Causal Compiler Syntax

# RULES Block

All of your causal relations should be defined within a single RULES block.

Ex. 0

RULES {

/\* YOUR RELATIONS HERE \*/

}

# Causal Relations

A causal relation can be broken into an action implied by a set of other actions. For example, in Ex.1, moveTo is implied by a grasp and release. It should be noted that all causal relations should end with a “;” EXCEPT for the last causal relation defined.

Ex. 1

moveTo(obj, dest, dx, dy, dz, da) := grasp(obj), release(obj, dest, dx, dy, dz, da);

#### Actions

Each action in a causal relation has a set of parameters which allows the user to define certain characteristics of actions. In the following example, the action moveTo is defined as moving an object with name obj to a destination dest with destination coordinates and angle dx, dy, dz, and da.

Ex. 2

moveTo(obj, dest, dx, dy, dz, da)

#### Action Implications

Each causal relation defines an action as *implied* by a set of other actions. Implication is shown with the := operator. It should be noted that the arguments to the action on the left of the := should be some subset of the set of arguments in all actions to the right of the :=. There can be any number (> 0) actions to the right of the :=, separated by commas as shown in Ex.1.

Ex. 3

IMPLIED\_ACTION(PARAMETERS) := ACTION0(PARAMETERS),ACTION1(PARAMETERS);

# Conditional Causal Relations

Some causal relations may be defined as valid only under some pre-defined condition. This conditional will be defined with an if-block. Each if-block starts on its own line with the conditional statement. An indented causal relation will be placed on the next line. See example 4. Indentation and newlines are not actually vital to compilation but they will make the code more readable!

Ex. 4

if (CONDITIONAL\_STATEMENT):

CAUSAL\_RELATION

#### Conditionals

Conditionals are implemented similar to many languages and may be chained together as such using the && (logical AND) and || (logical OR) operators. Parentheses may also be used to group certain conditionals together.

Ex. 5: The causal relation will only hold if CONDITION0 and CONDITION1 hold.

if (CONDITION0 && CONDITION1):

CAUSAL\_RELATION

It should be noted that using the || (logical OR) operator is essentially equivalent to writing multiple conditional statements. For example, Ex.6 and Ex.7 pictured below are equivalent.

Ex. 6: The causal relation will only hold if CONDITION0 or CONDITION1 hold.

if (CONDITION0 || CONDITION1):

CAUSAL\_RELATION

Ex. 7: The causal relation will only hold if CONDITION0 or CONDITION1 hold.

if (CONDITION0):

CAUSAL\_RELATION;

If (CONDITION1):

CAUSAL\_RELATION

Similarly due to the distributive property, Ex.8 and Ex.9 are also equivalent.

Ex. 8: The causal relation will only hold if CONDITION0 or CONDITION1 and CONDITION2 hold.

if ((CONDITION0 || CONDITION1)&& CONDITION2):

CAUSAL\_RELATION

Ex. 9: The causal relation will only hold if CONDITION0 or CONDITION1 and CONDITION2 hold.

if (CONDITION0 && CONDITION2):

CAUSAL\_RELATION;

If (CONDITION1 && CONDITION2):

CAUSAL\_RELATION

#### Types of Conditionals

There are six types of comparisons that a user may use in a conditional statement. The variables used in these conditionals must be present in the parameters of the actions used in the associated CAUSAL\_RELATION.

1. **Variable Comparison**Compare two variables to each other. This is an identity comparison of the two variables. In the following comparison, if the object referenced by obj is the same object referenced by obj1 then the comparison is true.  
     
   Ex. 10  
    obj = obj1

It should be noted that transitivity does not necessarily apply as one might think it would. For example, writing the following:

obj = obj1 && obj1 = ‘hello’

will not necessarily enforce that obj is equal to ‘hello’ in all instances. Luckily, examples such as this can be easily fixed by writing them as such:

obj = ‘hello’ && obj1 = ‘hello’

1. **Literal Comparison**Similar to the variable comparison, the literal comparison compares a variable to a string literal. In the following comparison, if the string referenced by dest is equal to ‘room’ then the comparison is true.  
     
   Ex. 11  
    dest = ‘room’
2. **TYPE Operator**Get the TYPE of an object. This can then used as a comparison to some other value or another TYPE operator. The following example is a comparison checking if the TYPE of the object referenced by obj is block. The type is accessing a field defined in the object XML. For example, you might define a custom XML object for SMILE with an id = ‘custom\_shape’ and a type = ‘block.’ Using this operator is dependent on the type being stored in a specific index of the state array. If adjustments are made to the format of the state array, this may get a different value that is not type.  
     
   Ex. 12  
    TYPE(obj) = block
3. **ALL Operator**Get a list of all objects with a given TYPE. This can then be used as a comparison to a list of objects (See the section on Lists below). The following example is a comparison checking if obj1 and obj2 reference the only objects of TYPE block.  
     
   Ex. 13  
    ALL(block) = [obj1, obj2]
4. **STATE Operator**Get a value in the CO-PCT stage of a value in the state array. Specifically, the STATE keyword takes a list of n arguments, which I’ll denote as a1, …, an. The value the statement holds will be state[a1][a2][a3]…[an]. This can be used as a placeholder for a variable, which means it can be used in both conditionals and as arguments. This keyword can be helpful for accessing values stored in the array that no other keywords give you access to. This functionality will only affect the CO-PCT stage; any conditionals containing it will be disregarded for imitation and any arguments in the intention will be replaced with placeholders in the imitation stage. When possible, other keywords should be used as this keyword requires intimate knowledge of the format of the state array in Python; it is only intended to be used in those instances when the syntax of the language cannot encapsulate the user’s desires. The following two examples show two different use cases of the keyword.  
     
   Ex. 14  
    TYPE(STATE(‘gripping’, 1)) == block   
    move-object(STATE(4, ‘arm’, ‘left’), obj) := grasp(obj)
5. **Python Operator**Allows user to inline Python code into CO-PCT stage. This can be used as both a conditional or an argument, however it has not been significantly tested as an argument. In general, users should avoid using the PYTHON keyword. Similar to STATE, it is intended to allow the user to write code that cannot quite be captured in the custom language. It is probably best in this instance to manually edit the generated scripts, however this is an option for the less programming-inclined user. Syntax notes: the beginning and end of the inlined Python statement should be marked with a single “#.” Any variable that is referenced in the inlined Python statement should be surrounded by “$”’s. PLEASE NOTE: This keyword is meant to be a last ditch effort to achieve desired results. It should be used sparingly and only when all other aspects of the language fail to meet the required output. The example below is one where the inlined Python code is performing a complex computation of a given variable that could not be done in the custom language.

Ex. 15  
 PYTHON(#if (sum([x\*\*2 for (x,) in $dt$])\*\*0.5) > 1:#)

# Peripheral Syntax Features

1. **White Space**

The language is entirely whitespace agnostic.  
Ex. 16  
 ACTION0 (PARAMETERS)  
is equivalent to  
 ACTION0(PARAMETERS)

1. **Lists**Lists are implemented as comma-separated variables wrapped by brackets. They are used in conjunction with the ALL operator. See example 9 for typical usage.
2. **CONT# Operator**The CONT# operator can be used to continue a list of objects. The following example represents checking that the list of ALL objects of TYPE block does in fact contain obj1.  
     
   Ex. 17  
    ALL(block) = [obj1, CONT1]  
     
   A given CONT# operator must be fed the correct information via the # part of the operator. For the above example, the number 1 is passed as the list is a continuation from obj1. In the next example, we show a continuation from obj1 and obj2 once more but this time within the set of parameters of an action stack. This can be interpreted as a “continuation” on obj1 and obj2 but not on the rest of the parameters. The reason for this distinction is so that both obj1 and obj2 can be referenced in conditionals. If you did not need to use obj2 in a conditional, then Ex.12 would be equivalent to Ex.13.  
     
   Ex. 18  
    stack(dest, dx, dy, dz, da, obj1, obj2, CONT2)

Ex. 19  
 stack(dest, dx, dy, dz, da, obj1, CONT1)

While subtle, proper use of the CONT# operator is important. Improper use can lead to indexing errors in the outputted Python scripts if the script attempts to access an element of an array that does not exist. I.e. For a causal relationship involving the action from Ex.12, the stack action need only have one obj in the final list, despite being written with an explicit obj2 argument. This would cause an indexing error if not for the 2 in CONT2.

# Complete Examples

The following example Ex.20 describes the causal relations that define stacking blocks. The floats passed to the stack action in the third causal relation define the relative position of the blocks (dx, dy, dz, and da). As one can see, the only variation in the blocks relative position is in the dz coordinate, which makes sense because they are all stacked up at the same x, y coordinates. In the following example, one should imagine that the lowest level actions (the leaves of the Causal Knowledge Tree) are grasp and release. All other actions are built from these base actions and other actions defined lower on the tree.

Ex. 20

RULES {

move-to(obj, dest, dx, dy, dz, da) := grasp(obj), release(obj, dest, dx, dy, dz, da);

if (TYPE(obj)=block):

stack(dest, dx, dy, dz, da, obj) := move-to(obj, dest, dx, dy, dz, da);

if (TYPE(obj1) = block && obj = obj1):

stack(dest, dx, dy, dz, da, obj1, obj2, obj3, CONT3) := move-to(obj, dest, dx, dy, dz, da), stack(obj1, 0, 0, 0.5, 0, obj2, obj3, CONT2);

if (ALL(block)=[obj1, CONT1] && dest = 'room'):

stack-all(dx, dy, dz, da) := stack(dest, dx, dy, dz, da, obj1, CONT1)

}

The following example Ex.21 is a far more complicated example. It represents the causal knowledge used in the set of tests run by Garrett Katz in testing CO-PCT for the experiments with the Baxter robot. You can see this at work at <https://github.com/jhomble/electron435/tree/master/python_causal_compiler/acceptance_test_2_results>.

Ex. 21

RULES {

if ((TYPE(obj) != DockCase && TYPE(obj) != DockDrawer)&&arm=1):

move\_unobstructed\_object(obj, 'left', NONE, NONE) := move\_arm\_and\_grasp(arm, obj);

if ((TYPE(obj) != DockCase && TYPE(obj) != DockDrawer)&&arm=2):

move\_unobstructed\_object(obj, 'right', NONE, NONE) := move\_arm\_and\_grasp(arm, obj);

if (arm=1):

move\_unobstructed\_object(STATE('gripping', 0), dest, dM, dt) := put\_down\_grasped\_object(arm, dest, dM, dt);

if (arm=2):

move\_unobstructed\_object(STATE('gripping', 1), dest, dM, dt) := put\_down\_grasped\_object(arm, dest, dM, dt);

if ((TYPE(dest)=DockCase || dest='discard-bin') && (dest != 'left' && dest != 'right')):

move\_unobstructed\_object\_to\_free\_spot(obj, dest) := move\_unobstructed\_object(obj, dest, dM, dt);

move\_object(obj, dest, dM, dt) := move\_unobstructed\_object(obj, dest, dM, dt);

if (dest != 'left' && dest != 'right' && (dest = 'dock\_case\_6' || dest = 'discard-bin')):

move\_object\_to\_free\_spot(obj, dest) := move\_object(obj, dest, dM, dt);

if (dest = 'discard-bin'):

discard\_object(obj) := move\_object\_to\_free\_spot(obj, dest);

if (TYPE(STATE('gripping', 0)) != DockCase && TYPE(STATE('gripping', 0)) != DockDrawer && arm1 = 1):

put\_down\_grasped\_object(arm1, dest, dM, dt) := move\_grasped\_object(arm1, dest, dM, dt), release(arm2);

if (TYPE(STATE('gripping', 1)) != DockCase && TYPE(STATE('gripping', 1)) != DockDrawer && arm1 = 2):

put\_down\_grasped\_object(arm1, dest, dM, dt) := move\_grasped\_object(arm1, dest, dM, dt), release(arm2);

if (arm0 = arm1 && TYPE(obj) != DockDrawer):

move\_unobstructed\_object(obj, dest, dM, dt) := move\_arm\_and\_grasp(arm0, obj), put\_down\_grasped\_object(arm1, dest, dM, dt);

if (arm0 = arm1 && arm1 = arm2 && TYPE(obj) = DockDrawer && PYTHON(#if (sum([x\*\*2 for (x,) in $dt$])\*\*0.5) > 1:#)):

open\_dock\_drawer(obj, PYTHON(#states[2]#)) := move\_arm\_and\_grasp(arm0, obj), move\_grasped\_object(arm1, dest, dM, dt), release(arm2);

if (arm0 = arm1 && arm1 = arm2 && TYPE(obj) = DockDrawer && PYTHON(#if (sum([x\*\*2 for (x,) in $dt$])\*\*0.5) <= 1:#)):

close\_dock\_drawer(obj) := move\_arm\_and\_grasp(arm0, obj), move\_grasped\_object(arm1, dest, dM, dt), release(arm2)

}