Guide to CHIPL

`http://InformationFromProcesses.org`

Robert M. Losee
University of North Carolina,
Chapel Hill, NC 27599-3360, USA

`http://InfomationFromProcesses.org`
email: *losee at unc period edu*

February 25, 2013

Contents

I CHIPL **6**

1 Installing and Basics of CHIPL **8**
1.1 Introduction 8
1.2 Purpose and Assumptions of CHIPL 8
1.2.1 Types of Statements 9
Process Definitions 9
Instance Definitions 9
Executable Statement 9
Pre-Processed Statements 9
Information 10
1.3 Using CHIPL 10
1.4 Downloading and Installation 10
1.4.1 Compiling the program yourself 11
1.5 Using CHIPL & Executing the program 11
1.5.1 Setting things up 11
1.5.2 Executing CHIPL 11
1.6 LaTeX 11
1.7 Gnuplot 12
Builtin Library 12
1.8 Example Programs 13
A Single Process 13

2 Manipulating the Stack **17**
Placing items on the top of the stack 17
2.0.0.1 dup 18
2.0.0.2 2dup 18
2.0.0.3 3dup 19
2.0.0.4 drop 19
2.0.0.5 2drop 19
2.0.0.6 3drop 19

2.0.0.7 swap . . . 19
2.0.0.8 rot . . . 20
2.0.0.9 -rot . . . 20
2.0.0.10 over . . . 21
2.0.0.11 clearstack . . . 21
2.0.0.12 isstackempty . . . 21

3 Mathematical Operations **22**
3.1 Arithmetic . . . 22
3.1.1 Addition . . . 22
3.1.2 Subtraction . . . 23
3.1.3 Multiplication . . . 23
3.1.4 Division . . . 23
3.1.5 Remainder . . . 24
3.1.6 Maximum . . . 24
3.1.7 Minimum . . . 24
3.2 Logical Operations . . . 24
3.2.1 Comparative Operators: Greater and Less Than . . . 24
3.2.2 Manipulating Truth Values . . . 25
3.3 Checking for Type . . . 26
3.4 Random Operators . . . 26
3.4.0.1 randombinary . . . 27
3.4.0.2 randominteger . . . 27
3.4.0.3 randomfloat . . . 27
3.4.0.4 noisybinarycopy . . . 27
3.4.0.5 noisyintegercopy . . . 27
3.4.0.6 concatenate . . . 28

4 Input and Output **36**
4.1 General Input and Output . . . 36
Variables and # . . . 36
4.2 Input . . . 36
4.3 Random Inputs . . . 37
4.3.0.7 randombinaryinput . . . 37
4.3.0.8 randomintegerinput . . . 37
4.3.0.9 randomfloatinput . . . 38
4.4 File Input . . . 38
4.5 Output . . . 38
Output . . . 38
same . . . 38
4.6 Print to Screen . . . 39

print 39
printspace 39
printcr 39
printf 39

5 Flow of Control **41**
5.1 Conditional Statements 41
5.2 Conditional Loops 43
5.2.1 For Loops 44
Multiplication Table Example 45

6 MetaInformation **47**
6.0.2 addMI 47
6.0.3 addOrReplaceMI 47
6.0.4 labelExistsMIQ 48
6.0.5 booleanExistsMIQ FINISH 48
6.0.6 findMI 48
6.0.7 removeMI 48
6.0.8 sizeMI 49
6.0.9 clearMI 49
6.1 Example 49

7 Statements **54**
7.0.1 makeemptyvector and makeemptystatement 54
7.0.2 getobjectatposition 54
7.0.3 removeposition 54
7.0.4 insertobjectatend 54
7.0.5 insertobjectatstart 55
7.0.6 sizeofstatement and sizeofvector 55

8 Miscellaneous Instructions **56**
debug 56
Abnormal program termination 56

9 Pre-processing **57**
9.1 Cycles 57
9.2 Non-Random Generation of Processes 57
9.3 Random Repetition of Processes 58
9.3.1 Random 58
9.3.2 Small Worlds 58
9.3.3 Random Graph Networks 59

10 System Variables **62**
10.0.3.1 pause 62
10.0.3.2 printvalueon 62
10.0.3.3 printvalueoff 62
10.1 Gnuplot File Format 62

II More Information Theory with CHIPL **64**

11 Measures of Performance **66**
11.1 The Amount of Information 66
11.1.1 Self-Information 66
11.1.2 Entropy 66
11.1.3 Joint Entropy 67
11.1.4 Conditional Entropy and Mutual Information 67

12 Information in Relational Databases **69**

Index **73**

Part I

CHIPL

Chapter 1

Installing and Basics of CHIPL

1.1 Introduction

CHIPL, the Chapel Hill Information Processing Language, allows one to develop processes and use the processes so as to manipulate data while providing numbers about the amount of information being manipulated. The language is structured so as to ease the development of process based programs, helping make explicit the information that comes from processes. There is no assumption that CHIPL will push aside Java or C++; CHIPL is intended to be used by those trying to explore information in all its beauty, ranging from simple communication circuits to discussing beliefs and knowledge to studying economic value and how information can be understood to support social interaction to to support war. It provides the ability to simulate informative processes.

1.2 Purpose and Assumptions of CHIPL

The programming language CHIPL is designed to support the modeling of information systems. When using the phrase "information system," we are not using the cliche applied to something that comes out of a computer, but instead mean any group of processes whose informational characteristics are of interest. By easing the development of processes and the examination of the information within them, describing an information system in CHIPL is often easier or more useful or both for those studying information than using other programming languages.

1.2.1 Types of Statements

There are several types of statements in CHIPL. Some statement define instances of points in a conceptual space, others define processes, and some cause actions to take place.

Process Definitions Processes execute when called to do so. Defining a process consists of entering a set of commands that are executed, in order, until the end of the process is reached or the process is abnormally terminated. Normally, most programming in CHIPLconsists of defining these processes, while the statements that cause the execution of these processes are usually a few statements to a very few dozen, while processes may be very complicated and, for larger programs, will be the vast majority of the programming effort.

Instance Definitions An instance and a label together may be conceptualized as a location in reality from which information may be accepted by a process or to which information may be placed by a process. One might declare

Bob is a human.

This establishes that Bob is of class human.

Executable Statement A CHIPL program will have one or more executable statements. These are not definitions of processes or instances, but represent the names of processes that will execute with the indicated input and output.

Users may supply their own processes or they may use processes defined in *builtin.pro*.

Executable statements are executed in a random order. Statements are shuffled randomly once they are all entered. They are then executed in the same order each cycle.

Pre-Processed Statements Some statements are executed, in whole or in part, before the other statements. These statements are best understood as setting values and establishing groups of processes. These statements have commands that are in uppercase.

For examples, the statement

CYCLE 2.

establishes that the system will cycle two times, with most of the statements executing twice.

Preceding a statement with **RANDOM** provides a mechanism by which the statement can be randomly replicated with different values, allowing for the development of networks of processes.

Information When statements are executed, the input and the output to the statement are used in calculating the information that enters the process, the amount of information that is produced by the process at the output, and the information characteristics of the process itself.

1.3 Using CHIPL

CHIPL provides an environment in which one can study information. Processes are individually defined and then may be executed.

1.4 Downloading and Installation

CHIPL may be obtained from `http://InformationFromProcesses.org` It is usually best to initially create a directory or file folder that will be dedicated to CHIPL use. CHIPL should be downloaded into this directory or folder, or it may be downloaded elsewhere and then moved to the folder to be used for CHIPL. Once the downloaded CHIPL file has been placed in its folder, it should be unzipped, placing the CHIPL program, the program files, configuration files, and extraneous files to assist CHIPL in interfacing with other programs (e.g., LaTeX.) It is strongly recommended, although not required, that CHIPL installations have LaTeXand GNUPLOT installed, as using these will improve the quality and the range of outputs produced.

Once CHIPL has been downloaded and installed, the user is encouraged to try it initially by clicking on or typing the name of the program that corresponds to the system being used (all below are for Intel 32 bit and higher processors):

System	*Command*
Windows	chipl.exe
Ubuntu Linux	ubuntuchipl or ubuntu

There are several programs that are included in the zipped file. If no file name is entered, the default file is *default.pro*. One file, *test.pro*, is a sample program that the user might want to use when initially testing the installation. This program should complete executing within a second or two.

1.4.1 Compiling the program yourself

The source code for CHIPL is included and the user is certainly welcome to recompile the program if they wish to. This is probably easiest for those using a linux system with the commonly present gnu c++ compiler. Such a linux user would type something like

> c++ -pthread -D LINUX -std=c++0x -o ./runme -Os ./information-program.cpp

This produces an executable program in the file *chipl*. To "run" chipl, you need to just type "runme" (without the quotes).

1.5 Using CHIPL & Executing the program

1.5.1 Setting things up

To begin, a CHIPL user should obtain a copy of CHIPL and have it installed, usually by unzipping it and by possibly recompiling CHIPL. One might begin by writing a chipl program file using a text editor that saves the program as a text file, not as a formatted word processing document! Once the program file has been produced, execute CHIPL. Once this has completed, the user might bring up a browser to examine the latex output. One might also execute the gnuplot program and examine its output, although running GNUPLOT should be saved for later after one has more experience with CHIPL. In summary, have an editor that saves the text file of the user's program, be able to execute the CHIPL program with the text file of the user's program as input, and examine the output in the brower, possibly by hitting the "refresh" icon on the browser to examine the latest output.

1.5.2 Executing CHIPL

The program may be executed by entering the name of the program that corresponds to the operating system, e.g., chipl.exe or redhatchipl. The system will normally prompt for a chipl program file, but if the user types the name of the program file after the chipl.exe or redhatchipl command, with an intervening space, will result in chipl executing with that particular ".pro" file being used.

1.6 LaTeX

A word processor, LaTeX, has the ability to produce publication quality material using scientific notation and drawings, and is widely used in the physical

and computer sciences and mathematics. TEX, the basic system upon which LATEX is built, is supported and based at the American Mathematical Society (`http://ams.org`). Note that the LATEXoutput will not be generated when the size of the data is so large that the LATEXoutput would be unprintable on several pages.

1.7 Gnuplot

Gnuplot is a program that produces graphical output. Files may be produced by CHIPL that can, in turn, be read as data files by gnuplot when the networking of processes occurs. Gnuplot can be obtained from `http://www.gnuplot.org`. Users who are examining information in a network context are encouraged to read gnuplot documentation supplied with the program or to look at simple descriptions of gnuplot that exist on the World Wide Web. Gnuplot is easy to manipulate, and it can produce powerful displays of network characteristics when using output from CHIPL, such as one can see below for diffusion using small world network models.

Using gnuplot can be beneficial when studying information moving through networks. Each cycle of process executions results in a single line of data, suitable for processing with gnuplot or other packages. Each line contains some basic data, described in comments in the datafile, followed by optional sets of data associated with each metainformation tag for the output.

Examining the gnuplotchipl file will show a header, the first line, that shows the type of value for each position. These values are repeated, after the original data for the object, with the values for each metainformation tag.

The gnuplot statements that user must supply should be in a file named "gnuplot.default" (without the quote marks). If one wishes to manually execute the gnuplot program, one might enter

```
gnuplot gnuplot.default
```

where gnuplot.default accepts input from the gnuplotchipl file.

The gnuplot output will not be generated when there are only a few processing cycles.

Builtin Library Several basic functions are included in the *builtin.pro* file that is automatically loaded by CHIPL before executing any user supplies files. Users may examine the code in the *builtin.pro* library to consider how functions may be coded. Some of the basic functions in *builtin.pro* may be modified based upon user needs. As with any programming language, it is probably best not to modify basic functions but, instead, to add a modified

copy of the original function to the users own files, with the modified process having its own unique name.

1.8 Example Programs

A number of example programs are provided with CHIPL. Many have a 3 digit number within the file name. The first digit represents that chapter number of related material of the electronic book Information From Processes.

A process operates, possibly accepting input and possibly producing output. CHIPL enables one to model the process, accept input, produce output, and measure some of the characteristics of the input, the output, and the processing.

A Single Process A single process representing a traditional Shannon channel can be coded in CHIPL using the code in Figure 1.1, with the output shown in Figure 1.2. Here the first line contains a comment: any text preceded by

Figure 1.1: A sample program in CHIPL.

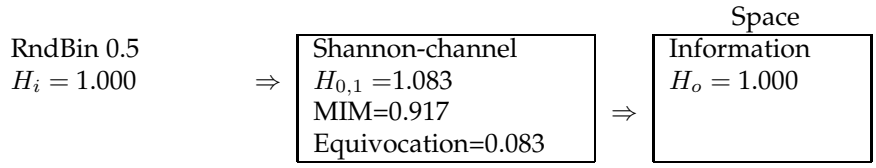
```
//  examplenoisyshannon.pro
CYCLES 5000.

Process shannon-channel
Code
0.5 randombinaryinput input
0.01 noisybinarycopy
# # output
Endcode.

Space is a entity.
Space's point-a shannon-channel Space's information.
```

"//" is treated as a comment, and only material to the left of "//" is processed by CHIPL. The CYCLES statement is optional, but allows one to specify the number of cycles that should occur. A large number will result in simulations producing numbers closer to their final or estimated values, while at the same time, a larger number of CYCLES results in more executing time and sometimes more use of storage space.

Note that all statements end with a period. Statements may be broken over several lines, with a new line starting at any point where a space is ac-

ceptable. Variables, such as "shannon-channel" may contain, after an initial letter, letters, numbers, or dash characters.

Figure 1.2: Output produced by 1.1.



The Shannon channel is defined as a processes by those operations between the words Code and Endcode. the first three data items, *0.5 randombinaryinput input*, generates a random binary number with a 1 occurring with probability 0.5. This then becomes input to the process. Within the process, information is copies from the input to the output, with noise being generated that switches the value of the bit being processed in 1% of the cases. This value is then outputted, with the hash marks indicating that those positions in the output (the same happens with input statements) will be taken up by entities and labels taken from a statement, such as the last statement. Thus the first # will be replaced with *Space* and the second # will be replaced with *point-b*.

Below is a different kind of program, whose focus is just to produce output. This is closer to what one finds with other programming languages, where the output itself is the focus of the language. The emphasis in CHIPL is on information phenomena, especially measurements of information. Figure 1.3 is a simple payroll program (examplepayroll.pro).

The output for the program is in Figure 1.4.

Figure 1.3: A sample payrollprogram in CHIPL.

```
// examplepayroll.pro
CYCLES 1.

Process sampleprocess
Code
printcr printcr
Employee %15s printf printspace drop
Rate %8s printf printspace drop
Hours %8s printf printspace drop
Gross %8s printf printspace printcr drop
======== %15s printf printspace drop
==== %8s printf printspace drop
===== %8s printf printspace drop
===== %8s printf printspace drop printcr

last
10.0 10.00 dave
35.0 30.00 charlize
37.0 23.00 bob
40.0 15.00 alice
dup
while
last ne
endwhilecondition
%15s printf printspace drop
2dup * -rot
%8.2f printf printspace drop
%8.2f printf printspace drop
%8.2f printf printspace drop
printcr dup
endwhile
printcr
Endcode.

Thisprogram is a nullprocess.
Thisprogram's null sampleprocess thisprogram's null.
```

Figure 1.4: Output produced by 1.3.

```
employee     rate    hours    gross
========     ====    =====    =====
   alice    15.00    40.00   600.00
     bob    23.00    37.00   851.00
charlize    30.00    35.00  1050.00
    dave    10.00    10.00   100.00
```

Chapter 2

Manipulating the Stack

CHIPL is based upon a stack as one of the basic units of storage. Imagines a pile of plates in a plate storage holder with a spring at the bottom. When there are no plates in the storage rack, there is usually a metal surface upon which one can place the first plate. Plates are placed, one on top of each other, and the spring compresses, with the weight of the plates moving the stack of plates downward. The last plate placed on the plate stack is the first plate to be removed, as it is on top of the stack. This is referred to as a last-in first-out device.

Placing items on the top of the stack Programs are always processed from left to right, and items that do not represent commands are placed on the stack. As an example, consider the following statements in a "program":

```
22 33
```

The first number read, 22, is moved onto the stack by CHIPL:

	Stack	
	Before	*After*
top:	*a*	*22*
	b	*a*
	c	*b*
	d	*c*
	⋮	⋮

Given the remaining number in the program, which now looks like:

```
33
```

the next number is placed on the stack, producing

	Stack	
	Before	*After*
top:	*22*	*33*
	a	*22*
	b	*a*
	c	*b*
	⋮	⋮

2.0.0.1 dup

Consider the following set of commands:

7 **dup**

The first number, *7*, is moved onto the stack by CHIPL. The next command, **dup**, causes the top item on the stack to be duplicated, with the duplicate being placed on top of the stack:

	Stack	
	Before	*After*
top:	*7*	*7*
	a	*7*
	b	*a*
	c	*b*
	⋮	⋮

The duplicate command is often helpful in making copies of data objects to be used for other operations.

2.0.0.2 2dup

In some cases, it is useful to have a copy of more than one item from the top of the stack. The **2dup** command duplicates the top two items. Consider the following set of commands:

4 5 **dup**

The first number, *4*, is moved onto the stack by CHIPL, followed by the second number, *5*. The next command, **2dup**, causes the top 2 items on the stack to be duplicated, with the duplicates being placed on top of the stack:

	Stack	
	Before	*After*
top:	*4*	*4*
	5	*5*
	a	*4*
	b	*5*
	⋮	⋮

2.0.0.3 3dup

The **3dup** command works like **2dup**, with the top 3 items on the stack being duplicated.

2.0.0.4 drop

The **drop** command removes the top item from the stack and disposes of it. Note that one of the major problems that beginners experience programming in stack based languages is using too many or too few drops. Too few drops and the stack progressively fills up to its predefined system limit. Too many drops and the stack is emptied and commands that use data on the stack find the stack empty and terminate the program.

2.0.0.5 2drop

The **2drop** command removes the top two items from the stack and disposes of them.

2.0.0.6 3drop

The **3drop** command removes the top three items from the stack and disposes of them.

2.0.0.7 swap

Consider the following set of commands:

7 8 **swap**

The first number, *7*, is moved onto the stack by CHIPL, followed by the number *8* which is placed on the top of the stack. The next command, **swap**, causes the top item on the stack to be swapped with the second item on the stack:

	Stack	
	Before	*After*
top:	*8*	*7*
	7	*8*
	a	*a*
	b	*b*
	⋮	⋮

2.0.0.8 rot

Consider the following set of commands:

7 8 9 **rot**

The first number, *7*, is moved onto the stack by CHIPL, followed by the number *8* which is placed on the top of the stack, followed by the number *9* which is placed on the top of the stack. The next command, **rot**, causes the third from the top item to be the top item on the stack, with the previous top and second to the top items each moving down one position on the stack:

	Stack	
	Before	*After*
top:	*9*	*7*
	8	*9*
	7	*8*
	a	*a*
	⋮	⋮

2.0.0.9 -rot

Consider the following set of commands:

7 8 9 **-rot**

The first number, *7*, is moved onto the stack by CHIPL, followed by the number *8* which is placed on the top of the stack, followed by the number *9* which is placed on the top of the stack. The next command, **-rot**, causes the top item on the stack to become the third item on the stack, with the other two items each moving up one position. Thus, the previous second item on the stack becomes the top item on the stack.

	Stack	
	Before	*After*
top:	*9*	*8*
	8	*7*
	7	*9*
	a	*a*
	⋮	⋮

2.0.0.10 over

In some cases, it is useful to have a copy of the second item on the stack. While **dup** is able to obtain a copy of the top of the stack, **over** provides the original stack with a new item on top that is a copy of the original second item on the stack. Assume the following code.

4 5 **over**

The first number, *4*, is moved onto the stack by CHIPL, followed by the second number, *5*. The next command, **over**, causes a copy of the second item on the stack being placed on top of the stack:

	Stack	
	Before	*After*
top:	*5*	*4*
	4	*5*
	a	*4*
	b	*a*
	⋮	⋮

2.0.0.11 clearstack

The command **clearstack** clears out all contents of the stack. This command should be used with caution, as one can argue that it should never be needed if programs are developed and tested properly.

2.0.0.12 isstackempty

The command **isstackempty** returns whether there is or is not something left on the stack.

Chapter 3

Mathematical Operations

3.1 Arithmetic

The basic arithmetic operators are written as though the first variable written has after it the operator followed by the second variable, and thus the command

10 2 **divide**

corresponds to the arithmetic problem, written in English, "ten divided by 2." Consider such math problems as written in their normal order but with the operator placed at the end of the variables (postfix notation.) These are implemented so that the first operand in normal written arithmetic is the second item on the stack and the top of the stack is the item after the operation, as normally written.

3.1.1 Addition

Addition uses either the *+* (plus) operator or the operator **add**. Given the instructions

7 6 5 4 +

with 4 and 5 on the top of the stack, the two values are removed from the stack and the sum, 9, is placed on the stack:

	Stack	
	Before	*After*
top:	*4*	*9*
	5	*6*
	6	*7*
	7	*x*
	⋮	⋮

If neither of the top two values on the top of the stack are valid numbers, an error message is produced and the program halts.

3.1.2 Subtraction

Addition uses either the - (minus) operator or the operator **subtract**. Given the instructions

```
7 6 5 4 -
```

with 4 and 5 on the top of the stack, the two values are removed from the stack and the difference, 9, is placed on the stack:

	Stack	
	Before	*After*
top:	*4*	*9*
	5	*6*
	6	*7*
	7	*x*
	⋮	⋮

If neither of the top two values on the top of the stack are valid numbers, an error message is produced and the program halts.

3.1.3 Multiplication

Multiplication uses either the * (asterisk) operator or the operator **multiply**. As with addition and subtraction, the two numbers are removed from the stack and the answer, the product of the two numbers, is placed on the stack.

3.1.4 Division

Division uses either the / (slash) operator or the operator **divide**. As with addition and subtraction, the two numbers are removed from the stack and

the answer, the first of the numbers (second on the stack) is divided by the second number (the top of the stack). This answer is placed on the stack. In the case of dividing two integers, only the whole number portion of the answer is produced.

3.1.5 Remainder

The **remainder** or **mod** or **%** operator divides one integer by another integer and returns the remainder. As with addition and subtraction, the two integers are removed from the stack and the answer, the first of the integers (second on the stack) is divided by the second integer (the top of the stack). The remainder is placed on the stack.

3.1.6 Maximum

Maximum uses either the *max* operator or the operator **maximum**. Two numbers are removed from the stack and the maximum of the two numbers is placed on the stack.

3.1.7 Minimum

Maximum uses either the *min* operator or the operator **minimum**. Two numbers are removed from the stack and the minimum of the two numbers is placed on the stack.

3.2 Logical Operations

Many Boolean operators used in CHIPL are already familiar to those who have programmed in other languages. These operators return either the value *true* or *false*, two value that have an interpretation to humans in terms of ethics but, for a computer, the values *true* and *false* are both arbitrary values with no particular meaning.

3.2.1 Comparative Operators: Greater and Less Than

Several operators compare the two values on the top of the stack, comparing the second on the stack to the top of the stack, as though the operator were written between them. All these operators return either the values *true* or *false* after removing the two values from the stack. As an example, the greater than operator, **gt**, determines whether the second element on the stack is greater than the item on the top of the stack:

	Stack	
	Before	*After*
top:	*3*	*false*
	4	*a*
	a	*b*
	b	*c*
	⋮	⋮

and

	Stack	
	Before	*After*
top:	*4*	*true*
	3	*a*
	a	*b*
	b	*c*
	⋮	⋮

Most of the operators can be written in a couple ways, and these are shown in parentheses below: equality (=, **equals**), inequality (<>, ! =), greater than (>, **gt**), greater than or equal to (>=, **ge**), lesser than (<, **lt**), and lesser than or equal to (<=, **le**).

Figure 3.5 provides some examples, with the output shown in Figure 3.6.

3.2.2 Manipulating Truth Values

Certain logical operations (explained in Chapter 4 of *Information From Processes*) [2] allow for the combination of truth values on the stack. The **and** operator given the values *true* and *false*

	Stack	
	Before	*After*
top:	*true*	*false*
	false	*a*
	a	*b*
	b	*c*
	⋮	⋮

returns the value *false*, while two *true* values produce the value *true*:

	Stack	
	Before	*After*
top:	*true*	*true*
	true	*a*
	a	*b*
	b	*c*
	⋮	⋮

The operators that work with two truth values include and (**and**), or (**or**), and exclusive or (**xor**).

Some of these operators only work with a single truth value. The **not** operator takes the top of the stack and replaces it with the logical inverse: *true* is replaced with *false* and *false* is replaced with *true*.

3.3 Checking for Type

A number of commands can be used to check for data types. For each of these, the object is removed from the stack and the value true or false is placed on the stack, indicating whether the removed object is of the type checked for.

The following commands check to see if the object is of the associated type:

Command	*Question Asked*
isnullq	object has never been assigned a value?
isstringq	object is a string?
isintq	object is an integer?
isintegerq	object is an integer?
isfloatq	object is a floating point number?
isvectorq	object is a statement?
isstatementq	object is a statement?

Figure 3.3 provides some examples, with the output shown in Figure 3.4.

3.4 Random Operators

Numeric and logical values may be changed randomly during processing.

3.4.0.1 randombinary

The **randombinary** command accepts a floating point value p from the stack, removes it, and produces an object that has a value 1 with probability p and a 0 with probability $1 - p$.

3.4.0.2 randominteger

The **randominteger** command accepts a floating point value n from the stack, removes it, and produces an object that has a value in the range from 0 to n, inclusive, with an equal probability of each integer being produced. The object with this generated number is then placed on the stack.

3.4.0.3 randomfloat

The **randomfloat** command accepts a floating point value f from the stack, removes it, and produces an object that has a value in the range from 0 to, but not including f with an equal probability of each floating point number being produced. The object with this generated number is then placed on the stack. Examples of these three types of randomly generated numbers are provided in Figure 3.7, with output given in Figure 3.8.

3.4.0.4 noisybinarycopy

When trying to introduce noise into binary data, one may use the **noisybinarycopy** command. Taking a probability of change from the stack and a data object, it transforms the binary value into its binary complement (0 becomes 1 and 1 becomes 0.) The object is then placed on the stack

0 *0.01* **noisybinarycopy**

Here the probability, in this case 0.01, is removed from the stack, as is the object with the initial value 0. The object's value 0 is changed with probability 0.01 and the object value, whether still 0 (and there is a 99% chance it is still 0) or whether changed to 1, is placed on the stack.

3.4.0.5 noisyintegercopy

When trying to introduce noise into integer data, one may use the **noisyintegercopy** command. Taking a probability of change from the stack, followed by the maximum value of integers, and a data object, integer value may be

transformed into another integer value with the probability of change being the probability obtained from the stack. The integer value is changed with this probability to the integer value above the original value or the integer value below the original value, the values above and the values below being equally probable. Values above the maximum value are wrapped around to 0, and values below 0 are wrapped around to the maximum value. The resulting object is placed on the stack. Consider the example

3 *5* *0.33* **noisyintegercopy**.

Here the probability of change is one third. The object's value 3 is changed with probability 0.33 and the object value, whether still 3 (and there is a two third chance chance it is still 3) or whether changed to 2 with probability of one sixth or to 4 with a probability of one sixth. This resulting value is placed on the stack.

3.4.0.6 concatenate

When trying to "glue" two strings together to produce a new string, composed of the two other strings, the **concatenate** command is used. The two items are removed from the stack and the first placed on the stack is placed on the left, the second placed on the stack on the right, and then the resulting string is returned to the stack. Consider the example

LL *RRRR* **concatenate**.

After this command has executed, the string "LLRRRR" is placed on the stack.

Figure 3.1: Sample CHIPL program illustrating mathematical operations.

```
// examplearithmetic.pro

CYCLES 1.

Process sampleprocess
Code
arithmetic_examples print drop printcr printcr

Value_3_plus_4_equals print printspace drop
3 4 add print printcr drop

Value_19_minus_7_equals print printspace drop
19 7 subtract print printcr drop

Value_3_times_7_equals print printspace drop
3 7 multiply print printcr drop

Value_3_times_negative_7_equals print printspace drop
3 -7 multiply print printcr drop

Value_7_divided_by_3 print printspace drop
7 3 divide print printcr drop

Value_7.0_divided_by_3 print printspace drop
7.0 3 divide print printcr drop
printcr
Below_are_examples_with_printf print drop printcr printcr

Value_7_divided_by_3_with_printf_format_%6.3f print printspace drop
7 3 divide %6.3f  printf printcr drop

Value_7_divided_by_3_with_printf_format_%4d print printspace drop
7 3 divide %4d  printf printcr drop

Value_7.0_divided_by_3_with_printf_format_%15.3f print printspace drop
7.0 3 divide %15.3f  printf printcr drop

Endcode.

Thisprogram is a nullprocess.
Thisprogram's null sampleprocess thisprogram's null.
```

Figure 3.2: Output from the program in Figure 3.1.

```
arithmetic_examples

value_3_plus_4_equals 7
value_19_minus_7_equals 12
value_3_times_7_equals 21
value_3_times_negative_7_equals -21
value_7_divided_by_3 2
value_7.0_divided_by_3 2.33333

below_are_examples_with_printf

value_7_divided_by_3_with_printf_format_%6.3f  0.000
value_7_divided_by_3_with_printf_format_%4d    2
value_7.0_divided_by_3_with_printf_format_%15.3f          2.333
```

Figure 3.3: Examples of type checking in CHIPL.

```
// exampledatatypes.pro
CYCLES 1.
Process sampleprocess
Code
type_checking_examples print drop printcr

printcr ========================= print printcr drop
is_string_q print printcr drop
fred dup isstringq swap print printspace drop print printcr drop
3 dup isstringq swap print printspace drop print printcr drop
0.3 dup isstringq swap print printspace drop print printcr drop
printcr ========================= print printcr drop
is_int_q print printcr drop
fred dup isintq swap print printspace drop print printcr drop
3 dup isintq swap print printspace drop print printcr drop
0.3 dup isintq swap print printspace drop print printcr drop
printcr ========================= print printcr drop
is_float_q print printcr drop
fred dup isfloatq swap print printspace drop print printcr drop
3 dup isfloatq swap print printspace drop print printcr drop
0.3 dup isfloatq swap print printspace drop print printcr drop
.3 dup isfloatq swap print printspace drop print printcr drop

Endcode.

Thisprogram is a nullprocess.
Thisprogram's null sampleprocess thisprogram's null.
```

Figure 3.4: Output from the program in Figure **??**

```
type_checking_examples

=========================
is_string_q
fred true
3 false
0.3 false

=========================
is_int_q
fred false
3 true
0.3 false

=========================
is_float_q
fred false
3 false
0.3 true
0.3 true
```

Figure 3.5: Example of comparison operators in CHIPL.

```
// examplecomparisons.pro
CYCLES 1.

Process sampleprocess
Code
printcr comparative_examples print drop printcr printcr
Value_7.0_lt_3 print printspace drop
7.0 3 lt print printcr drop
Value_7_lt_3 print printspace drop
7 3 lt print printcr drop
Value_7_le_3 print printspace drop
7 3 le print printcr drop
Value_7_ge_3 print printspace drop
7 3 ge print printcr drop
Value_7_ge_3 print printspace drop
7 3 gt print printcr drop
Value_7_lt_3 print printspace drop
7 3 lt print printcr drop
Value_7_le_3.0 print printspace drop
7 3.0 le print printcr drop
Value_7_equal_3.0 print printspace drop
7 3.0 = print printcr drop
Value_3_equal_3.0 print printspace drop
3 3.0 = print printcr drop
Value_3.0_equal_3.0 print printspace drop
3.0 3.0 = print printcr drop
Value_3_equal_3 print printspace drop
3 3 = print printcr drop
Value_3_notequal_3.0 print printspace drop
3 3.0 != print printcr drop
Value_3.0_notequal_3.0 print printspace drop
3.0 3.0 != print printcr drop
Value_3_notequal_3 print printspace drop
3 3 != print printcr drop
Value_4_notequal_3.0 print printspace drop
4 3.0 != print printcr drop
Value_4.0_notequal_3.0 print printspace drop
4.0 3.0 != print printcr drop
Value_4_notequal_3 print printspace drop
4 3 != print printcr drop
Endcode.

Thisprogram is a nullprocess.
Thisprogram's null sampleprocess thisprogram's null.
```

Figure 3.6: The output from the program in Figure 3.5.

```
comparative_examples

value_7.0_lt_3 false
value_7_lt_3 false
value_7_le_3 false
value_7_ge_3 true
value_7_ge_3 true
value_7_lt_3 false
value_7_le_3.0 false
value_7_equal_3.0 false
value_3_equal_3.0 true
value_3.0_equal_3.0 true
value_3_equal_3 true
value_3_notequal_3.0 false
value_3.0_notequal_3.0 false
value_3_notequal_3 false
value_4_notequal_3.0 true
value_4.0_notequal_3.0 true
value_4_notequal_3 true
```

Figure 3.7: Statements that generate random numbers.

```
// examplerandom.pro
CYCLES 1.
Process sampleprocess
Code
random_numbers print drop printcr printcr

binary_0.75: print printspace drop
15 1 beginfor
0.75 randombinary print printspace drop
endfor  printcr printcr

integer_3: print printspace drop
15 1 beginfor
3 randominteger print printspace drop
endfor  printcr printcr

float_2.0: print printspace drop
5 1 beginfor
2.0 randomfloat print printspace drop
endfor  printcr printcr


Endcode.

Thisprogram is a nullprocess.
Thisprogram's null sampleprocess thisprogram's null.
```

Figure 3.8: Output from program in Figure 3.7.

```
random_numbers

binary_0.75: 0 1 0 0 0 1 1 0 1 1 1 1 1 1 0

integer_3: 2 0 2 0 0 3 0 3 1 2 2 2 3 3 3

float_2.0: 1.02586 1.67822 1.22528 0.592063 1.2751
```

Chapter 4

Input and Output

Input and output commands in CHIPL work with the stack, with input placing an object on the stack and output taking an object from the stack and sending it out. A key point in CHIPL is the acceptance of input into a process and the output produced by the process and analyzing the amount of information produced by the process.

Before beginning on specifics of the input and output instructions, it might be worthwhile to consider a statement like

Bob's ear hears Lee's voice.

The two people, Bob and Lee, are *instances* and the points on their bodies where sounds move from, and to, are *labels*.

4.1 General Input and Output

Both the **input** and **output** and their ancillary commands have two operands.

Variables and # The stack may contain either the exact name of either the instance or the label or both or the character "#" in lieu of an exact name. When the character # is encountered as the instance or the label for either inputs or outputs, the corresponding instance or label is used from the executable statement. The top value on the stack is treated as the instance and the second item on the stack is the label, an item or label within the instance.

4.2 Input

The basic input command, **input** takes an object from the *instance* and the *label* and places it on the stack. When the following is executed,

mouth bob **input**

the object that is stored in the instance *Bob* at label, or location *mouth*.

Note that the order here of label followed by instance is the opposite of the order in executable statements, where one might find something such as

Bob's mouth speaksto Fred's ear.

4.3 Random Inputs

Instead of accepting input from another user-defined process, a process may accept data from a random data generator

4.3.0.7 randombinaryinput

The **randombinaryinput** command accepts a floating point value p from the stack, removes it, and produces an object that has a value 1 with probability p and a 0 with probability $1 - p$. The commands

0.75 **randombinaryinput input**

results in a value that has the binary value 1 three quarters of the time on the stack, with a 0 being placed one quarter of the time.

4.3.0.8 randomintegerinput

The **randomintegerinput** command accepts an integer n from the stack, removes it, and produces an object that has a value in the range from 0 to n, inclusive, with an equal probability of each integer being produced. The object with this generated number is then placed on the stack. The commands

4 **randomintegerinput input**

produce and place on the stack one of the equally probable values: 0, 1, 2, 3, or 4.

4.3.0.9 randomfloatinput

The **randomfloatinput** command accepts a floating point value f from the stack, removes it, and produces an object that has a value in the range from 0 to, but not including f, with an equal probability of each floating point number being produced. The object with this generated number is then placed on the stack.

4.4 File Input

Input may accept integers, floating point numbers, or words from an existing file, with one data item per line. This is done by, within the **input** command, supplying the filename and the keyword **file**. For example, given

mydata123.txt file input # # output

the **input** command will read numbers from the file *mydata123.txt*, a user supplied file and filename. Note that no provisions are made for end-of-file detection, and the number of CYCLES should be set to not attempt to read past the end of the file. An example is provided in file example225.pro.

4.5 Output

Output When the following is executed,

Brian's ear **output**

the object that is stored on the top of the stack is removed from the stack and placed in the instance *Brian* at label, or location *ear*.

When the system attempts to write multiple outputs to a single label, only one of the outputs is actually written and used when statistics are computed. When multiple outputs are written to a file, however, all are written to the file.

same In some cases, the output instance of a statement may need to be the same as the input instance. This is achieved by using the **same** keyword as the output instance. Note that this will not execute if the input instance in the process is something other than #, the number sign. If one uses something else to generate input, such as a statement that generates random input, the **same** at the output instance will not execute as desired. In this situation, the programmer might generate a random number and place it in a temporary

label in the desired instance. This label may then be processed and placed into the desired label on the output (**same**) instance.

While this may seem silly in the case of working with single instances, working with large numbers of instances at once, using the CHANGE syntax, may make this necessary. In this case, a single generating process may produce output at a number of instances whose names are determined by the CHANGE syntax. A second statement can then use these labels as input and produce desired output at correct label for the **same** instance.

4.6 Print to Screen

Several commands print to the screen, which is useful for producing some kinds of output or for debugging programs. These commands also simulataneously print their content to a file titled "printtoscreen.txt". Examples in this manual that were produced without "boxes" being displayed were likely produced by this form of print instruction.

Note that using these options allows one to insert literals that are to appear in the output. After printing, these literals must be removed from the stack using the **drop** command.

print Consider the following set of commands:

sample_text **print**

The value *sample_text* will be printed both to the screen and to the "printtoscreen.txt" file.

printspace The **printspace** command produces a single space character on the screen and in the "printtoscreen.txt" file.

printcr The **printspace** command produces a single space character on the screen and on the "printtoscreen.txt" file.

printf Consider the following set of commands:

8 %d **printf**

The value *8* will be sent to the screen consistent with the printf formatting command used in other C and C++ related programming languages with "printf"

commands. Note that no quote marks should be used around the formatting string, unlike most other C-related languages where the format would be within quotes. Similarly, only one format is allowed and only one object is printed at a time. Printing strings is handled poorly by C++ and thus handled poorly here; specifically, left justification is not handled properly. Users unfamiliar with such a command from other languages are recommended to search for "printf" in most search engines, as it is a commonly discussed output formatting in languages such as C and C++.

An example file illustrating formatting is *examplepayroll.pro*.

Chapter 5

Flow of Control

Statements execute in sequence from beginning to end or as read, from left to right. However, certain statements break out of the sequential paradigm, modifying the flow of control, the order in whcih statements execute.

5.1 Conditional Statements

Conditional statements are statements that are executed when a certain condition exists. This sort of "if . . . then . . . " statement is implemented in CHIPL by first checking for a truth value on the stack and then executing zero or more statements based on whether the truth value was true or false. Consider the following:

true **truebranch** *itistrue print printcr drop* **endbranch**

The value *true* on the top of the stack is seen by the **truebranch** command and the statements following the **truebranch** command, up to the **endbranch** command are executed. In this case, *itistrue* is printed. Note that the truth value remains on the stack at the end of this process. This is useful because it one to check for two conditions, whether the truth value is true or whether it is false. For example,

3 2 gt **truebranch** *itistrue print printcr drop* **endbranch falsebranch** *itisfalse print printcr drop* **endbranch** drop

prints out the *itistrue* message, but not the *itisfalse* message. On the other hand

Figure 5.1: Program showing conditional execution (an "if" statement) with the output below.

```
// exampletruefalse.pro

CYCLES 1.

Process sampleprocess
Code
This_should_be_a_true_value: print drop
true
truebranch True_Value endbranch
falsebranch False_Value endbranch
drop
print
drop
printcr printcr

This_should_be_a_false_value: print drop
false
truebranch True_Value endbranch
falsebranch False_Value endbranch
drop
print
drop
printcr printcr
Endcode.

Thisprogram is a nullprocess.
Thisprogram's null sampleprocess thisprogram's null.
```

```
this_should_be_a_true_value:true_value

this_should_be_a_false_value:false_value
```

2 3 gt **truebranch** *itistrue print printcr drop* **endbranch falsebranch** *itisfalse print printcr drop* **endbranch** drop

the *itisfalse* message will print but not the *itistrue* message. Note the final **drop** command, which removes the truth value from the stack so that the stack is as it was before the conditions began executing.

Figure 5.2: Program with a while statement.

```
// examplewhileloop.pro
CYCLES 1.

Process sampleprocess
Code
printcr while_loop print printcr drop

65 dup
while
70 lt
endwhilecondition
1 add dup print printspace
endwhile
printcr
Endcode.

Thisprogram is a nullprocess.
Thisprogram's null sampleprocess thisprogram's null.
```

```
while_loop
66 67 68 69 70
```

5.2 Conditional Loops

A general form of conditional repetition may be achieved through use of what are often referred to as "while" loops. Three commands are necessary for the execution of while loops. The first is the **while** command, which occurs at the beginning of the while loop. Following the **while** command but before the **endwhilecondition** command is the test condition. If the test condition is true, the **endwhilecondition** command transfers control to the statements fol-

lowing it, with those statements, up to the **endwhile** command. When the **endwhile** command is reached, execution control is returned to the **while** command, which

The example in Figure 5.2 shows the program described above along with the output from the program shown at the bottom.

Figure 5.3: Program that produces a "for" loop, with output below.

```
// exampleforloop.pro
CYCLES 1.

Process sampleprocess
Code
Looping_example_showing_66_through_70_inclusive
print printcr drop
printcr for_loop print printcr drop
//debug
70 66
beginfor
getindex print printspace drop
endfor
printcr
Endcode.

Thisprogram is a nullprocess.
Thisprogram's null sampleprocess thisprogram's null.
```

```
looping_example_showing_66_through_70_inclusive

for_loop
66 67 68 69 70
```

5.2.1 For Loops

Another form of loop is commonly referred to as a "for loop." A for loop executes a predetermined number of times and is most useful when the number of executions of the loop is known before entering the loop.

At the beginning of the for loop is the **beginfor** command, while at the end of the for loop is the **endfor** command. Before the **beginfor** command are two integers on the stack. The second one is the final value for a hidden variable

when the for loop executes, while the top of the stack contains the initial value for the hidden variable when the for loop executes. If one wants to access the hidden variable, during the execution of the for loop, the **getindex** command may be used to place the value of the variable on the top of the stack.

The example in Figure 5.3 shows a for loop with the final value of the loop being 70 and the initial value being 66. When executed, the output shown at the bottom of Figure 5.3 is produced.

Multiplication Table Example The example in Figure 5.4, with the output in Figure 5.5, provides a multiplication table with the values for the row position times the column position. By placing one for loop within another for loop, and only moving to a new line when all the data for a first line is printed. Thus, the first numeric line in Figure 5.5 shows the values $1, 2, 3, 4$. As each number is printed, a following space is printed. Once the final number on the line is printed and the inner for loop ends, the **printcr** (outside the inner loop but still inside the outer loop) causes the next output to appear on a new line.

Figure 5.4: Program that produces a multiplication table using nested "for" loops.

```
// examplenestedfor.pro
CYCLES 1.

Process sampleprocess
Code
Multiplication_example print printcr drop
printcr table_of_row_value_times_column_value print printcr printcr drop
//debug
4 1
beginfor
getindex
4 1
beginfor
dup
getindex
multiply
print printspace drop
endfor
printcr
endfor
Endcode.

Thisprogram is a nullprocess.
Thisprogram's null sampleprocess thisprogram's null.
```

Figure 5.5: Multiplication table produced by the program in Figure 5.4.

```
multiplication_example

table_of_row_value_times_column_value

1 2 3 4
2 4 6 8
3 6 9 12
4 8 12 16
```

Chapter 6

MetaInformation

6.0.2 addMI

Given the operation

addToThisObject addThis metaInformationTag **addMI**

the command **addMI** adds metainformation *addThis* to the metainformation tag *metaInformationTag* for the object *addToThisObject*. All three variables are removed from the stack, and the revised object, with the metainformation, is added to the stack. If the metainformation tag already has a value, the value is replaced by *addThis*.

6.0.3 addOrReplaceMI

Given the operation

addToThisObject addThis metaInformationTag **addorreplacemi**

the command **addorreplacemi** adds metainformation *addThis* to the metainformation tag *metaInformationTag* for the object *addToThisObject*. A prior occurrence of *metaInformationTag* will be replaced and the new value inserted. All three variables are removed from the stack, and the revised object, with the metainformation, is added to the stack.

6.0.4 labelExistsMIQ

Given the operation

object labelToSearchFor **labelexistsmiq**

the command **labelexistsmiq** looks for the metainformation label *labelToSearchFor* for the object *object*. The object containing the Boolean True or False are returned depending on the presence or absence of the *labelToSearchFor* in *object*.

Both variables are removed from the stack, and the object with true or false is added to the stack.

6.0.5 booleanExistsMIQ FINISH

Given the operation

object metaInformationLabel **booleanexistsmiq**

the command **booleanexistsmiq** looks for the presence of the metainformation tag *metaInformationLabel* for the object *object*. Both values are removed from the stack. If the *metaInformationLabel* exists and the metainformative value associated with the label is either the Boolean true or false, then an object with the Boolean value true is placed on the stack.

6.0.6 findMI

Given the operation

objectToSearch metaInformationTag **findmi**

the command **findmi** searches the object *objectToSearch*. for the value associated with the metainformation tag *metaInformationTag*. Both variables are removed from the stack, and the value of the metainformation is added to the stack. The operation assumes that the *metaInformationTag* exists in object *objectToSearch*. If there is doubt about this existence, the **labelexistsmiq** operation should be used first.

6.0.7 removeMI

Given the operation

object metaInformationLabel **removemi**

the command **removemi** searches the object *object* for the label *metaInformationLabel* and then removed the label and the associated metainformative value. Both variables are removed from the stack, and the modified form of *object*is added to the stack. If the *metaInformationLabel* does not exist in *object* then *object* is placed on the stack unmodified.

6.0.8 sizeMI

Given the operation

object **sizeMI**

the command **sizeMI** searches the object *object* for the number of metainformative labels. The original object is removed from the stack and the number of labels is placed on the top of the stack.

6.0.9 clearMI

Given the operation

object **clearmi**

the command **clearmi** removes all metainformative labels from *object*. The original *object* is removed from the stack and the modified form of *object* is placed on the stack.

6.1 Example

A further set of examples testing the operation of the metainformation operators is in file *example395.pro.*

Figure 6.1: Sample program using metainformation.

```
// examplemetainformation.pro
CYCLES 1.
Process testMI Code sampleobject
inputvalueis print drop printspace print printcr
true label1 addMI false label2 addMI
samplevalue label3 addMI
dup sizemi print printspace drop
is-number-of-labels print drop printcr printcr
dup label3 booleanexistsmiq print printspace drop
is-whether-label3-is-boolean print drop  printcr printcr
dup label1 labelexistsmiq print printspace drop
is-whether-label1-exists print drop printcr
dup label2 labelexistsmiq print printspace drop
is-whether-label2-exists print drop printcr printcr
dup label1 findmi print printspace drop
is-value-of-label1 print drop printcr
dup label2 findmi print printspace drop
is-value-of-label2 print drop printcr
dup label3 findmi print printspace drop
is-value-of-label3 print drop printcr printcr
dup label1 findmiq print printspace drop
is-truth-value-of-mi-label1 print drop printcr
dup label2 findmiq print printspace drop
is-truth-value-of-mi-label2 print drop printcr
dup label3 findmiq print printspace drop
is-truth-value-of-mi-label3 print drop

EndCode.
Fred is a entity.
Fred's elbow testMI Fred's shoulder.
```

Figure 6.2: Output from example metainformation program in Figure 6.1.

```
inputvalueis sampleobject
3 is-number-of-labels

false is-whether-label3-is-boolean

true is-whether-label1-exists
true is-whether-label2-exists

true is-value-of-label1
false is-value-of-label2
samplevalue is-value-of-label3

true is-truth-value-of-mi-label1
false is-truth-value-of-mi-label2
false is-truth-value-of-mi-label3
```

Figure 6.3: Example metainformation.

```
// examplemetainformation2.pro
CYCLES 1.
Process testMI Code sampleobject
inputvalueis print drop printspace print printcr
true label1 addMI false label2 addMI
samplevalue label3 addMI

false label1 addorreplacemi true label2 addorreplacemi
secondvalue label3 addorreplacemi
dup label1 findmi print printspace drop
is-value-of-label1 print drop printcr
dup label2 findmi print printspace drop
is-value-of-label2 print drop printcr
dup label3 findmi print printspace drop
is-value-of-label3 print drop printcr printcr printcr
abnormal-condition print printcr drop
label2 removemi label2-removed print printcr drop
dup label1 findmi print printspace drop
is-value-of-label1 print drop printcr
dup label2 findmi print printspace drop
is-value-of-label2 print drop printcr
dup label3 findmi print printspace drop
is-value-of-label3 print drop printcr printcr printcr
dup clearmi clear-mi print drop
sizemi print printspace drop is-number-of-labels print drop
EndCode.
Fred is a entity.
Fred's elbow testMI Fred's shoulder.
```

Figure 6.4: Output from Figure 6.3.

```
inputvalueis sampleobject
false is-value-of-label1
true is-value-of-label2
secondvalue is-value-of-label3


abnormal-condition
label2-removed
false is-value-of-label1
INVALIDINVALID is-value-of-label2
secondvalue is-value-of-label3


clear-mi0 is-number-of-labels
```

Chapter 7

Statements

Statements are list of objects that, taken together, form a larger object. A statement is also referred to as a vector.

7.0.1 makeemptyvector and makeemptystatement

The commands **makeemptyvector** and **makeemptystatement** produce an empty vector or statement and place this empty object on the stack.

7.0.2 getobjectatposition

The command **getobjectatposition** takes a statement and, on the top of the stack, an integer n. These two are removed, and the object at location n is placed on the top of the stack. The position of the first item in the statement is 0, the second 2, etc.

7.0.3 removeposition

The command **removeposition** takes an object and, on the top of the stack, an integer n. These two are removed, and the object has removed from it the n^{th} item in the object. The shorter object is then placed on the top of the stack.

7.0.4 insertobjectatend

A statement has above it an object, both of which are removed from the stack by the **insertobjectatend** command. The object is added to the end of the vector or statement. The revised, longer statement is then placed on the top of the stack.

7.0.5 insertobjectatstart

A statement has above it an object, both of which are removed from the stack by the **insertobjectatstart** command. The object is placed added at the beginning of the vector or statement, and other objects are all moved down one position. The revised, longer statement is then placed on the top of the stack.

7.0.6 sizeofstatement and sizeofvector

The commands **sizeofvector** and **sizeofstatement** produce an integer that indicates the size of the vector or statement. If the object is empty, the size is zero and the integer 0 is returned.

Chapter 8

Miscellaneous Instructions

debug The **debug** command causes the system to provide some basic debugging information. The command does not modify the stack and the program continues execution when the **debug** command completes executing.

Abnormal program termination A program may be artificially terminated by using either the **quit** command or the **emergency** command. The **quit** command performs a normal "peaceful" termination, while **emergency** causes termination by executing the error handling routine which accepts other errors and terminates as gracefully as possible.

Chapter 9

Pre-processing

9.1 Cycles

The **CYCLES** command controls the number of times that the set of executable statements is executed. For example

CYCLES 20.

will result in the program executing 20 times.

9.2 Non-Random Generation of Processes

Multiple occurrences of statements may be generated through the use of **CHANGE** commands embedded in statements. Consider the following example:

Z[CHANGE1TO100BY1]yz's pointx mycopy Z[CHANGE1TO100BY1]yz's pointy.

The system generates different versions of the statement, with what is between the left and right square brackets ("[" and "]"), as well as the brackets themselves, being replaced by an integer. The expression **CHANGE1TO100BY1** generates 100 statements, with the expression being replaced by a value of 1, then 2, and so on to the last statement, with 100. One statement will be generated with the instance being *X1yz*, the second *X2yz*, the third *X3yz*, and so on. The **CHANGE** expression results in a "for" loop being executed as one finds in many programming languages, with the starting value, the ending value, and the increment being used as in most "for" loops. There should be no spaces between the left and right square brackets. Note that in this code example,

there are two **CHANGE** expressions; since each one generates a value from 1 to 100, there will be 100×100 statements generated.

9.3 Random Repetition of Processes

9.3.1 Random

Statements beginning with the **RANDOM** command, which is followed by a probability that the statement will be generated, might look like

RANDOM 0.048 Z[CHANGE1TO100BY1]yz's pointx mycopy Z[CHANGE1TO100BY1]yz's pointx.

Instead of $100 \times 100 = 10000$ statements being produced, all statements are generated but an individual statement is only included in the final result with the probability that follows the **RANDOM** expression. That is, the probability is used by a random truth generator, with true being generated with the provided probability. Only true statements are then included in the output result. This is useful in implementing networks of processes.

9.3.2 Small Worlds

The Watts-Strogatz model [3] of network generation, described more fully in *Information from Processes*, produces a set of statements (and associated processes) that are linked so as to model something between a random network, in which instances are randomly connected to other instances, and a small world network, in which most or all of the instances are linked only to their neighbors. The **WATTS** command might appear in a context such as this:

WATTS 50 7 0.020 x[CHANGE]yz's perception mycopy x[CHANGE]yz's perception.

Three numbers occur after the **WATTS** command: the first represents the number of statements generated (in this case 50, the second represents the number of neighbors (in this case 7), and the third represents the probability that a given link from the instance being generated to a neighbor is changed to a link to a randomly selected instance in the network.

Note that the **CHANGE** in the Watts-Strogatz **WATTS** command is different than in the other examples provided earlier. Because the Watts-Strogatz model needs to control more variables, the **CHANGE** element is simpler and

the appropriate values are inserted where the **CHANGE** element and the matching square brackets occur. An example of a program using this form of small world network is given in Figure 9.1.

The output of this program, when processed by GNUPLOT, can produce the output graphics given in Figure 9.2.

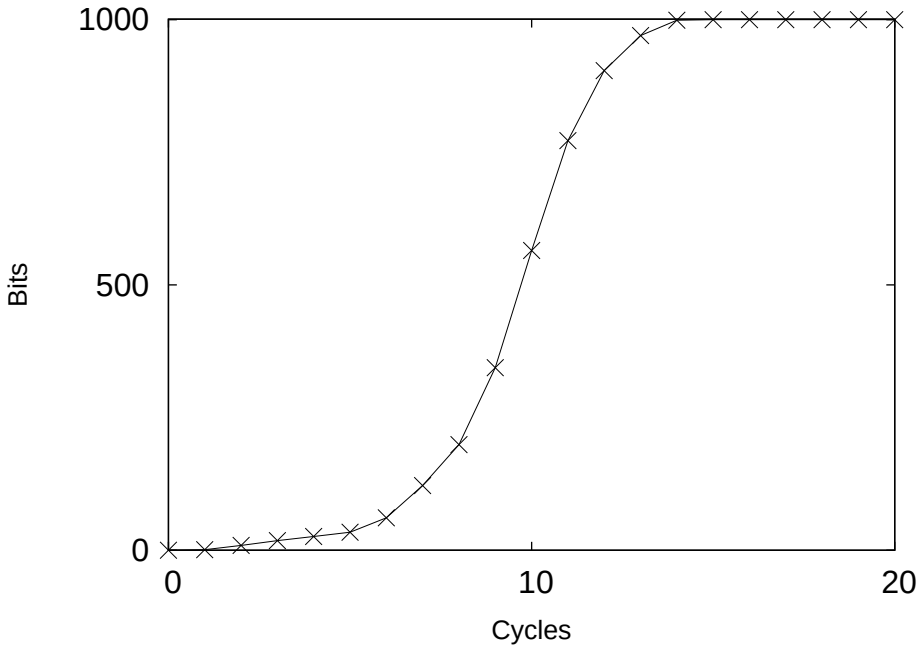
9.3.3 Random Graph Networks

The Random Graph model may be obtained when the third parameter of the **WATTS** command, the probability that a pre-established link to a neighbor is changed to a randomly selected network node, is changed to 1.0.

Figure 9.1: A sample small world program in CHIPL.

```
// examplesmallworld.pro
// ================================================
Process cyclezero
// first cycle only, all modes
Code
cyclenumber 0 = truebranch
0 x # output
endbranch drop
Endcode.
// ================================================
Process cycleone
// second cycle only, all modes
Code
cyclenumber 1 = truebranch
1 x # output
endbranch drop
Endcode.
// ================================================
Process copyx
Code
cyclenumber 1 > truebranch
//  only copy true x
x # input 1 =
  truebranch
    1 x # output
  endbranch
  drop
endbranch
drop
Endcode.
// ================================================

CYCLES 30.

null is a entity.
node[CHANGE0TO1000BY1] is a entity.
null's null cyclezero node[CHANGE0TO1000BY1]'s x.
null's null cycleone node50's x.
WATTS 1000 5 0.030 node[CHANGE]'s x copyx node[CHANGE]'s x.
```

Figure 9.2: The output of a sample small world program in CHIPL.



Chapter 10

System Variables

10.0.3.1 pause

The **pause** command causes the program to delay for approximately 1 second.

10.0.3.2 printvalueon

The **printvalueon** command causes the program to begin printing the latest value as input or output.

10.0.3.3 printvalueoff

The **printvalueoff** command causes the program to stop printing the latest value as input or output. Not printing the latest value is the system default.

10.1 Gnuplot File Format

The gnuplot file contains a number of different values on each line. There is a repetitive structure for each line's entry.

The first item on each line is the cycle number. Following this is the repetitive structure, with the following being included for each output label (with the labels being combined across all instances for that cycle.)

1. The label.
2. The number of times to which this label has been written in this cycle.
3. The average amount of below.

4. The sum of the numeric values outputted into this label during this cycle. In the case of Boolean values, the value *true* is represented by a 1 in accumulating the total and *false* is represented by a 0 in accumulating the total.
5. The average entropy of this output variable during this cycle.
6. The total of the entropy values for this output variable during this cycle.
7. Undefined
8. Undefined
9. Undefined
10. Undefined

Part II

More Information Theory with CHIPL

Chapter 11

Measures of Performance

There are several measures of the amount of information present in the input, processing, and informative output characteristics. These are described non-quantitatively in `http://InformationFromProcesses.org` Chapter 2.

11.1 The Amount of Information

11.1.1 Self-Information

The amount of self-information that is contained in or associated with an object when the probability of the object X having value x is $\Pr(x)$ or $\Pr(X = x)$, is the logarithm of the inverse of the probability, or

$$I_x = \log 1/\Pr(x). \tag{11.1}$$

The choice of a logarithmic base corresponds to the choice of a unit for measuring self-information. If the base 2 is used the resulting units are in binary digits, or *bits*. This form of information may be understood as how much one learns when the state of an object is revealed, that is, when the uncertainty about the actual value of the object is removed.

11.1.2 Entropy

The average information of a random variable, which can take on a number of values, is computed as

$$H_X = \sum_{x_i \in X} \Pr(x_i) \log_2 \frac{1}{\Pr(x_i)}, \tag{11.2}$$

where x_i is the i^{th} possible value for random variable X.

Entropy is shown in CHIPL for inputs to processes (information that was produced by other processes) and for outputs from a process.

11.1.3 Joint Entropy

The average information of two random variables, taken together, which can take on a number of discrete values, is computed as

$$H_{X,Y} = \sum_{x_i \in X, y_i \in Y} \Pr(x_i, y_j) \log_2 \frac{1}{\Pr(x_i, y_j)}, \tag{11.3}$$

where x_i is an instance of random variable X and y_i is an instance of random variable Y.

11.1.4 Conditional Entropy and Mutual Information

Assume that one represents the input to a process by random variable X and the output information produced by the process is denoted as random variable Y. The conditional entropy or equivocation is the average information of a random variable, given a second variable, and is computed as

$$H_{Y|X} = -\sum_{x_i \in X, y_j \in Y} \Pr(x_i, y_j) \log_2 \frac{\Pr(x_i, y_j)}{\Pr(x_i)} = H_{X,Y} - H_X. \tag{11.4}$$

This equivocation of process output Y about process input X, $H_{Y|X}$, measures what Shannon refers to as the "average ambiguity" of the information at the output of the process due to input X. This ambiguity can be a form of information loss.

The rate of processing of information through the process may be computed as the entropy of the input minus the equivocation at the processes' output, that is, the entering information minus the loss. This amount of information is the rate of transmission and is referred to as the mutual information between the input and the output of the process. The mutual information between the input and the output of a process is also referred to as the transinformation. This may be interpreted as the amount of uncertainty or information that remains when we consider the amount of information or uncertainty that exists about process output Y after the ambiguity at Y about X is removed. If Y has a large quantity of equivocation or ambiguity about X, then $I_{X,Y}$ will be relatively smaller, whereas if Y has little equivocation or ambiguity about X, $I_{X,Y}$ will be relatively larger.

The mutual information may be computed as

$$I_{X,Y} = H_X - H_{Y|X} = \sum_{x_i \in X, y_j \in Y} \Pr(x_i, y_j) \log_2 \frac{\Pr(x_i, y_j)}{\Pr(x_i)\Pr(y_j)}, \tag{11.5}$$

which uses the relationship that exists when two variables are taken independently $\Pr(x_i)\Pr(y_j)$ and when two variables are taken jointly $\Pr(x_i, y_j)$ in computing the information that one variable has about the other.

Chapter 12

Information in Relational Databases

Informational representations, such as in books, pictures, or music, often contain unstructured information. An informative representation of a tree could take any of a number of forms and still represent the same tree, and ambiguity is often found in unstructured information. In many situations, information also may be represented in a more structured form, as in a table, with labeled columns indicating what can be included in each column. The column headers may serve as a quick guide as to where readers should search. These labels and the architecture of the table provide structure, as opposed to the lack of labels or lack of a fixed architecture on a painting or in nature language, making them unstructured. A set of information values is often referred to as a *database*.

Representational information may contain variables that take on a number of values. For example, the variable *house owner* might be any of a number of individuals, as well as organizations, corporations, and groups of individuals in a query or a specific database entry. We assume that a single variable can have only one value at a given time. Any of these possible house owners, whether one person or some other entity, may be represented as a single value held by the house owner variable.

An ordered set of related variables is referred to as a *relation*. A *tuple* is a set of specific values for the relationship, such as that Arwen lives at 53 Mallard Court. A printed telephone directory often contains the names for individuals or organizations, as well as their address and their telephone number. This would be referred to as a 3-tuple.

Such a relation can be presented as a table, such as in Table 12.1. In tables, each column represents a specific variable (called an *attribute*) and each row a

Name	*Address*	*House Owner*
Arwen	53 Mallard Court	Mike
Caitlyn	53 Mallard Court	Mike
Abigail	53 Mallard Court	Mike
Roger	8 Hilltop Circle	Sally
Simon	8 Hilltop Circle	Sally

Table 12.1: Unnormalized sample information.

Name	*Address*
Arwen	53 Mallard Court
Caitlyn	53 Mallard Court
Abigail	53 Mallard Court
Roger	8 Hilltop Circle
Simon	8 Hilltop Circle

Address	*House Owner*
53 Mallard Court	Mike
8 Hilltop Circle	Sally

Table 12.2: Two tables that normalize the data in Table 12.1. These are from *Information From Processes* [2].

specific tuple. One of the variables is the *key*, and serves as the access point to the relation by allowing one to access one or more specific tuples. For example, a telephone directory is usually accessed by looking up the name of a person or organization. This first column of the directory, the name, is the key for the table.

Relationships may contain redundant information, that is, information that is repeated after its first occurrence [1]. If Mike owns the house at 53 Mallard Court and this is entered into a table, it would be redundant and repetitive for the table to contain the information that Arwen's house is at 53 Mallard Court if it was also entered that Arwen's house is owned by Mike if it were elsewhere indicated that Mike owns the house at 53 Mallard Court. We note that this information about Mike owning the house at 53 Mallard Court is in the records about Arwen, Caitlyn, and Abigail. To remove redundancies from relationships, they may be normalized so that no relationship exists between any non-key variables, except for that induced by the presence of the key variable itself. Normalization often occurs to avoid anomalies that may

occur when facts are added or deleted from a database.

In Table 12.1, if Roger and Simon both moved out of the house at 8 Hilltop Circle, the fact that Sally owned the house would be deleted. If this table were normalized so that the information in it were instead in the form of Table 12.2, Roger and Simon could move out of the house at 8 Hilltop Circle and the fact that Sally owned the house at this address would remain in the set of relationships.

Structured information has advantages over unstructured information for representing reality, but the inverse also holds. While structured information may provide a simple, unambiguous representation for reality, and it may be easy to retrieve information from a table, structured information may not be as successful as unstructured information at capturing all types of data, such as the beauty in DaVinci's painting of the *Mona Lisa* or pain expressed in a human language. Species of animals at a variety of levels communicate successfully by providing auditory and visual clues of an unstructured nature. Clearly, both unstructured and structured information are important in practice when representing reality.

Figure 12.1: A program that reads in the datafile in Figure 12.1 and processes the data.

```
// exampledatabase10.pro
// database entropy
CYCLES 5.

Process database-tuples
Code
printvalueoff
exampledatabase10data.pro file input
exampledatabase10data.pro file input
concatenate
exampledatabase10data.pro file input
concatenate
# # output
Endcode.

Space is a entity.
Space's point-a database-tuples Space's tuple-entropy.
```

Figure 12.1 is a program that represents analysis of the entropy in the first relation where all data is present. The entropy in this case is computed as 2.322 which is also $\log_2 5 = (\log 5)/(\log 2) = 2.322$. Because there are 5 lines or

tuples here, this makes sense.

Note that there are 3 input statements in the program, one for each attribute in a tuple. These strings are concatenated together so that when the three values have been inputted, they are concatenated into a single large string, whose entropy is then calculated. If one wanted to not process the first attribute, it would be inputted and then the **drop** command used. Only one **concatenate** command would be needed in this case, to bring together the second and third attributes. In general, by using **drop** commands and **concatenate** commands, one can input all fields and drop or include those fields of interest in computing entropy values.

Note that any table where there is n lines without an entire tuple being a duplicate, then the entropy associated with the table is $\log +2n$. Each line or tuple thus has $\log_2 n$ bits of information, that is, the amount of information about which of the n tuples in the table this is is computed as $\log_2 n$.

If one computes the entropy for the two subtables in Figure 12.2, the first, with 5 lines, has the entropy of 2.322, while the second, with only 2 lines, has an entropy of $\log_2 2 = 1$ bit.

The entropy of various components may be combined to produce the joint entropy of the entire table or database. For example, the entropy of the three combined columns Name, Address, and House Owner may be computed as the entropy of the Name and Address plus the entropy of the Address and House Owner minus what is in common and duplicated between them, the entropy of the Address. In this case, using only the first letters of the Attribute or column name, $H(N, A) + H(A, HO) - H(A) = H(N, A, HO)$ This may be computed as $2.32 + 1 - 1 = 2.32$. This addresses how two smaller tables may be joined into a larger table, with the entropy of the two being combined, less the entropy of a column that is included twice.

To understand the different relationships that exist in tuples, the reader is strongly encouraged to examine [1].

Bibliography

[1] T. T. Lee. An information theoretic analysis of relational databases, parts I and II. *IEEE Transactions on Software Engineering*, SE-13(10):1049–1072, October 1987.

[2] R. M. Losee. *Information From Processes: Information Creation, Use, and Representation*. Springer, New York, 2012. `http://InformationFromProcesses.org`.

[3] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, June 4 1998.

Index

<, 25
<=, 25
<>, 25
=, 25
>, 25
>=, 25
*, 23
+, 22
-, 23
-rot, 20
. (period), 13
/, 23
%, 24
“ (space), 14
2drop, 19
2dup, 18, 19
3drop, 19
3dup, 19

add, 22
addMI, 47
addorreplacemi, 47
ambiguity, 67
and, 25, 26

beginfor, 44
booleanexistsmiq, 48

CHANGE, 57–59
CHANGE1TO100BY1, 57
clearmi, 49
clearstack, 21
concatenate, 28, 72
conditional entropy, 67
CYCLES, 57

database, 69
debug, 56
divide, 22, 23
drop, 19, 39, 43, 72
dup, 18, 21

emergency, 56
endbranch, 41, 43
endfor, 44
endwhile, 44
endwhilecondition, 43
equals, 25
equivocation, 67

falsebranch, 41, 43
file, 38
findmi, 48

ge, 25
getindex, 45
getobjectatposition, 54
gt, 24, 25

information
    unstructured, 69
input, 36–38
insertobjectatend, 54
insertobjectatstart, 55
isstackempty, 21

labelexistsmiq, 48
le, 25
lt, 25

makeemptystatement, 54
makeemptyvector, 54
maximum, 24
minimum, 24
mod, 24
multiply, 23
mutual information, 67

network
    random graph, 59
    small world, 58
new line, 14
noisybinarycopy, 27

noisyintegercopy, 27, 28
not, 26

or, 26
output, 36, 38
over, 21

pause, 62
period (.), 13
print, 39
printcr, 45
printf, 39
printspace, 39
printvalueoff, 62
printvalueon, 62

quit, 56

RANDOM, 10, 58
random graph network, 59
randombinary, 27
randombinaryinput, 37
randomfloat, 27
randomfloatinput, 38
randominteger, 27
randomintegerinput, 37
remainder, 24
removemi, 48, 49
removeposition, 54
rot, 20

same, 38, 39
sizeMI, 49
sizeofstatement, 55
sizeofvector, 55
small world network, 58
space, 14
subtract, 23
swap, 19

transinformation, 67
truebranch, 41, 43

WATTS, 58, 59
while, 43, 44

xor, 26