

The Juk User Manual:
Visual Programming
with Functional Abstraction

Mayer Goldberg (gmayer@cs.bgu.ac.il)
Assaf Shemesh (shemeas@cs.bgu.ac.il)

July 31, 2005

Contents

1	Introduction	5
2	Juks Data Types	7
2.1	Atomic Data Types	7
2.1.1	Integers	7
2.1.2	Strings	7
2.1.3	Booleans	7
2.1.4	Null	7
2.2	Compound Data Types	8
2.2.1	Closures	8
2.2.2	Arrays	8
2.3	Run-Time Type Checking	8
3	The Semantics of the Juk	9
3.1	The structure of the Juk	9
3.2	Data Flow	9
3.3	Applying Closures	10
3.4	Compound Juks	11
3.5	Control Flow- How Juks are Evaluated	11
4	Built-in Juks	15
4.1	Language	15
4.1.1	<i>Const</i>	15
4.1.2	<i>Application</i>	15
4.1.3	<i>If</i>	15
4.1.4	<i>Observer</i>	15
4.1.5	<i>Sequence</i>	16
4.2	Arithmetic	16
4.2.1	$+$	16
4.2.2	$-$	16
4.2.3	$*$	16
4.2.4	$/$	16
4.2.5	<i>Mod</i>	17
4.3	Comparison	17
4.3.1	$==$	17
4.3.2	$!=$	17
4.3.3	$<$	17
4.3.4	$<=$	17
4.3.5	$>$	18
4.3.6	$>=$	18
4.4	Boolean Operators	18
4.4.1	<i>and</i>	18

4.4.2	<i>or</i>	18
4.4.3	<i>not</i>	18
4.4.4	<i>xor</i>	19
4.5	Type testing	19
4.5.1	<i>isBoolean</i>	19
4.5.2	<i>isClosure</i>	19
4.5.3	<i>isInt</i>	19
4.5.4	<i>isNull</i>	19
4.5.5	<i>isString</i>	19
4.5.6	<i>isArray</i>	20
4.6	Strings	20
4.6.1	<i>stringAppend</i>	20
4.6.2	<i>intToString</i>	20
4.6.3	<i>stringLength</i>	20
4.6.4	<i>asciiValueToString</i>	20
4.6.5	<i>stringRef</i>	20
4.6.6	<i>substring</i>	21
4.7	Arrays	21
4.7.1	<i>createArray</i>	21
4.7.2	<i>arrayGet</i>	21
4.7.3	<i>arraySet</i>	21
4.7.4	<i>arrayLength</i>	21
5	Juk Libraries	23
5.1	Class Paths	23
5.2	Creating Juk Libraries	23
5.3	Using Juk Libraries	23
5.4	Editing Juk Libraries	23
5.5	The Standard Juk Library	24
A	Running the Juks	25
A.1	Prerequisites	25
A.2	Running the Juks	25
	Index	26

Chapter 1

Introduction

This manual describes the a visual programming platform known as the *Juk*.¹The Juk is both a general-purpose visual and functional programming language. The Juk system comes with a GUI-based interactive development environment. Programming with this IDE consists of constructing and inter-connecting self-contained software components known as Juks. Juks are inter-connected at their docking points to describe data flow. Any number of Juks can be encapsulated within a larger Juk to provide a functional interface and hide internal complexity. Juks are saved in files, and can be used to construct elaborate libraries of re-usable software components.

¹ The term “Juk” comes from the colloquial Hebrew term for “microchip”, which originally comes from the Russian word for “beetle”. The term “Juk” was selected for this project because Juks carry a superficial graphical resemblance microchips.

Chapter 2

Juks Data Types

Like other functional language such as Lisp and Scheme, the Juks language uses dynamic (or latent) typing. This means that it is values that have types associated with them and not the language constructs. However, the language does expect specific value types for many kinds of operations, and errors will occur if the correct types of values are not supplied as inputs.

2.1 Atomic Data Types

Atomic data types are like primitive data types in languages such as C or Java. They are elemental and cannot be further subdivided into parts. The Juks' set of atomic data types includes numbers(at the moment only numbers of type Integer are supported), strings, booleans and the Null object.

2.1.1 Integers

Currently the language supports only 127-bit signed integer values. In the future, we are planning to add support for a full numeric tower including arbitrary-precision integers and floating point numbers, and fractions. The built-in operations on integers includes the basic arithmetics and comparison.

2.1.2 Strings

The language provides fixed length immutable strings. At the moment the character type is not supported. The language uses a 1 length strings instead. Among the The built-in operation on strings are `stringAppend`, `stringLength` and more.

2.1.3 Booleans

A value of a boolean can have one of the values `true` or `false`. One of the uses of booleans is as the test condition of the `If Juk`. The built-in operations on booleans includes the logic operations (like `and`, `or`).

2.1.4 Null

The Null is the default value for any uninitialized value. Applying most of the built-ins with Null would result with a run time error.

2.2 Compound Data Types

Compound data are the set of built-in data structures supported by the Juks. These contain, or are composed of, other compound or atomic data.

2.2.1 Closures

A closure is an abstraction of the functionality defined by some Juk. *Every Juk* can be abstracted as a closure using its abstraction docking point. The Juks language treats closures as first-class objects, which means that closures can be handled like any other primitive value; they can be returned as the value of another Juk, can be passed as arguments to other Juks, can be stored in data structures, and so on.

2.2.2 Arrays

Arrays are fixed-sized containers for values of any Juk data type. The Array is the only data type in the language that is mutable; the values of its cells can be changed at any time. The Array data type is polymorphic, which means that its cells can have values of any data type, and different cells can have values of different types. Currently the maximum possible length of an array is 2147483647 ($2^{31} - 1$). Multidimensional arrays can be created using arrays nested inside arrays.

2.3 Run-Time Type Checking

Many of the built-in Juks expects values of certain types as inputs. Applying values of different types would result with a run-time error. Preprocessing the program in order to detect these kind of errors is not done since the language is dynamically typed. For a list of the built-ins and their expected input types please refer to Chapter 4. Compound Juks, however, cannot force certain input values types.

Chapter 3

The Semantics of the Juk

3.1 The structure of the Juk

Graphically speaking, a Juk is a 2D rectangle, with small circles drawn on the border. The position of the Juk as well as its size has no effect on its semantics. The circles on the top border are the input docking point, from which the Juk reads its inputs. The circles on the bottom border are the output docking points, where the Juk writes its output. Each output docking point represents a functionality of the Juk (a thing that the juk can compute or do). The circle on the right border is the abstraction docking point (we will discuss it on Section 3.3), and is always named "λ". Each Juk has a name, but the name is not a part of the language semantics and is used only for documentation. Each docking point has a name as well. The names of the input and output docking point are used only when a closure is applied (Section 3.3). In Figure 3.1 we see the Juk whose name is "greatest common divisor" which has two input docking points: "a" and "b", and one output docking point named "gcd".

3.2 Data Flow

The Juk language uses the data flow model. Depending on the control flow of the program (Section 3.5), connections between docking points create flow of values. An output docking point of one Juk can be connected to an input docking point of another. Each connector is directed - it has *source* docking point and a *destination* docking point. The data always flows from the source to the destination.

As we can see in Figure 3.2, the value 1 flows from the output docking point "value" of the Juk "1" (the source) to the input docking point "a" of the Juk "+" (the destination). Similarly, the value 2 flows from the output docking point "value" of the Juk "2" to the input docking point "b" of the Juk "+". Both "a" and "b" are calculated in order to calculate the output docking point of the "+" (the arithmetic plus operation) to be 3. In turns, This value flows to the output docking point "result" of the surrounding compound Juk (Section 3.4) "calc 1+2".

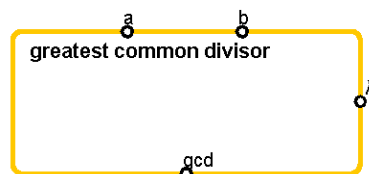


Figure 3.1: The GCD Juk

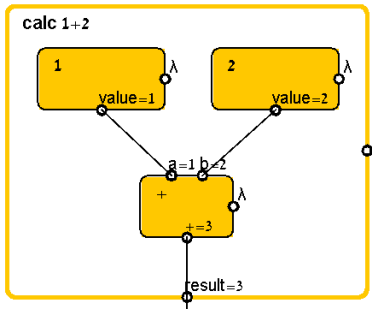


Figure 3.2: the "calc 1+2" Juk

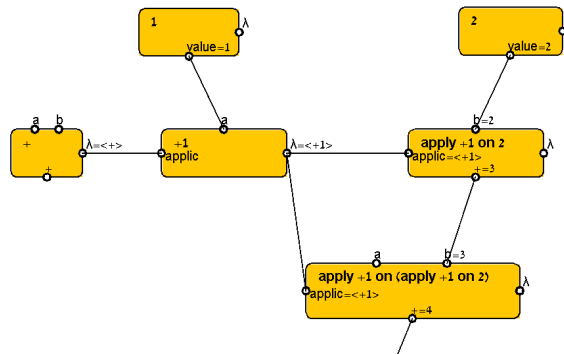


Figure 3.3: Applying "+"

3.3 Applying Closures

Applying closures is the Juks' way of calling functions, like languages such as C or Java do, with some important differences: (1) Closures are first-class objects, which means that they can be treated as any other data type of the language. (2) Any Juk in the language can be captured as a closure (for example, in Java an if statement is not a function call and cannot be passed to a method as a parameter). (3) Any piece of code can be refactored easily into a closure application without the programmer having to add declarations and statements to his code (Section ??). (4) A closure captures the functionality of a Juk together with its inputs (the environment).

The only way to create closures is to use the abstraction docking point. Closures are the only legal input for the application docking point of the application Juk. The application Juk is the only kind of Juk that has such an input docking point, it is located on the left border and its name is "applic". The application Juk has zero or more input docking points and zero or more output docking points to the Juk abstracted by the closure. Applying both zero input docking points and zero output docking points is possible, though semantically meaningless. When applying a closure, the input values applied by the application Juk are flowed as the input values of the applied closure if both of the following conditions met: (1) The name of the input docking point of the application Juk is the same as the name of the input docking point of the closure. (2) The applied Juk input docking point has no connection ending with it. The application Juk initiates the evaluation of an output docking point of the applied Juk if and only if it has the same name as an output docking point of the application Juk in the course of the data flow. The closure being evaluated might use the input values supplied by the application Juk input docking points as described above.

An example is shown in Figure 3.3. The arithmetic + is being abstracted as input of the "+1" Juk, which in turn applies the value 1 to the input docking point "a". Now the Juk "+1" is the arithmetic +1 operation, and the Juks "apply +1 on 2" and "apply +1 on (apply +1 on

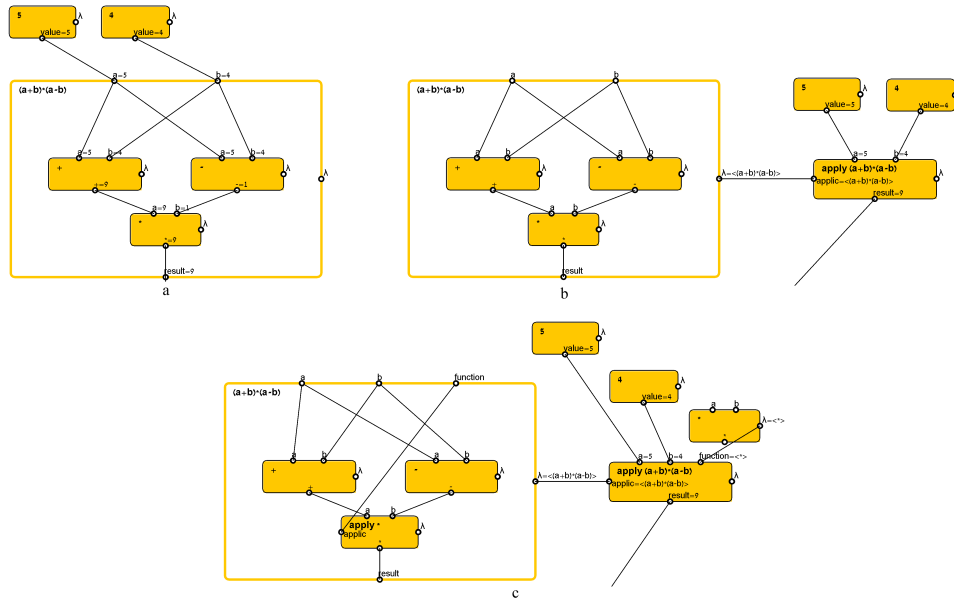


Figure 3.4: Compound Juks

2) use it this way in order to add 1 to the values they apply on the input "b". Note that "+1" has no output docking points since it is not used to evaluate but to create another closure, and that the other two applications does not apply a value to "a" because that input docking point was already associated with an input with "+1". Finally, we can see from this example that the application Juk can be abstracted using its abstraction output docking point as any other Juk.

3.4 Compound Juks

Compound Juks are containers of other Juks and are used to encapsulate functionality. They may contain any number of Juks (both compound and non-compound), and have any number of input docking points and output docking points as the programmer wants. However, a compound Juk without output docking points can never be used nor applied.

An input docking point of a compound Juk can be connected to an input docking point of any of the contained Juks, or to an output docking point of the compound Juk itself. Similarly, an output docking point of any contained Juk can be connected to an output docking point of the compound Juk. The contained Juks can be connected to each other as described in Section 3.2.

For example, let's look on how an expression like $(a + b) * (a - b)$ is programmed in the Juk language. In Figure 3.4 we can see three versions of applying this formula on $a = 5, b = 4$. 3.4a shows direct use of the compound Juk "(a+b)*(a-b)" with the inputs 5, 4. In 3.4b we see application of that compound Juk with the same arguments. 3.4c goes further with functional abstraction when it passes the Juk "*" as an argument.

3.5 Control Flow- How Juks are Evaluated

Whereas the data in a Juks language program flows along the connectors, as in the data flow model, from a source docking point to a destination docking point (Section 3.2), the control flow works the other way around. Only when a destination docking point is required for the evaluation, the source docking point is evaluated.

Program evaluation always starts with the output docking point "output" of the main Juk "Main". Every program has exactly one main Juk, which is a special compound Juk. In addition

to the one output docking point "output", the main Juk has one input docking point "input" that is used to pass the program arguments.

When an output docking point of a compound Juk is evaluated, one of the followings applies:

- (1) If that docking point is connected to another docking point, then the other docking point (the source) is evaluated. Then the result is written to the destination docking point being evaluated.
- (2) If the docking point is not connected then it is evaluated to Null.

When an output docking point of a built-in Juk is evaluated, the flow of control depends on that Juk. Usually, a Juk will evaluate all of its input docking points, make some processing with their values and write the results to the output docking points. The arithmetics Juks, for example, works this way. However, there are some important exceptions. The Juks If, And, Or and the application Juk behave differently since they have special policies of input docking points evaluation. For example, the Juk If evaluates only one of his input docking points "then" and "else" depending on the value of the evaluation of the input docking point "test". Read Chapter 4 for more details.

When an input docking point of a built-in or compound Juk is evaluated, the rule is similar to output docking point of a compound Juk evaluation. If it is connected then the source docking point is evaluated, and the result is written to the destination input docking point. If the input docking point is not connected then it is evaluated to Null. input docking points of the application Juk are evaluated when the applied Juk input docking point needs them (through the course of data flow) to be evaluated (assuming that the names are equal and that there is no connection ending with the applied Juk input docking point as described in Section 3.3).

Figure 3.5 shows a complete Juks program. Note that only the necessary docking points are evaluated. For this reason the *or* operation between a boolean and the number 17 is not an error. Since the input docking point "a" was evaluated to true, the input docking point "b" was never evaluated. Applying the Juk "x*y,x/y" with 5 and 0 does not yield a "division by zero" error since only the output docking point "x*y" is evaluated.

In textual functional languages this type of evaluation is called delayed or lazy evaluation. It means that an expression is not evaluated as soon as it gets bound to a variable, but when the evaluator is forced to produce the expression's value. In our case, a value of a docking point is evaluated only when it's needed for evaluating another docking point.

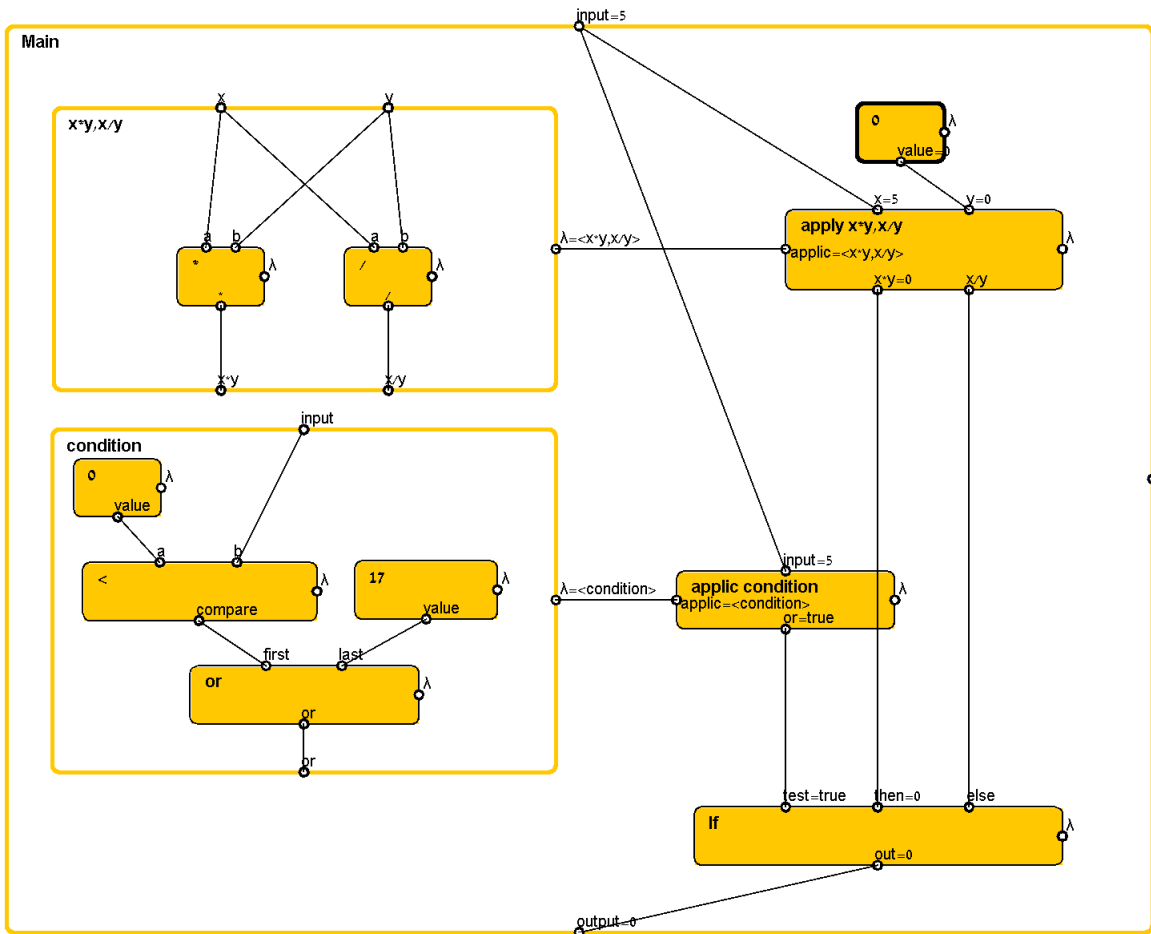


Figure 3.5: Control Flow

Chapter 4

Built-in Juks

This chapter lists all of the language built-in Juks. Starting with the basic language constructs, through operations for manipulating data types. For every Juk we list the names of its input docking points and output docking points with their expected data type. Unless specified else, *all* of the Juk's inputs are evaluated in an *undefined order*.

4.1 Language

4.1.1 *Const*

Outputs:

value: any

4.1.2 *Application*

Inputs:

applic: Closure

Inputs order of evaluation: *applic*, other input docking points depending on the control flow

4.1.3 *If*

Inputs:

test: Boolean

then: any

else: any

Outputs:

out: any

Inputs order of evaluation: *test*, *then* (only if *test* evaluated to true), *else* (only if *test* evaluated to false)

4.1.4 *Observer*

Inputs:

in: any

Outputs:

out: any

4.1.5 *Sequence*

Inputs:

first: any

last: any

Outputs:

last: any

Inputs order of evaluation: *first*, *last*

4.2 Arithmetic

4.2.1 +

Inputs:

a: Integer

b: Integer

Outputs:

+: Integer

4.2.2 −

Inputs:

a: Integer

b: Integer

Outputs:

-: Integer

4.2.3 *

Inputs:

a: Integer

b: Integer

Outputs:

***: Integer

4.2.4 /

Inputs:

a: Integer

b: Integer

Outputs:

/: Integer

4.2.5 *Mod*

Inputs:

a: Integer*b*: Integer

Outputs:

mod: Integer**4.3 Comparison****4.3.1** $==$

Inputs:

a: Integer*b*: Integer

Outputs:

compare: Boolean**4.3.2** $!=$

Inputs:

a: Integer*b*: Integer

Outputs:

compare: Boolean**4.3.3** $<$

Inputs:

a: Integer*b*: Integer

Outputs:

compare: Boolean**4.3.4** $<=$

Inputs:

a: Integer*b*: Integer

Outputs:

compare: Boolean

4.3.5 $>$

Inputs:

 a : Integer b : Integer

Outputs:

 $compare$: Boolean**4.3.6** $>=$

Inputs:

 a : Integer b : Integer

Outputs:

 $compare$: Boolean**4.4 Boolean Operators****4.4.1** and

Inputs:

 $first$: Boolean $last$: Boolean

Outputs:

 and : BooleanInputs order of evaluation: $first$, $last$ (only if $first$ evaluated to true)**4.4.2** or

Inputs:

 $first$: Boolean $last$: Boolean

Outputs:

 or : BooleanInputs order of evaluation: $first$, $last$ (only if $first$ evaluated to false)**4.4.3** not

Inputs:

 in : Boolean

Outputs:

 out : Boolean

4.4.4 *xor*

Inputs:

a: Boolean

b: Boolean

Outputs:

xor: Boolean

4.5 Type testing

4.5.1 *isBoolean*

Inputs:

value: any

Outputs:

isBoolean: Boolean

4.5.2 *isClosure*

Inputs:

value: any

Outputs:

isClosure: Boolean

4.5.3 *isInt*

Inputs:

value: any

Outputs:

isInt: Boolean

4.5.4 *isNull*

Inputs:

value: any

Outputs:

isNull: Boolean

4.5.5 *isString*

Inputs:

value: any

Outputs:

isString: Boolean

4.5.6 *isArray*

Inputs:

value: any

Outputs:

isArray: Boolean

4.6 Strings

4.6.1 *stringAppend*

Inputs:

string1: String

string2: String

Outputs:

append: String

4.6.2 *intToString*

Inputs:

integer: Integer

Outputs:

string: String

4.6.3 *stringLength*

Inputs:

string: String

Outputs:

length: Integer

4.6.4 *asciiValueToString*

Inputs:

ascii: Integer

Outputs:

string: String

4.6.5 *stringRef*

Inputs:

string: String

index: String

Outputs:

ascii: Integer

4.6.6 *substring*

Inputs:

string: String*from*: Integer*length*: Integer

Outputs:

substring: String**4.7 Arrays****4.7.1** *createArray*

Inputs:

length: Integer

Outputs:

array: Array**4.7.2** *arrayGet*

Inputs:

array: Array*index*: Integer

Outputs:

value: any**4.7.3** *arraySet*

Inputs:

array: Array*index*: Integer*value*: any

Outputs:

array: Array**4.7.4** *arrayLength*

Inputs:

array: Array

Outputs:

length: Integer

Chapter 5

Juk Libraries

In order to encourage code reuse and good program design the Juks languages uses a library mechanism. Library Juks are Juks that had been written into separate files and thus are used by more than one program. A special Juk called the *Library Juk* is responsible for linking your program to the library Juk files. The input docking points and output docking points of the Library Juk are identical to those of the imported Juk.

5.1 Class Paths

Like the environment variable CLASSPATH in Java, the Juks language uses a list of paths in which it looks for library Juks at runtime. This list can be configured from the IDE. To add a path, right-click the Juks tree view and select Add Class Path. After inserting the new path, all the *.juk files in that path would appear in the tree view. To remove a path, select it on the Juks tree view, right click to activate to popup menu, and then select *Remove Class Path*.

Class paths cannot include one another. Every path cannot appear more than once.

5.2 Creating Juk Libraries

To create a Library juk, select a compound Juk and then use the main menu *Refactoring->Export as Library Juk*. A dialog box would appear asking for a file name. The file (*.juk) must be included within one of the class paths. After inserting a file name and location, a new file would be created containing the selected Juk. The compound Juk itself would be replaced by a library Juk linked to the new file. The result is semantically identical to the original program.

5.3 Using Juk Libraries

Look for the Juk that you insert to your program in the Juks tree view. Then right click it and select *Insert to Juk*. A Library Juk linked to the selected file would be inserted to the current selected Juk in the editor.

5.4 Editing Juk Libraries

To edit an existing library Juk file, look for it in the Juks tree view. Then right click it and select *Edit Library Juk*. This would open a new tab in the editor containing the selected Juk file.

5.5 The Standard Juk Library

The Standard Juk Library contains Juks that perform a lot of frequent programming tasks, such as add 1, subtract 1, etc. At the moment the library contains arithmetic, boolean, collections and control operations.

Appendix A

Running the Juks

A.1 Prerequisites

Since The Juk IDE is written in the Java programming language (language level 1.5), Java 1.5 or higher is required.

A.2 Running the Juks

The Juk IDE is available as a single JAR file. Once the Java run-time environment is installed on your system, you should be able to run the Juk IDE from a Unix, Linux, or MS Windows command line by issuing:

```
% java -jar Juks.jar
```

Alternately, on MS Windows, you may associate JAR files with the `javaw.exe` program (which is available in the `bin` sub-directory of your Java installation), so that clicking on the JAR file will automatically start up the Juk IDE. You can also create a shortcut or *alias* for invoking the Java byte-code interpreter with specific stack and heap sizes. Consult the documentation for the Java byte-code interpreter for additional details on how to do this.

Index

`*`, 16
`+`, 16
`-`, 16
`/`, 16
`==`, 17
`<=`, 17
`<`, 17
`>=`, 18
`>`, 18

and, 18
Application, 15
Array, 8
 multidimensional, 8
arrayGet, 21
arrayLength, 21
arraySet, 21
asciiValueToString, 20

Boolean, 7

Class Path, 23
Closure, 8
 applying, 10
Const, 15
createArray, 21

If, 15
Integer, 7
intToString, 20
isArray, 20
isBoolean, 19
isClosure, 19
isInt, 19
isNull, 19
isString, 19

Juk
 compound, 11
 definition, 5
 library, 23
 main, 11
 name, 9

Mod, 17

not, 18
Null, 7

Observer, 15
or, 18

Sequence, 16
String, 7
stringAppend, 20
stringLength, 20
stringRef, 20
substring, 21

xor, 19