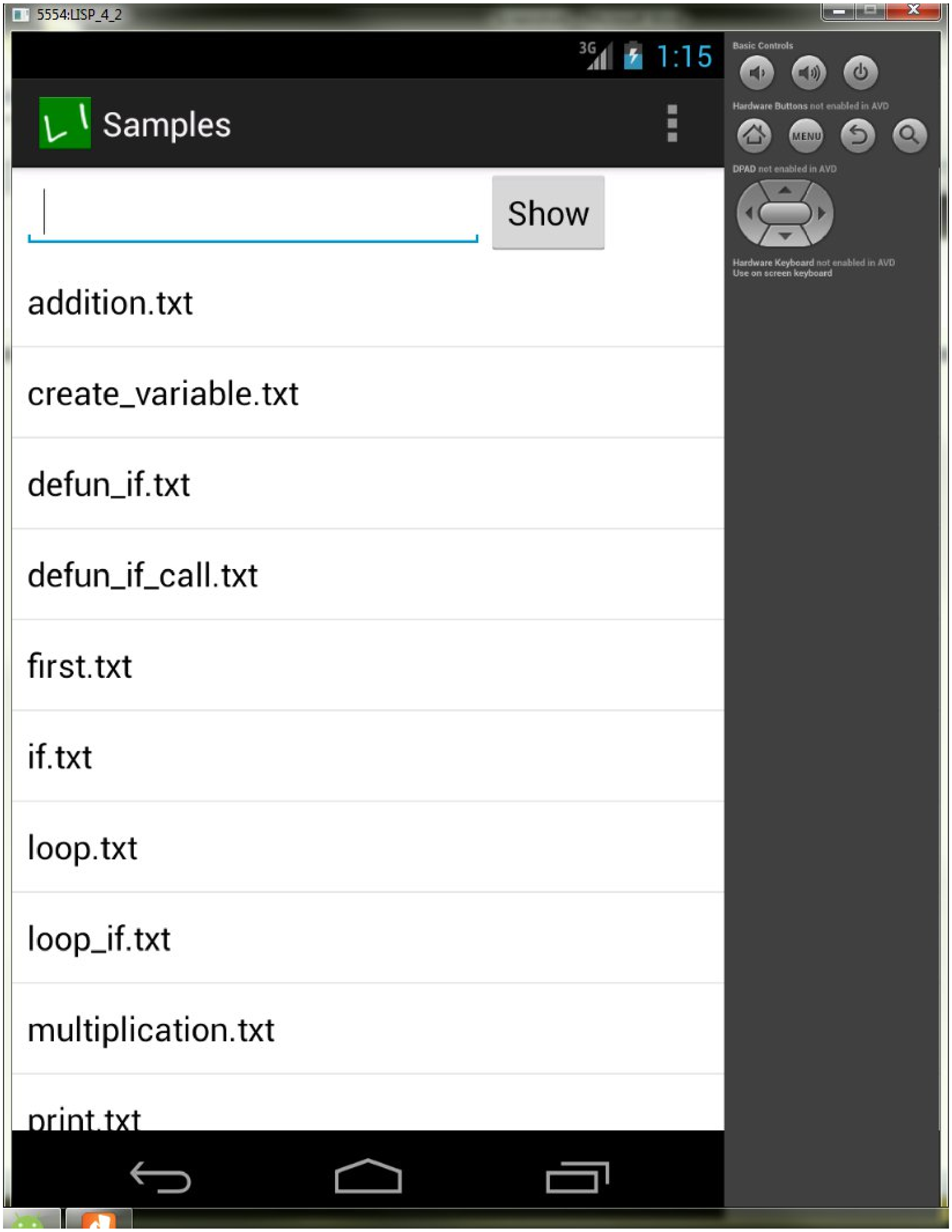
APPEARANCE

Line Numbering ✓
On/Off

Font size
Select the font size you want

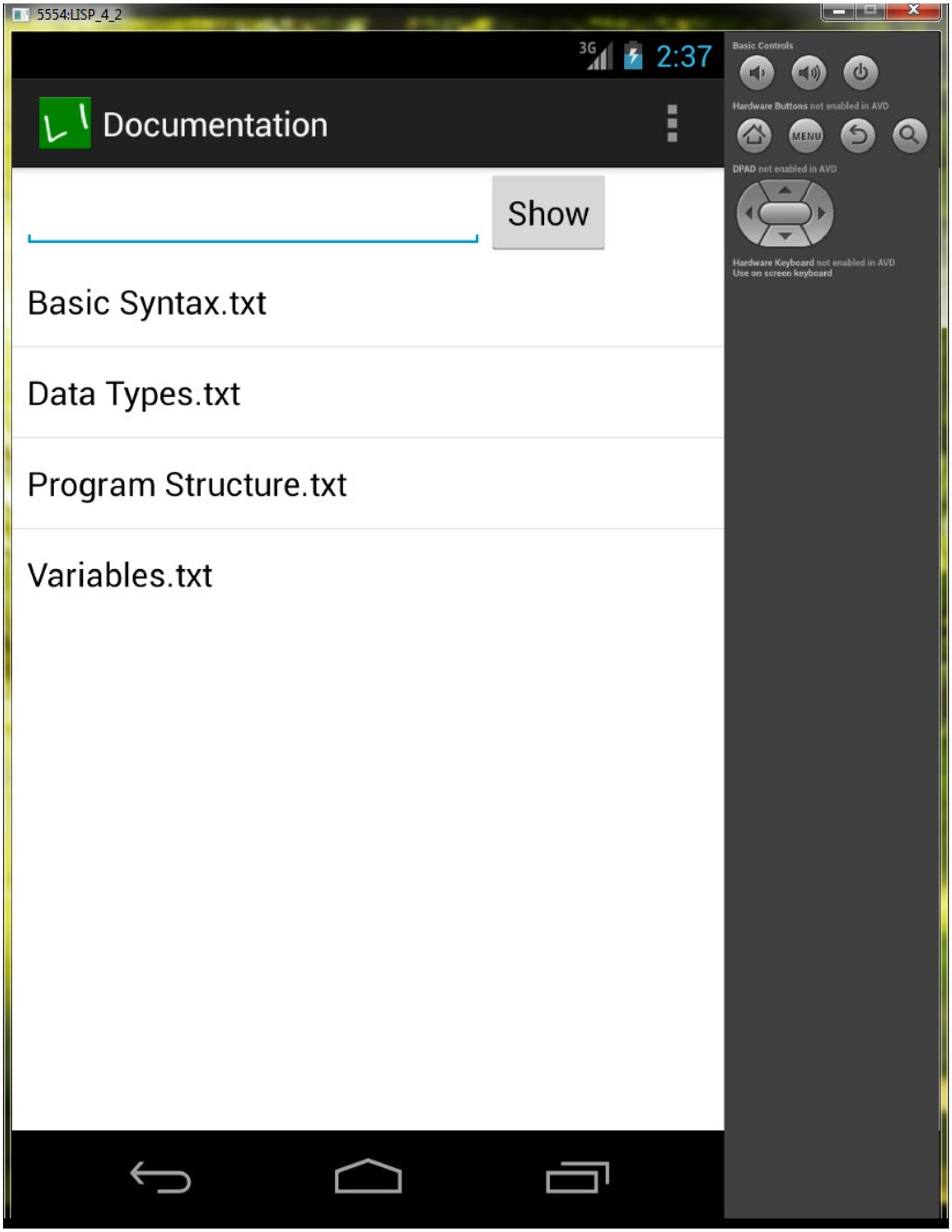
BEHAVIOUR

Lisp Interpreter sample codes screen



addition.txt
create_variable.txt
defun_if.txt
defun_if_call.txt
first.txt
if.txt
loop.txt
loop_if.txt
multiplication.txt
print.txt

Lisp Interpreter documentation screen



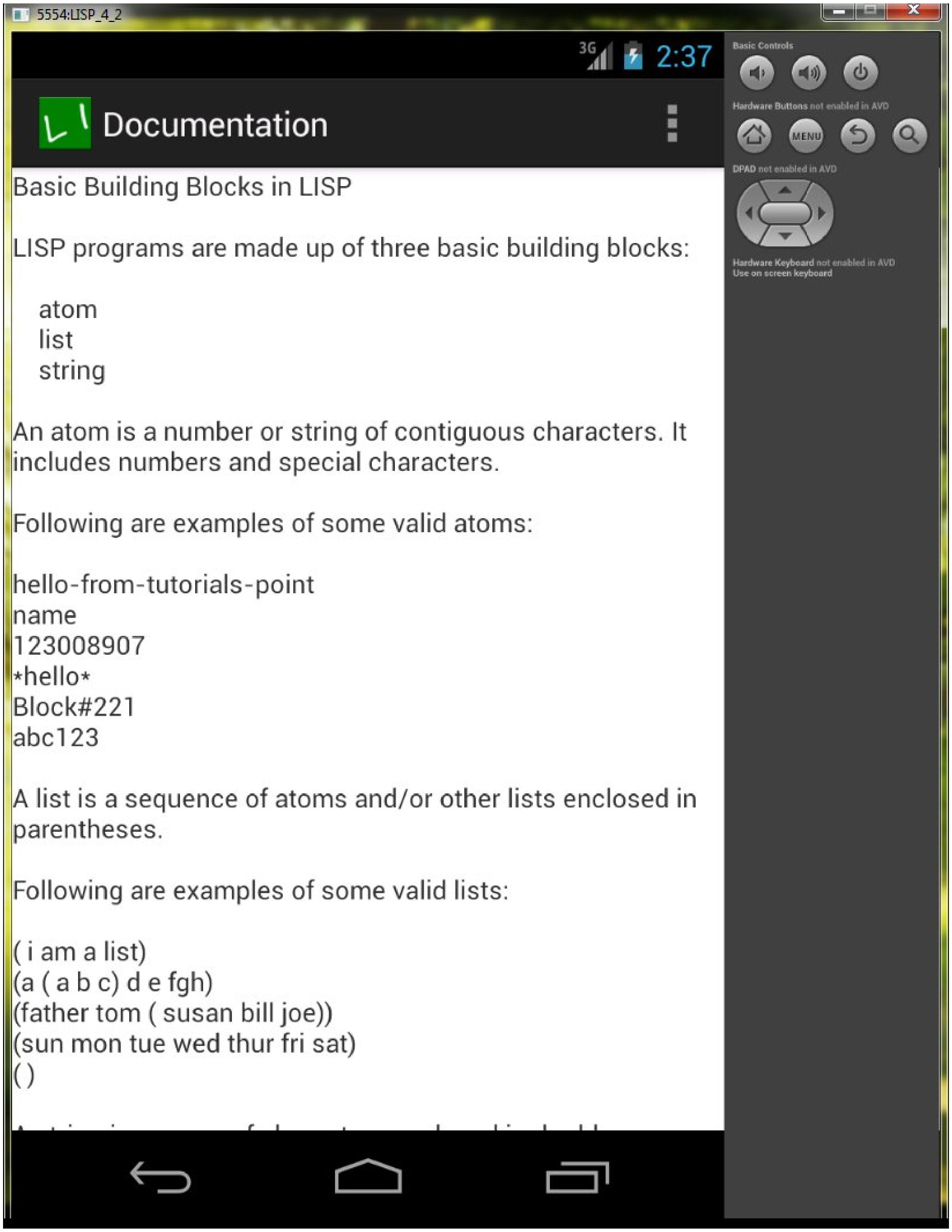
Basic Syntax.txt

Data Types.txt

Program Structure.txt

Variables.txt

Lisp Interpreter documentation output screen



Documentation

Basic Building Blocks in LISP

LISP programs are made up of three basic building blocks:

atom
list
string

An atom is a number or string of contiguous characters. It includes numbers and special characters.

Following are examples of some valid atoms:

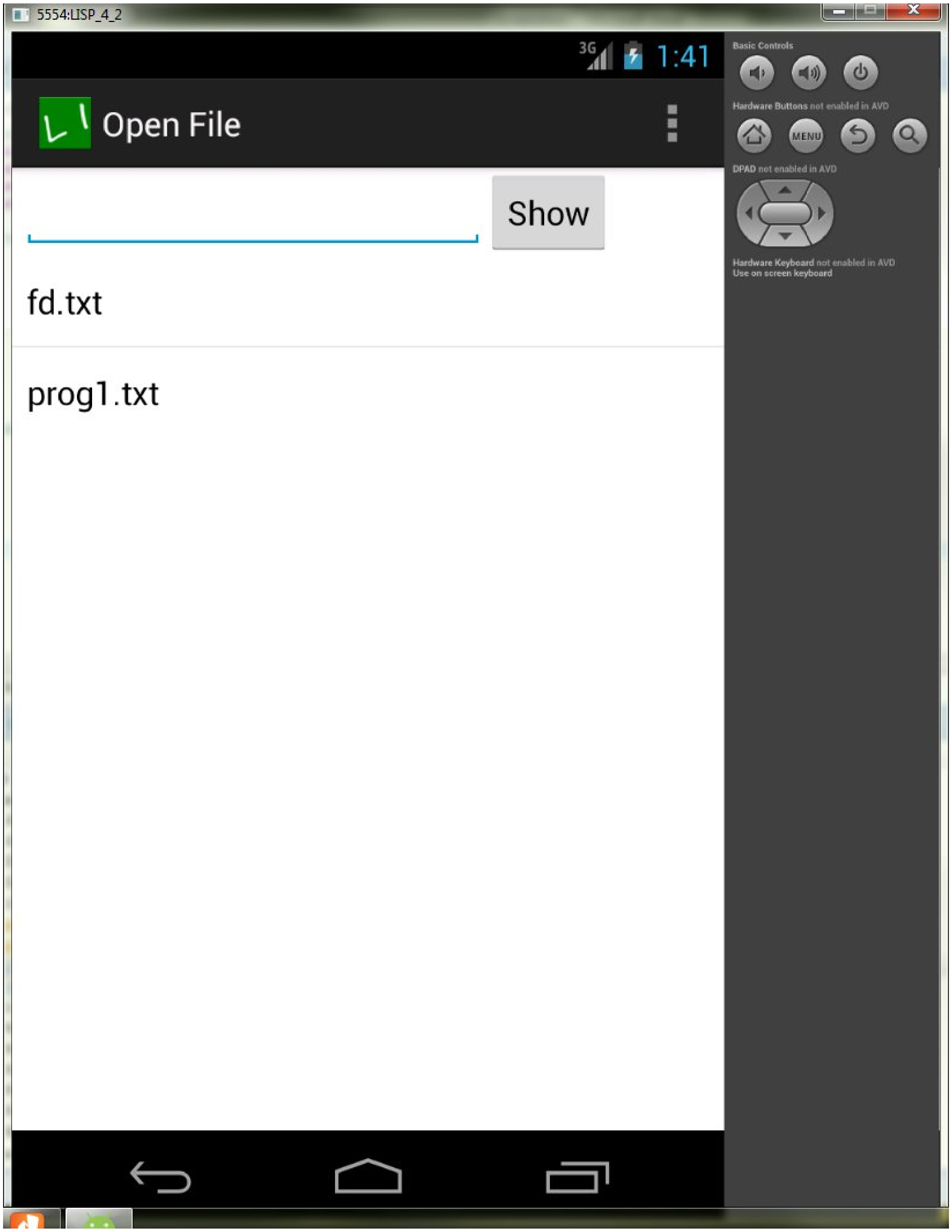
```
hello-from-tutorials-point
name
123008907
*hello*
Block#221
abc123
```

A list is a sequence of atoms and/or other lists enclosed in parentheses.

Following are examples of some valid lists:

```
( i am a list)
(a ( a b c) d e fgh)
(father tom ( susan bill joe))
(sun mon tue wed thur fri sat)
( )
```

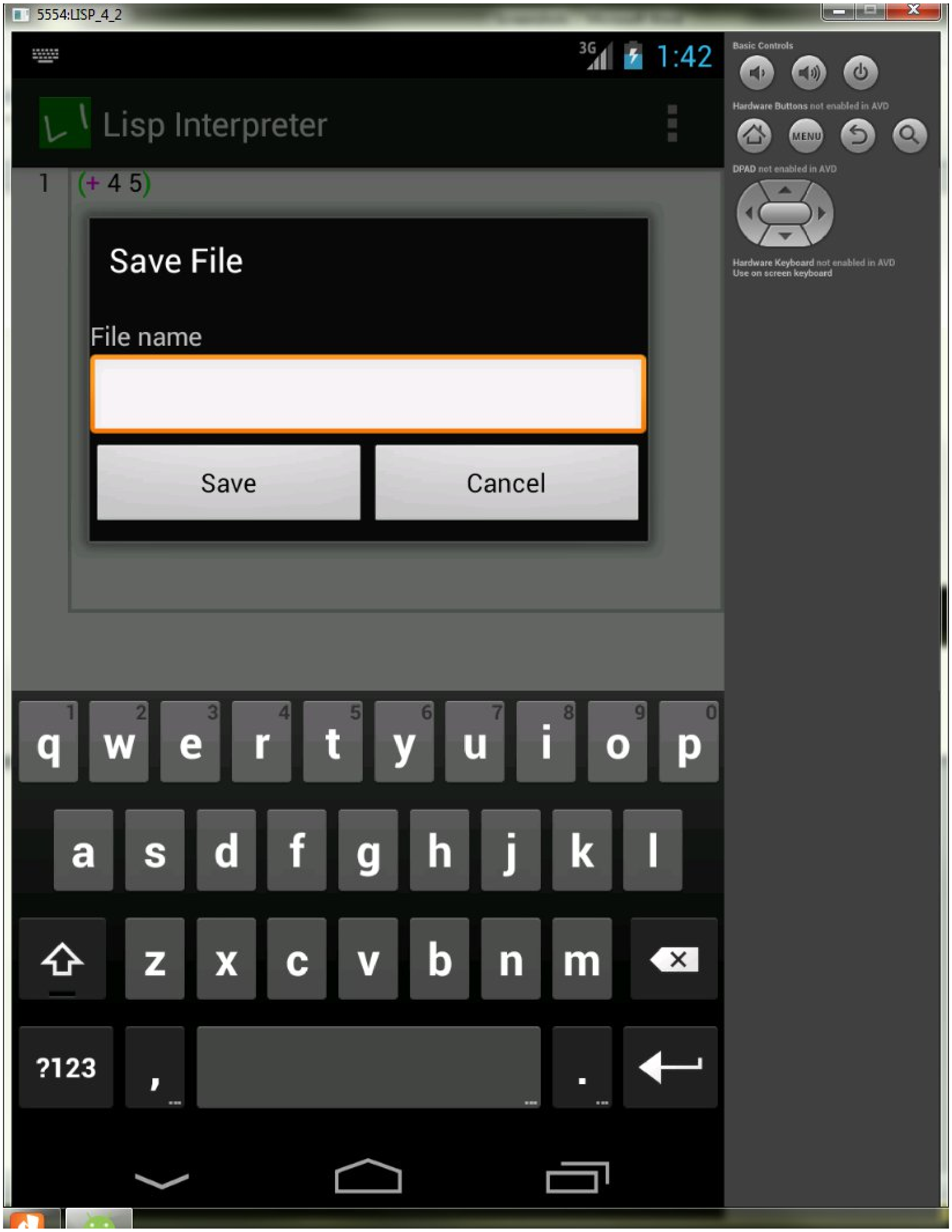
Lisp Interpreter open file screen



fd.txt

prog1.txt

Lisp Interpreter save file screen



References

[1] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart and Michael I. Levin. LISP 1.5 Programmer's Manual, 2st edition, The M.I.T. Press,1962, ISBN 0-262-13011-4

[2] Thomas D, McFarland, and Reese Parker. Expert Systems in Education and Training, 1st edition, Educated Technology, 1990, ISBN 0-877-78210-5

[3] Paul Graham. On Lisp: Advanced Techniques for Common Lisp, 1st edition, Prentice Hall, 1993, ISBN 0-130-30552-9

[4] Guy Steele. Common LISP. The Language, 2nd edition, Digital Press, 1990, ISBN 1-555-58041-6

[5] R Kent Dybvig. The Scheme Programming Language, 4th edition, The M.I.T. Press, 2009, ISBN 0-262-51298-5

[6] Rouse Margaret. Parser Definition, Accessed at: http://searchsoa.techtarget.com/definition/parser, Accessed on: 2015