JiPi Language Specification

Abstract

This document describes the syntax, semantics, and design of the JiPi programming language.

Authors

Jimmy Dahl

Navigation

To navigate within this document, use the Contents page links or just scroll down.

Contents

1. Introduction

JiPi is a simple and modern high-level programming language. It will be fairly familiar to Python programmers, even though it is much simpler.

Because of it's simple grammar that pretty much looks like talked English, the JiPi programming language makes it easy for everyone to write short and simple scripts in no time. The syntax is inspired by the most common high-level scripting languages in the world which makes it easy for a experienced programmer to start writing code in JiPi.

The JiPi compiler makes it easy to develop and use programs written in the JiPi programming language and because of it's intuitive and common appearance it's easy for both the new and the experienced programmer to execute their programs. The compiler translates the JiPi source code to Ruby source code. Ruby is a well spread and commonly used programming language and it's interpreter is preinstalled in several linux distributions.

The rest of this chapter gets you started, while later chapters describe rules and exceptions of the language. The intent with the "Getting started" chapter is to provide the reader with an introduction to the language that will facilitate the writing of early programs and the reading of later chapters.

1.1 Getting started

The canonical "hello, world" program can be written as follows:

```
write("hello, world")
```

The source code for a JiPi program is typically stored in one or more text files with a file extension of .jp as in hello.jp

Using the command-line compiler provided with JiPi, such a program can be translated to Ruby by starting the compiler and follow its instructions.

2. Lexical structure

This chapter describes the lexical structure of the JiPi programming language. If you want to read about the JiPi grammar please read chapter 12.

2.1 Programs

A JiPi program consists of one or more source files. A source file is an ordered sequence of Unicode characters and can be written with a text editor.

2.2 Escape sequences

At this moment JiPi only has one unicode character escape sequences and that one is to enable new line in a string. This could be needed when writing to a file.

The escape sequences starts with a backslash followed by a letter. The escape sequence for a new line is `\n`

2.3 Keywords

All predefined functions, operators and some other identifiers are keywords in JiPi. These words can not be used to anything else. Except for the operators (see 2.4) and the predefined functions (see 10.1) these are the keywords:

`if`, `elseif`, `else`, `for`, and `function`.

2.4 Operators

These are the operators in JiPi:

`+`	The arithmetic addition operator.
`-`	The arithmetic subtraction operator.
`*`	The arithmetic multiplication operator.
`/`	The arithmetic division operator.
`=`	The assignment operator.
`>`	The “bigger then” operator.
`<`	The “less then” operator.
`>=`	The “bigger or same as” operator
`<=`	The “less or same as” operator
`==`	The “equal” operator
`return`	The return operator that returns variables from a function.

3. Basic concepts

This chapter defines basic concepts that are required for understanding subsequent chapters.

3.1 Writing code

The JiPi programming language does not ship with any development environment. All you need to write source code is a regular text editor.

3.2 Save program

To save your source code for later use you save it as an ordinary file with any name and the filename extension .jip. The .jip extension tells that the file is a JiPi source code file and is not necessary for the file to compile. The compiler will try to compile every file you give it, no matter what filename extension you use.

3.3 Compile program

Before you'll be able to run the source code you have to compile it. You compile the source code using the JiPi compiler which translates the JiPi source code to Ruby source code.

3.4 Run program

When the source code is written, saved and compiled you can run it as a usual Ruby program with the Ruby interpreter.

4. Types

JiPi supports two kinds of types: value types and reference types. Value types differ from reference types in that variables of the value types directly contain their data, whereas variables of the reference types store references to objects.

4.1 Value types

JiPi uses different data types for different data when it's stored in a variable. The programmer does not define what type a variable is and does not have to care to much about it. The predefined value types include integers, floats, and the types bool and string. Booleans can either be true or false. Strings are sequences of characters.

All value types are declared automatically when they are first used.

The only difference between these types that the programmer needs to know about is that they support different operations. Read more about the operators and what types they work on in chapter 7.

4.2 Reference types

The only predefined reference type is `list`. A list is a data structure that contains zero or more variables. The variables contained in a `list`, also called the elements of the `list`, can be of any other type. Every element in a `list` is a reference to another object, that's why it's called a reference type.

The list type is, just like the value types automatically declared when first used. To separate a `list` from a value type brackets are used before and after the list elements. Read more about this in chapter 7 where you can read about the assignment operator and how it works for lists.

The `list` is indexed by number and the first elements position is 0.

5. Variables

Variables represent storage locations. Every variable has a type that's being set automatically when the variable gets assigned a value. The value, and type, of a variable can be changed through assignment.

There are two kinds of variables, local and global. This is decided by what namespace they're in. Read more about this in chapter 11.

6. Conversions

Sometimes you want to convert numbers to letters and letters to numbers. For this need JiPi has some predefined conversion functions. For every possible conversion there's a function to call with the value you want to change as an argument.

The possible conversions are from string to integer, from integer to string and from float to integer. Conversion between integers and floats are done automatically when needed. If you do a calculation with both integers and floats the integers will be converted to floats. The functions names are `string_to_int(...)`, `int_to_string(...)` and `float_to_int(...)` and can be called from anywhere in the program.

7. Expressions

An expression is a sequence of operators and operands. This chapter defines the syntax, order of evaluation of operands and operators, and meaning of expressions.

7.1 Operators

Expressions are constructed from operands and operators. The operators of an expression indicate which operations to apply to the operands. The order of evaluation of operators in an expression is determined by the precedence and associativity of the operators. Operands in an expression are evaluated from left to right if the operators precedence doesn't tell otherwise.

7.2 Arithmetic operators

The `*`, `/`, `+`, and – operators are called the arithmetic operators. The `*` operator does multiplication, the `/` operator does division, the `+` operator does addition and the – operator does subtraction.

The order of evaluating arithmetical operators is determined by the precedence of the operators which in JiPi follow the usual rules for mathematical operations.

To use the arithmetical operators you just put them between operands of a type that supports the operation. Every numerical data type supports all arithmetical operations.

You may also use the addition operator to add strings to each other. This is done in the same way, put the addition operator between two strings.

7.3 Logical operators

There are two logical operators, the `and` and the `or`. These are to be used in Boolean expressions. A Boolean expression with the `and` operator evaluates true if and only if both operands are true else it's false. The `or` operator evaluates true if at least one of them are true, else it is false.

7.4 Assignment operators

In JiPi there's only one assignment operator, the `=` operator. Whatever type of value you want to assign to a variable you use this operator between the variable name and the value.

7.5 Boolean expressions and comparing operators

A boolean-expression is an expression that yields a result of type bool. The controlling conditional expression of an `if` statement or `for` statement is a boolean expression. Boolean expressions are built on operands and the comparing operators `>`, `<`, `>=`, `<=` and `==`.

8. Statements

Your programs will be formed by one or more statements and every statement will consist of expressions. In JiPi there are 3 types of statements, expression, selection and iteration statements.

A statement may also embed other statements.

8.1 End point

Every statement, but the expression statements, has an end point. In intuitive terms, the end point of a statement is the location that immediately follows the statement. The execution rules for composite statements (statements that contain embedded statements) specify the action that is taken when control reaches the end point of an embedded statement. For example, when control reaches the end point of a statement in a block, control is transferred to the next statement in the block.

End points is in JiPi marked with the keyword `end`. Whenever the compiler comes to an *end* declaration the current statement is ended.

8.2 Expression statements

An *expression-statement* evaluates a given expression. The most common is the assignment of a value to a variable.

8.3 Selection statements

Selection statements select one of a number of possible statements for execution based on the value of some expression. In JiPi the only selection statements are the `if` statements.

The `if` statement selects a statement for execution based on the value of a Boolean expression. Associated with the `if` statement you may use the `elseif` and `else` statements.

The `elseif` statement does the same thing as the `if` statement, but only if the preceding `if` statement did not evaluate. The `else` statement is only selected when neither preceding `if` or `elseif` hasn't been evaluated.

8.4 Iteration statements

Iteration statements repeatedly execute an embedded statement. The one iteration statement in JiPi is the `for` statement.

The `for` statement loops through every element in a list. It takes a list of elements and, starting with the first element and stepping one element forward for every loop, gives the value of the element to a variable and run all statements inside the `for` statement. The variable with the elements value makes it possible to use all elements in the list in the statements embedded in the `for` statement. This makes the `for` statement ideal for tasks like printing all elements in a list, move them to another list or maybe sum them together.

9. Lists

As mentioned in chapter 4 JiPi has a reference data type called lists. JiPi lists works as arrays does in other languages. A `list` is a data structure that contains references to a number of variables. The list is indexed by number and the first elements position is 0.

9.1 Creating lists

To create a `list` you assign a group of elements, separated with commas, to a variable using a bracket before and after the group of elements like this:

```
variable = [ element1, element2 ]
```

You may also create an empty `list` by passing the variable a `list` with a starting and an ending bracket with no elements in between.

9.2 Add element to lists

If you want to add an element to an existing `list` you use the predefined `append` function. The append function takes two arguments, the `list` and the new element, and adds the new element at the end of the `list`.

9.3 Accessing elements

To access an element in a `list` you use the elements indexed position and surround it with brackets. To get the first element in a `list` you call the lists variable name with the number 0 inside the brackets like this:

```
first = listname[0]
```

If you want to access all elements in a `list` the easiest way is to loop through it with the `for` statement.

10. Functions

JiPi ships with a bunch of predefined functions to makes coding easier for the programmer. But these functions barely covers the most basic needs for writing programs. In addition to the predefined functions programmers have the ability to define their own functions. This is a quite easy process and can simplify and shorten the source code.

10.1 Predefined functions

The predefined functions are: length, append, write, read, open, string_to_int, int_to_string, float_to_int

10.1.1 Append

The predefined `append(…)` function adds an element at the end of an existing `list`. The function takes to arguments, the existing list and the new element. This example explains it:

```
variable1 = “hello”
variable2 = [ “I”, “said” ]
variable3 = append( variable2, variable1 )
```

10.1.2 Length

The `length(…)` function works for both strings and lists and returns the number of letters if you give it a `string` and the number of elements if you give it a `list`. The function takes the `string` or `list` as argument.

10.1.3 Write

The `write(…)` function takes a `string` as argument and writes it to either the terminal where the program runs or to a file, if a file is open.

10.1.4 Read

The `read(…)` function is the one predefined function that doesn’t take any argument. Instead you just call it with empty parentheses. The `read` function reads a line either from the terminal where the program runs or from a file, if a file is open.

10.1.5 Open

The `open(…)` function opens file so that the `read` and `write` functions can write to or read from it. The open function takes the path to the file as argument. Whenever a file is open the `read` and `write` functions only work on the opened file.

10.1.6 string_to_int

This function transforms a `string` that only contains numbers to an `int`.

10.1.7 int_to_string

This function transforms an `int` into a `string`.

10.1.8 float_to_int

This function rounds a `float` to the closest integer.

10.2 Define a function

To define your own function you need to give it a name. The function definition starts with a line with the keyword `function` followed by the functions name and the possible arguments inside parentheses. If the function doesn't take any arguments you just leave the space between the parentheses blank.

After the definition-line you are free to put whatever statements you want. At the end of the function you have to put the `end` declaration that tells the compiler that the function has ended.

To return anything to the statement calling the function you use the `return` operator.

10.3 Call a function

To call a function you simply type its name followed by the arguments you wish to pass (and the function needs) inside parentheses.

11. Namespaces

JiPi uses the usual namespaces approach. There can only be one identifier for a name in the same namespace. In practice this means there can't be two different variables with the same name in the same namespace.

Because of the simple language layout and the thought of the whole language as a language for quit small programs the JiPi programming language only uses two levels of namespaces. The first and most important is the global one. This namespace includes all identifiers declared outside a function. These identifiers are reachable from everywhere in the program.

The other namespace level is the one for identifiers declared inside a function. Every function has its own namespace. Variables declared inside a function can only be reached from inside this function. Calls from outside a function can't reach identifiers inside the function. But when you call for a identifier name inside a function that isn't declared inside the function this will check for this identifier in the global namespace .

This means the global namespace can be reached from anywhere in the program, regardless of which namespace the call comes from.

12. Grammar

The idea of this chapter is to show the languages grammar with some well chosen examples. It contains examples of every thought way of using expressions, statements and functions.

12.1 Expressions

Expressions are built of to operands and an arithmetic operators.

12.1.1 Arithmetic expressions

```
a + 5
b – a
a + b – c * d  / 7
```

12.1.2 Logical expressions

```
a or b
a and c or b
```

12.1.3 Assignment expressions

```
a = 5
b = a + 4
c = “Hello”
```

12.1.4 Boolean expressions

```
b > a
a <= c
d == 5
```

12.2 Statements

Statements are built of expressions, read about the expressions grammar above.

12.2.1 Selection statements

```
if a = b
  write “same”
elseif a > b
  write “bigger”
else
  write “smaller”
end
```

12.2.2 Iteration statements

```
a = [ 1, 2, 3 ]
for element in a
  write “loop: “
  write element
end
```

12.3 Lists

12.3.1 Create a list

```
a = [ 1, 2, 3 ]
```

12.3.2 Append element

```
b = append( a, 4 )
```

12.3.3 Access elements

```
for element in b
  a = a + element
end
```

12.4 Functions

12.4.1 Define a function

```
function test( argument )
  variable = argument * 5
  return variable
end
```

12.4.2 Call a function

```
new = test( 5 )
```