Variables, Decisions, and Scripting in Construct

Brian R. Hirshman and Kathleen M. Carley
September, 2009
CMU-ISR-09-126

Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Center for the Computational Analysis of Social and Organizational Systems
CASOS technical report.

Report Documentation Page		*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **SEP 2009**	2. REPORT TYPE	3. DATES COVERED **00-00-2009 to 00-00-2009**
4. TITLE AND SUBTITLE **Variables, Decisions, and Scripting in Construct**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Carnegie Mellon University ,School of Computer Science,Pittsburgh,PA,15213**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**68**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

Abstract

Simulation designers benefit from a flexible system for creating scenarios that are easy to modify, expressive, and allow for more complex interventions to be assessed. This technical report introduces a C-like scripting language that can be used with Construct in order to support numeric variables as well as user-specified decisions. This scripting language can be used to specify outputs in a targeted manner, allowing the user to modify the type of output created without modifying the underlying code. The scripting language also allows the simulation to be self-modifying, allowing the knowledge, beliefs, or decisions of an agent to influence the evolution of the overall simulation. Such abilities greatly increase the power of Construct and extend the scope of the simulation.

Table of Contents

List of Figures

1 Introduction & Motivation

Construct is a complex, agent-based simulation system grounded in sociology and cognitive science which seeks to model the processes and situations by which humans interact and share information. Construct is an embodiment of constructuralism (Carley 1986), a theory which posits that human social structures and cognitive structures co-evolve so that human cognition reflects human social behavior, and that human social behavior simultaneously influences cognitive processes. Recent work with Construct has sought to influence both the social and cognitive richness of the Construct model. To do so, this work has introduced a number of input and output mechanisms in order to better allow users to build more expressive models. As these inputs and outputs have grown in complexity, it eventually became necessary to write a series of scripts which could assist in the setup of Construct input and output. The variable, decision, and scripting systems described in this technical report were developed to support such work.

The purpose of this technical report is to discuss the variable, scripting, and decision systems in Construct. It is not necessarily designed to introduce readers to the key processes or algorithms in Construct. Such an introduction can be found in other publications, including technical reports on specifying agents in Construct (Hirshman and Carley 2007), the core algorithms in Construct (Hirshman, Kowalchuck et al. 2009), the literacy and information access system in Construct (Hirshman and Carley 2008), and the help files associated with executable. Nor is it expressly designed to discuss the properties of the problems that can be solved using variables or the scripting language. Additionally, several conference and journal publications may also provide key insight as to how Construct has been used in past research. While the earliest work with the tool focused on defining the constructuralism meta-theory and examining the processes by which subgroups form and change (Carley 1991), work in the late 1990s focused on organizational applications and non-human agents (Carley 1997; Carley 1999); more recent work has focused on building and describing more cognitively complex agents (Hirshman, Martin et al. 2008; Carley, Martin et al. 2009). While the scripting language described in this technical report is indeed new, it builds off of nearly twenty years of work in Construct input design.

It is important to note that the variable, decision, and scripting system described here are current to Construct version 3.9, released in the summer of 2009. However, the work herein is not wholly new. Earlier Construct versions had support for some of the functionality specified – specifically, the variable system outlined in Section 2 has been supported since at least Construct version 3.5 – although many features have been added in the last year in order to improve expressiveness and ease of use. Importantly, however, the input deck specifications described in this document – though not the functionality – may be changed or modified in future releases of the tool. A GUI interface is in development which may make interaction with the variables, decisions, and scripting language easier, but it might require changes to some of the semantics of the input file in order to support such a tool. Please consult the Construct help files for the most up-to-date methods for creating and using the technical features described in this document.

This technical report seeks to describe both the uses and the rationale for the variable and decision functions provided in Construct. While the initial part of the document describes why certain design decisions were made and provides examples of the types of possible decisions, the details of these decisions are provided in appendices. This technical report has sought to quote from actual examples of Construct input decks when possible, though several of the examples are simplified in order to illustrate points more clearly. Examples of input decks, such as

descriptions of variables and decisions, use a `monospace font` will be used to more clearly differentiate the example from the surrounding text. For many of the examples, it should be possible to copy the text written in `monospace font` and paste them directly in Construct input files in order to verify the result discussed in the text.

It is worthwhile to briefly outline important sections of the ConstructML input format in order to better understand the variable, decision, and scripting systems. ConstructML is a modified version of the DynetML framework used in other tools such as ORA (Carley and Reminga 2004) and is a specific version of XML used in the CASOS center. While the names of the ConstructML tags are specified in other technical literature (e.g. Hirshman and Carley 2007), the majority of the functionality is specified by attributes on those tags, especially the 'name' and 'value' attributes. When these tags attributes are read by the ConstructML parser, they are evaluated and the simulation is loaded. Construct variables are defined in the first of these tags, the `construct_vars` tag, and allow the user to have substantial flexibility in declaring constants. Decisions are defined in the last tag, the `operations` tag, though they represent an improvement over previous output formats. Lastly, scripting is used by both the variable system, the decision system, and any other parts of the ConstructML input deck (where flexible input may be needed. Each of these features will be described in more detail throughout this document.

The remainder of this document is structured as follows. Section 2 describes what variables can do in Construct, and specifies how to create and use them in Construct version 3.9. Section 3 describes the customizable decision functionality available in Construct, and how it can be used to create a designer-specified per-agent output. Section 4 provides an overview of the scripting language which can be used to create variables and decisions, while a more technical description of the language is provided as an appendix. Section 5 provides several applications and examples of the variables, decisions, and scripting language in use. Section 6 presents the summary and conclusions.

2   Support for Variables

Construct has included support for input variables, as part of the ConstructML input format, since Construct version 2. Such variables allowed constants to be specified at the top of the input deck, which made it relatively simple and straightforward to modify to the deck. However, these variables were limited to experimenter-specified constants and were rather brittle. Construct 3.5 and above included support for mathematical operations using variables. The method of creating, initializing, and using variables described in this section is current as of Construct 3.9 and is the most powerful and robust variable specification model yet used in Construct. Since there have been many changes between Construct 3.5 and 3.9, the variable syntax discussed in this section may not be supported by older Construct executables.

The remainder of this section introduces Construct variables and is organized as follows. Section 2.1 discusses how to specify variables in the ConstructML input deck, and their usefulness. Section 2.3 describes how to reference variables once they have been created. Section 2.4 introduces the `with` statement, which can greatly increase the expressivity of an individual variable. Section 2.5 provides some stylistic guidelines for using variables in the Construct input deck. Section 2.6 discusses minor caveats regarding variables.

Figure 1: Creation of Construct variables

Variable

```
<construct>
  <construct_vars>
    <var name="var1" value="1"/>
    <var name="var2" value="'variable 2'"/>
    <var name="var3" value="variable 2"/>
    <var name="var4" value="construct::intvar::var1"/>
    <var name="var5" value="construct::intvar::var3+1"/>
  </construct_vars>
  ...
</construct>
```

2.1 Specifying variables

Variables in Construct are specified in a `construct_vars` tag in the Construct input file. This tag should be the first tag under the enclosing `construct` tag in order to make the variables visible and easily accessible. Variables are specified using a `var` tag beneath the `construct_vars` tag. There should be only one `construct_vars` tag per Construct input file, though the number of `var` tags that can be contained under that tag is effectively unlimited. An example input deck with four Construct variables (`var1`, `var2`, `var3`, and `var4`) can be seen in Figure 1.

All `var` tags should have a `name` attribute, which specifies the unique name of the variable, and a `value` attribute, which specifies the value that should be assigned to this variable. Variable names should consist of a string of alphanumeric characters and underscores; including other characters within the name may cause problems for variable interpolation. Variable values should consist only of valid script lexemes as described in Section 4 and Appendix B. A variable must have only one name attribute and one value attribute; neither the name attribute nor the value attribute can be blank.

Any non-`var` tag placed as a subtag of the `construct_vars` tag will be silently ignored.

Variables can be initialized in three ways: they can be initialized with constants, in terms of other variables, or using mathematical or logical combination of constants and variables. All three types of variables can be seen in Figure 1.

Creation of a variable with a constant value is straightforward. Any number can be in the `value` section of the `var` parameter, as seen in the declaration of the variable `var1` in Figure 1. Both positive and negative integers, as well as positive and negative floating point values, can be initialized in this fashion. String values are slightly more complicated. The creation of a string variable can be done in two fashions.   To do so explicitly, as in `var2` in Figure 1, surround the variable value in single quotes (`'`). These single quotes must go between the double quotes which specify the ConstructML variable, as can be seen in the initialization of variable `var2` in Figure 1. However, it is also possible to declare variables implicitly without surrounding the value in single quotes, as seen in variable `var3` in Figure 1. If construct does not recognize a bare word term in a variable declaration, it is converted silently into a string. However, this implicit conversion process will remove white space. Thus, while the value of variable `var2`

Figure 2: Examples of macros and with statements in variables

Variable	Value
`<var name="var1" value="$i$" with="$i$=1"/>`	`"1"`
`<var name="var2" value="construct::intvar::var$i$"` `    with="$i$=1"/>`	`"1"`
`<var name="var$i$" value=` `    "construct::intvar::var$i:int-1$+1" with="$i$=3"/>`	`"2"`
`<var name="var4" value="$j$"` `    with="$i$=3,$j$=construct::intvar::var$i$,verbose"/>`	`"4"`

would be `variable 2`, including the space, the value of variable `var3` is `variable2` without the space.

Variables can also be defined in terms of other variables, as occurs for `var3` in Figure 1. The general syntax for such an expression is `construct::<TypeExpr>::<VariableExpr>`, where `<TypeExpr>` is a valid type expression and `<VariableExpr>` is the name of a construct variable. This will retrieve the variable value for `<VariableExpr>` and store it as the value for the current variable. This syntax is discussed more thoroughly in Section 2.3.

Lastly, variables can be defined in terms of mathematical, logical, or other types of expressions. For instance, it is possible to use addition, subtraction, multiplication, division, and logical statements (among others) in order to define variables. However, there are several caveats to the evaluation order, as discussed in Section 2.3.

2.2 Evaluating variables

Variables in Construct, with one exception which will be discussed at the end of the section, will be evaluated immediately after being read by the Construct parser. When a Construct input deck is read, the first item that is read is the `construct_variables` tag and the included `var` tags. These `vars` are loaded in the order specified by the file. This means that the variable declared first in the `construct_variables` tag will be evaluated first, then the second variable, and so forth until all variables are loaded. Within each variable, the `var` name is loaded before the value.

The names and values in the `var` tag, like most expressions in Construct, are evaluated left-to-right from the beginning of the script. Parentheses can be used in order to specify that certain expressions should be evaluated before others. However, mathematical and casting operations will be evaluated in a right-to-left fashion, which may be confusing for new users. Both of these are worthy of further note.

As can be seen for `var4` in Figure 1, mathematical and relational statements can be used when defining a variable. At this time, however, there is no specific notion of operator precedence at this time, and by default math all math operations are parsed from right to left. In the absence of explicit instruction, expressions are evaluated from right to left, meaning that the expression `2/4.0+1`, as seen in variable `var1` in Figure 3, will yield the value `0.4`. This is because the rightmost operation, the addition, is evaluated first while the leftmost operation is evaluated second. Thus, the operation is ultimately analogous to evaluating `2/5.0`. To modify the operation in order to yield the result would probably be expected with standard operator precedence orders, parentheses can be explicitly added. This is done in variable `var1` in Figure

4

Figure 3: Caveats in evaluation order

Variable	Value
`<var name="var1" value="2/4.0+1"/>`	`"0.4"`
`<var name="var2" value="(2/4.0)+1"/>`	`"1.5"`
`<var name="var3" value="4-1-1"/>`	`"4"`
`<var name="var4" value="4-1+1"/>`	`"2"`
`<var name="var5" value="(2/4):float"/>`	`"0.0"`
`<var name="var6" value="(2/4.0):float"/>`	`"0.5"`
`<var name="var6" value="2/4.0` `with="delay_interpolation"/>`	`"2/4.0"`

3 in to create the expression `(2/4)+1`. When the parentheses are added, the division is performed prior to the addition and the resulting value is changed. Note that this right-to-left evaluation order may be most confusing when subtraction is involved. For instance, the evaluation of variable `var3` will yield a result of `4` because the rightmost subtraction is performed first. The result of the subtraction `1-1` will yield `0`, which will then be subtracted from 4. Similarly, the evaluation of variable var4 will yield a result of 2, since the addition is performed before the subtraction. Adding parentheses to these expressions is, therefore, highly recommended in order to correct for this unexpected behavior.

Second, the types of the variables used are extremely important, since the type of the result is due to the types of the expressions which create it. Writing the expression `2/4` as a variable value will yield the value zero, as seen in `var5` in Figure 3, since both variables are integers. When this zero is then cast to a float using the cast operation (as discussed in Appendix B, part VII), the value remains zero. However, if a floating point value is desired, one of the values must initially be a floating point. Casting either value using the cast operation (by using a `:float` cast in this case) or explicitly writing either value as a float (for instance, `2/4.0` as seen in variable `var6`) will lead to a floating point result.

In order to delay the evaluation of Construct variables, the `delay_interpolation` parameter can be provided as a `with` variable (with variables are discussed further in Section 2.4). The `delay_interpolation` with variable will ensure that the expression is not evaluated immediately after being parsed, which may hide parsing errors until the variable is evaluated at a later time. This may be useful if the variable references a variable which has not yet been created, or if the variable holds information which may be related to a decision. If the variable is eventually evaluated as an `expressionvar`, as discussed in Section 2.3, the evaluation rules described above will be used in the evaluation of the statement. Until that point, the variable will be treated as a string of characters.

2.3   Referencing variables

Construct variables can be referenced in the other sections of input deck – the `construct_params`, `nodeclasses`, `networks`, or `operations` tags, as discussed in other technical reports such as "Specifying Networks in Construct" (Hirshman and Carley 2007). However, they can also be referenced when constructing other `construct_vars` as long as the referenced var has already been defined; this allows for the creation of more complex variables. A variable can be referenced by writing

`construct::<TypeExpr>::<VariableExpr>`, where `<TypeExpr>` is a valid type expression and `<VariableExpr>` is the name of a construct variable.

All variables are defined as strings in the input deck, and are cast to a specific type `<TypeExpr>` when in use. There are five possible values for `<TypeExpr>`, as briefly introduced below.

- `boolvar`, which references the variable as a Boolean variable. The variable being referenced can be any string, but will evaluate to true only if the string is "true" or a non-zero number value. If the Boolean variable is used in a numeric context, it is automatically converted to 0.0 if false and 1.0 if true.
- `intvar`, which references the variable as an integer variable. If the value cannot be cast to a number, then the value 0.0 is used. Otherwise, if the number is a floating-point value, the number is silently truncated to (i.e. not rounded) to the nearest positive or negative integer. An example of an invar reference can be seen in Figure 1, where `variable3` is defined as a reference to the value of `variable1`.
- `floatvar`, which references the variable as a float. If the value cannot be cast to a number, then the value 0.0 is used. If an integer value is used as a float variable, any math performed on this variable will have a float result.
- `stringvar`, which references the variable as a string. All variable types can be converted to strings.
- `expressionvar`, which treats the referenced variable as an expression to evaluate. The result of this expression's evaluation is used and returned. The `expressionvar` allows for variables to serve as meta-variables and to collect commonly-used expression subparts in a particular location. Note that the expression can evaluate to a bool, int, float, or string; the expressionvar is agnostic as to which is used. Thus, if an `expressionvar` is used in a larger expression, it may be necessary to cast the result (see Appendix B, part VII) to a desired type in order to manipulate it further.

Two other types, `agentgroupvar` and `knowledgegroupvar`, cannot be used in the creation of variables because they rely on the membership networks (which have not been created when the variables are initialized). For additional information about any of these types, see the appropriate subsections of Appendix B which describes the syntax of variable references in more detail.

Note that a variable can be used in multiple different contexts. A variable can be used as an integer in one context, a float in another, and a string in a third. The location at which the variable is used will determine the type of the variable.

2.4   Macros and `with` statements

Construct variables can be defined using macros. These macros allow for substitution into a variable name and/or value, which can allow a complex set of variables to be created. Macro variables are defined by placing the variable to be substituted within dollar signs (`$`); for instance, the any expression containing `$i$` will depend upon the value of the macro variable `i`. The value of `i` must be specified in a `with` attribute which is attached to the same `var` tag as the variable name and value.

6

Figure 4: Creating multiple variables simultaneously

Variable	Value
`<var name="letters" value="a,b,c"/>`	`"a,b,c"`
`<var name="numbers" value="1"/>`	`"1"`
`<var name="variable$i$" value="$i$"` `   with="$i$=construct::stringvar::numbers"/>`	`"1"`

`<var name="variable$letter$$number$"`
`   value="$letter$$number$"`
`   with="$letter$=construct::stringvar::letters,`
`       $number$=construct::stringvar::numbers"/>`
	`"a1"`,
	`"b1"`,
	`"c1"`

Macro variables can use most strings of lower-case characters, upper case characters, numbers, and the underscore. However, there are some restrictions. First, all macro variables must start with an alphabetic character; variables beginning with a number or an underscore may not be interpreted properly. Second, macro names are case sensitive, so the macro variable `$i$` is different than the variable `$I$`. Third, macro variables cannot be the same as any of the reserved words in the scripting language; variables with names such as `$foreach$` will cause unpredictable behavior and likely will cause Construct to error and exit. Fourth, macro names must be unique within each `var` tag; if a `with` variable is declared twice, it will be treated as the same variable. Last, the macro variable only applies to one specific ConstructML variable and does not carry over into any other subsequent variables.

When the value of a macro is specified within a `var` name or value, the macro value is substituted directly into the name or variable value. Thus, if the value of macro variable `i` is set to 1, then the expression `variable$i$` will evaluate to `variable1`, as can be seen in the third example of Figure 4. The substitution of the variable will occur before any expressions are evaluated. Thus, if the previous example was actually embedded in the expression `construct::intvar::variable$i$`, the value returned by the macro would be the value of the variable named `variable1`, not the string "variable1".

When macro variables are being evaluated, as discussed in Section 2.2, it is possible to perform math on the variable as it is being evaluated. For instance, the variable `var3` in Figure 2 (page 4) contains operations to modify the macro variable, as specified. In the evaluation of this variable, `$i$` is initialized to 3 but subtraction is performed in order to modify the value to 2. Evaluating the expression `construct::intvar::var2`, then will return the value 1, and the addition operation will increment the value to 2. This type of modification is extremely powerful, and can be used to chain together a large number of variable declarations quickly. For instance, using this method, it is possible to create a sequence of variables, each of which references the previously-created variable and defines a new one; one can define `var4` in Figure 2 in terms of `var3`, a fifth var in terms of `var4`, and so forth.

Construct variables can specify the values for macro parameters using a `with` statement. If a macro parameter is not defined in a with statement, Construct will output an error message and exit. If a single with value is provided, the `with` statement must contain the name of the variable surrounded by dollar signs ($), followed by an equals sign (=), followed by the value. This can be clearly seen in the three examples provided in Figure 2. If more than one `with` value are provided – for instance, if there are multiple macro variables within the same

expression, as can be seen in the declaration of `var4` in Figure 2 – the `with` values must be provided in a comma separated list.

If one wanted to create a sequence of variables similar to var3, as defined in Figure 2, one could either copy-paste the variable value and change the with expression every time. Alternatively, however, it is possible to specify that a macro variable iteratively take on multiple values. An example of such a syntax can be seen in the last example of Figure 4.

In this example, there are two macro variables, `$letter$` and `$number$`, the former of which iterates over the values provided in the construct variable letters (defined in the first row) and the latter of which iterates over numbers (defined in the second row). The value of each variable is iteratively assigned during three different macro evaluations. In the evaluation, `$letter$` is a while number is 1, leading to the creation of a variable named a1 with value a1. Since there are no more numbers to evaluate, the next letter is examined. In the next evaluation, `$letter$` is b while number is 1, leading to the creation of variable b1 with value b1. Again, all numbers have been exhausted, so the third letter is examined. A third evaluation creates variable c1 with value c1, and at this point, all letters and numbers have been used, so all possible variables have been created.

This multiple-macro variable system has a number of benefits. First, if one desires to change the number of variables generated, one need only modify the number of values set for the `with` attribute `$number$` as opposed to generating separate variables. Thus, if the number of numbers increases from 1 to 2 (perhaps by changing the value of number to 1,2, the requisite variables will be created. This can save a substantial amount of time when making changes, as variables which are dependent on other variables may be automatically created. Secondly, this saves a substantial amount of space in the input deck, and can help to minimize typos. One possible downside, however, is that it may take some debugging to verify that the macro has been set up correctly.

Note that the use of a multiple-value macro will require multiple values to be specified in the `with` expression, as seen in Figure 4. However, it is not possible to rewrite the with statement of Figure 4 directly including these values, as the statement `with="$letter$=a,b,c"` will not create three different variables. Construct will not know whether the commas are meant as separators between variable values or between different with parameters. By referring to the list of variables using another variable, though, Construct will be able to distinguish between with variables and lists of values over which to iterate. If the user wishes to place the list of values directly in the with statement, which is not recommended, the values over which to iterate should be specified in parentheses, such as `with="$letter$=(a,b,c)"`.

There are a number of bare-word `with` variables which can be used when writing variables or other decisions. These keywords should not be surrounded by dollar signs or else the parser will treat them as macro variables and not as debugging commands. Note that all of these parameters are case-sensitive.

- ▪ <u>verbose</u>. The `verbose` parameter will perform a verbose evaluation of the parameter. The value of the parameter both before parser initialization, after parser completion, and after evaluation are printed for diagnostic and debugging purposes. Additionally, should the parser encounter an error, this keyword tells the parser to provide a more verbose error message. An example of the use of this keyword can be seen in Figure 5, where the verbose parameter is provided for the debug variable. When Construct evaluates the debug parameter, it will print out the name of the variable as it is

Figure 5: Common uses for variables

Variable	Value
`<var name="debug" value="false" with="verbose"/>`	`"false"`
`<var name="human_agent_count" value="if(construct::boolvar::debug){30}else{300}"/>`	`"30"`
`<var name="human_agent_begin" value="0"/>`	`"0"`
`<var name="human_agent_end" value= "construct::intvar::human_agent_begin+ construct::intvar::human_agent_count-1"/>`	`"29"`

encountered, the value of the variable as it is encountered, and mention that the bareword 'false' has been converted to a string. Note that the verbose keyword is more targeted than the Construct parameter `verbose_initialization`. While a verbose initialization will provide verbose information for every variable initialized, the `verbose` keyword for a particular variable will ensure than only the particular variable is evaluated in verbose mode. Thus, for the examples in Figure 5, only the debug variable will print verbose output.

- <u>details</u>. The `details` parameter can be used in conjunction with the verbose parameter in order to determine the values of macro substitution parameters. While the `verbose` keyword can be used to debug a simple math expression, it may be necessary to see additional information about the state of the parser as it evaluates macro expressions. The details parameter prints out any information about variables in use, in addition to some very specific information about the internal state of the parser as it examines the input string. Should the parser encounter an error, it should also provide more information about the parameter value.

- <u>preserve_all_white_space</u> (note: includes three underscores). This parameter specifies that white spaces are important to the expression, and that tabs, returns, spaces, and comments should not be removed as the expression is evaluated. By default, all white space is removed during the creation of variables. If included as a keyword, however, any white space in the value will be preserved when the parser is run.

- <u>preserve_spaces_only</u> (note: includes two underscores). This parameter specifies that spaces are important to the expression, and thus should be preserved, but that returns tabs, and comments will be removed. Using this parameter will allow for newlines to be placed in scripts which must preserve spaces.

- <u>delay_interpolation</u> (note: includes one underscore). If this keyword is included in a `with` statement, then the variable will not be evaluated when Construct first parses the variable. Instead, it will be evaluated every time the variable is referred to with an expressionvar (using the `construct::expressionvar::<VarName>` syntax). This is useful in two cases. First, it can be useful for defining macros which depend on variables which are not yet defined. Thus, one could put such an expression at the top of the input deck for editing clarity, and use it as an expression when the other parameters have been defined later on. Second, it is useful in specifying the values of decisions (Section 3) as variables; by specifying `delay_interpolation`, one can ensure that the variable will only be evaluated when the decision is called.

2.5   Using variables

There are a number of common uses for variables, some of which are listed below. While this list is not exhaustive, it may be useful for understanding input decks created in the CASOS lab as well as helpful for experimenters who are beginning to design their own decks.

Common variable uses include:

- Variables as logical flags. One common use for variables is to create logical flags in the input deck. For instance, on can define and set a debug flag at the top of the program and then define a number of variables in terms of logical operations on the debug flag, as seen in the `debug` variable of Figure 5. As can be seen in the figure, the number of human agents (the `human_agent_count`) is greatly decreased when debugging is active. This allows testing and debugging runs to be performed in a reasonable amount of time. Similarly, one might use a debug flag to turn variable debugging on or off or to set certain `construct_parameter` values appropriately for debugging.
- Variables for important quantities. Another common use for variables is to specify the number of agents, knowledge facts, beliefs, or other nodeclass values. For instance, the `human_agent_count` variable in Figure 5 is a way of specifying a subgroup (in this case, the human subgroup) of the agent nodeclass. This provides an easy way to control and modify the number of values in the particular subset: if a user wants to double the number of human agents in a Construct experiment, only a single line of the input deck needs to be modified. If other variables, such as the bounds for particular nodeclasses, are written appropriately then all changes will propagate through the input deck and no other changes will be necessary.
- Variables defining bounds. In addition to specifying nodeclass counts variables are can be used for the specification of nodeclass bounds. When networks are initialized, they are initialized in ranges – a begin and end row, as well as a begin and end column are necessary to correctly initialize the network (Hirshman and Carley 2007). Two examples of bounds, the `human_agent_begin` and `human_agent_end`, can be seen in Figure 5; note that the former starts at the beginning of the network while the latter employs the previously-defined count variable. By using variables to specify the start and end values, it is possible to automatically create the bounds using count values as well as an experimenter-defined ordering of the variable list.
- Variables for key values. Variables can also be used to specify generator values at the top of the program. By placing all constants in variables, experiment designers gain two advantages. First, all constants are located at the same place (the top of the input deck), which allows for easy input deck modification if only the variable values have to be changed. Second, changes to one variable may be propagated to other variables. If a key value used at many locations in the input deck is implemented as a variable, it will only be necessary to change the variable's value once at the top of the program, rather than searching through the file and changing the value at multiple locations. This can make input decks easier to create and maintain.
- Redefinitions of key values for logical clarity. In some cases, the generator parameters used in Construct may not match the conceptual idea desired by the experiment designer. For instance, many of the network generation mechanisms require a network density parameter, but some experiment designers may instead choose to think about

Figure 6: The decision operation

```
<operation name="ReadDecisionOutput">
    <parameters>
        <param name="output_filename" value="[filename]"/>
        <param name="output_format" value="csv"/>
        <param name="run" value="all"/>
        <param name="time" value="[time]"/>
        <param name="verbose" value="<BoolExpr>"/>
        <param name="header_row" value="<BoolExpr>"/>
        <param name="applicable_agents" value="<ListExpr>"/>
        <param name="decision_names" value="<ListExpr>"/>
    <parameters>
</operation>
```

the average number of alters to which an ego is connected. These are both references to the same concept; however, the latter requires a little math in order to yield a density. A variable can help here by allowing the experiment designer to specify a `average_number_of_alters` variable at the top of the Construct input deck for easy modification as well as logical consistency, then perform the necessary math to calculate a density using other variables.

This list is not meant to be an exhaustive list, and Construct users are encouraged to find new uses for this input deck feature.

2.6 Caveats

Variables can be used in all portions of the input deck using the syntax described in Section 2.3. However, there is one place where a seemingly obvious syntax may lead to problems when the Construct parser examines the string.

- Network names. When Construct parses the names of networks during network generation, the white space is automatically preserved using the keyword `preserve_spaces_only`, which was presented in Section 2.4. The names of most networks, by default, include spaces. In order to preserve these spaces, the `preserve_spaces_only` keyword is provided by default for this parameter. When parsing most variables and other values, spaces are removed by default. Thus, if the name of a network must be saved in a Construct variable, it will be necessary to explicitly specify the `preserve_spaces_only` parameter in order to ensure that the space is not removed when the variable is parsed.

Other than these locations, white space is ignored and variable parsing and evaluation follows the rules previously described.

3   Support for Decisions

With Construct version 3.9, Construct is capable of interpreting an arbitrary expression in order to compute output. This method of creating Construct output is called a `decision`, since it is executed on a per-agent basis. Early work with construct have used decisions have involved functions of agent knowledge and/or interactions with others in order to compute a desired output. However, the decision system is not limited to such outputs. Decisions can take into

11

account multiple different types of inputs as well as arbitrary combinations of such inputs. Decisions can do more than passively output values; the network setter command can allow agent decisions to actively modify agent features as well as features of the overall simulation. Decisions are specified using a decision operation, as described in Section 3.1. The syntax for specifying decisions is specified in Section 3.2, while a series of special with variables is described in Section 3.3 which can be useful in determining the result of an agent's decision.

Decisions, like all operations, are evaluated at the end of the simulation time period. This means that the decision is evaluated after agents have chosen their interaction partner(s), selected message(s) to communicate with the partner(s), learned information from other sources, and updated any applicable beliefs. Because decisions are evaluated at the end of the time period, it is possible to record information about what went on in the time period – who interacted with whom, what types of knowledge was learned, and so forth. However, decisions also have the ability to modify the simulation dynamically by affecting the underlying networks that govern the simulation. Because the decision system only runs at the end of the period, though, any changes made to the simulation will not have effects until the next simulated time period when agents act on their modified knowledge, belief, and other important attributes.

While decisions are evaluated during every time period, the simulation results may only be printed during the time periods which the user requests (a point discussed further in Section 3.1). A decision is guaranteed to run at the end of the last simulated time period so that experiment designers will be able to gather final data and perform any necessary calculations.

3.1   The decision operation

The decision is one of multiple outputs that can be obtained from a running Construct simulation. Decisions, like other Construct outputs, can be obtained by creating the specified `operation` within the `operations` tag of the ConstructML. To create an arbitrary decision, the name of the operation should be `ReadDecisionOutput`. The syntax for an arbitrary decision is shown in Figure 6. The name `ReadDecisionOutput` is case sensitive.

The decision operation requires several parameters (`params`) in order to be parsed successfully. If any of these parameters are missing, Construct will output an error and exit. The order of the parameters is not important, though the order specified in Figure 6 is recommended for readability; this order groups the parameters common to most Construct operations at the top and the parameters specific to the decision operation at the bottom. Some of these parameters are required for other operations, while others are specific to the decision output. Each of these are discussed in turn.

- output_filename. The output file name specifies the name of the file. The file name can be any legal name allowed by the operating system on which Construct is running. The file will be created in the same directory where the input file is located; at this time, it is not possible to specify a separate output directory. Note that if the file cannot be created when the simulation is performed, Construct will exit with an error. The most common cause of this error is due to the fact that the user has opened the file but has not closed it before attempting to re-run Construct; to address this problem, close the file and allow it to be overwritten.
- output_format. The output format parameter specifies the file type of the output file. While both DynetML and comma separated valued output are supported for most operations, the output_format parameter for decisions must be set to `csv`. The csv file

Figure 7: Decision names

```
<operation name="ReadDecisionOutput">
    <parameters>
        ...
        <param name="decision_names" value="d1,d2,d3"/>
        <param name="d1"
          value="getKnowledgeNetwork[agent,1]" with="agent"/>
        <param name="d2"
          value="getKnowledgeNetwork[agent,2]" with="agent"/>
        <param name="d3" value="d4,d5"
          type="decision_name_list"/>
        <param name="d$i$"
          value="getKnowledgeNetwork[agent,$i$]"
          with="agent,$i$=(4,5),$i$"/>
    <parameters>
</operation>
```

format can be easily loaded into many different types of databases for multiple types of analysis. Other CASOS tools such as ORA (Carley and Reminga 2004) can import csv files, and these tools can be used to convert the output file into additional formats for network analysis.

- run. The run parameter must be included for legacy compatibility. For all work using Construct version 3.5 and later, this value should be set to "all".

- time. The time parameter specifies the time periods at which the decision will be printed. While the decision is evaluated during every simulated time step, it will only be printed out at the end of the time periods that the user requests. This parameter can be specified in one of three ways. First, the parameter can be specified as a comma-separated list of evaluation times. For instance, the string "1,3,5" will specify that the decision will be printed in the first, third, and fifth time periods (as long as the time periods are within the specified simulation duration). Second, the parameter can be specified as the string literal "first" or "last", which will print the decision during the first or last time periods. Lastly, the value "all" can be used as shorthand to ensure that the parameter is printed every time period. Note that the decision will be updated even in time periods that the user has not specifically requested as time periods to print the decision result.

- verbose. The verbose operation parameter (which is different than the verbose with parameter) prints out information related to the parsing of the decisions as they are loaded prior to the start of the simulation. Setting the verbose operation parameter to true is identical to specifying the verbose with parameter for all decision values, which can be useful for rapidly enabling and disabling of debugging on multiple decision parameters. For additional information on the verbose with parameter, see Section 2.4.

- header_row. The header row is a Boolean parameter which specifies whether or not a header row should be printed at the top of the file. If set to true, the name of the decision is printed as the first row in the csv file, otherwise no header row is printed.

A header row can be useful for debugging or for creating a set of headers to use when importing the decision results into a database.

- <u>applicable_agents</u>. The applicable agents parameter specifies the names of the agents to which the decision will be applied. This allows the decision to be performed to a subset of the total agent population – for instance, only to apply the intervention to human agents as opposed to the intervention agents present in the simulation. The applicable agents parameter requires a list of indices, which can be supplied by using the agent group reference syntax (`construct::agentgroup::<name>`) described in Appendix B, part XII.

- <u>decision_names</u>. The decision names variables specifies a comma-delimited list of decision names. These decision names must also be specified as decision parameters, as can be seen in Figure 7 where the decision names `d1` and `d2` are specified as the first two decisions. When the decisions are output to the csv file, `d1` will be the result in the first column, as it is the first decision printed. The decision `d2` will be second, and so forth. Note that, for forward compatibility, the `decision_names` param must have an attribute `type` and set its value to "`decision_name_list`".

3.2 Specifying decisions

Decisions are specified as additional `params` under the `parameters` tag of the `ReadDecisionOperation`. While there is no hard limit on the number of decisions which may be specified, practical and space limitations suggest that no more than two hundred decisions be created per Construct simulation. It is highly likely that most virtual experiments using Construct will use only one or two key outputs, so this restriction should not be limitation in practice.

To be evaluated, the decision must be specified in the `decision_list` param or reachable from a chain of decisions. If the decision is not reachable in this manner, it will not be parsed and thus will not be checked for errors. To determine whether or not the decision is being parsed, set the `header_row` parameter to `true` and run the simulation; if the name of the specific decision appears in the header row, then it is being evaluated.

When specifying decisions, it is possible to use all of the scripting features used when specifying variables. This means that the use of constants, addition and subtraction, multiplication and division, logical operations, string operations, conditional statements, all the operations described in Section 2.1 are usable in decisions. Five additional scripting features also become available: network getters, network setters, agent references, time period references, other decision references, and previous result references:

- <u>network getters</u>. One of the most important aspects of the decision system is that it can reach into any network and grab a value or set of values. This allows the user to establish an arbitrarily complex set of relationships between values in multiple networks. While users may find it most practical to use decisions to relate agent-based features such as agent knowledge and agent beliefs, it is possible to access more general networks such as the knowledge-by-knowledge group `knowledge group membership network` if so desired. The syntax for this operation looks like `getXXX[param1,param2]`, where XXX is the name of a network. The syntax is described more fully in Appendix B, part XI. The value returned by the network getter

14

is the same as the type of the network. Thus, a value retrieved from the knowledge network would be a float since the knowledge network is represented internally as a float. A list of the network names in Construct, as well as their corresponding types, can be found in other technical literature (Hirshman and Carley 2007; Hirshman and Carley 2007).

- network setters. The decision system can also set values in an arbitrary Construct network. This process allows an agent's decision to modify itself or modify another agent, and to do so dynamically as the simulation is running. For instance, an agent who learns about the existence of a web site (a knowledge fact) while interacting with one agent may then be able to consider that web site as an interaction partner. Prior to that, the agent would not have known that the web site exists and thus would have not been able to interact with the web site agent. Thus, knowledge can be made a prerequisite for interaction. When the knowledge is learned, a setter can be used to modify the interaction pattern of one or both agents. While Construct users may not find it necessary or practical to use network setters in many decisions, it remains an option that can be used to make simulations much more dynamic. The syntax for the operation looks like `setYYY[param1,param2,value]`, where YYY is the name of a network, and is described more fully in Appendix B, part XI. The value returned by the network setter is the same as the type of the network. Thus, a value set in the knowledge network must be a float since the knowledge network is represented internally as a network of floats. The return value of the setter will also be the same type as the network, and thus in the knowledge example would be a float as well. Note that any the evaluation of any decision operation, and thus any set operation, occurs at the end of a Construct interaction cycle – after agents have already exchanged information and learned any new information – and thus the set operation will only affect the network in the following simulation timeperiod.

- agent references. Decisions are evaluated iteratively by running through a list of applicable agents (Section 3.1). In the evaluation of these decisions, it is often useful to refer directly to the agent for which the decision is being evaluated. For instance, it may be useful to examine that agent's knowledge, that agent's beliefs, and other properties unique to that agent. This can be done using the keyword `agent`. Each decision is evaluated once per applicable agent per applicable time period, meaning that the agent reference is guaranteed to be set to every applicable agent during evaluation. Note that this functionality is only possible for decisions, unlike the more general behavior of network getters and setters.

- time period references. Decisions are evaluated once per applicable agent per applicable time period, and it is often useful to know which time period is currently being evaluated. The timeperiod keyword allows access to this current time period. Note that the time period value will only take on the values specified for that decision; if the time period is not specified as an evaluation period, then the time period reference will never be updated to that value.

- decision references. Decisions can also reference other decisions. This allows decisions to be built up of multiple smaller sub-decisions, the values of which can be recorded separately for subsequent analysis. Thus, if one wanted to analyze Boolean decision AAA, Boolean decision BBB as well the joint case when either value was true (AAA || BBB), one could define two independent decisions and then define the third

15

Figure 8: Using `with` variables in decisions

Decision
| ```
<operation name="ReadDecisionOutput">
    <parameters>
        <param name="decision_names" value="everTalkedTo0"
   type="decision_name_list"/>

        <param name="everTalkedTo0" value="
     getInteractionNetwork[agent,0] ||
       getInteractionNetwork[0,agent]  ||
       previousResult:bool
   }"
   with="agent"/>
     </parameters>
</operation>
``` |

as the logical or of the two earlier values. This can be done by specifying the name of the referenced decision in the text of the decision which is referencing it: thus, the syntax `AAA:bool || BBB:bool` would return the desired result (the casts to bool are necessary because, by default, the decision reference values are returned as generic strings). The advantage of using a decision reference as opposed to rewriting the decision is threefold. First, the decision need only be written and debugged once. This can save the hassle of keeping two separate decisions synchronized as well as correctly functioning. Secondly, the value of the referenced decision is only calculated once. If the referenced decision is expensive to calculate, which is often true if it involves multiple network getters, the running time of the operation is decreased. Lastly, any calls to random numbers are preserved between the referenced decision and the decision doing the reference. In the example above, if decision `AAA` required a random number during its execution, then the decision reference to `AAA` would use the result (and compute a value consistent with the referenced decision) while a rewrite of decision `AAA`'s code would cause a different random number to be used and may create an inconsistent decision.

- previous result references. Decisions can also refer to the previous result of the decision using the keyword `previousResult`. This will retrieve the previous value of the decision when it was last evaluated in the previous time period; in the first period, this result is initialized to the empty string. The previous result will always be retrieved as a string, but can be cast to any relevant data type. Note that this is deliberately different from the syntax that is usually used to refer to other decisions: while references to other decisions are done by referencing the name of the other decision, referencing the same decision must be done by specifically requesting the previous decision. This is to ensure that the user deliberately wished to use the previous result in the current decision. An example of the use of the `previousResult` keyword can be seen in Figure 8, where the previous result is used to determine whether or not the agent ever talked to agent 0. Because the interaction network is cleared after every time period, it is not possible to determine whether or not two agents ever communicated. By referring back to the previous

16

result of the decision, however, the simulation designer can use this decision as a way to keep track of whether an agent ever communicated with agent 0.

3.3 `With` variables for decisions

When specifying a decision, it is possible to include a large number of `with` variables. While some of these variables may be necessary given the type of decision being specified, others provide additional flexibility to the simulation designer.

First, all of the `with` variables available for `construct_vars` (Section 2.4) can be used when specifying decisions. This means, for instance, that it is possible to specify that one particular decision be run in verbose mode for additional debugging. In fact, it is strongly recommended that all users take advantage of the "`verbose`" `with` variables when writing scripts, as the debugging information provided can prove to be invaluable for understanding problems. The use of other `with` variables such as "`no_filtering`" may have situational use and are available to the user.

As introduced in Section 3.2, the decision system can create and modify a number of internal variables as it operates. If these variables are to be used in a decision, the experimenter must refer to these variables using `with` variables. This specification requirement is put in place primarily to ensure that the user did intend to use the variable and that the variable was not created due to a typing error. Construct will ensure that the variable value is updated as the simulation state changes.

- `agent`. The agent with parameter specifies that the decision will reference agents in the `iterable_agents` list. This creates an agent-specific counter to the agent being evaluated. An example of this with variable's use can be seen in Figure 8. The `agent` keyword allows this decision to evaluate differently for each of the applicable agents; for instance, for agent 1 in the simulation, the value would evaluate to true if the agent ever contacts agent 0 in the current period (the first line), was contacted by agent 0 during the current period (the second line), or ever had contact the agent during the immediately previous period (the third line, which by recursion will lead to a true value if any communication ever occurred). Since the value of agent will be different for each applicable agent, this result will have different values for agent 0, agent 1, and so forth.
- `timeperiod`. The time period keyword specifies that the decision references the current time period at the time of evaluation. This time period will be an integer index from zero (the Construct initialization time period) to the number representing the last time period.
- It is also possible to use `with` variables in order to set up macros for decisions. As with Construct variables, Construct decisions can be set up to require macro parameters. If only a single macro parameter is specified, then only one decision is created; if a list of parameters are specified, then one decision per parameter value is created. All macro declarations must begin with a macro variable name surrounded by dollar signs (`$`), followed by an equals sign, followed by the names of the possible values. An example of this variable declaration can be seen in Figure 7. In this case, the decisions named `d4` and `d5` are created using a `with` variable `$i$` in the latter manner; the second with parameter specifies that the value of `$i$` should be first 4,

17

then 5. It is important to remember, however, that both versions of the decisions (i.e. both `d4` and `d5` in this case) must be referenced in a way reachable from the `decision_names` parameter in order to be evaluated.

The user also has the ability to declare any non-quoted string as a `with` variable, if it is not recognized as a special keyword or the name of a previous decision. In variables, barewords are automatically recognized as seen in Figure 2 (page 4). In decisions, however, it is necessary to specify barewords as a string so that Construct will not exit with an error. There are two uses for this feature. First, future versions of Construct may introduce the ability to treat these bare words as non-strings, in the way that `agent` and `timeperiod` have special meanings in a decision. However, it may also be used to let Construct know that certain variables will be used as barewords within the script. In Figure 7, for instance, the variable `$i$` is used in this way in variable `d$i$`. Since the variable is used as an index into the knowledge network, the `$i$` must be specified as a bare word `with` variable in order for it to parse correctly. For the most part, however, it is always recommended that the user place any constant within quotes so that the decision parser will recognize the word as a string literal, this mechanism will facilitate forward compatibility with changes to the scripting language by allowing future users to specify new behavior for the decision.

4   Scripting

The Construct variable system (Section 2) and decision system (Section 3) are subsets of the scripting system available in Construct. The scripting system is able to perform a number of additional functions, including the evaluation of math or string expressions, the manipulation of Construct networks, and various logical and control operations in order to generate a specific result. This portion of the technical report provides an overview of the scripting language as well as a brief set of tips that may be useful when writing scripts in Construct. Section 4.1 discusses the types of lexemes that are possible in the scripting system, many of which have been partially introduced in previous sections. Section □ provides information about commenting in the Construct scripting language. Section 4.3 describes various features in the scripting language which can facilitate the debugging of scripts. Section 4.4 provides a discussion of several caveats that may occur with the scripting system and the associated `with` variables.

4.1   Types of scripting commands

A wide variety of script operations are supported in Construct. While this section introduces the classes and types of operations supported, it only provides a brief overview of the functionality. For a complete list of the lexemes supported, see Appendix A and Appendix B. The appendices also contain information on the suggested use of these lexemes. Additionally, they contain illustrative sample code for some of the more complicated lexemes.

The general types of operations supported include mathematical operations, string operations, logical operations, comparison operations, random number operations, cast operations, if statements and control operations, network getters and setters, variable references, and a few miscellaneous commands. Each of these is described in turn. Additional details can be found in the appendices.

- Comments and constants. C-style comments (`/* … */`) are used to provide a way to document a script. These are described further in Section 4.2 and the appendices. String

constants can be specified using single quotes (`'`) to mark the beginning and end of the string, while numbers need not be specified using special characters.

- Mathematical operations. Standard mathematical operations addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and exponentiation (`**`) are supported. The input to these operations must be bools, ints, or floats; the output will be a bool if both inputs are bools, an int if one or both inputs are ints but none are floats, and a float if at least one input is a float value. It is important to note that the scripting language does not obey the standard precedence rules at this time, and instead will evaluate all mathematical expressions from right to left. This means that the expression `3*2+1` will evaluate to `9`. However, since parentheses can be used to define the order of evaluation multiple consecutive math operations be explicitly parenthesized for clarity.

- String operations. Several generic string operations are supported. String concatenation operations (`+`) allow for the concatenation of two strings, as well as the concatenation of a string with the string representation of a bool (represented as a 0 or 1), float, or int. The enumeration operation (`..`) creates a list of integers from a lower bound to an upper bound and stores the resulting list in a string. Lastly, a sequence of characters surrounded by single quotes (`'` and `'`) can be used to define a string literal.

- Logical operations. Construct supports the logical and (`&&`), logical or (`||`), logical not(`!`), and logical exclusive or (`^`). The input to these operations must be numeric, the result returned will be either a zero for false or one for true. Logical operations are short-circuiting, meaning that the left-hand side will be evaluated first, and if the left-hand side will yield an unambiguous result the result is returned without evaluating the right-hand side.

- Comparison operations. The scripting language supports the equals (`==`), not equals (`!=`), greater than(`>`), less than(`<`), greater than or equal to (`>=`), and less than or equal to (`<=`) operation. All six operations can be performed on numeric values. Equality and inequality operations can also be performed on strings. The result of these operations will be either a zero for false or one for true, and will follow the same casting rules as the mathematical operations.

- Random number operations. The scripting language allows the user to directly generate numbers from a uniform (`randomUniform`) or a normal (`randomNormal`) distribution.

- Cast operations. The cast operation (`:`, a colon) allows the casting of values from one type to another. While some operations, discussed in Appendix B, will perform implicit conversions, downcasts from a less restrictive to a more restrictive type (or conversions of strings to any other type) will require explicit casts.

- If statements. Construct supports multiple types of control statements. The if statement (`if`) performs a logical test when the script is evaluated and executes the relevant branch of code based on the result of that test. The static if statement (`static_if`) performs the logical test when the script is first read by the Construct parser – which can be substantially earlier then when the script is evaluated – and only parses one particular branch of the if statement, which can lead to substantially faster but less flexible code.

- Assignment statemets. The assignment statement (`=`) allows for the assignments of values to variables. Every variable has a type that will be used in all subsequent expressions

19

using that variable. The first time a variable is used, it is given a type; all subsequent uses and assignments to the variable follow that type.
- Control operations. The foreach statement (`foreach`) sets up a loop and uses an accumulator in order to perform a requested operation multiple times. Note that the foreach loop can be used in conjunction with a series of assignment statements to create an accumulator. The return statement (`return`) returns a value or expression from somewhere within a script. This allows a variable to be returned, either at the end of a script or at a short-circuit point reached from an `if` statement.
- Network operations. The Construct scripting system can read values from arbitrary networks as well as set new values in those networks. The getter (`getXXX`) can obtain a single value from a network or the aggregated result from a part of a network. The setter (`setYYY`) can set the value of an arbitrary network. The getters and setters only work as long as the networks have been initialized, so they cannot be used when variables are being initialized; however, they can be used in decisions and scripting as discussed in Section 3.2.
- Variable reference. The variable reference (`construct::XXXvar::YYY`) was previously introduced in Section 2.3, and can be used in the scripting language as long as the variable has previously been defined. If the variable has not been defined, the script will exit with an error. Alternatively, variables can also be referenced as expressions to be evaluated, or as lists of nodes which are members of a particular group.
- Other commands. Additional specialized operations, such as the error operation (`error`) and macro operation (`$XXX$`) provide additional means of specifying the behavior of the script. Additionally, unknown lexemes are passed to a handler, which allows a user to extend the scripting language to handle other unknown lexemes by modifying the models used in the underlying Construct executable. These features are further discussed in Appendix B, part XIII.

4.2 Comments

Because the syntax of the Construct scripting language can be complex, it may be helpful to use well-placed comments in order to clarify and document the purpose of the script. While comments can be supplied either prior to the script using ConstructML comment tags (which are identical to standard XML comment tags `<!--` and `-->`), it is also possible to insert C-style comments directly into the input file. Within the Construct XML script, one can insert a comment using the character sequence `/*` to mark the beginning of a comment and `*/` to mark the end of it. Any characters contained in the comment, including newline characters, will be ignored by the parser.

4.3 Debugging

If there is one thing that is nearly certain regarding all things computer-related, it is that one's first implementation will never be completely successful. Instead, one must often follow a lengthy debugging procedure in order to understand the cause of an error and then to fix the erroneous input. The Construct scripting language contains several features which can assist with code debugging. These features include:

- The `verbose` keyword. The verbose keyword can be used in order to print out the value of a single ConstructML tag before and after execution. When the verbose keyword is present as a `with` variable, the input to the script as well as the resulting value after script execution are written to Construct's standard output. This can be helpful in two ways.
    - It can be used to verify what is actually being executed by the Construct lexer and script execution engine. This can help diagnose whether or not there are problems with variable interpolation which can occur if, for instance, a value depends on a second value set in another variable or alternatively if there is a problem with macro interpolation (which can be common with very complex expressions). Thus, it can illustrate a need to go back and modify the expression to ensure that the right variable is in use at the right occasion.
    - The `verbose` keyword can be used to help determine whether the expression is being evaluated in the manner expected. For instance, certain operations share common lexemes but may perform different functions. While the user may intend to use the / operation as a division operation, omitting a cast may lead the operation to be used as a subsequence operator (Appendix B, part III) and lead to a very different result. Thus, the verbose keyword would help the user see that division was not being performed and provide some guidance as to what process might be going on. While employing the verbose keyword itself will not address the underlying problem, it should be able to give the user additional resources which may be helpful in diagnosing and fixing it.
- Comments using /* and */. The comment characters can be used to temporarily remove certain operations from the script. For instance, one can use them to remove a complex mathematical statement and replace it with a temporary constant. Such an approach can be useful in conjunction with the verbose keyword as described earlier.
- The `error()` operation. The `error()` operation can be used to stop execution of a script in case an invalid state is reached. For instance, one can guarantee that a variable referred to in a script is greater than zero by invoking an error statement if the number is less than zero. When an error statement is executed, Construct will exit with a user-specified error message. While the error operation may not immediately pinpoint the source of the error, it can help determine where an error is occurring and what can be done to fix it. For additional information on the `error` operation, see Appendix B, part XIII.

In addition to the above debugging commands, it is always advisable to use good programming style to minimize the number and severity of errors. For instance, commenting can help improve script maintainability, especially if multiple users are involved in the creation of an input deck. Breaking a complex operation into two or more combined operations, perhaps with the help of multiple variables or sub-expressions, can also be useful at times

4.4 Caveats with macros and `with` variables

As described in Section 2.4 and again in Section 3.3, `with` variables can be used to specify macro parameters as well as supplemental parameters that can be helpful in setting up scripts. It

is useful to again discuss these issues in the broader context of scripting in Construct, as well as to offer a number of words of advice as to their use.

First, `with` parameters are not mandatory when using the Construct scripting language, but are needed when using macros or more complex features of the scripting language. For instance, most Construct variables can and will be written without any `with` parameters. If no `with` variables are needed, the attribute should be omitted from the ConstructML to improve clarity. This occurs in the declaration of most Construct variables. For instance, the variables in Figure 1 (page 3) and Figure 5 (page 9) are variable declarations which are used when initializing variables. These variables do not contain macros, and the math performed is relatively simple. While a `with` attribute for verbose initialization could have been provided, such an attribute is unnecessary. Thus, the `with` attribute is omitted in the ConstructML, and the code appears more readable.

Second, `with` variables that work in one context may not always be applicable in another. For instance, the `previousResult` lexeme, which can be used to refer to the result of a decision in the immediately prior time period, only makes sense in the context of a decision operation. Referring to the `previousResult` lexeme when initializing a Construct variable does not make sense. In fact, in that context, the `previousResult` lexeme is not recognized and will be treated as any other erroneous lexeme: it will cause Construct to output an error and exit. Thus, while the functionality of the scripting language is general, there are a number of macro and `with` variables that are specific to certain parts of the input file.

Third, while macros are an extremely powerful way to shorten a Construct input file, they often should first be written without macros and then generalized. Since macros can be very difficult to write, it is often best to focus on getting the correct behavior for one particular instance before moving on to generalize to multiple similar instances. For instance, the last decision declaration in Figure 7 (page 13) is one that uses a macro variable to initialize two similar decisions at the same time. However, this decision is similar to the ones that came before it; all of the declared decisions have the same structure. By first debugging the variables which do not use a macro parameter, one can first determine whether bugs that occur are due to the logic of the statements before tackling the question of whether the macro system is introducing further errors. While it may not be possible to write some variables or decisions without starting with the macro system – a few of the examples provided in Section 5 rely intrinsically on the macro system in order to function – one can generally speed the debugging process by writing single specific variables or decisions and then generalizing rather than focusing on writing everything all at once.

Fourth, it is important to note that macros are expanded immediately prior to use. This means that if a macro variable or decision has not yet been examined in the input file then it cannot be referenced by a prior variable or decision. Thus, if variable `varA$i$` and `varB$i$` exist in an input file where the former occurs before the latter, one cannot refer to any macro version of `varB$i$`, for instance `varB0`, when initializing `varA$i$`. This is because the variables are only expanded when they are initialized. For instance, if `$i$=0,1,2`, variable `varA0` to `varA2` will be generated before variables `varB0` to `varB2` and the variable varB0 will not exist until `varA2` has been fully initialized. Thus, the value of variable `varB$i$` can refer to one of the values of `varA`, but the values of variable `varA$i$` cannot refer to those of `varB` since the variables will not be generated. The same concept applies to decisions and other uses of the scripting language. This may impact how variable and decision code is written, because it

Figure 9: Creating a sequence of variables

Variable
| ```
<var name="intervention_agent_begin" value="0"/>
<var name="intervention$i$_agent_index"
  value="static_if($i$:int>0) {
      construct::intvar::intervention$i:int-1$_agent_index+1
    } else {
      construct::intvar::intervention_agent_begin
    }"
  with="$i$=(0..construct::stringvar::intervention_count-1)"/>
<var name="intervention_agent_end"
  value="construct::intvar::intervention$i$_agent_index"
  with="$i$=(construct::intvar::intervention_agent_count-1)"/>
``` |

might often be useful to refer to a variable before it has been expanded using a macro. Since this is not possible, it is often useful to create a series of intermediate variables which all derived variables can refer to; in this example, it may be useful to create the bounds for `varB` (i.e. a `varB_min` and `varB_max`) prior to initializing `varA$i$`.

5   Applications and Examples

The power of the Construct scripting system can be best illustrated by a number of examples, which are provided here. While a number of illustrative examples have been included earlier in this document (for example, Figure 2 on page 4 describes a number of example macros and how they can be used together), it is useful to use some extended examples to discuss how certain aspects of the scripting language can be used. The examples in this section are meant to be complete examples that can be plugged into user code.

This section describes five examples of the Construct scripting language in action: the creation of multiple variables to describe a group of identical agents (Section 5.1), the use of with variables when generating a network (Section 5.2), the use of the scripting language for network generation using the `lexer_based` generator (Section 5.3), the use of the decision system to count the number of times an agent interacts with particular alters in order to create a frequency-of-interaction list (Section 5.4), and the use of the decision system to modify agent behavior based on newly-learned knowledge (Section 5.5). All of these examples have been taken from actual Construct input decks, though some have been simplified for use as examples.

5.1   Creating bounds variables for a sequence of identical agents

Sometimes, when generating an input deck, it is useful to set up a sequence of variables, the exact number of which should not be hardcoded. For instance, in some virtual experiments, there should be two sequence variables values and in other experiments there should be three. However, the names of these variables must be (seemingly) hardcoded – for instance, there should be a variable1, then a variable2 and then in some cases a variable3. While this can be done using two distinct input decks, such a sequence can be tricky to generate and debug, since there may be other variables which depend on values at the beginning or end of the sequence.

An alternative to such hard-coding can be seen in Figure 9. In this example, three `var` tags can be used to define a sequence of `intervention_count` (abbreviated N) variables from

`intervention0` to `interventionN`. A begin and an end tag can be used to mark the entire block of variables, in case the values have to be set en masse in a generator; however, individual variables are also created, one for each variable index.

The creation of the `intervention_agent_begin` variable is simple, but is worthy of note. The variable from which this tag is derived, the Construct variable `human_agent_end`, indicates that these variables are not stand-alone. If a user wishes to use this sequence of variables as a stand-alone set, this particular value can be set to zero. However, this value is shown as a non-zero value to illustrate the fact that the intervention_agents can follow some other sequence of previously defined variables (for instance, those relating to human agents) and thus be part of a larger picture. Note that it will be necessary to define the values prior to the `human_agent_end var` tag used in the `intervention_agent_begin` definition.
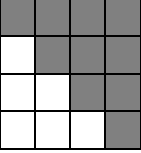
The `intervention$i$_index` variable is a complex statement, though a powerful one. The first thing to note is the `with` statement at the end of the `var` tag. The macro variable `$i$` is defined to be a sequence of variables from `0` to `intervention_count-1`, starting at `0`. When the value is zero, the `static_if` expression is evaluated and returns false, meaning that the value of `intervention0_agent_index` is set to be `intervention_agent_begin`. Next, the value is set to 1, meaning that `intervention1_agent_index` is set to be `intervention0_agent_index+1` by the macro expansion of the proper part of the `static_if` statement. Since the value of `intervention0` was defined before the value of `intervention1`, the if statement will recognize the variable and define `intervention1_agent_index` in the manner expected. With `intervention1_agent_index` defined, `intervention2_agent index` can be created, and so forth until all of the variables are defined in an increasing sequence.

Note that it is necessary to use a `static_if` (as opposed to a standard `if`) when defining the `intervention$i$_agent_index` variable. The first value of `$i$` is `0`, which would mean that the value of `intervention$i:int-1$_agent_index` would evaluate to `intervention$-1$_agent_index` since `0-1=-1`. The minus sign would then be considered a separate lexeme, meaning that the ConstructML parse would attempt to parse the expression `construct::intvar::intervention-1_agent_index` using the subtraction operation and an undefined variable `1_agent_index`. This will lead Construct to exit with an error. To prevent this from happening, it is necessary to convince Construct not to parse the true branch if the logical expression is actually false. A `static_if` is evaluated when the expression is first loaded, meaning that the false branch of a `static_if` is ignored by the scripting language and will not lead to an error.

The `intervention_agent_end` variable is noteworthy because it is defined in terms of the last intervention index. If there are four intervention indices, then the value of the `intervention_agent_end` tag will be identical to that of the intervention3_index variable. This allows the intervention end to be set dynamically. The `with` tag of this variable, however, is subtly different than that of the `intervention$i$_agent_index` variable. The with tag uses only a constant value, while the `intervention$i$_agent_index` uses an enumeration (`..`) to ensure that multiple values are set and multiple variables are created. Here, however, only one variable is desired and so a single variable should be initialized.

Note that the variable naming strategy defined here only makes sense if the variables will be used later in the ConstructML input file. If the number of variables will change in different

Figure 10: Using `with` variables in network generators

Network	Value

```<network name="interaction sphere network"```
``` src_nodeclass_type="agent"```
``` target_nodeclass_type="agent"```
``` link_type="float" network_type="dense">```
```   <generator type="constant"```
```   with="i=0..construct::intvar::agent_count-1">```
```   <rows first="i"```
```   last="construct::intvar::agent_count-1"/>```
```   <cols first="i" last="i"/>```
```   <param name="constant_value" value="1"/>```
```   </generator>```
```</network>```
	hierar-
	chical
	network

experimental conditions, then it will be necessary to have the number of network generators change as well. Thus, if there are four interventions, there should be four sets of network generators, one for each of the `begin` and `end` variables created above. Otherwise, the utility of using this type of syntax will be minimal (though it may help to save some space in the input file). This problem can be addressed by using a `with` variable to the network generator, as described in Section 5.2. By combining both of these strategies, it can be possible to expand and contract the sequence as needed, as well as have the Construct networks use this sequence effectively.

5.2 Using `with` variables when generating networks

It is possible to use `with` variables to initialize multiple network generators that share a similar structure. This allows multiple generators (even a modifiable number of generators, such as described in Section 5.1) to be defined simultaneously. The `with` variable ensures that multiple independent generators are created, then evaluated sequentially, in order to create a more condensed structure than would occur if each generator were defined separately. Although the end result is identical to that achieved using a sequence of generator definitions each slightly different from the last, the clarity and conciseness provided by the use of `with` variables is quite pronounced.

For example, consider the hierarchical network generator of Figure 10. The generator being defined creates an interaction sphere representing a hierarchical network (where agents are only contacted by "superiors" and may only contact "subordinates") using a `with` variable to condense multiple generators into a couple of lines in the input file. The `interaction sphere network` defines which agents can communicate with which others; ego agents (the `src_nodeclass`) can only contact the alter agents (the `target_nodeclass`) if the network value is 1.0 (Hirshman and Carley 2007). The generator in Figure 10 is set up to be a very strict hierarchical network with the first agent (agent0) able to contact everyone but only contactable by itself; the second agent (agent1) able to contact all but the first agent but only contactable by agent0 and itself; and so forth until the last agent (agentN) can be contacted by any of its superiors but only can choose to interact with itself. A pictorial diagram of the values set can be seen on the right, where the shaded values are set to 1.0 (meaning an contact is

possible) and the remaining values unchanged. While such strict hierarchies are unlikely (as research has indicated that strict organizational charts on paper are often supplemented or circumvented by personal and friendship links), the generator serves as a reasonable demonstration of the way in which `with` variables can be used in network generators.

As can be seen in Figure 10, the `network` tag is unchanged even when `with` variables are in use. The network tag contains the network `name`, `src_nodeclass_type`, `target_nodeclass_type`, `link_type`, and `network_type` as appear in any Construct 3.9 network. No modifications to the network tag are necessary.

To use a `with` expression with a generator in the network, the generator tag is written to include a `type` attribute and a `with` expression. The remainder of the generator can then be written to take advantage of the `with` attributes. For instance, any of the parameters in the `rows` tag, the `cols` tag, or the `param` tags can take advantage of macro variables and macro expressions. As can be seen in the hierarchical network generator in Figure 10, the rows and columns are defined in terms of variables. The row parameter is set to include all values between `$i$` and `agent_count-1`, which will include a different number of rows as the value of `$i$` changes when the `with` variable is evaluated. The column parameter will set only one column at a time, since both the first and last value are `$i$` (meaning that the `$i$`th agent will be initialized in each instance of the generator). Note also that it is possible to include generator values that do not reference any `with` variables. As can be seen in the example, the `constant_value param` is set to be a constant 1.0. This syntax ensures that the same value will be set for all instances of the generator, which would be the desired behavior here.

When the network is generated, the macro is expanded. This means than `agent_count` generators are created, starting with `$i$=0` and continuing until `$i$=agent_count-1`. Thus, the first network generator will generate column 0 (rows 0 to `agent_count-1`), the second network generator will generate column 1 (rows 1 to `agent_count-1`), and so forth until all of the networks have been generated following macro expansion. Note that it would have been possible to have written this out using `agent_count` number of generators and hard-coding each row and column. The primary advantages of employing the `with` variable system is to make this generator tractable, easily readable, and conceptually more coherent.

There are a couple of notes for using the macro system to generate networks. First, the `with` statement is associated with the generator and not alongside the `rows` tag where the macro variable is used. Because the whole generator is to be repeated multiple times for different values of macro variable `$i$`, the `with` variable is attached to the generator tag. If instead the with tag were included once for the `rows` tag and again for the `cols` tag, Construct will ignore all but the last `rows` and `cols` variables created, and the generator would only set the value for `$i$=agent_count-1`. Second, it is possible to use multiple macro variables when generating networks in this fashion. The macros should be comma-separated as described in Section 2.4. Note that using multiple with variables will greatly slow down the network setup time, as it is necessary to create a separate generator for each combination of variables (a relatively slow process). Lastly, it is possible to use the macro system to modify the type of network being generated, for instance by defining a macro variable `$type$` and setting the generator type attribute by saying `type=$type$`. However, since each generator has its own distinctive list of parameters, it will be necessary to define all of the necessary parameters for the macro expansion – even though some of these variables will not be used when the macro is

Figure 11: The `lexer_based` network generator

<div style="text-align:center">Network</div>

```
<network name="agent belief network"
  src_nodeclass_type="agent" target_nodeclass_type="belief"
  link_type="float" network_type="dense">
    <generator type="lexer_based">
        <rows first="0"
            last="construct::intvar::agent_count-1"/>
        <cols first="0"
            last="construct::intvar::belief_count-1"/>
        <param name="lexer_string" with="delay_interpolation"
            value="if(getHoldsPositiveBeliefNetwork[row,col]){
                randomNormal(0,0.2,-1,0)
              } else {
                randomNormal(0,0.2,0,1)
              }"/>
    </generator>
</network>
```

expanded. Thus, to create a macro generator that conditionally expands to either a constant value generator or a tied generator, one would have to define a `constant_value` parameter (for the constant generator) and the `tied_row` and `tied_col` parameter (for the tied generator). Then, depending on the value of the macro variable `$type$`, the relevant parameters would be used to generate the network.

5.3   Using the lexer for network generation

While Section 5.2 described how `with` variables could be used to generate networks, in truth the entire scripting language can be used for network generation. This type of generator is called the `lexer_based` generator. Unlike the standard network generators described in earlier technical reports (i.e., Hirshman and Carley 2007), the `lexer_based` generator will generate a network based upon an arbitrary script as defined using the Construct scripting language. This allows the user to create an arbitrary network using all the power that scripting can provide. Specifically, this network generator allows the user to define a network in terms of other (previously defined networks) using the `getXXX` scripting language command.

The `lexer_based` generator introduces two new `with` lexemes, `row` and `col`, which specify the row position and column position currently being manipulated. The `row` and `col` lexemes are integer variables, and as such do not have to be explicitly cast in order to use in a `getXXX` statement. The `row` and `col` variables are automatically set by the generator and cannot be exogenously manipulated by the user; however, their values can be read in order to locate a particular value in a network. Note that the `row` and `col` lexemes are only defined for the `lexer_based` generator and do not have meaning for variable declarations (Section 2) or the decision system (Section 3).

An example using the `lexer_based` generator can be seen in Figure 11, in which a `lexer_based` generator is used to set up the `agent belief network` – the who-initially-

believes-what network as discussed in other technical reports (Hirshman and Carley 2007). This `lexer_based` generator sets the value of the agent's belief to a random value drawn from a particular side of the normal distribution based on whether a particular agent should hold a positive belief. For instance, if the value of the belief is supposed to be positive (as defined in the not-shown "`holds positive belief network`"), then the belief should be initialized from the positive side of a normal distribution with mean `0.0` and variance 0.2. On the other hand, if the value of the belief is supposed to be negative, then the belief is set up from the negative side of the same normal distribution. This generator ensures that agents are initialized to a particular belief, but that most agents' beliefs are relatively weak.

In Figure 11, all rows and all columns in the `agent belief network` are set by the generator. The `lexer_string` is set up such that, upon evaluation, it reads the value in the `holds positive belief network`, determines whether the value is `true` or `false`, then chooses the value from the proper side of the distribution. The logic for doing this is set up using the scripting language as described earlier.

Note that the functionality of this generator could not have been created using other generators available in earlier versions of Construct. Though the normal distribution generator itself was not available, there were additional reasons why such a network generation strategy was impossible. First, it was previously not possible to conditionally set up a network generator. This meant that a used could not set up a network like the `holds positive belief network` and use values in that network to generate other values. Instead, Construct users were limited to initializing individual networks in isolation, and any dependencies between values would have to have been set up using external CSV files. With the lexer_based generators, expressions of arbitrary complexity can be used to initialize network values, a change which has the potential to provide even greater power to the experiment designer.

There are two caveats with using the lexer_based generator. First, note that the `lexer_string` is set up with the `delay_interpolation with` variable in Figure 11. This syntax should be used with all `lexer_based` generators. The lexer string only makes sense when evaluated by the lexer based generator, since the `row` and `col` lexemes only have meaning within the `lexer_based` generator. However, in the absence of the `delay_interpolation` parameter, the standard Construct lexer examines the values and will fail with an error. In order to avoid this error, it is necessary to include the `with` variable expression. Secondly, the evaluation of a lexer string takes slightly more time to run than the evaluation of a built-in generator. This means that users should attempt to use the built in generators described in other Construct literature (Hirshman and Carley 2007) where possible and to use the lexer only where needed. However, since the initialization time is usually a minor fraction of total simulation running time for large experiments, this overhead is minimal.

5.4 *Employing the decision mechanism to count agent interactions*

The decision system described in Section 3 can be used to observe and record information about the behavior of agents – complex functions of knowledge and beliefs that lead to a certain type of behavior (e.g. Carley, Martin et al. 2009; Hirshman and St. Charles 2009). However, decisions can be used to aggregate simpler information about particular agents in order to facilitate later analysis. Thus, decisions systems can capture how likely an agent is to interact with a particular alter, the number of agents that have at least a certain amount of similar

28

Figure 12: Using the decision system to count interactions

<table>
<tr><td align="center">Decision</td></tr>
</table>

```
<operation name="ReadDecisionOutput">
    <parameters>
        <param name="output_filename" value="decision.csv"/>
        <param name="output_format" value="csv"/>
        <param name="run" value="all"/>
        <param name="time" value="last"/>
        <param name="verbose" value="false"/>
        <param name="header_row" value="true"/>
        <param name="applicable_agents"
            value="construct::intvar::human_agent_list"/>

        <param name="decision_names" value="
  interacted_with_whom /* record cumulative information */"
  type="decision_name_list" with="agent"/>


        <param name="interacted_with_whom" value="
 $result$ = ''; /* initialize value to return */
 foreach $i$ (construct::stringvar::human_agent_list)
 {
     $result$ = $result$ +
        if($result$!='') { ','} else { '' } +
        interacted_with_agent$i$; /* create value for agent i*/
 }
 return $result$;"
 type="decision_name_list" with="$result$"/>


        <param name="interacted_with_agent$i$" value="
 /* make an accumulator, using this period and past periods */
 previousResult:int +
    (getInteractionNetwork[agent,$i$] ||
       getInteractionNetwork[$i$,agent])"
 with="agent,$i$=construct::stringvar::human_agent_list"/>
    </parameters>
</operation>
```

knowledge to an agent, and (in this example) how many times that agent has interacted with a particular alter. Specifically, this example demonstrates how the decision system can be used as an accumulator for interaction.

Figure 12 presents one way of employing the decision system to build an accumulator and maximum value counter. The initial parameters at the top of Figure 12 provide important groundwork for the decision setup, as described in Section 3. The output will be saved in the csv file `decisions.csv`. Output will be printed only in the last time period, meaning that only one set of values will be printed and that the accumulator will run for the entire experiment. Verbose output is disabled since this example code should not need debugging; users, however, may wish to enable debugging to see how each of the decisions is recognized and interpreted by

the Construct scripting system if they choose to implement this example. A header row will be printed, and this header row will contain the names of the decision printed in that column. The indices of the applicable agents must be provided in the construct variable `human_agent_list`, which should be a comma-delimited list such as "0,1,2."

The `decision_names` parameter specifies the names of the decisions to be evaluated, as discussed in Section 3.1. Here, there is only one set of decision names to be evaluated: the `interacted_with_whom` set of decisions. Note that the `interacted_with_whom` set of decisions, when defined, have a `type` parameter with value "`decision_name_list`" in order to indicate that they list the names of other decisions as opposed to defining decision scripts themselves. The advantage of using an intermediate parameter such as `interacted_with_whom` is that the original `decision_names` parameter can be easily extended and appears relatively uncluttered; for instance, a set of decisions could be added to determine which of the accumulator values is the maximal with minimal overlap to the older decisions.

The `interacted_with_whom` decisions are used to count how many times a (dynamically defined) agent interacts with a (statically defined) alter. If an agent's interaction with a particular alter should be counted, an `interacted_with_agent$i$` name (for example, `interacted_with_agent0`) should be created for that alter agent. Thus, the decision name `interacted_with_agent0` will count the number of times each of the `applicable_agents` interacts with agent index `0`. The list of decision names is built using a `foreach` loop as seen in Figure 12. In the loop, the `human_agent_list` keeps track of the alter agent indices whose interactions should be counted; for each agent in that list, the agent index is appended to the string "`interacted_with_agent`" in order to create the decision name `interacted_with_agent0`, `interacted_with_agent1`, and so forth. The eventual loop is a comma-separated list.

Note that the value returned from the `foreach` loop is the value `$result$`. Initially, at the beginning of the `foreach` loop, this value is set to be the empty string '', an assignment that types the `$result$` variable as a string. As the loop proceeds, the result string will have the value `interacted_with_agent$i$` appended for different values of loop variable `$i$`. During the first iteration of the loop, a comma will not be prepended to the result since the string will initially be empty and the `if` statement will execute the false branch. After the first decision name is added, however, commas will be prepended before all subsequent decision names since the string will be non-empty.

The param with name "`interacted_with_agent$i$`" specifies the actual decision that will be evaluated on a per-agent basis, and is the last parameter specified in Figure 12. A separate parse tree for each `interacted_with_agent$i$` decision will be created for each alter `$i$` in the `human_agent_list`, since `$i$` is specified as a `with` variable. The value of the `interacted_with_agent$i$` decision at a particular point in time is equal to the last period result, plus 1 if the agent interacted or received communication from the specific alter agent. The `interaction sphere network` is a Boolean network that contains a 1 if the ego (the first parameter) initiated contacted the alter (the second parameter), and a zero otherwise. The macro parameter `agent` specifies that the agent in question will be dynamic and thus will be different when the macro is evaluated for agent 0, agent 1, and so forth. The macro parameter `$i$`, however, is static.

When Construct runs, the decision becomes a simple accumulator. Each decision in the decision system is evaluated for every agent during every time period. Even though the decision is not printed until the last time period (due to the fact that the printing is specified for the "last" time period by the decision parameter `time`), the decision is still evaluated and updated internally during every time period. Initially, all decision values are initialized to zero (the numerical representation of the empty string) for each `applicable_agent`. When the decision is evaluated in the first time period, the `previousResult` is this initialized zero value. The `previousResult` value is incremented if the agent interacts with alter on which decision is based, meaning that if the decision is `interacted_with_agent0` then the value is incremented if the agent interacted with agent `0`. In the following period, the second time period, the `previousResult` is either one or zero based on whether there was interaction in the first time period. The new `previousResult` – the value from the first time period – is incremented based on whether a new interaction occurred (as the `interaction network` is always cleared between time periods to only capture interaction in the most recent time period). This process continues until the end of the simulation, meaning that the final value must be between 0 if there was no interaction between an ego and alter and the number of time periods in the simulation if there was continuous interaction. At the end of the simulation, the value is printed, as requested by the user.

Several interesting observations about the decision system are captured in this set of accumulator decisions. First, the decisions that are possible are highly dynamic. Even though Construct does not have an internal mechanism for keeping track of how many times agent A interacted with agent B, the user is able to program this using the decision system and thus "instruct" Construct on how to efficiently keep track of relevant data. Though all the information is available at one point in the simulation, only by using the decision system is the experimenter able to aggregate information in a useful fashion and record the information at the end of the simulation. Second, it is possible to use macros in the creation of both decisions and decision names. This has the advantage of making the macros very general, since it does not matter whether there are ten applicable agents in the simulation or ten thousand. By writing a decision in this fashion, one is able to make a flexible input deck. Lastly, this decision illustrates why it is useful to have the decisions evaluated every time period even if they are only printed out intermittently. By allowing the decision system an opportunity to update every time period, it is possible to have an active accumulator. If the decision were only active during the periods in which it was printed, then it would only be possible to gather this data dynamically using Construct with a tremendous waste of storage space.

While this use of the decision system may not seem overly impressive, it can be used to do a number of more interesting things. For instance, one can use these results (and a new decision, of course) in order to figure out the agent with which each `applicable_agent` interacted most. One could then choose to decrease the interaction probability between that most frequent partner using the `setXXX` operation to modify any one of the static interaction factors. Alternatively, one could use this to figure out the mean number of interactions with other agents, and determine which agents were below the mean after some number of time periods had passed. These agents could then be made inaccessible to the `applicable_agent`, a factor which could speed up the rest of the simulation by allowing each agent to have fewer alters to consider. Thus, while the decision described here is primarily designed for counting and output, it can serve as the backbone of a dynamically self-modifying simulation as well.

5.5 Enabling agent interactions based on knowledge transmission

In the decisions discussed in Section 5.4, an accumulator was used to passively keep track of agent interactions. However, decisions can be much more active and can transform the way the simulation performs and how agents evolve. Decisions can modify knowledge, beliefs, weights, interaction patterns, and a variety of other factors which previously could only be defined statically in the input deck. This effectively allows a simulation to be self-modifying, as agents in the network will be able to interact with each other and by their behavior change the nature of the space in which they interact. In the example discussed in this section, the decision system is used to enable interaction between a series of human agents and an intervention as long as the human agents learn an intervention existence fact. This example has been used in the implementation of interventions such as web pages in forthcoming research.

The decision script outlined in Figure 13 illustrates this decision. It examines a list of `applicable_agents` (which is initialized to a `human_agent_list` similar to that of the example in Section 5.4) and determines if the agents know sufficient facts in the `existence` fact group. If the agents know sufficient facts and are cleared by the experimenter to access the intervention, then agents are allowed to access the intervention in future periods. Note that just because an agent is granted access to the intervention does not mean that the agent will immediately seek out the intervention; instead, the agent will then be able to use its transactive memory of the intervention agent to evaluate the intervention as a possible interaction partner. It may then choose the intervention as its interaction partner based on its evaluation of all other possible interaction partners.

In this set of decisions, only one decision, the `intervention_access` decision, is to be performed; therefore, for brevity the name of the decision to be evaluated is included directly in the `decision_names` tag. This decision will modify the agent's `access network`, the network which supplements the `interaction sphere` in order to determine which agents can interact with which. The interaction sphere serves as a hard constraint for knowledge-based interactions in Construct. Agents who do not have a particular alter in their interaction sphere will not be able create transactive memories of that alter and therefore will not seek to interact with it. The access network supplements the interaction network by specifying which agents can interact with each other. Alters who are in the interaction sphere of one agent, but do not have access to it, will be able to exchange information with other (via third parties) but cannot directly interact. If the access network is modified, agents will retain their transactive memory of the alters but will either gain or lose the ability to interact with the alter.

At the beginning of the simulation, it is assumed that every `applicable_agent` has the intervention agent in its interaction sphere (i.e. values set to `1` or `true`) but does not have access to the intervention (values set to `0` or `false`). This must be initialized when the respective Construct networks are set up.

When the `intervention_access` decision is first run, then the access network contains zeros for every applicable agent, and the decision will set this value to true if the agent learns sufficient knowledge. The `intervention_access` decision starts out with a `previousResult` value of `false`. This occurs because result is initialized to the empty string which, when casted to a bool, has a value of false.

When the initial outer `if` statement is first executed, the `else` branch will be taken and the inner `if` statement will be performed. It will then examine whether or not the agent has knowledge of the facts in the `existence` fact group. This fact group must be set up using the

Figure 13: Using decisions to modify agent behavior

<div align="center">Decision</div>

```
<operation name="ReadDecisionOutput">
    <parameters>
        <param name="output_filename" value="decision.csv"/>
        <param name="output_format" value="csv"/>
        <param name="run" value="all"/>
        <param name="time" value="last"/>
        <param name="verbose" value="false"/>
        <param name="header_row" value="true"/>
        <param name="applicable_agents"
            value="construct::intvar::human_agent_list"/>

        <param name="decision_names" value="
  intervention_access /* determine if can talk to intvn */"
  type="decision_name_list" with="agent"/>

        <param name="intervention_access" value="
  if(previousResult:bool) /* if already set to true */
  {
      previousResult      /* preserve, do not modify */
  }
  else                    /* otherwise */
  {
      /* if the agent knows a sufficient number of facts */
      /*   AND the experimenter said that access is possible */
      if((getKnowledgeNetwork[agent,
          construct::knowledgegroupvar::existence] >= 1.0) &&
        (getInternetAccessNetwork[agent,0]:bool))
      {
          /* allow the agent to interact with the intvn */
          /*   set result to true to avoid modification */
          (setAccessNetwork[agent,
              construct::agentgroupvar::intervention,1] > 0)
      }
      else                /* otherwise */
      {
          0               /* set to false */
      }
  }" with="agent,decision"/>
    </parameters>
</operation>
```

knowledge group membership network in order to specify which facts are associated
with the existence group. The construct::knowledgegroupvar::existence
reference will then instruct Construct to get the indices of the existence facts in a comma-

delimited list, as described in Section 2.3. These indices are then fed to the `knowledge network` in order to determine whether the agent knows the particular facts.

If the agent knows a fact, the knowledge network will have a value of 1 for the agent at that fact location. If the agent knows at least one of the facts in the `existence` fact group, the value returned by the `get` operation will be at least 1 and the inner if statement will return true. If this occurs, the decision will perform a `set` operation in order to allow the agent to interact with the agents specified in the `intervention` agent group. This set operation will set the access value to 1 between the source `applicable_agent` and the target `intervention` agents. Because the `intervention` reference is set up as a reference to an agent group (as opposed to a single index), access to multiple agents can be manipulated at the same time. The trailing greater than (>) comparison operator will allow the set statement to return a 1 if the set is successful, as opposed to the number of values set. This is purely cosmetic, but allows the experimenter to quickly count the number of agents who made the decision.

After the agent has made the decision and the `access network` has been modified, the result of the decision is one. When the decision is evaluated in the next period, the outer `if` statement retrieves this value and casts it to a bool. Since any non-zero value, when cast to a bool, is `true`, the then part of the outer `if` statement will be executed. Since the `previousResult` value will be returned from the then statement, the value of 1 will be preserved.

If the agent did not have sufficient knowledge in the inner `if` statement, then the inner `else` statement is executed, the access network is left unchanged, and a value of zero is returned. During subsequent periods, the check will be performed again in order to determine if the agent should have access to the interventions. A check is performed each time period until an agent has access to the intervention, after which the outer `if` statement will avoid having to check against agent knowledge and thus improve performance.

The use of these nested `if` statements in this decision has several advantages. First, because the setting of the matrix value is a relatively expensive operation that only needs to be performed once, the use of the outer `if` statement prevents the setting operation from being performed multiple times. This helps improve the running time of the decision. Second, it will be easy to see which agents have made the `intervention_access` decision: those with a value of 0 will not have made the decision, while those that have a value of 1 – the preserved value of the `previousResult` – did. However, it is important to note that none of these logical statements can be `static_if` operation. A `static_if` operation will always return the same value since it chooses a branch when it is initially computed. Since the outer `if` changes behavior when the agent's access has been set, its behavior cannot be defined at simulation start. Similarly, the inner if statement will be modified based on agent knowledge; as the agent learns new information over the course of the simulation, the value read in the inner if statement may change. Thus, it is necessary to use `if` statements for both statements. While these if statements are slower than `static_if` statements, they allow the decision to have the highly dynamic behavior that allows the simulation to be self-modifying.

While this example is complex, it illustrates a new and exciting area for Construct decisions. The decision described in Figure 13 is one which describes a logic-based means for modifying agent behavior, but does not explicitly specify the exact agents for whom this decision will be applied. Instead, as the simulation evolves, some agents learn sufficient information so that this decision is applied to them, while many agents will not learn this knowledge and thus will never

have their access networks modified. Decisions like this one allow the experiment to dynamically self-modify based on the characteristics of the agents in the simulation. For instance, using a similar decision with different networks, it should be possible to block the most frequent connections between each pair of agents in order to understand what the effect of this isolation would be on information diffusion. Because it may not be possible to predict *a priori* which connection would be activated most frequently, it might not be possible to specify such a modification in older versions of Construct. The dynamic nature of the decision system, however, opens up exciting new possibilities for simulations which will be able to modify themselves.

6 Conclusion

Construct variables and decisions, both of which are defined using the Construct scripting language, are quite powerful. They allow the user to customize a variety of important parts of the Construct system and to set up experiments more efficiently. This technical report has sought to introduce the variable, decision, and scripting systems available in the Construct simulation system as implemented in Construct version 3.9. Future versions of Construct may expand upon these systems in order to make the overall simulation more powerful. For instance, new scripting lexemes may be added in the future. These lexemes and other changes will be added to the help files for the tool.

The most immediate advantage of these systems is that they can greatly increase the speed of input deck development and analysis. The variable system described in this document allows a user to quickly modify constants to Construct. By using variables that reference other variables, one can quickly propagate any changes from one variable to the next. Such a change greatly simplifies modifications to input files, allowing instantaneous changes to core functionality. Additionally, the decision system allows the user to print out only the information that is relevant to the user's research needs. Thus, rather than printing out a host of information and relying on a post-processing program to extract relevant information, one can use the decision system to specify a small number of outputs which synthesize multiple factors.

However, the future advantage of these systems is only beginning to be explored. The decision system provides a way for users to specify decisions that allow the simulation to self-modify. Decisions could specify changes to agents if or when the agents meet some pre-specified criteria as opposed to specifying the agents by index. This allows for more realistic behavior modeling as well as more interesting and exciting intervention modeling. Because the decision system allows the experimenter to have access to the core levers of the simulation while the simulation is running, the system thus gives the experiment designer significantly more power than previously. Such potential will be explored with the current as well as future versions of Construct.

Appendix A   Reserved words in the Construct Scripting Language

The following words are reserved in the Construct scripting language, and should not be used as macro or variable names.

```
construct                if                    randomUniform
delay_interpolation      preserve_all_         randomNormal
details                    white_space         return
else                     preserve_             set*
error                       spaces_only        static_if
foreach                  random*               verbose
get*
```

There are four caveats with the above list. First, the word preserve_white_space, without spaces, is a keyword and should not be used (though for formatting purposes had to be distributed over two lines). Second, any string that begins with the word "get" or "set" will be treated as a network reference. The remainder of the network reference must be a valid network name specified in CamelCase, as specified in Section Appendix B, part XI. Third, all words beginning with the word "construct" or "random" are reserved for future extensions. Last, all other words beginning with an alphabetic character will be treated as variables, though in many cases the user will have to specify that the variable is a valid variable by declaring it as a `with` variable.

The following words are reserved by the decision system, and should not be used as macro or variable names when writing decisions. Additional information on these reserved words can be found in Section 3.

```
agent                    timeperiod
```

Appendix B   Operations in the Construct Scripting Language

The remaining appendix sections discuss the operations in Construct. An overview of the scripting language commands presented in this appendix can be found in Section 4.1, which begins on page 18. Commands are organized thematically so that, for instance, all relational operators are grouped together in a particular section.

It is worthwhile to briefly mention several features of the language syntax used in the remainder of this appendix, as the presentation format used in this section is nonstandard when compared to other language specifications. First, all scripting or ConstructML keywords are specified in a `monospace` font, while explanations are provided in a standard Times font. This behavior is identical to that observed in the remainder of the document, but may differ from the manner found in other reference manuals. Second, an arbitrary expression in monospace font surrounded by angle brackets, such as `<text>`, is an indication that any valid expression can be used at that location. Sometimes these arbitrary expressions will be specific – for instance, `<BoolExpr>` means that a Boolean expression must be provided at that location – but the expression modifiers should be clear. Third, any use of square brackets ([]) or curly braces ({}) is deliberate and must be included in the scripting language. While in some language

36

descriptions these characters are used to indicate optional characteristics, this is not the behavior intended in the remained of this description.

Appendix B, part I.   Comments and Constants

- **comment:** `/* <Text> */`

In the Construct scripting language, it is possible to include comments internally within the expression to be evaluated. Such comments are in addition to – and thus need not replace – the standard XML comments which can occur throughout the input file. However, because some expressions may become complex, it can be useful to provide comments in line with the test. C++-style comments can be used within the expression to provide readability.

Newlines can be included before, after, or during the comment. They will be ignored by the parser. The newlines included in the input file, however, may be printed when debugging.

Commas (`,`) and quotes (`'`) included in comments may cause problems. Certain sub-modules of Construct may attempt to split a string into comma-separated values without first stripping comments; commas embedded within comments may be falsely recognized and lead to errors. Similarly, the Construct parser may have difficulty determining whether a particular quoted string is fully included in a comment or not. For this reason, users are strongly encouraged not to use commas or quotes within comments.

As of Construct 3.9, it is NOT possible to use the C-style double-backslash (`//`) notation to provide comments. This feature may be supported in a future version of the system.

- **quoted literal:** `'<Text>'`

To unambiguously specify strings, arbitrary strings of characters can be included within single quotes (`'`).   All quoted literals are treated as strings, and conversions to other types will require cast operations.

In order to quickly alert the user to quoting errors, the maximum length of a quoted string is limited to one hundred characters. In order to specify strings that contain more than one hundred characters, the concatenation operator can be used to combine multiple substrings.

All but two types of characters can be included in string literals. Currently, there is no escape character in place, so it is not possible to include the `'` character as part of the string. Additionally, embedding commas in a sequence of quoted text may confuse the parser in some places.

- **numbers:** `<Number>` or `<Number>.<Number>` or `-<Number>.<Number>`

Numbers are used to generate numeric constants. Numbers can be specified in two forms, integers (which lack decimal points) and floats (which have them). Numbers specified using an optional minus sign, then a required sequence of values using the numerals 0 to 9, then an optional decimal point, then an optional decimal component.

Internally, numbers are stored as integers when possible and floats otherwise. Note that Construct will attempt to represent the number using the smallest type possible. To explicitly create a floating point value from what would otherwise be an integer, the decimal point should

clearly be specified. Thus, a value of `1` will be represented as an integer, while `1.0` should be represented as a float.

Negative numbers can be specified by including a negative sign (`-`) immediately prior to the number.

Construct can represent any sixteen-bit, twos complement signed integer numbers, as well as C-style float.

- **white space:** `<space>` or `<tab>` or `<newline>`

In the current implementation, white space is ignored to improve clarity. This means that input deck designers can include whitespace and newlines in order to better comment their work. Spaces, tabs, and newlines are automatically ignored; they are not used to represent breaks in the script as parsed. The only place in which white space is not ignored occurs in quoted strings; in quoted strings, white space is interpreted literally.

It is important to note that the removal of white space may have unintended consequences. For instance, some network names in Construct are written with internal spaces. When specifying variables which refer to these names, it is necessary to write such names as quoted literals. If spaces are not supplied, the system will fail with an error.

- **sub-expression:** `(<Expr>)`

Parentheses are used to separate subexpressions and to specify the order of evaluation as well as to represent important subexpressions for other tokens. Some expressions, such as the if statement, require subexpressions; in other cases, subexpressions are used to specify the order of operation. The type returned by the evaluation of the parentheses is the same as that of the internal expression; adding parentheses has no effect other than to order operations.

At this point, the Construct scripting language does not have an operator-precedence parser; equality expressions, multiplicative expressions and additive expressions all have the same precedence levels. Users will need to use parentheses and subexpressions in order to express these precedence levels. Future versions of Construct should correct this issue and therefore minimize the numbers of parentheses necessary.

Appendix B, part II.   Math Operations

- **addition:** `<Expr>+<Expr>`

The addition (+) operator adds numbers and concatenates strings. If either of the operands are strings, the addition operation converts the other operand to a string and performs a string concatenation; for instance, adding "test" and "1" will create the string "test1". If neither operand is a string, then numeric addition is performed; the addition result will be a float if one at least one operand is a float and will be an integer otherwise. These casts are performed implicitly and do not have to be specified by the user.

- **subtraction:** `<Expr>-<Expr>`

The subtraction (-) operator takes the difference of two numeric lexemes. If either of the operands is a float, the value will be returned as a float; otherwise, the value will be returned as an integer. Note that subtraction cannot be performed with strings.

- **multiplication:** `<Expr>*<Expr>`

The multiplication (*) operator is used to multiply two numeric lexemes. If either of the operands is a float, the value will be returned as a float; otherwise, the value will be returned as an integer. Note that multiplication cannot be performed with strings.

As of Construct version 3.9, the scripting language does not include precedence. In addition, math expressions are evaluated right-to-left. Thus, evaluating the expression `3*4+1` will result in the value `15` since the addition is performed before the multiplication (as opposed to `13` which would be expected with standard precedence rules). To correct for this problem, use parentheses around the multiplication.

- **division:** `<Expr>/<Expr>`

The division (/) operator is used to divide one numeric lexeme by the other. If either of the operands is a float, the value will be returned as a float; otherwise, the value will be returned as an integer. Note that division cannot be performed with strings; instead, the subsequence operator (see below) is applied and a string result is created. If the divisor for the division operation evaluates to zero, the operation will fail; however, at this time, Construct does not check for division errors.

- **exponentiation:** `<Expr>**<Expr>`

The exponentiation (**) operator is used raise one numeric lexeme to the power of another. If either of the operands is a float, the value will be returned as a float; otherwise, the value will be returned as an integer. Note that exponentiation cannot be performed with strings. All expressions will be evaluated with the C-function pow(), which will evaluate both positive and negative numbers.

Appendix B, part III.   String Operations

- **concatenation:** `<Expr>,<Expr>`

The concatenation (,) operator is used to separate two values in a sequence. This operator has two important uses.

First, the concatenation operator is often used in order to create lists of entries. For instance, in most with statements (i.e. Section 0 and Section 4), the sequence of iterated variables is specified using a comma-delimited sequence. The way the list is eventually parsed will depend upon the way in which the list is being used, and will depend upon how the underlying Construct code has been written. For instance, in a foreach loop, each element in the list of iterable

39

parameters is parsed as an integer. In other cases, such as the listing of parameters in with statements, each element is parsed and interpreted as a string.

In addition, the concatenation operator is employed internally by the parser to separate parameters to other scripting commands. For instance, commands to get and set values in Construct networks use commas as separators for internal parameters.

Unlike most other operators, which return a type equivalent to that of their subexpressions, the return type of the sequence operator is a string. The sequence operator can be used multiple times in sequence to create a string containing integer, float, and non-numeric values, though if the string will eventually be parsed by another portion of the Construct system it will be necessary to specify the string components as per what the other component requires. The concatenation operator will not keep track of the components in the string – i.e. there is no explicit representation for "string of strings" or "string of ints".

- **subsequence:** `<StringExpr>/<StringExpr>` or
  `<StringExpr>|<StringExpr>`

The subsequence (`/`or `|`) operator is used to specify a group of related items in a sequence. This operator can be specified as either a single backslash or a single bar. This functionality allows the script designer to specify a list-within-a-list.

The subsequence operator is often used in conjunction with specific network generators in Construct (Hirshman and Carley 2007). For instance, the socio-demographic proximity generator is used to construct similarity scores for agents depending on the number of identical attributes that they share. Most agents will only have one value per attribute, but some agents may need two or more values to ensure overlaps with other agents. To create agents with multiple attributes, it is possible to use the subsequence operator to specify these overlapping attributes. However, other uses for lists-within-lists may be implemented in future versions of Construct.

It is important to note that the subsequence operator is the same as the division operator and will be used in place of the division operator if one or both of the expressions are strings. Thus, it is possible for a subsequence operator to supplant a division operator if the script user does not rigorously check and debug what is written. On the other hand, users may wish to be wary when trying to create a subsequence that contains multiple numeric values: the subsequence `1/2/3` will be treated as a number unless the experiment designer makes clear that the value should be treated as a string (for instance, by writing `1/2/3:string`).

- **enumeration:** `<Expr>..<Expr>`

The enumeration (`..`) operator is used to create a sequence of integers between a minimum and maximum value. This operator allows a comma-separated list of integers to be created quickly and efficiently without using a foreach loop.

The enumeration operator creates a sequence of integers only. The list is generated from the minimum value, inclusive, up to the maximum value, inclusive – i.e. the operation `1..5` will generate the sequence `1,2,3,4,5`. If the minimum value is greater than the maximum value, an empty string will be returned.

At this time, enumeration can only occur for integer inputs and will enumerate all values between the minimum and maximum value. It is not possible to specify an increment value

other than 1 at this time, though a `foreach` loop can mimic such an action if it is given the proper inputs. It is not possible to enumerate strings at this time using this operator.

When Construct evaluates an enumeration expression, it first evaluates the expression on the left-hand size of the enumeration, then the right-hand side of the enumeration, and then performs the enumeration. The left and right-hand expressions are evaluated in right to left fashion.

Note that enumeration has the same priority as addition, subtraction, and other mathematical operations. This means that may be important to correctly parenthesize any written code in order to ensure that a desired result is obtained. For instance, the enumeration `3+1..5` will generate the sequence `31,2,3,4,5`, evaluating the subsequence operator and prepending the number 3, since the subsequence is the rightmost operation. To create the result which was probably desired, it is necessary to enclose this minimum value with commas to create `(3+1)..5`, which evaluates to `4,5`. On the other hand, the enumeration expression `1..5+3` will lead to the result `1,2,3,4,5,6,7,8` since the addition is performed prior to the enumeration evaluation, since the addition is the rightmost operation.

Appendix B, part IV. Logical Operations

- **logical and:** `<Expr>&&<Expr>`

The logical and operator (`&&`) performs a logical and of the left-hand side and right-hand side values. If both expression values are not strings, it will cast the expression on the left-hand side and that on the right-hand side to bool values before performing the operation. If either side is a string, the operation will fail. The logical and of two numeric or Boolean values will be a 1.0 if both of the inputs are non-zero; otherwise, a value of zero will be returned.

It should be noted that certain XML editors may warn when displaying the ampersand (&) character that is used in this operation, since the ampersand is a reserved keyword in XML. Construct input decks are written in ConstructML, which is a derivative of XML. Such parsing errors may be observed if the user works with XML editors such as Microsoft Visual Studio. However, the ConstructML reader in the Construct executable will NOT have problems with this syntax.

Future versions of Construct will allow the ampersand character to be represented as the string "&" for compatibility with standard XML editors. At this time, though, this feature has not been implemented in Construct and attempting to use the "&" string in place of an ampersand will result in a parsing error.

- **logical or:** `<Expr>||<Expr>`

The logical or operator (`||`) performs a logical or of the left-hand side and right-hand side values. If both values are not strings, it will cast the expression on the left-hand side and that on the right-hand side to bool values before performing the operation. If either side is a string, the operation will exit and error. The logical or of two numeric or Boolean values will be a 1.0 if either of the two numeric inputs is non-zero; otherwise, a value of zero will be returned.

- **exclusive or:** `<Expr>^<Expr>`

The exclusive or operator (^) performs an exclusive or of the left-hand side and right-hand side values. If both values are not strings, it will cast the expression on the left-hand side and that on the right-hand side to bool values before performing the operation. If either side is a string, the operation will exit and error. The exclusive or of two numeric or Boolean values will return 1.0 if exactly one of the numeric or Boolean inputs is non-zero; if both inputs are zero or both inputs are non-zero, a value of zero will be returned.

- **negation:** `!<Expr >`

The negation operator (!) performs a negation of the right-hand side and value. If the value is not a string, it will cast it to bool values before performing the operation. If the right-hand side is a string, the operation will exit and error. A negation of a numeric or Boolean value will return zero if the numeric input is non-zero, and will return 1.0 if the numeric input is zero.

Appendix B, part V.   Comparison Operations

- **equality:** `<Expr>==<Expr>`

The equality operator (==) performs a logical comparison between the result of two expressions, returning the Boolean value 1.0 if the values are the same and 0 if they are different. In order to perform the comparison, both expressions must have the same type. If either value is a string, any non-string value is cast to a string before a string comparison is performed. Otherwise, if either value is a float, the non-float value is cast to a float prior to performing the comparison. If this is not the case, then an integer comparison is performed unless both values are Booleans (in which case a Boolean comparison is performed).
   Note that Construct does not implement operator precedence rules and evaluation proceeds in a right-to-left order among relational, multiplicative, and additive operators. Thus, the resulting value from evaluating the expression `2+1==3` will be the integer value 2, because the script will perform a comparison between the integer values 1 and 3 before casting the resulting value (the bool value 0, since the values are equal) to an integer and adding it to the leftmost 2. In order for the script to return the value that would result from conventional precedence rules, it is necessary at this time to use explicit parentheses: the expression `(2+1)==3` will return the bool value 1, which would be the result in most other programming languages.

- **inequality:** `<Expr>!=<Expr>`

The inequality operator (!=) performs a logical comparison between the two expressions and returns the Boolean value 1.0 if the values are not equal.
   Note that Construct does not implement operator precedence rules and evaluation proceeds in a right-to-left order among relational, multiplicative, and additive operators. Thus, the resulting value from evaluating the expression `2+1!=3` will be the integer value 3, because the script will perform a comparison between the integer values 1 and 3 before casting the resulting value (the bool value 1, since the values are not equal) to an integer and adding it to the leftmost 2. In order for the script to return the value that would result from conventional precedence rules, it is

necessary at this time to use explicit parentheses: the expression `(2+1)!=3` will return the bool value 0, which would be the result in most other programming languages.

- **less than:** `<Expr><<Expr>`

The less than operator (<) performs a logical comparison between the two expressions and returns the Boolean value 1.0 if the value on the left-hand side is less than the value on the right hand side. Note that it is not possible to compare strings using the less than operator. Attempting to do so will result in an error.

Note that Construct does not implement operator precedence rules and evaluation proceeds in a right-to-left order among relational, multiplicative, and additive operators. Thus, the resulting value from evaluating the expression `2+1<3` will be the integer value 3, because the script will perform a comparison between the integer values 1 and 3 before casting the resulting value (the bool value 1, since 1 is less than 3) to an integer and adding it to the leftmost 2. In order for the script to return the value that would result from conventional precedence rules, it is necessary at this time to use explicit parentheses: the expression `(2+1)<3` will return the bool value 0, which would be the result in most other programming languages.

It is worth noting that standard XML parsers reserve the less than character (<) and that expressions including a logical and may result in parsing warnings or errors. However, the ConstructML reader supplied with Construct will silently ignore these warnings and load Construct input decks containing this lexeme. Future versions of Construct will allow the greater than character to be represented as the string ">" for compatibility with such parsers, although at this time such behavior is not supported.

- **greater than:** `<Expr>><Expr>`

The greater than operator (>) performs a logical comparison between the two expressions and returns the Boolean value 1.0 if the value on the left-hand side is greater than the value on the right hand side. Note that it is not possible to compare strings using the greater than operator. Attempting to do so will result in an error.

Note that Construct does not implement operator precedence rules and evaluation proceeds in a right-to-left order among relational, multiplicative, and additive operators. Thus, the resulting value from evaluating the expression `2+1>3` will be the integer value 2, because the script will perform a comparison between the integer values 1 and 3 before casting the resulting value (the bool value 0, since 1 is less than 3) to an integer and adding it to the leftmost 2. In order for the script to return the value that would result from conventional precedence rules, it is necessary at this time to use explicit parentheses: the expression `(2+1)>3` will return the bool value 0, which would be the result in most other programming languages.

It is worth noting that standard XML parsers reserve the greater than character (>), and that expressions including a logical greater than may result in parsing warnings or errors. However, the ConstructML reader supplied with Construct will silently ignore these warnings and load Construct input decks containing this lexeme. Future versions of Construct will allow the greater than character to be represented as the string ">" for compatibility with such parsers, although at this time such behavior is not supported.

- **less than or equal to:** `<Expr><=<Expr>`

The less than or equal to operator (<=) performs a logical comparison between the two expressions, and returns the Boolean value 1.0 if the value on the left-hand side is less than or equal to the value on the right hand side. Note that it is not possible to compare strings using the less than or equal to operator. Attempting to do so will result in an error.

Note that Construct does not implement operator precedence rules and evaluation proceeds in a right-to-left order among relational, multiplicative, and additive operators. Thus, the resulting value from evaluating the expression 2+1<=3 will be the integer value 3, because the script will perform a comparison between the integer values 1 and 3 before casting the resulting value (the bool value 1, since 1 is less than 3) to an integer and adding it to the leftmost 2. In order for the script to return the value that would result from conventional precedence rules, it is necessary at this time to use explicit parentheses: the expression (2+1)<=3 will return the bool value 1, which would be the result in most other programming languages.

- **greater than or equal to:** `<Expr>>=<Expr>`

The greater than or equal to operator (>=) performs a logical comparison between the two expressions, and returns the Boolean value 1.0 if the value on the left-hand side is greater than or equal to the value on the right hand side. Note that it is not possible to compare strings using the greater than or equal to operator. Attempting to do so will result in an error.

Note that Construct does not implement operator precedence rules and evaluation proceeds in a right-to-left order among relational, multiplicative, and additive operators. Thus, the resulting value from evaluating the expression 2+1>=3 will be the integer value 2, because the script will perform a comparison between the integer values 1 and 3 before casting the resulting value (the bool value 1, since 1 is less than 3) to an integer and adding it to the leftmost 2. In order for the script to return the value that would result from conventional precedence rules, it is necessary at this time to use explicit parentheses: the expression (2+1)>=3 will return the bool value 1, which would be the result in most other programming languages.

Appendix B, part VI.   Random Number Operations

- **generate random number from a uniform distribution:**
  `randomUniform(<MinExpr>,<MaxExpr>)`

The `randomUniform` expression creates a random float value drawn from a uniform distribution. The random number generator generates a new random number each time it is invoked, meaning that the expression is evaluated as Construct is executed and not when the statement is parsed. The random number generator invoked by the script is the same random number generator employed by Construct, and is governed by the seed parameter specified in the input file. Both the scripting language random numbers and the simulation random numbers are initially set by the random seed parameter, a simulation parameter.

The `MinExpr` and `MaxExpr` expressions can be any expressions which evaluate to integer or float values. If both subexpressions and the comma are omitted, the minimum and maximum values will default to zero and one; thus, the no-parameter call to `randomUniform()` is

effectively a shorthand for the invocation `randomUniform(0,1)`. Note that the value returned by both types of invocations will be a float.

The random uniform number generator generates a random float value between the value of `MinExpr` and `MaxExpr`, inclusive. For instance, the expression `randomUniform(2,5)` will generate a random float value between 2.0 and 5.0 with uniform probability. If an integer value is desired, such a value can be obtained by generating a random value between `MinExpr` and `MaxExpr+1` and then casting the resulting value to an integer. Thus, to generate a random integer between 2 and 5, the appropriate call would be `randomUniform(2,6):int`.

A fresh random number will be generated each time the script makes a call to either `randomUniform(<MinExpr>,<MaxExpr>)` or `randomUniform()`. If one wishes to generate only a single random value, or to cache a random number for later use, it is best to save the random number to a Construct variable and refer to that variable at a later point. Such a scenario can, for instance, occur when using the scripting language as a generator or in the decision system.

- **generate random number from a normal distribution:**
  `randomNormal(<MeanExpr>,<VarianceExpr>,<MinExpr>,<MaxExpr>)`

The random normal number generator generates a float value from a normal distribution with mean `MeanExpr` and variance `VarianceExpr`. The random number generator generates a new random number each time it is invoked, meaning that the expression is evaluated as Construct is executed and not when the statement is parsed. The random number generator invoked by the script is the same random number generator employed by Construct and is governed by the seed parameter specified in the input file. Both the scripting language random numbers and the simulation random numbers are initially set by the random seed parameter, a simulation parameter.

The `MeanExpr`, `VarianceExpr`, `MinExpr`, and `MaxExpr` expressions can be any expressions which evaluate to integer or float values.

The random normal number generator generates a number from a normal distribution between the value of `MinExpr` and `MaxExpr`, inclusive. For instance, the expression `randomNormal(0,1,-3,3)` will generate a random float from a normal distribution with mean zero, variance one, minimum value negative three, and maximum value positive three. If an integer value is desired, it will be necessary to cast the resulting value to an integer using a cast operation. Note, however, that the casting operation is a truncation, so users seeking to create integer values from a normal distribution may prefer to generate only one side of the normal distribution based on a random uniform value and with that probability negate the value generated by the normal distribution.

If no minimum and maximum expressions are provided, the number generated can range from negative infinity to positive infinity. However, for most Construct applications, it is necessary to bound the range of the random numbers that can be generated. For example, the socio-demographic proximity weights of individual agents must be between zero and one (technically, it and a number of other factors must sum to one, as described in (Hirshman and Carley 2007), although the general point that the value has an upper bound is still valid). In order to ensure that this value does not exceed a particular range and thus create malformed input, a minimum and maximum value for the random number can be provided. Note that the minimum and maximum numbers do not have to be equidistant from the mean; it is perfectly

Figure 14: Examples of cast operations

Variable	Value
`<var name="var1" value="(2/4):float"/>`	`"0.0"`
`<var name="var2" value="(2/4:float)"/>`	`"0.5"`
`<var name="var3" value="construct::floatvar::var2"/>`	`"0.5"`

valid to create a distribution where, for instance, the mean is much closer to the minimum than the maximum.

The (inclusive) minimum and maximum bounds on the random number serve as post-processing bounds on random number generation. This means that a candidate number is initially drawn from a normal distribution with the user-specified mean and variance, and then compared to the bounds. If the number is within the bounds, it is returned; if it exceeds the bounds, a new candidate number is drawn. This process is repeated until a valid random number is obtained. Note that this process may be very slow if the variance greatly exceeds the random number bounds, since it may require the generation and retiring of many candidate random numbers.

The random normal random number generator can technically be run without a mean and variance. In such a scenario, the mean employed would be `0.5`, the valiance would be `0.25`, the minimum value `0` and the maximum value `1`. However, as this generator is likely of limited value, it is recommended that the user specify the mean and variance if only for clarity of the input file.

Different random numbers will be generated each time the script makes a call to either `randomNormal(<MeanExpr>,<VarianceExpr>,<MinExpr>,<MaxExpr>)` or related formulations. If a single random value is desired, or to cache a random number for later use, it is best to save the random number to a Construct variable and refer to that variable at a later point. Such a scenario can, for instance, occur when using the scripting language as a generator or in the decision system.

Appendix B, part VII.   Cast Operations

- **cast to type:** `:<CastType>`

The cast (`:castType`) postfix operator is used to cast from one type to another. At this time, the scripting language supports four acceptable types for casting: `bool`, `int`, `float` and `string`. While future extensions to the lexer may support additional types, specifying a value other than bool, int, float, or string will currently result in a fatal error.

The casting rules are as follows. Any non-zero integer, as well as the string "true", will result in a true value when cast to a bool, represented internally as 1.0. All other values will result in a false value and will be stored internally as 0.0. Conversion of the value "true" from string to a bool and back to a string will result in a string value of "1.0". The C-function atoi() is used to convert the string representation of any value to any (signed) int and the C-function atof() is used to convert the string representation of any value to any (signed) float.

Two examples of variable casting can be seen in variables `var1` and `var2` in Figure 14. In variable var1, the cast occurs after the (integer) division, so the resulting `0` from the division is cast to a float to become the value `0.0`. On the other hand, in variable `var2` the cast occurs

prior to the division operation, so a floating point division is performed. Thus, the location of the cast can be extremely important.

Note that the cast operation is different than the variable reference operation. In the cast operation, there is only one colon (:), while the variable reference uses two back-to-back colons. Also note that it is not necessary to use the cast operator to convert a variable reference from one type to another. Instead, it is significantly simpler to specifically cast to the desired type. Thus, while one could have evaluated variable `var4` in Figure 14 as a `stringvar` followed by a cast to a float, it is significantly more straightforward to treat the values as a `floatvar` and to omit the casting.

Appendix B, part VIII. If Statements

- **if expression:**
  ```
  if(<BoolExpr>) { < Expr> } else { <Expr> }
  or
  if(<BoolExpr>) { < Expr> } else if(<BoolExpr>) { < Expr> }
    else { <Expr> }
  ```

The `if else` expression allows the scripting language to execute conditional statements and to return conditional results. If the conditional statement following the initial if statement evaluates to true, the value of the first expression is returned; otherwise, each `else if` statement is examined in turn to determine whether the corresponding expression should be returned. If none of the expressions are true, the `else` value is returned.

The syntax of an `if` expression is as follows. The lexeme `if` must be present, followed by a Boolean expression in parentheses (`()`). A required set of curly braces (`{}`) must enclose an expression to be evaluated if the Boolean expression evaluates to true. The closing curly brace must be followed by the keyword `else`, though this `else` may be part of an `else if`. If one or more `else if` conditions are in use, a Boolean expression must follow the `else if`, and in turn must be followed by curly-brace delimited expression for evaluation if the `else if` expression is true. Eventually, any sequence of else if commands must be followed by a terminal `else`. Following this `else`, a set of curly braces enclosing the else expression must be included. Curly braces are required between all expressions; it is not possible at this time to write C-like statements where the curly braces are omitted.

When a conditional test is being performed, the return value of that test must be a Boolean. This means that it is not sufficient to leave a float, integer, or string value as the if expression. The user is required to explicitly cast the conditional expression to a bool in order to verify that an important part of the expression (i.e. a comparison operator) has been omitted.

The returned types of all expressions should be equivalent. While the script should perform some basic checking and casting, it is always useful to examine the values returned from each sub expression of the `if` statement. As with the additive and multiplicative operations, if any of the expressions would return string values then all expressions will be automatically promoted to return strings.

It is also possible to chain zero or more else if statements together to simplify the legibility of scripts. Note, however, that it is always necessary to specify an else condition when an if statement is used, since all script expressions must return a value.

- **static if expression:**
  ```
  static_if(<BoolExpr>) { <Expr> } else { <Expr> }
  or
  static_if(<BoolExpr>) { <Expr> }
     else if(<BoolExpr>) { <Expr> } else { <Expr> }
  ```

The `static_if` expression differs from the standard `if` expression because it is evaluated in a static context: the logical section of the `if` conditional statement is evaluated when the if statement is parsed, not when the if statement is executed. This change has a number of useful yet subtle properties.

First, the `static_if` statement can be used to dramatically speed up code execution. If the experimenter is certain that a given logical expression will never change, the experimenter can use a `static_if` to indicate that one of these values should be used. This is useful when designing efficient decisions (Section 3). A normal `if` statement will be evaluated every time the decision is executed, which is likely to be once every time period for every agent. On the other hand, a `static_if` statement will be evaluated only once, when the expression is parsed saving a significant number of logical tests. If what is being tested is a constant value – for instance, the value of a Boolean variable set as a `construct_var`, then these extra evaluations are unnecessary. The Boolean value should not change and therefore the same branch of the if statement would be used each time. Use of a static if expression allows the experimenter to design decisions which rely on the values of specific `construct_vars`, but do not have a runtime cost to evaluate.

Second, and perhaps more importantly, a `static_if` will only evaluate one of its branches, the branch for which the logical statement evaluates to `true`. The other branch will be ignored by the parser: the parser will check to make sure that there is a block of text surrounded by curly braces, but the commands within that block will not be evaluated. Thus, if the conditional statement of the `static_if` evaluates to true, the "then" part of the statement must contain syntactically valid scripting commands while the "else" portion of the command will be ignored. Thus, the value of the not-taken branch can refer to `construct_vars` that do not exist or cannot be defined in the current context. This can be useful for the creation of base cases for a recursive variable: for instance, if one wishes to create a variable `var$i$` (i.e. `var0`, `var1`, etc) that is defined in terms of a previous value `var$i:int-1$`, one can use the `static_if` syntax to create all variable values greater than zero in one branch while creating the zero value in another branch. If one had used a standard `if` as opposed to a `static_if`, the code would not parse because a `$i$` value of `0` would create the variable expression `var-1`, which would be treated as a subtraction operation as opposed to a single (and likely unintended) variable name.

It is perhaps simplest to illustrate the difference between an `if` statement and a `static_if` statement using an example. Consider a decision which contains either the expression `if(timeperiod > 0)…` or the expression `static_if(timeperiod > 0)…`. In the first case, the if statement would be evaluated every time the statement is executed, which should be every time period of the simulation. This statement would be true for each time period except the first. However, the second statement would be evaluated when the decision is initialized. Since the `timeperiod` variable will be initialized to zero at simulation start, this expression would never be true; the "then" branch of the statement would be ignored and only the "else"

branch would be built in to the decision. Thus, the `static_if` statement would always be false and the else branch would always be executed.

Appendix B, part IX.   Assignments Operations

- **assignment:** `$variable$ = <Expr>;`

The assignment statement (=) allows for the assignment of a variable to a value. The assignment statement allows for the creation of scratchpad variables in scripts in order to facilitate more complex calculations.

The syntax for the assignment statement is as follows. The variable that is being modified must be listed on the left-hand side of the equals sign (=) and surrounded by dollar signs (`$`). The variable name must be unique within each script, contain only alphanumeric characters and underscores, and cannot have the same value as any of the reserved words in the scripting language. The right-hand side expression can be any scripting expression, including expressions with mathematical and logical operations embedded in them, although it must end with a semicolon (`;`). The expression can also include variables that have been declared and have had values assigned to them.

The assignment statement (=, with one equals sign) should not be confused with the equality statement (==, with two equals signs). Since assignment statements require a semicolon at the end of the expression while equality statements do not, using an equality operator in place of assignment should cause the script parser to fail. However, there may be certain situations in which only the user's watchful eye will prevent a mistake.

When a variable is used, it is given a variable type. If the right-hand side expression is a Boolean the first time the variable is initialized, the variable will be type as a Boolean. Otherwise, if it is an integer, float, or string, the variable will be typed as an integer, float, or string (respectively). The most specific type that can be used for a variable will be used to type the variable. If a specific variable type is to be used, the right-hand side can be cast to the desired type using the cast (`:`) operation. When the variable is used, or if it is modified using an assignment statement one or more times, the variable will maintain the same type. Thus, it is imperative that the user cast the variable to the desired type when it is first assigned.

At this time, assignment variables are global in scope within the ConstructML attribute where they are created. This means that variables declared and first used in a loop can be accessed outside the loop. However, a variable declared in one ConstructML attribute cannot be directly used within another ConstructML attribute; a variable declared in a 'name' attribute will not be identical to that used in a 'value' attribute even if both have the same name. Additionally, variables initialized by one set of `with` variables are independent of those variables in use when another set of `with` variables are in use. Thus, if any of the with variables change, a new variable will be created. If the user wishes to refer to a variable in multiple contexts or locations, it will be necessary to use a construct `var` in order to do so.

When variables are first declared on the left-hand side of an assignment statement, they are often not recognized by the parser. This is because the parser is sensitive to unrecognized lexemes and (in most cases) is hesitant to cast bare words to strings. Thus, it is necessary to declare any variable as a `with` variable to ensure that the parser recognizes the variable and does not fail with an error. For this reason, the example `loop1` in Figure 12 declares the variable

Figure 15: Examples of `foreach` loops

Variable	Value

``
<var name="loop1" value="
   $result$ = '';
   foreach i ('a','b','c','d') {
       $result$ = $result$ + i;
   }
   return $result$;"
 with="$result$"/>
``
	"abcd"

``
<decision name="loop2" value="
    foreach col (0..10) {
        setKnowledgeNetwork[row:int,col:int,0]
    }"
 with="row=2"/>
``
	""
	(sets network values)

`result` to be a `with` variable, surrounded by dollar signs, so that the parser will not error on this value. Note that no value is assigned to $result$ in the `with` statement, which will prevent macro expansion of this variable when the script is parsed.

Note that when an assignment statement is in use, it will be necessary to include a `return` statement in the script in order to specify which ultimate result (or, more usually, which variable to return). If the end of a script is reached that does not contain a `return` statement, the parser will fail with an error. Rather than return the last statement executed, or return the last variable value assigned, a design decision was made to require that the user include a `return` statement to explicitly specify the value to return. This decision has the added advantage of generating parser errors if a return statement is omitted when a script is being debugged so as to point out places where a script is incomplete (thus hopefully speeding the debugging process).

Appendix B, part X.   Control Operations

- **foreach expression:** `foreach $iterator$ (<IterableExpr>)`
        `{ <Expr> }`

The `foreach` accumulator loop allows for an experiment designer to specify a list of operations to be repeated in order to create an aggregate result. This aggregate result is returned from the loop and can be used in subsequent computation.

To use the foreach loop, it is necessary to specify the keyword `foreach`, then specify the name of the iterator parameter for the loop, then specify the list of values over which to iterate and finally specify the statements to repeat in curly braces. The command must begin with the word foreach and then be followed by the name of the iterator parameter surrounded by dollar signs ($). This parameter can be any valid macro name that is not already in use. The list of variables over which to iterate is then specified in parentheses (()). These values are specified in a comma-separated list and will be treated as strings in the loop expression. Thus, an expression such as "`foreach $val$ (1,2,3)`" will create an iterator variable `val` which will be assigned the value 1 in the first iteration of the loop, 2 in the second, and 3 in the third. Note that this syntax and style is analogous to that found in the Perl language.

50

Following the closing parenthesis of the iteration parameters, the expression to iterate must be contained within curly braces (`{}`). Any sequence of statements can be placed inside the foreach loop, including statements which employ the iterator value as a macro variable or computation aid. Often, it is useful to have a variable declared outside of the loop, such as an accumulator variable `$result$` and then modify `$result$` using assignment statements in the loop. Loops may be used perform `setXXX` operations and thus may not need to have an accumulator. Multiple set or assignment statements can be executed in a single loop iteration.

Two examples of foreach loops can be seen in Figure 15. The first example, `loop1`, uses a foreach loop in order to incrementally create a more complex string from individual string characters. Initially, the accumulator `$result$` is initialized to the empty string. Since the initial value of the result variable is a string, it remains a string and is gradually increased in length as the loop is evaluated. The first value assigned to the iterator variable `$i$` is the character 'a'. This value is appended to the empty string in order to create the string 'a' during the first iteration of the loop. In the next iteration, the iterator variable `$i$` is 'b' and the appending operation creates result 'ab'. The process continues until, after four iterations, the string 'abcd' is created.

In the second example, `loop2` of Figure 15, a foreach loop is used as part of a decision to modify an existing network. The loop iterator variable, `col`, iterates over the values 0, 1, 2 … 10 as the iterator variables are specified using an enumeration. A `with` variable, `row`, is initialized outside the loop and is used to help specify the location to modify. Note that both the row variable, the with variable, and the column variable, the loop iterator, should be cast to integers to ensure that both values can be used as indices into the network. The result of executing this loop is that the first eleven knowledge values for the second agent are cleared (set to zero), since the knowledge network is an agent-by-knowledge network.

It is important to note that foreach loops are currently implemented by unrolling the list of statements embedded in the loop. This design decision has several implications. First, it means that a foreach loop containing a list of assignment statements, such as `loop1` in Figure 15, will be unrolled in order to create a list of four assignments to perform. While this unrolling can be fast for small loops, it may slow down the computation speed for large loops. Second, the unrolling of the loop allows the loop parameter to be used as a macro parameter in specifying variable names; thus, the value `$i$` in `loop1` could have been used to help specify the name of a variable reference. Most other variables that are modified at execution time, such as those to which values can be assigned, cannot be used in this fashion since they will not have a value when parsed. Third, it means that the iterator values must be defined before the loop is ever examined by the Construct parser. This means that the iterated values must either be constants or be specified by variables. Loops which iterate over a variable or modifiable number of values will be implemented in future versions of the scripting language.

Note that foreach loops can be embedded in more complex calculations. For instance, `foreach` loops can be nested inside other `foreach` loops, as long as the iterator variables are distinct in the two loops. However, the outer iterator variable can be used to define the inner foreach loop in such a situation; the iterator variable can even be used as part of a larger string expression when defining a variable reference. Additionally, the variables modified in a `foreach` loop can be further modified by statements that follow the loop. This means that a variable, such as an "`average`" variable, can be incremented in the loop and then, after the loop finishes, divided by the number of loop entries in order to create an average. Lastly, results

can be returned from the `foreach` loop using a `return` statement, which will break the calculation and specify a script result immediately. Such return statements will likely be embedded in an `if` statement to ensure that a value is returned from the loop only if a particular condition is met.

- **return:** `return <Expr>;`

The `return` statement allows for an arbitrary value (or the result of an arbitrary expression) to be returned from anywhere within a script. It is primarily intended for more complicated scripts, such as those which include assignment statements, `if` statements and `foreach` loops.

A return statement must begin with the six-character word "`return`". All values after the word return, up to a trailing semi-colon (`;`), will be considered part of the return expression. The return expression can be any valid Construct expression that can include both with and accumulator variables. Note that if a return statement occurs at the end of a script, the trailing semicolon is optional and all characters between the lexeme "`return`" and the end of the script will be considered part of the return expression. However, all return statements in the middle of loops and if statements must contain a trailing semicolon.

Due to the fact that Construct ignores spaces, any characters after the return statement will be considered part of the value to return; thus, variables with names like "`return_val`" will be (mis)interpreted by the scripting language as a statement to return the variable "`_val`" and thus should not be used. However, entire expressions can be placed after the `return` lexeme in order to specify the value to be returned. Thus, expressions like `return $count$+1;` will return the value of the variable count increased by one. Constants (e.g. `1` or `'abc'`), construct variable values (e.g. `construct::intvar::agent_count`), script variables (e.g. `$average$`), or expressions containing one or more of such elements can be used as parts of valid return expressions.

Return statements must be present in any script that contains assignment statements, `if` statements, or `foreach` loops. Since these expressions do not create a return value by default, it is necessary to supply a return statement in order so that the script returns a value. If an `if` statement or `foreach` loop is present and no `return` statement is provided at the end of the script, Construct should exit and error.

Note that values evaluated as part of an `expressionvar` (Section 2.5) are considered to be part of the script and are not considered separate. Instead, when the `expressionvar` is encountered, the code from the expression expansion is inserted directly the body of the `expressionvar` invoker. This means that any return statements in the `expressionvar` will serve as returns from the entire script. Thus, an `expressionvar` serves as a macro-like placeholder for complex code that need only be written once. Fully-fledged subroutines which calculate intermediate in support of a more complex script may be supported in future versions of Construct. At this time, however, they are not supported.

Appendix B, part XI.   Network Operations

- **get network value:** `get<NetworkName>[<RowExpr>,<ColExpr>]`

The get network value operation retrieves the value as a specific location in a network. The location to be retrieved is indexed by two integers, the row and the column. The return type of the value is the same as the type of the network – i.e. if the network is float network, such as the knowledge network, then the retrieved value will be returned as a float; if it is a string network, like many attribute networks, then the retrieved value will be returned as a string.

This command, when executed, requires the name of a network in order to execute. The name of the network is written in CamelCase following the initial `get`, meaning that the command `getKnowledgeNetwork` will result in an examination of the knowledge network, an agent-to-knowledge network of what facts are known by which agents. Because most network names have spaces in them, it is necessary to use this syntax in order for the parser clearly determine that what is being parsed is indeed the name of the network. The CamelCase syntax specifies that the first letter of each word be capitalized and remaining letters remain in lower case; no spaces are present in the word. The script parser will reconfigure the name appropriately in order to retrieve the specific network. If the network does not exist, Construct will exit with an error.

The indices into the network are specified in enclosing bracket characters (`[]`) and separated by a comma (`,`). The row and column indices retrieved must be integers and also must be valid for the particular matrix. If the values are not valid, Construct will exit with an appropriate error message.

Note that the network is re-examined each time that this command is called, meaning that if this command is called for in a decision then the decision will check the network every time period to examine whether or not there was a change in the underlying network. Such calls are relatively time consuming and thus should be minimized, but make it possible for decisions to check for updates to network properties.

This command cannot be used when initializing a construct variable, because the nodeclasses and networks have not yet been initialized. If a `get` command is called during the setup of a variable, or if the requested network cannot be found, then Construct will exit with an error.

- **aggregate network values:** `get<NetworkName>[`
        `<RowExpr>,<ColExprString>]`

The aggregate network value operation is shorthand combination of the get network value operation and the foreach operation. It will get a series of values from a particular network. The series of values are passed as a list of indices in the `ColExprString`, a string containing a series of comma-separated integers. These integers should be valid indices into the particular network in question. The value returned by this expression is of the same type as the underlying network which is to be retrieved

When this operation is evaluated, the specified network is first retrieved. Next, the row expression is evaluated and the column expression string is retrieved. The column expression string is then retrieved and parsed into a set of n indices. Following this, the n retrieved values – all of which are in the same row, but differ in column – are added together using the addition (+) operator.

It should be noted that the `ColExprString` is meant to be a nonempty, column-separated list of indices. This string can be generated in one of two ways. First, a direct list of values (created either by the concatenation operator or enumeration operators) can be used in order to build up a list of indices. If this is done though, it is best to place the comma-separated list

53

within parentheses so it appears visually as a sub expression rather than having more than one comma in the aggregate network values expression. Alternatively, one can refer to an agent group or fact group using the group reference syntax described earlier.

One of the most common ways in which this method is called is to get the network values for a specific agent group or fact group. For instance, if binary facts are in use, one can count the number of facts that agent `0` knows in the knowledge group `group`, one can write the expression `getKnowledgeNetwork[0,` `construct::knowledgegroupvar::group]`. This syntax is shorter, and arguably clearer, than embedding a series of `get` operations in a `foreach` loop.

- **get value from csv file:** `readFromCSVFile[` `<FileExpr>,<RowExpr>,<ColExpr>]`

The read from CSV file command opens a comma-separated file for reading and parsing. The location of the file must be specified relative to the location of the running directory; it is recommended that the file be located in the same folder as both the Construct executable and the input deck. If the CSV file is not present, or cannot be opened for reading, Construct will exit with an error.

The syntax of the operation is as follows. It must begin with `readFromCSVFile` (the syntax differs from the `get` operations which examine networks since it will eventually be possible to read from DynetML files as well), then in square brackets (`[]`) contain a comma-separated list with the name of the file to read, the integer index of the row, and the integer index of the column. If the file is invalid, the row is out of bounds, or the column is out of bounds, Construct will error and exit. Otherwise, the operation will return the value in the CSV file at that particular location.

Note that the rows in the CSV file do not have to have the same number of columns; unlike graphs, the columns in a CSV file can have varying number of values. However, if Construct attempts to read from a non-existent value, the simulation will exit and error.

The value returned from the CSV file is returned as a string. In order to cast the value to another type, it is necessary to use a cast operation (`:`).

Note that this operation is very time consuming, since it requires opening a CSV file, reading a single specific value, and then closing the file once the value is read. The use of this command should be minimized where possible for performance reasons; ideally, it should only be used in the initialization of Construct variables, and those variables then referred to in other expressions to avoid unnecessary opening and closing of files. Nevertheless, it is possible to call this command from the decision system or any applicable location in the input file in order to support dynamic updating of Construct inputs from files. This mechanism, for instance, can allow an external program to have a hook into an executing Construct process as long as the external program can write its output to a CSV file.

- **set network value:** `set<NetworkName>[` `<RowExpr>,<ColExpr>,<ValueExpr>]`

The set network value command allows a network value to be modified by the value expression. While this modification can be used to set up exogenous changes – for instance, to change a particular agent's knowledge at a predefined time period – it can also be used to allow

for extremely powerful emergent changes within the system: for instance, the experimenter can create a decision operation which allows an agent to interact with additional agents only when learning a particular piece of knowledge.

This command, when executed, requires the name of a network in order to execute. The name of the network is provided in CamelCase following the initial `get`, meaning that the command `setKnowledgeNetwork` will result in a modification of the knowledge network, an agent-to-knowledge network of what facts are known by which agents. Because most network names have spaces in them, it is necessary to use this syntax in order for the parser to be clear that what is being parsed is indeed the name of the network. The CamelCase syntax specifies that the first letter of each word be capitalized and remaining letters remain in lower case; no spaces are present in the word. The script parser will reconfigure the name appropriately in order to retrieve the specific network. If the network does not exist, Construct will exit with an error.

The indices into the network, as well as the value to set, are specified in enclosing bracket characters (`[]`) and separated by a comma (`,`). The row and column indices must be integers and also must be valid for the particular matrix. If the values are not valid, Construct will exit with an appropriate error message. The value expression must be the same type as expected by the network and is separated from the indices by a comma. Note that the set network command, unlike assignment statements, does *not* require a semicolon after the final bracket. Including a semicolon will create a parse error at this time.

It is important to verify that the `ValueExpr` should be of the same type as the network which is being modified. If the two are not of the same type, Construct will exit with an error. Construct will not try to explicitly cast from one type to another: for instance, Construct will not cast from an integer to a string, even though such a cast is legitimate, in order to force the user to reconsider the syntax and to check the script for errors.

While this command is a mutator, it also has a return value that can be used in mathematical expressions. The value returned is of the same type, and of the same value, as that of the `ValueExpr`. Thus, the return value of this scripting command is the same value as is set in the network upon success.

- ▪ **set aggregate network values:** `set<NetworkName>[`
    `<RowExpr>,<ColExprString>,<ValueExpr>]`

The set aggregate network values operation is a similar to the set network value operation, but uses the aggregation functionality that has been described for the get operation. The syntax for this operation is a hybrid of both approaches: the network to be modified is specified in CamelCase after the value `set` and then square brackets (`[]`) enclose a row, a list of column elements and then a value to set. The value of the value expression is then set at the specified locations of the network. Note that the set aggregate network values expression also does not require a semicolon after the final bracket.

While this command is a mutator, it also has a return value that can be used in mathematical expressions. The value returned is of the same type, and of the same value, as that of the `ValueExpr`. Thus, the return value of this scripting command is the same value as is set in the network upon success. However, the command is also an aggregation command, as it serves to set multiple network values simultaneously. For this reason, the value returned by the command is equivalent to the value obtained when applying the addition operator (+) to all of the newly set

55

Figure 16: Interpolation differences between `intvars` and `stringvars`

Variable	Value
`<var name="a" value="1"/>`	`"1"`
`<var name="b" value="construct::intvar::a+1"/>`	`"2"`
`<var name="c" value="construct::intvar::b+1"/>`	`"3"`
`<var name="d" value="construct::stringvar::a+1"/>`	`"1+1"`

values. Thus, if multiple string values are set in a network, the value returned would be the concatenated string containing all of the set values sequentially; alternatively, if the values were numeric, the value returned by the operation would be the sum. In most scripts, this value is usually not of interest, as the primary use of the set aggregate network values command will be to modify a second network; however, all operations must return a value and some experiment designers may find this return value helpful.

If the value expression `ValueExpr` references a non-static value – for instance, if it makes calls to the random number generator – then the value used will be recomputed for each of the values set. If one wishes to set the same random number value in each location, a `foreach` loop should be employed. For additional information on the use of random numbers in Construct scripting, see Appendix B, part VI.

Appendix B, part XII. Variable Reference Operations

▪ **direct variable reference:** `construct::<TypeExpr>::<VariableExpr>`

The variable reference syntax allows the scripting system to access some of the previously declared Construct data structures, by name and use that value when computing another value. Such variable references allow variables and decisions to be constructed based on simple expressions. The general syntax for such an expression is the word `construct`, followed by two colons `::`, followed by a type expression, followed by another two colons `::`, followed by the name of a variable.

For direct variable references, there are four possible values for `TypeExpr`: `boolvar`, `intvar`, `floatvar`, and `stringvar`, which cast the variable to a Boolean, integer, float, and string, respectively. Note that it is not necessary to type the variable when the variable is created, but only when it is used; this allows a variable to be used as an int in one context and a string in another. While this interchange between types may cause some problems that are not checked by the scripting language – for instance, Construct will not warn if one can attempt to use an arbitrary string as an integer – it can allow for additional power in some circumstances. For instance, in Figure 16, it is possible to use the value `b` both as an integer and a string.

Direct variable references can be chained; in Figure 16, variables `c` and `d` both refer to `VariableExpr` values which are themselves references. However, the variable addressed must be declared prior to use in a direct variable reference. Because the Construct input deck is parsed from top to bottom when configuring a Construct simulation, this means that any variable reference must occur below the variable declaration. This property is true recursively: if a direct variable reference points to another direct variable reference, the second direct variable reference must also be previously declared. This also guarantees that there are no self-loops.

Figure 17: Interpolation differences between `stringvars` and `expressionvars`

Variable	Value
`<var name="w" value="1+1" with="delay_interpolation"/>`	`"1+1"`
`<var name="x" value="construct::stringvar::w"/>`	`"1+1"`
`<var name="y" value="construct::expressionvar::w"/>`	`"2"`
`<var name="z"` `    value="(construct::expressionvar::w):string+'1'"/>`	`"21"`

Variable lookup is case sensitive. However, it is recommended as good programming practice that variable names not depend solely on differences in case.

Note that the direct variable reference operation uses a pair of colons (`::`) as a delimiter. If only one colon is provided, Construct will misinterpret the colon as a cast. This should result in a fatal error. Furthermore, both sets of colons are required. If one set of colons is missing, Construct will fail; the type must be explicitly specified each time any variable is used.

- **expression reference:** `construct::expressionvar::<VariableExpr>`

The behavior of the expression reference is different than the direct variable reference – specifically, the `stringvar` reference – in several ways. A `stringvar` reference will get the value of the previously defined variable as a string literal, and will not attempt to interpolate the value of the retrieved value. The `expressionvar` reference , on the other hand, will retrieve the value of the previous variable and then attempt to interpolate the variable using any additional parameters available in the script which invoked the expressionvar.

Several examples of `expressionvars` can be seen in Figure 17. As can be seen, variable `w` has been declared as a string whose value should not be interpolated at the time of declaration. The variable `x`, which uses the value of w as a string, will then be interpolated as the string `1+1`, which is the string value of what is being stored in w. The variable `y`, on the other hand, will treat the value in x as an expression and interpolate it, leading `y` to have the value `2`.

Note that the example in Figure 17 is trivial, since there is unlikely to be a reason to interpolate `w` immediately. However, the decision system may rely on parameters that are computed as the simulation is running (for instance, an agent's knowledge), and it may be useful to write expressions that perform such a computation with that value in the `construct_vars` in order to facilitate code reuse. The `expressionvar` syntax allows this reuse to occur and incurs no runtime penalty.

Note that the `VariableExpr` part of the expression must follow the same rules as for the direct variable expression: any referenced variable must have been previously declared. If an expressionvar is evaluated whose expression refers to a variable, nodeclass, or network which has not yet been created, Construct will output an error and exit.

The return type for an `expressionvar` will be the type resulting from the computation, and is dependent on the value of the other variable. In some cases, though, it may be useful to cast the type resulting from expression evaluation in order to ensure correct behavior in the script. For instance, variable `z` in Figure 17 relies on variable `w` in order to generate its result. However, `w`'s return value is an integer, since the value can be typed as an int. By explicitly casting the value to a string using the cast operation, the value can be converted to a string and then a string concatenation can be performed.

- **nodeclass reference:** `nodeclass::<TypeExpr>::<CountExpr>`

An alternative type of variable expression can be used find a property about a node class as opposed to returning a construct variable value. This allows the experimenter to quickly and efficiently determine the size of a particular node class – the number of nodes created for that node class and thus the number of rows or columns in any network involving that node class.

The `TypeExpr` value can be any valid node class that has been declared; if the node class has not been declared, the lookup will fail and Construct will exit with an error. Thus, it is not possible to use this syntax when creating construct variables, as they are created before any of the node classes are initialized; however, it may be useful to use this syntax when generating networks since the node classes are initialized before networks are generated.

The `CountExpr` can be either `count` or `count_minus_one`. The former is provided to determine the number of nodes in the node class, while the latter returns the index of the last element in the node class. The index of the first element is always zero (0).

Note that both `count` and `count_minus_one` return integer values; to convert these values into another type, it is necessary to use an explicit cast. Also note that the expression `count_minus_one` returns the same value as the mathematical expression `count-1`. The `count_minus_one` is provided to support legacy input decks.

- **membership reference:** `construct::<NodeExpr>groupvar::<NameExpr>`

A variable expression to retrieve the node indices of all values in a particular nodeclass membership group. The nodeclass membership group is specified in a nodeclass membership network (i.e. the `agent group membership network`). The indices are then concatenated together using the concatenation (`..`) operator in order to create a string of indices. This string is then returned.

The `<NodeExpr>groupvar` Section of this expression indicates the type of nodeclass group which is being retrieved. For instance, using `agentgroupvar` will cause the `agent group membership network` (an agent by agent group network) to be examined, while using `knowledgegroupvar` will cause the `knowledge group membership network` to be searched (a knowledge by knowledge group network). While these node sets are the only networks that are supported at this time, future versions of Construct will allow access to other types of nodeclass groups.

The nodeclass membership network is a Boolean network in the current implementation, meaning that each node is either wholly in or wholly out of a particular membership group. Thus, the list of indices returned contains only those agents which are wholly in the membership group. The values returned will be in increasing numeric order. For instance, if agents 3, 5, and 7 are in the agent group "group1", then a call to `construct::agentgroupvar::group1` will return the value `3,5,7`.

Note that the `NameExpr` must be the name of the particular group. If the name of the group contains internal spaces, it will be necessary to refer to the name via a Construct variable. Supplying either the direct group name or a variable that describes the direct group name is sufficient for the purposes of this expression.

Appendix B, part XIII. Other Operations

- **error:** `error(<StringExpr>)`

The `error` expression will cause Construct to output the string expression and exit when evaluated. This can help with debugging a macro, since an `error` expression can be used to test whether input to a particular function is valid. For instance, one can use an `if` statement to evaluate a certain type of input and if the input exceeds a specified range, an `else` statement can be used in order to have the program exit with an appropriate error message.

The current error implementation will only exit if the five characters `e-r-r-o-r` are encountered immediately prior to an open parenthesis (`(`). This implementation scheme was chosen for several reasons. First, it is possible to have variables or decisions named `error_name` and or `inadvertent_error`, as these strings will not be interpreted as names of error, as long as the characters after the word "error" are not open parenthesis. Second, in order to support extensions to the error system, it may allow for more meaningful subclasses of errors (such as an `IOError`) to be implemented in the future.

The string expression provided to the error lexeme will be used as the error message and will be returned to the user if the error is ever called. If no string is provided, a default message (`"<no error message provided>"`) will be return. Note that the string expression can be a quoted string, which is a constant, or a dynamically evaluated string which can provide additional information on the cause of the error.

- **macro variable expression:** `$<Name>$`

The macro variable expression may be both powerful and confusing for the user. When writing expressions containing macro variables, it is advisable to use comments within the expression as well as to use the `verbose` output mode when initially testing the expression results. However, the macro variable expression can significantly reduce the number of lines in a ConstructML file and can greatly enhance the maintainability of an input deck as it is being developed.

A macro is defined in two parts, as seen in the three examples given in Figure 18. First, within the actual text of the script, a case-sensitive macro identifier is placed between two dollar sign characters (`$`). This identifier should be limited to alphanumeric characters and the underscore character; it should not contain spaces. Secondly, the variable value should also be specified. For most variables, this specification will occur in the `with` tag of the variable, decision, or other enclosing part of the ConstructML file. The with tag must define the same case-sensitive macro identifier between dollar signs. The second dollar sign must be followed by an equals sign and then by the value of the particular variable, as seen in the first three examples of Figure 18.

The results of a macro expansion contribute directly to the text of the expression. Thus, in variable `x1` of Figure 18, the expansion of variable `$i$` is immediately converted to an integer value and then incremented. However, the substituted variable can also contribute to the creation of a new lexeme. For instance, the expansion of variable `x2` will substitute the value `1` for `$i$`, leading to the creation of the value `construct::intvar::x1`. When this expression is expanded and evaluated, the value of variable `x1` will be returned, leading `x2` to have the value

59

Figure 18: Examples of macros

Variable	Value
`<var name="x1″ value="$i$:int+1" with="$i$=1"/>`	`"2"`
`<var name="x2" value="construct::intvar::x$i$` `   with="$i$=1"/>`	`"2"`
`<var name="x3" value="$2*i:int$+1" with="$i$=1"/>`	`"3"`
`<var name="x$i$" value="$i$" with="$i$=(4,5)"/>`	`x4="4"` `x5="5"`

2. Thus, the macro system can be used to contribute to mathematical equations as well as to lead to the creation of new variables. The macro system, then, serves as a supplement to the script system and allows the script to proceed using whatever results occur from macro evaluation.

Note that it is also possible to perform simple calculations within the macro expansion; the macro variable does not have to be used immediately. For instance, it is possible to perform math operations on, cast, and modify the macro variable. The macros in variable `x3` of Figure 18 illustrate how macro variables can be used in internal expressions. In variable x3, the macro variable is converted to an int and then doubled before it used. Nevertheless, it is not possible to call a macro variable within a macro; since the dollar sign character both begins and ends a macro, a second dollar sign following an initial dollar sign will be treated as a macro-ending character and not as the beginning of a second, internal macro.

Most variables should be defined in the `with` tag of the ConstructML file. As can be seen in variables `x1`, `x2`, and `x3` in Figure 18, the value for the macro is specified in a with statement. The syntax for the with statement consists of a variable name surrounded by dollar signs (`$`), the equals sign (=) to indicate assignment, and then the value of the macro variable. The name of the macro variable must be the same as the macro variable specified in the script itself; if the name is different, Construct will exit with an error. The value of the macro variable must be written as a string but can be cast to any type in the expansion of the macro (see, for example, variable `x3`). To create multiple macro variables in the same with statement, specify them in a comma separated list. Note that a comma (`,`) and not a semicolon is used as the list separator between multiple values, a factor which differs from the assignment statement discussed earlier.

It is also possible to create a list of variables which should be iteratively applied to a macro, an example of which can be seen in variable `x4` in Figure 18. This is done by supplying a list of variables after the equals sign (=) of a variable assignment. The macro will then be expanded once for each value of the macro. Note that the list of variables must be isolated using parentheses to differentiate them from the other `with` variables. Also note that it is possible to have multiple macros variables which have separate variable lists; if this is the case, then one macro expansion will be performed for each set of variables. This means, for instance, if there are three possible values for macro variable `$i$` and four for macro variable `$j$`, then twelve different expansions will be performed. Note that no semicolon is needed after the equals sign, unlike the assignment variable.

However, not all values surrounded by dollar signs are macro variables. For instance, some variables may be used as assignment variables in the script, meaning that the user will wish to assign values to them and return them at the conclusion of the script. These variables also begin and end with dollar signs (`$`), but will be modified dynamically as the script is evaluated. Such variables are usually specified as `with` parameters to ensure that they are recognized by the

Construct system. However, since they are used as arguments to assignment statements, these variables are not constant and will be modified during the execution of the script. While such variables have the same initialization syntax as macro variables, macro variables are defined statically and can be embedded in more complex names; the variables which are defined dynamically cannot be used in such a fashion.

Macro expansions can be useful in creating concise variable and decision names. Any variable defined in the `with` tag will be available when both the name and the value are being interpolated. As can be seen in Figure 18, the last example contains a variable which has a macro in both its name and its value. The expansion of this macro over both of the associated with values ends up generating two separate variables: the variable `x4` has value `4`, while the variable `x5` has value `5`. These variables can then be referred to in the standard way. No variable with name `x$i$` will ever be created in the system and attempting to reference a variable of that name will result in an error.

- **other expression:** `[A-Z][a-z]`

If a lexeme begins with a letter and does not match one of the pre-defined reserved words, it is passed to a lexer handler for parsing. This lexer handler is defined by the programmer who built the Construct binary, and can allow for forward compatibility with new types of user-defined decisions. Users may specify that certain words are to be treated as specified values.

At its core, the decision system discussed in Section 3 is actually a powerful lexer handler, the `decision lexer handler`. For instance, the decision lexer handler will take the lexeme `agent` and map it to the current agent being processed when the lexer is evaluated. At compile time, the handler has been defined to map this lexeme to the currently evaluated agent. Thus, the handler – and not the lexer – understands and interprets the lexeme `agent`; using the lexeme `agent` when a different handler is active will likely have no effect. If the lexeme is misspelled when passed to the correct handler, it will not be recognized and an error will occur. Most handlers will cause an error if an inappropriate lexeme is provided in order to assist with debugging, though this behavior may be determined by the handler designer.

By default, the lexer handler that is in use is a `silent lexer handler`. This handler will take any unrecognized lexeme and convert it into a string literal then return it to the lexer for additional parsing. Thus, it will treat any unrecognized lexeme identically to a quoted (`''`) string. This handler, for instance, is used as the `construct_variables` are loaded (Section 2); verbose output from the lexer during this process will show the silent lexer handler converting all new variables into strings. Other handlers can be specified for different code sections. For instance, the `decision lexer handler` is a lexer handler that is specific to the decision system. The decision lexer handler will recognize certain lexemes, such as the lexeme `agent`, and specify that these lexemes have specific behavior. Unrecognized lexemes will cause Construct to exit with an error. Other lexer handlers can be specified, though it is necessary to modify the Construct code (as opposed to the input deck) in order to specify both their specific lexemes and their behaviors.

References

Carley, K. (1986). "An Approach for Relating Social Structure to Cognitive Structure." Journal of Mathematical Sociology **12**(2): 137-189.

Carley, K. (1991). "A Theory of Group Stability." American Sociology Review **56**(3): 331-354.

Carley, K. (1997). "Organizational Adaptation." Annals of Operations Research **75**: 25-47.

Carley, K. (1999). "On the Evolution of Social and Organizational Networks." Special Issue of Research in the Sociology of Organizations on Networks In and Around Organizations: 3-30.

Carley, K., M. Martin, et al. (2009). "The Etiology of Social Change." Topics in Cognitive Science **1**(3).

Carley, K. and J. Reminga (2004). ORA: Organizational Risk Analyzer. Technical Report number CMU-ISRI-04-101. Pittsburgh, PA, Carnegie Mellon University School of Computer Science.

Hirshman, B. and K. Carley (2007). Specifying Agents in Construct. Technical Report number CMU-ISRI-07-107. Pittsburgh, PA, Carnegie Mellon University School of Computer Science.

Hirshman, B. and K. Carley (2007). Specifying Networks in Construct. Technical Report number CMU-ISRI-07-116. Pittsburgh, PA, Carnegie Mellon University School of Computer Science.

Hirshman, B. and K. Carley (2008). Modeling Information Access in Construct. Technical Report number CMU-ISR-08-114. Pittsburgh, PA, Carnegie Mellon University School of Computer Science.

Hirshman, B., M. Kowalchuck, et al. (2009). Core Algorithms in Construct. Carnegie Mellon University School of Computer Science, Carnegie Mellon University.

Hirshman, B., M. Martin, et al. (2008). The Impact of Educational Interventions on Real & Stylized Cities. Technical Report number CMU-ISR-08-115. Pittsburgh, PA, Carnegie Mellon University School of Computer Science.

Hirshman, B. and J. St. Charles (2009). Simulating Emergent Multi-Tierd Social Ties. Proceedings of the 2009 Human Behavior and Computational Intelligence Modeling Conference. Oak Ridge National Laboratory, TN.