# The Syntax of CycL

CycL is a formal language whose syntax derives from first-order predicate calculus (the language of formal logic) and from Lisp. In order to express common sense knowledge, however, it goes far beyond first order logic. The vocabulary of CycL consists of terms. The set of terms can be divided into constants, non-atomic terms (NATs), variables, and a few other types of objects. Terms are combined into meaningful CycL expressions, which are used to make assertions in the CYC® knowledge base.

# Constants

Constants are the "vocabulary words" of the CYC® knowledge base. The CYC® KB is an attempt to model the world as most sane, adult humans perceive it, so each constant stands for some thing or concept in the world that we think many people know about and/or that most could understand.

The KB contains constants that denote collections of other concepts, such as #$AnimalWalkingProcess (the set of all actions in which some animal walks) or #$Typewriter (the set of all typewriters).

It can have constants that denote individual things, some of which are more-or-less permanently in the KB, like #$InternalRevenueService, and some of which might get created only when reasoning about some state of affairs, like #$Walking00036 (a particular case of walking).

Some of the individuals represented in the KB are predicates, such as #$isa or #$likesAsFriend, that allow one to express relationships among constants. Others are functions, such as #$GovernmentFn, which can take constants or other things as arguments, and generate new concepts (i.e., (#$GovernmentOf #$Canada)).

Each constant has its own data structure in the KB, consisting of the constant and the assertions which describe it.

# Constant Names

Most CYC® constants have a unique name, such as #$BillJ, #$massOfObject, or #$MapleTree.

CYC® constants are referred to with the prefix "#$“. These characters are sometimes omitted in documents describing CycL, and they may be omitted by certain interface tools.

# Naming Conventions

The name of a CYC® constant - the part after the "#$" prefix - must follow these rules:

All CYC® constant names must be at least 2 characters long (not including the #$ prefix).

Constant names can include any uppercase or lowercase letter, any digit, and the symbols "-", "_", and "?". No other characters, such as "!", "&", or "@" are allowed. This policy is enforced in the CYC® Functional Interface and in the CYC® Web Interface.

CYC® constant names are case-sensitive: #$foo is not the same as #$Foo. However, distinguishing two constant names solely on the basis of capitalization is prohibited by the system.

All CYC® predicate names must begin with a lowercase character. (This does not include all the things that are presently instances of #$Predicate in CYC®. Some of these latter things are more like functions, and their names begin with uppercase letters).

All non-predicate constant names must begin with an uppercase character. Non-predicate constant names may also begin with a numeric character (e.g., #$3MCorporation). We may also allow predicates to begin with numeric characters, if someone makes a compelling argument for why this should be allowed.

All CYC® constant names should be composed of one or more meaningful "words" in sequence, with no breaks except for dashes or underlines (e.g. #$isa and #$SportsCar). A sequence of numeric characters may count as a "word" (e.g., #$FrontOfficeOf123Corp). With the exception noted above for predicate names, each (non-numeric) "word" in a sequence must begin with a capital letter.

Hyphens are used to set off parts of names which restrict or refine the meaning of the name, as in #$Fruit-TheWord or #$Horse-Domesticated.

# Naming Strategies

In general, it's best to give related constants names which are alphabetically proximal. Some of our interface tools make it easy to search for all constants whose name begins with a certain string of characters, and it's easier to find all constants having to do with horses if they have been given names like #$Horse-Domesticated and #$Horse-Wild than if they have been given names like "DomesticatedHorse" and "WildHorse".

When naming a constant, it's important to assign a name that distinguishes the denoted concept from other concepts it might get confused with. So "Bow" would be a terrible name for a constant. Instead, names like "Bow-BoatPart", "BowTheWeapon", "Bowing-BodyMovement" should be used, depending on the underlying concept denoted.

Sometimes it is possible to take this principle of specificity in names to an extreme, and attempt to embody the whole meaning of the constant in its name. This is discouraged. For example, one might be tempted to give the constant #$physicalParts the name "distinctIdentifiablePhysicalParts", but it is better to leave the name a bit terser since it isn't easily confused with some other concept, and put the additional information in the constant documentation.

# Variables

Quantified CycL expressions (discussed below) contain one or more variables which stand for constants whose identities are not specified. A variable may appear (nearly) anywhere a constant can appear.

# Variable Names

Variable names must begin with a question mark and are ordinarily written in capital letters ("?FOO"). Variable names are subject to the same restrictions on usable characters as constant names.

# Naming Conventions

In formulas in which only one variable is used, it is common to use a single-letter variable, such as "?X". However, when a formula contains more than one variable, it will be much more readable if you give the variables mnemonic names. Here's an example:

```
(#$implies
    (#$and
      (#$isa ?TRANSFER #$TransferringPossession)
      (#$fromPossessor ?TRANSFER ?FROM))
    (#$isa ?FROM #$SocialBeing))
```

"The initial possessor in a possession transfer is a social being."

# Formulas

CycL formulas combine terms into meaningful expressions.

Every formula has the structure of a Lisp list. It is enclosed in parentheses, and consists of a list of objects which are commonly designated ARG0, ARG1, ARG2, etc.

The object in the ARG0 position may be a predicate, a logical connective, or a quantifier. The remaining arguments may be atomic constants, non-atomic terms, variables, numbers, English strings delimited by double quotes ("), or other formulas.

# #$CycFormula

This is the class of well-formed formulas in CycL. If a CycL formula satisfies all the constraints on the number and types of arguments to the relations that appear in it, the system will recognize it as an instance of the collection #$CycFormula.

# Atomic Formulas

The simplest kind of formula is an atomic formula: a formula in which the ARG0 position is occupied by a predicate, and all the other argument positions are filled with terms:

(#$likesAsFriend #$DougLenat #$KeithGoolsbey)

(#$skillCapableOf #$LinusVanPelt
#$PlayingAMusicalInstrument #$performedBy)

(#$colorOfObject ?CAR ?COLOR)

The first two of the atomic formulas above are ground atomic formulas (GAFs), since none of the terms filling the argument positions ARG1, ARG2, etc. are variables.

# Predicates

Every CycL atomic formula must begin with a predicate in order to be well-formed.

# Predicate Arity

Most predicates are defined to take a fixed number of arguments. There are no optional predicate arguments in CycL. However, a few predicates, such as #$different, can take a variable number of arguments. Such predicates are elements of the collection #$VariableArityRelation.

In most cases, arity is automatically inferred by CYC® when a relation or predicate is made an instance of a certain type of collection (e.g. #$BinaryPredicate). However, arity can also be asserted directly, via the binary predicate #$arity.

The number of arguments a predicate takes is determined by its arity. A predicate is described as unary, binary, ternary, quaternary, or quintary, according to whether it takes 1, 2, 3, 4, or 5 arguments. Currently, no CycL predicate takes more than 5 arguments; however, if some representation required a predicate to take more arguments, CycL would be changed to allow this.

To be well-formed, an atomic formula must have the right number of arguments for the predicate filling the ARG0 position. So,

(#$likesAsFriend #$DougLenat
#$KeithGoolsbey #$Fido)

is not well-formed, since the arity of #$likesAsFriend is 2, but this formula gives 3 arguments to #$likesAsFriend.

# Predicate Argument Types

The type of each argument must be specified in the definition of the predicate, using the predicates #$arg1Isa, #$arg2Isa, etc. For example, suppose the predicate #$residesInDwelling is defined by the following:

(#$isa #$residesInDwelling #$BinaryPredicate)

(#$arg1Isa #$residesInDwelling #$Animal)

(#$arg2Isa #$residesInDwelling #$ShelterConstruction)

To be well-formed, every formula which has #$residesInDwelling in the ARGO position must have a term which is an instance of #$Animal in the ARG1 position, and term which is an instance of #$ShelterConstruction in the ARG2 position.

So,

(#$residesInDwelling #$PottedPlant37 #$KarensHouse)

is probably not well-formed. Though we can never be absolutely certain just from the names, #$KarensHouse could be an instance of #$ShelterConstruction, but #$PottedPlant37 is probably not an instance of #$Animal.

# Logical Connectives

Complex formulas can be built up out of atomic formulas or other complex formulas by using logical connectives, which are special constants analogous to the logical operators of formal logic. The most important logical connectives in CycL are #$not, #$and, #$or, and #$implies.

# #$not

The connective #$not takes a single formula as an argument. Like the "not" of formal logic, it reverses the truth value of its argument.

Thus,

(#$not (#$colorOfObject #$FredsBike #$RedColor))

will be true if and only if

(#$colorOfObject #$FredsBike #$RedColor)

is false.

Likewise,

(#$not (#$not (#$colorOfObject #$FredsBike
#$RedColor)))

will have the same truth value as

(#$colorOfObject #$FredsBike #$RedColor)

# #$and

The connective #$and takes one or more formulas as arguments. Like the "and" of formal logic, it returns true if and only if each of its arguments evaluates to true.

# #$or

The connective #$or takes one or more formulas as arguments. Like the "or" of formal logic, it returns true if and only if at least one of its arguments evaluates to true.

# #$implies

The connective #$implies takes exactly two formulas as arguments. Like the "if-then" statement of formal logic, it returns true if and only if it is not the case that its first argument is true and its second argument is false.

Here's an example:

```
(#$implies   (#$owns #$Fred #$Bike001)
             (#$colorOfObject #$Bike001 #$RedColor))
```

This assertion states that if #$Bike001 is owned by #$Fred, then it is red. Newcomers to formal logic may misinterpret #$implies as implying a causal relationship. But, strictly speaking, a #$implies assertion says only that either the first argument is false, or the second argument is true.

So, for example, the assertion

```
(#$implies
    (#$isa #$RichardNixon #$Fruit)
    (#$colorOfObject #$BillJ #$PastelMintGreen))
```

is true, because the first argument is false.

Assertions involving #$implies are very common in the CYC® KB. We also call them conditionals or rules, and we often refer to the first argument as the antecedent and the second argument as the consequent. Note, however, that the particular formula above is *not* representative of assertions likely to be found in the CYC® KB. We will come to some more representative examples in a moment.

# Well-Formedness of Complex Formulas

Any complex formula formed by using the logical connectives will be well-formed if the formulas given as arguments to the connectives are also well-formed and if the right number of arguments are given. Another way of saying this is that #$not, #$and, #$or and #$implies produce CycFormulas when they are given arguments which are also CycFormulas.

Suppose A and B are syntactically legal, and C is not. Then,

(#$not A)

(#$and A)

(#$and A B)

(#$or A)

(#$or A B)

(#$implies A B)

would all be CycFormulas.

But

(#$not A B)

(#$and)

(#$and A C)

(#$implies A)

would NOT be CycFormulas.

It should also be noted that #$and and #$or are elements of #$VariableArityRelation.

# Quantification

So far, we have only looked at ways to make statements about specific objects, like #$FredsBike. But CycL, like first-order predicate calculus, also gives us two ways to talk about objects without being specific about the identity of the objects involved: universal quantification and existential quantification.

Universal quantification corresponds to English expressions like *every*, *all*, *always*, *everyone*, and *anything*, while existential quantification corresponds to English expressions like *someone*, *something*, and *somewhere*.

CycL contains one universal quantifier, #$forAll, and four existential quantifiers, #$thereExists, #$thereExistAtLeast, #$thereExistAtMost, and #$thereExistExactly.

# forAll

The quantifier #$forAll takes two arguments, a variable and a formula in which the variable appears. In practice, the formula is almost always a conditional in which the antecedent is used to restrict the scope of the variable.

Here's an example:

```
(#$forAll ?X
    (#$implies
        (#$owns #$Fred ?X)
        (#$objectFoundInLocation ?X #$FredsHouse)))
```

This formula states that it is true, concerning every object in the CYC® ontology, that if #$Fred owns that object, then that object is located in #$FredsHouse. In other words, all Fred's stuff is in his house.

# Multiple Quantification

Formulas may contain more than one quantifier, as in the following:

```
(#$forAll ?X
  (#$forAll ?Y
    (#$implies
      (#$and
        (#$owns #$Fred ?X)
        (#$owns #$Fred ?Y))
      (#$near ?X ?Y))))
```

which says that any two things owned by Fred are near each other. Note that each quantifier introduces a new variable, and that each variable must have a different name.

# Unbound Variables

Normally, variables need to be introduced ("bound") by a quantifier before they are used. Each quantifier binds exactly one variable, and every variable used should be bound by exactly one quantifier. Furthermore, a variable has no meaning outside the scope of the quantifier which binds it.

However, if a unbound variable appears in a CycL formula, it is always assumed to be universally quantified, with the result that

```
(#$implies
   (#$owns #$Fred ?X)
   (#$objectFoundInLocation ?X #$FredsHouse))
```

is exactly equivalent to

```
(#$forAll ?X
    (#$implies
        (#$owns #$Fred ?X)
        (#$objectFoundInLocation ?X #$FredsHouse)))
```

Since the former is easier to write and read, it is almost always preferred in practice, and you will rarely see a #$forAll while browsing the CYC® KB.

Note, however, that unbound variables which appear only in the consequent of a conditional, and not in the antecedent, may have drastic and undesired consequences. Take, for example, the following:

```
(#$implies
    (#$owns #$Fred ?WHATEBER)
    (#$objectFoundInLocation ?WHATEVER #$FredsHouse))
```

Because of the typo, the variable ?WHATEVER will range over the entire CYC® ontology. In other words, the assertion above states that as long as Fred owns one thing, *everything* is located in #$FredsHouse-probably not what we wanted.

# #$thereExists

The quantifier #$thereExists takes two arguments, a variable and a formula in which the variable appears. In practice one uses #$thereExists only in certain ways, of which the following is a good example:

```
(#$implies
      (#$isa ?A #$Animal)
      (#$thereExists ?M
            (#$mother ?A ?M)))
```

This assertion states that, for every animal, there exists at least one object which is that animal's mother. The object which is the animal's mother may be an object which is already represented by a CYC® constant, or it may be a new object of which CYC® has no knowledge. But unless and until it is told otherwise, CYC® will assume that the object is a new one not identical with any "known" object.

# #$thereExistExactly, #$thereExistAtLeast, #$thereExistAtMost

These three quantifiers are similar to #$thereExists, but provide greater quantitative expressiveness. Each of them takes three arguments: a positive integer, a variable, and a formula in which the variable appears. Their meaning should be fairly self-explanatory. Look at the following examples:

```
(#$implies
      (#$isa ?P #$Person)
      (#$thereExistExactly 2  ?LEG
            (#$and
                  (#$isa ?LEG #$Leg)
                  (#$anatomicalParts ?P ?LEG))))
```

```
(#$implies
      (#$isa ?T #$Table)
      (#$thereExistAtLeast 3 ?LEG
            (#$and
                  (#$isa ?LEG #$Leg)
                  (#$anatomicalParts ?T ?LEG))))
```

```
(#$implies
      (#$isa ?P #$Person)
      (#$thereExistAtMost 1 ?SPOUSE
            (#$spouse ?P ?SPOUSE)))
```

# Well-Formedness of Quantified Formulas

As you probably by now expect, any formula beginning with a quantifier is well-formed if and only if its arguments are of the right number, of the right types, in the right order, and its formula argument is well-formed.

# Skolemization

People writing assertions for entry into the CYC® KB use #$thereExists quite frequently. But when you browse the KB, you rarely see #$thereExists in an assertion. That's because once assertions are entered into the KB, occurences of #$thereExists are automatically converted into Skolem functions. The only exceptions are certain cases where #$thereExists is used within an expression that is an argument to a predicate.

Thus, an assertion which was entered as:

```
(#$implies
      (#$isa ?A #$Animal)
      (#$thereExists ?M
            (#$and (#$mother ?A ?M)
                  (#$isa ?M #$FemaleAnimal))))
```

will appear in the KB as 4 different assertions:

```
(#$isa #$SKF-8675309 #$SkolemFunction)
(#$arity #$SKF-8675309 1)
(#$implies
      (#$isa ?A #$Animal)
      (#$mother ?A (#$SKF-8675309 ?A)))
(#$implies
      (#$isa ?A #$Animal)
      (#$isa (#$SKF-8675309 ?A) #$FemaleAnimal))
```

# Non-Atomic Terms

A non-atomic term (NAT) is a way of specifying a term as a function of some other term(s). Every NAT is composed of a function and one or more arguments to that function.

Consider, for example, the function #$FruitFn, which takes as an argument a type of plant and returns the collection of the fruits of that type of plant. This function can be used to build the following NATs:

(#$FruitFn #$AppleTree)

(#$FruitFn #$PearTree)

(#$FruitFn #$WatermelonPlant)

. . .

Note that there may or may not be a named CYC® constant corresponding to the collection of apples (that is, a constant called #$Apple). The NAT (#$FruitFn #$AppleTree) provides a way of talking about this collection even if the corresponding constant does not exist.

NATs can be used anywhere a constant can be used.

# Function Arity

Like predicates, most functions have a fixed arity. A function is described as unary, binary, ternary, quaternary, or quintary, according to whether it takes 1, 2, 3, 4, or 5 arguments. No CycL function currently takes more than 5 arguments.

A few functions do not have a fixed arity, but can take a variable number of arguments. Mathematical functions like #$PlusFn are one example. And in Cyc-10, IBQEs are now treated as NATs in which the units of measure are functions which can take either one or two arguments, according to whether they are intended to denote a single value or a range.

# Function Argument Types and Result Types

Functions with fixed arity are similar to predicates in that the definition of the function must specify the type of each argument, using the predicates #$arg1Isa, #$arg2Isa, etc.

Functions with no fixed arity are defined using the predicate #$argsIsa, which specifies a single type of which every argument must be an instance.

Functions differ from predicates in that they return a CYC® term as a result. Accordingly, function definitions must also describe the type of the result to be returned, using the predicate #$resultIsa. Consider, for example, the function #$GovernmentFn:

(#$arity #$GovernmentFn 1)

(#$arg1Isa #$GovernmentFn #$GeopoliticalEntity)

(#$resultIsa #$GovernmentFn #$RegionalGovernment)

The argument to #$GovernmentFn must always be an instance of #$GeopoliticalEntity, and a NAT created using #$GovernmentFn will always be an instance of #$RegionalGovernment. So, for instance,

(#$isa

(#$GovernmentFn #$UnitedStatesOfAmerica) #$RegionalGovernment)

# Reifiable Functions vs. Non-Reifiable Functions

Many CycL functions are instances of #$ReifiableFunction. Each time an instance of #$ReifiableFunction is used with a new set of arguments to build a NAT, that NAT is reified, that is, preserved in the CYC® ontology as a constant.

Constants which are reified NATs don't start out with proper constant names, but can always be referred to by their NAT expression. They can later be assigned constant names if desired.

- Skolem functions are reifiable.
- Non-reifiable functions include mathematical functions like #$PlusFn. Just because we use a NAT like (#$PlusFn 59 64) doesn't mean we want to add to the KB a unit denoting the number 123. If we want to talk about the number 123, we'll just refer to it directly.

# Assertions

So far this document has dealt mostly with the syntax of formulas in CycL. This is the syntax used by people or external programs when they assert things into a version of the CYC® KB or query the KB. Now we will shift our focus to what formulas look like once they have been asserted into the KB.

The CYC® KB consists of a large number of assertions. When a formula is successfully asserted into the KB, it is stored as one of these. Each assertion is composed of a number of elements:

- a formula
- a microtheory
- a truth value
- a direction (or access level)
- a support

# Formulas

You are already familiar with formulas-they are the CycFormulas we use to state things in the CYC® KB.

# Microtheories

Every assertion is contained in some microtheory. A particular formula may be asserted into (or concluded in) more than one microtheory; when this is the case, there will be an assertion which has that formula in each of those microtheories. The largest number of assertions are currently in the #$BaseKB.

Microtheories are covered in more detail here, as well as in the constant vocabulary, under #$Microtheory. Where does the microtheory information on assertions come from? That depends on the origin of the assertion. If an assertion is added to the KB by the inference engine as the result of firing a rule, the inference engine code decides what microtheory the conclusion should be added in and records it at add time.

If an assertion is the result of a person or external program asserting a formula into the KB, at that time the asserter must specify which microtheory the formula is to go in. Some interfaces for knowledge entry may not require the user to specify a microtheory for new assertions, and will then either try to choose the right one or will use #$BaseKB as a default. If you use such an interface make sure you know what the default behavior is.

# Truth Values

Attached to every assertion is a truth value that indicates its degree of truth. CycL contains five possible truth values, of which the most common are default true and monotonically true.

Assertions that are monotonically true are held to be true in every case, that is, for every possible set of bindings to the universally quantified variables (if any) in the assertion, and cannot be overridden. In the case of a monotonically true assertion with universally quantified variables in its formula, if an object is found for which the assertion is not true, an error is signalled.

In the case of a ground assertion that is monotonically true, if the negation of that formula is ever asserted or arrived at during inference (in the same microtheory), an error is signalled.

Assertions that are default true, in contrast, are held to be true in most cases, and can be overridden. If the negation of an existing ground, default assertion is asserted in the same microtheory, or is arrived at through inference, no error is signalled. Instead, the argumentation mechanism is invoked to decide what the final truth value of the assertion will be.

By default, GAFs which begin with the predicates #$isa and #$genls are monotonically true, while all other assertions (including rules) are default true.

# Directions

Direction is a value associated with every assertion that determines when inferencing involving that assertion should be performed. There are three possible values for direction: forward, backward, and code.

Inferencing involving assertions with direction forward is performed at assert time (that is, when a new assertion is added to the KB), while inferencing involving assertions with direction backward is postponed until a query occurs and that query allows backward inference. By default, GAFs have direction forward, while rules have direction backward. Only in very special cases should rules have direction forward.

# Supports

Attached to every assertion is a support, which consists of one or more justifications which form the support for the presence of the assertion in the KB. In many cases, at least one of the supporting justifications is local, indicating that the assertion was added to the KB from an outside source (most commonly, a human KEer).

In other cases, a supporting justification is a source which indicates the assertion was inferred and which outlines the final step of some argument, or chain of reasoning, which supports the assertion.