В.А. Лапшин

Система Cyc и ее библиотека онтологий

Аннотация. Работа посвящена описанию библиотеки онтологий, предоставляемой программной системой под названием Cyc. На сегодняшний день, онтологии системы Cyc представляют собой наиболее объемное собрание фактов в мире, описывающих различные аспекты человеческой деятельности. Система Cyc довольно объемна как по предоставляемой функциональности, так и по объему содержащейся в ней информации. Поэтому описание системы разделено на две части. Данная статья представляет собой первую часть, в которой дана общая информация о системе, а также приведено описание языка онтологий CycL, который используется в системе Cyc для добавления новых фактов.

Ключевые слова: базы знаний, онтологии, Cyc, CycL.

Введение

В последнее время стало очевидным возрастающее значение онтологий и связанных с их разработкой программных инструментов в современной компьютерной науке. В области изучения онтологий наблюдаются процессы, которые очень похожи на то, что происходило в 70-х годах прошлого века в области построения теории баз данных. Базы знаний [2] являются естественным развитием баз данных, а онтологии выполняют специфическую роль, позволяя базам знаний описывать свое содержание машинам и людям. Но задача описания знаний может быть сформулирована и вне связи с программной системой управления знаниями. Ярким примером этого является исследовательская активность [14], наблюдаемая в области Semantic Web [9].

Хотя, большинство современных исследований в области изучения онтологий обращено к Semantic Web, роль традиционных баз знаний и проблем построения онтологий этих программных систем остается актуальной и в наши дни. Эти системы, помимо того, что выполняют свое традиционное предназначение, являются своего рода испытательным полигоном, на котором отрабатываются различные подходы к решению проблем, возникающих при описании знаний различного рода, моделируются новые задачи и появляются пути их решения. В этом смысле традиционные базы знаний представляют собой «адронные коллайдеры» для изучения проблем, возникающих в области описания знаний.

В данной работе описана одна из старейших баз знаний в мире и, вероятно, наиболее объемная — система управления знаниям Cyc [5]. К сожалению, литературы по системе Cyc на русском языке совсем немного. Можно упомянуть разве что вводную статью М. Алексеевой [1]. Данная работа предназначена для того, чтобы заполнить этот пробел, предоставив русскоязычному читателю подробный обзор системы Cyc на родном языке.

1. Что такое Cyc

Cyc[1] [5] представляет собой программную систему, библиотека онтологий которой создана для описания знаний, необходимых для рассуждений на бытовом уровне, т.е. таких знаний, которые человек использует в своей каждодневной деятельности.

Разработка Cyc началась в 1984 г. под руководством Дугласа Лената в компании

[1] Имя «Cyc» представляет собой выдержку из английского «enCYClopedia» (энциклопедия) и произносится как «Цик».

Microelectronics and Computer Technology Corporation (MCC)[2]. В 1994 г. была учреждена компания Cycorp, целью которой являлась коммерческая деятельность в области, связанной с проблематикой искусственного интеллекта.

В системе Cyc для обозначения системы, разрабатывавшейся до 1994 г., используется термин Cyc-9, а для системы, разработанной позже – Cyc-10. Таким образом, Cyc-10 представляет собой название текущей реализации системы Cyc. Первая версия Cyc-10 появилась в марте 1995 г. и была разработана под руководством Кейта Гулсби (Keith Goolsbey), архитектора системы и разработчика процедур логического вывода.

Знания в библиотеке онтологий Cyc поделены на так называемые *микротеории*. Микротеорий в проекте Cyc тысячи. Микротеория содержит факты, касающиеся той или иной области знаний. В этом смысле микротеория представляет собой аналог онтологии. Для записи фактов в микротеориях Cyc использует специальный язык под названием CycL (Cyc Language). Язык CycL представляет собой язык исчисления логики предикатов первого порядка с аксиомами эквивалентности, возможностями расширения процедур логического вывода и сколемизацией. Язык также поддерживает некоторые свойства языка второго порядка (например, квантификацию по предикатам с некоторыми ограничениями). База знаний Cyc состоит из *термов* (terms) – словаря языка CycL и *суждений* (assertions), связывающих эти термы. Первая часть описания системы Cyc, представленная в данной статье, посвящена в основном языку CycL.

Система Cyc используется не только, как хранилище для суждений, но также предоставляет возможности по осуществлению логического вывода по этим суждениями.

[2] Компания MCC была создана в 1982 г. крупнейшими производителями компьютерной техники США. Основной целью создания компании был ответ на японский проект создания компьютера пятого поколения (Проект пятого поколения), который японцы планировали создать к 1991 г. Директором компании был назначен адмирал Боби Рей Инман, который до этого возглавлял Агентство национальной безопасности США, а также являлся заместителем директора ЦРУ. Компания занималась поддержкой проектов, связанных с разработками в области искусственного интеллекта. В июне 2000 г., совет директоров компании проголосовал за ее роспуск.

Библиотека онтологий проекта Cyc довольно обширна. Так, известно, что еще в 1994 г., когда была создана компания Cycorp, библиотека содержала порядка 250 тыс. фактов. Очевидно, что за прошедшие 15 лет, объем знаний, содержащихся в библиотеке, увеличился. На данный момент, онтология Cyc содержит более 2 млн. суждений. Для систематизации такого обширного комплекса знаний в Cyc используется своя онтология верхнего уровня.

Вторая часть описания системы, представленная отдельной статьей, содержит описание систем логического вывода Cyc, а также структуры онтологии системы — так называемой онтологии верхнего уровня.

2. Язык CycL

CycL – это язык, использующийся в проекте Cyc для описания онтологий. CycL – формальный язык, т.е. язык, синтаксические конструкции которого точно описаны и имеют однозначное истолкование. Синтаксис языка базируется на двух источниках: языке логики предикатов первого порядка [3] и функциональном языке LISP [15]. Однако семантика языка CycL, как мы увидим в дальнейшем, отличается от семантики языка логики предикатов.

Словарь языка CycL состоит из *термов* (terms). Множество термов состоит из *констант* (constants), *составных термов* (non-atomic terms (NATs)), *переменных* (variables) и некоторых других типов объектов. Термы составляют семантически значимые *выражения* (expressions) языка CycL, которые используются для записи *суждений* (assertions) в базах знаний проекта Cyc.

Ниже мы рассмотрим основные элементы языка CycL, в большей степени уделяя внимание наиболее часто использующимся конструкциям. Для более детальной информации о языке заинтересованный читатель может обратиться к «Руководству по языку CycL» [11] на сайте проекта системы. Автор основывает свое изложение свойств языка CycL на этом документе.

2.1. Константы

Константы представляют собой элементы словаря базы знаний проекта Cyc. Разработчики проекта пытаются моделировать человеческое

восприятие мира, поэтому каждая константа обозначает некоторую вещь или понятие. Некоторые константы обозначают коллекции других понятий, такие константы так и называются – *Коллекции* (Collections). Например, константа `#$AnimalWalkingProcess` означает множество действий, производимых животным во время ходьбы, а константа `#$Typewriter` – множество всех пишущих машинок. Другие константы обозначают конкретные объекты, которые могут более или менее постоянно находиться в базе знаний (например, `#$InternalRevenueService` – Налоговое управление США), или возникают там время от времени (например, `#$Walking00036` – конкретный случай описания процесса ходьбы). Некоторые константы обозначают предикаты, такие как `#$isa` или `#$likesAsFriend`, использующиеся для обозначения связей между объектами, другие константы обозначают функции (функция `#$GovernmentFn`, получающая в качестве аргументов константы или другие объекты языка и генерирующая в результате новое понятие – `#$GovernmentOf #$Canada`). Каждая константа имеет в базе знаний Cyc свою собственную структуру данных, содержащую, кроме всего прочего, имя данной константы и множество суждений о ней.

Большинство констант идентифицируются в языке CycL уникальными именами. Однако, как мы увидим ниже, некоторые константы могут не иметь постоянных имен. Имена таких констант формируются как результат выполнения функции, т.е. имя представляет собой составные выражения. Этот способ именования мы подробно рассмотрим в дальнейшем. Имена констант должны начинаться с префикса «#$» (произносится как «хэш-доллар» – hash" dollar).

За префиксом должны следовать по крайней мере два символа имени константы. Иначе говоря, длина имени константы не может быть менее четырех символов (включая префикс). Для именования констант позволено использовать буквы верхнего и нижнего регистров, цифры, дефис «-», подчеркивание «_» и вопросительный знак «?». Имена, различающиеся только регистром символов, считаются различными, т.е. язык CycL чувствителен к регистру (case sensitive).

Все имена предикатов должны начинаться со строчной буквы, а имена других элементов языка, – либо с прописной буквы, либо с цифры. Внимательный читатель уже, вероятно, заметил соглашение об именовании констант, применяющееся в языке CycL. Для выделения значимых элементов имени используются символы верхнего регистра. Иначе говоря, имя константы состоит из значимых слов, начала которых выделяются прописными буквами. Например, если мы хотим использовать константу для обозначения автора романа «Морской волк» [4], то она должна иметь вид `#$JackLondon`, но не `#$Jacklondon` или `#$Jack?London`. Символы подчеркивания, дефиса и вопросительного знака нужно использовать для разделения семантически разнородных компнонентов имени. Например, в имени `#$Fruit-TheWord` разделены компоненты «`Fruit`» (фрукт) и «`TheWord`» (слово), чтобы отделить обозначение имени объекта (фрукт) от его характеристики (слово).

Сами по себе имена ничего не значат в языке CycL. Совершенно неважно, каким именно именем будет обозначаться понятие «зеленый цвет»: `#$Green` или `#$GreenColor`. Значение константы определяется не его именем, а суждениями, которые имеются в системе относительно данной константы. Конечно, будет проще и понятнее использовать имена, по смыслу согласующиеся с их значениями в базе знаний Cyc, в противном случае можно внести путаницу. Например, если использовать имя `#$Green` для обозначения цвета, которое в суждениях базы знаний Cyc обозначает понятие «красный цвет». Но, всегда следует помнить, что смысл константы определяется не его именем, а суждениями относительно этой константы, содержащимися в базе знаний Cyc. Например:

```
(#$isa #$Green #$Color)
(#$colorOfObject #$Grass37 #$Green)
(#$forAll ?O (#$implies (#$isa ?O #$Okra)
                        (#$colorOfObject ?O
#$Green)))
```

Здесь имеются три суждения относительно константы `#$Green`:

- #$Green – это цвет;
- цвет объекта #$Grass37 (конкретный экземпляр понятия «трава») – это #$Green;
- для всех объектов, имеющих тип «охра» (#$Okra), их цвет должен быть #$Green.

2.2. Переменные

В языке CycL выражения с кванторами могут содержать переменные, использующиеся для обозначения констант, идентификаторы которых не определены. Имена переменных должны начинаться с символа «?» и записываться только прописными буквами. Например, можно написать «?FOO», но не «FOO» или «?Foo».

Если в формуле используется одна переменная, то для ее именования обычно используют только один символ, например, «?X». Если формула содержит несколько переменных, то принято давать им мнемонические имена. Например,

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

2.3. Формулы

Формулы в языке CycL объединяют термы в осмысленные выражения. Ввиду того, что синтаксис CycL унаследован от языка LISP, каждая формула в языке CycL имеет вид списка. Список представляет собой последовательность элементов, заключенную в круглые скобки. Например, формула (?A ?B ?C) представляет собой список из трех переменных. Элементами списка могут быть также и другие формулы, т.е. другие списки. Например, список (?A (#$isa ?B #$Green) ?C) – это формула, содержащая три элемента, причем второй элемент сам является формулой (списком из трех элементов). Понятно, что элементы вложенной формулы также могут быть формулами и такое вложенное структурирование может продолжаться сколь угодно долго. В результате, можно построить формулу любой сложности. Списочная структура проста и логична, но трудно читаема. Поэтому, при написании сложных формул стараются использовать разметку, записывая элементы списков в отдельных строках и выделяя элементы одного уровня вложенности одними и теми же отступами.

Элементы списка формулы называются *аргументами* (arguments). По соглашению, нумерация элементов списка начинается с нуля, т.е. на первый аргумент ссылаются, как на ARG0, на второй – как на ARG1 и т.д. В качестве ARG0 могут выступать предикат, логический оператор или квантор. Остальными аргументами могут быть термы, составные термы, переменные, числа, заключенные в двойные кавычки строки символов английского алфавита или другие формулы.

Для формул в языке CycL определен специальный класс (Collection) под именем #$CycFormula. Элементами этого класса являются все правильные формулы, т.е. правильно сформированные синтаксически и удовлетворяющие всем семантическим ограничениям.

Формулы простейшего вида называются *атомарными формулами* (atomic formulas). В атомарной формуле в качестве ARG0 выступает предикат, а остальные аргументы представляют собой термы. Например,

```
(#$likesAsFriend #$DougLenat #$KeithGoolsbey)
(#$skillCapableOf #$LinusVanPelt
                  #$PlayingAMusicalInstrument
                  #$performedBy)
(#$colorOfObject ?CAR ?COLOR)
```

Первые две формулы из примера выше представляют собой *базовые атомарные формулы* (ground atomic formulas – GAF). Базовая атомарная формула содержит в качестве ARG0 имя предиката, а остальные аргументы представляют собой константы (но не переменные).

2.4. Предикаты

Абсолютное большинство предикатов в языке CycL имеют фиксированное число аргументов. Число аргументов предиката называется его *местностью* (arity) – (одноместный предикат, двуместный предикат и т.д.). Необязательные аргументы задавать нельзя, хотя некоторые предикаты, такие как #$different, могут иметь переменное число аргументов. Для предикатов с переменным числом аргументов в языке CycL имеется специальный класс #$VariableArityRelation. Предикаты с фиксированным числом аргументов также принадлежат своим классам, ограничивающим число аргументов. Например, все предикаты с двумя аргументами должны быть элементами класса #$BinaryPredicate. Местность предиката можно также указать непосредственно с помощью бинарного предиката #$arity, который берет имя предиката в качестве первого аргумента и количество его аргументов в качестве

второго. На данный момент в языке CycL не существует предикатов с числом аргументов больше пяти.

Атомарная формула считается *правильно построенной* (well formed), если в качестве ARG0 выступает имя предиката, а число остальных аргументов совпадает с местностью этого предиката. Кроме того, типы аргументов формулы обязательно должны совпадать с типами аргументов предиката, который выступает в качестве Arg0. Например, предикат `#$residesInDwelling` определен следующим образом:

```
(#$isa #$residesInDwelling #$BinaryPredicate)
(#$arg1Isa #$residesInDwelling #$Animal)
(#$arg2Isa #$residesInDwelling
#$ShelterConstruction)
```

Первая формула объявляет константу `#$residesInDwelling` элементом класса `#$BinaryPredicate`, т.е. предикат `#$residesInDwelling` объявляется как двуместный. Вторая формула задает тип первого аргумента (`#$Animal`), третья формула – тип второго (`#$ShelterConstruction`). Формула вида `(#$residesInDwelling #$PottedPlant37)` заведомо не является правильно построенной, так как количество аргументов в ней не совпадает с местностью предиката `#$residesInDwelling`. Формула

```
(#$residesInDwelling #$PottedPlant37
#$KarensHouse)
```

будет правильно построенной только тогда, когда тип константы `#$PottedPlant37` будет `#$Animal`, а тип константы `#$KarensHouse` – `#$ShelterConstruction`.

Из формы определения предикатов в языке CycL становится понятно, почему невозможно объявить предикаты с местностью, большей пяти. Дело в том, что обязательно необходимо определить тип каждого аргумента предиката. Для этого используются предикаты вида `#$arg1Isa`, `#$arg2Isa` и т.д. Таких предикатов в языке CycL всего пять: от `#$arg1Isa` до `#$arg5Isa`. Поэтому определение предикатов местности большей пяти требует изменения языка CycL.

Кроме того, необходимо задать тип предиката, т.е. его принадлежность одному из классов предикатов. В языке CycL все предикаты должны принадлежать классу `#$Predicate`. От этого класса унаследованы классы `#$UnaryPredicate`, `#$BinaryPredicate`, `#$TernaryPredicate`, `#$QuaternaryPredicate` и `#$QuintaryPredicate`, определяющие одноместные, двуместные, трехместные, четырехместные и пятиместные предикаты, соответственно. Из этого также следует, что определение предиката с местностью, большей пяти, синтаксически невозможно.

2.5. Логические операторы

Сложные формулы строятся из атомарных или других сложных формул с помощью *логических операторов* (logical connectives). Логические операторы представляют собой строго определенный набор констант, имена которых совпадают с названиями логических операторов, которые используются в логических языках. Наиболее часто использующиеся логические операторы – `#$not`, `#$and`, `#$or` и `#$implies`. Рассмотрим эти операторы подробнее.

Оператор `#$not` эквивалентен логическому отрицанию. В формуле оператор `#$not` всегда выступает как Arg0 и имеет один аргумент – формулу, возвращающую истину или ложь. Если аргумент есть истина, то оператор возвращает ложь, и наоборот. Например, формула `(#$not (#$colorOfObject #$FredsBike #$RedColor))` является истинной, только, если формула `(#$colorOfObject #$FredsBike #$RedColor)` ложна, а формула `(#$not (#$not (#$colorOfObject #$FredsBike #$RedColor)))` возвращает то же значение, что формула `(#$colorOfObject #$FredsBike #$RedColor)`.

Оператор #$and аналогичен оператору конъюнкции, использующемуся в математической логике. Но, в отличии от логического «и», этот оператор может брать на вход любое число аргументов – один, два и более. Оператор возвращает истину только, если все аргументы истинны. Например,

```
(#$and
   (#$colorOfObject #$FredsBike #$RedColor)
   (#$objectFoundInLocation #$FredsBike
#$FredsGarage))
```

возвращает истину только в том случае, когда мотоцикл Фреда красного цвета `((#$colorOfObject #$FredsBike #$RedColor))`

и Фред держит этот мотоцикл в гараже ((#$objectFoundInLocation #$FredsBike #$FredsGarage)).

Оператор #$or также берет на вход один и более аргументов. Так же, как и логический оператор «или», оператор #$or возвращает истину, если хотя бы одни из его аргументов является истинным. Рассмотрим, например, формулу

```
(#$or
   (#$colorOfObject #$FredsBike #$RedColor)
   (#$objectFoundInLocation          #$FredsBike
#$FredsGarage)
   (#$owns #$Fred #$FredsBike))
```

Данная формула возвращает истину, если хотя бы одно из высказываний «мотоцикл Фреда красного цвета», «Фред держит мотоцикл в гараже» и «Фред – владелец своего мотоцикла» истинно. Могут быть истины более одной формулы, в этом случае оператор #$or все равно возвращает истину.

Оператор #$implies аналогичен логической импликации (⇒). Так же как оператор импликации, оператор #$implies имеет в точности два аргумента. Аргумент Arg1 является логической посылкой, а аргумент Arg2 – логическим следствием. Логически формула (#$implies A B) эквивалента ($A \Rightarrow B$) и читается так: «если A, то B». Оператор возвращает ложь только в том случае, когда первый аргумент есть истина, а второй аргумент – ложь. Например, формула

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

возвращает ложь только тогда, когда Фред является собственником мотоцикла не красного цвета.

Суждения с оператором #$implies часто используются в языке CycL. Такие суждения еще называют *условиями* (conditionals) или *правилами* (rules). Следует также заметить, что формулы, подобные той, что приведена в примере выше, не часто встречаются в базе знаний Cyc. Более репрезентативные примеры применения оператора #$implies будут даны ниже.

Формула с логическими операторами является *правильной* (well formed), если все формулы, выступающие в качестве аргументов, являются правильными, и число аргументов логического оператора совпадает с разрешенным. Предположим, что A и B – правильные формулы, а формула C – нет. Тогда, формулы

```
(#$not A)
(#$and A)
(#$and A B)
(#$or A)
(#$or A B)
(#$implies A B)
```

правильные, а формулы

```
(#$not A B)
(#$and)
(#$and A C)
(#$implies A)
```

не правильные. Читателю предлагается оценить самостоятельно в качестве упражнения, почему формулы, приведенные в примере выше, не являются правильными.

2.6. Кванторы

До сих пор мы рассматривали только суждения о конкретных объектах, таких как #$FredsBike. Но, язык CycL, как и всякое исчисление предикатов первого порядка, позволяет говорить об объектах в общем, не определяя в формуле их конкретные имена. Формулы такого типа выглядят как высказывания, начинающиеся с «существует объект, для которого истина формула...» или «для любого объекта выполняется...». Такие формулы используют так называемые *операторы квантификации* (quantifications) или кванторы. В языке CycL имеется два вида кванторов: *универсальные кванторы* (universal quantification) и *кванторы существования* (existential quantification). Универсальные кванторы используются в суждениях, соответствующих русским предложениям «для любого» или «для всех», а кванторы существования – предложениям вида «существует», «найдется», «какой-то», «где-то» и т.п.

Язык CycL содержит один оператор универсальной квантификации (универсальный квантор) #$forAll и четыре оператора квантификации существования (квантора существования): #$thereExists, #$thereExistAtLeast, #$thereExistAtMost и #$thereExistExactly. Рассмотрим эти операторы подробнее.

Оператор #$forAll имеет два аргумента: переменную и формулу, в которой эта переменная содержится. На практике формула

обычно представляет собой некоторое *условие*, в котором посылка используется для того, чтобы ограничить область применения переменной. Например, формула

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

говорит о том, что для любого объекта в онтологии, собственником которого является Фред, этот объект должен располагаться в доме Фреда.

Для **квантификации по нескольким переменным** можно использовать вложенные друг в друга формулы с кванторами. Например, формула

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

говорит о том, что любые два объекта, находящиеся в собственности Фреда, должны находиться рядом.

Необходимо также отметить, что имена переменных, использующихся во множественной квантификации, не должны совпадать. Иначе говоря, не позволяются формулы вида:

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

Свободные переменные. Обычно, если переменная появляется в формуле, то она присутствует в операторе квантификации, как это было продемонстрировано в примерах выше. Такие переменные называются *связанными* (bound). Каждая переменная связана со своим квантором и каждый квантор связан в точности с одной переменной. Поэтому применение связанной переменной где-то вне данной формулы бессмысленно. Язык CycL позволяет использование в формулах *свободных* (unbound) переменных (т.е. переменных, которые не связаны ни с каким квантификатором), но такие переменные все равно трактуются как связанные оператором универсальной квантификации. Например, формула

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

имеет в точности ту же интерпретацию, что и формула

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

Ввиду того, что формулы со свободными переменными выглядят яснее и проще, оператор `#$forAll` используется редко. Но, следует иметь в виду, что использование свободных переменных не в условных формулах, или в условиях, но не в посылке, может привести к непредсказуемым последствиям. Например, формула

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

является истинной для любого объекта в онтологии. Формула, фактически, говорит о том, что любая вещь, собственником которой является Фред, находится в доме Фреда, а это, конечно, в общем случае неверно.

Оператор `#$thereExists` имеет два аргумента: переменную и формулу, в которой эта переменная содержится. На практике, данный квантификатор используется довольно специфически:

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

Данная формула говорит о том, что всякий экземпляр класса «Животное» (`#$Animal`), должен иметь мать. Представляющий мать объект может уже иметься в онтологии в качестве константы, а может быть новым объектом, о котором Cyc еще не знает.

Кванторы `#$thereExistAtLeast`, `#$thereExistAtMost` и `#$thereExistExactly` подобны квантору `#$thereExists`, но предоставляют большие выразительные возможности. Каждый из операторов имеет три аргумента: положительное целое число, переменную и формулу, содержащую эту переменную. Смысл операторов проще понять на примерах:

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

Здесь говорится, что каждая персона имеет в точности две ноги[3].

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

Данная формула содержит суждение о том, что каждый стол должен иметь по крайней мере три ноги[4]. И, наконец, формула

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

говорит о том, что каждая персона имеет не более одного супруга[5].

Сколемизация. Хотя суждения с кванторами существования широко используются в онтологиях Cyc, при просмотре этих онтологий суждения такого вида не показываются. Это происходит из-за того, что система Cyc автоматически преобразовывает каждую формулу, содержащую кванторы существования в сколемовскую нормальную форму[6]. Рассмотрим пример. Формула

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

будет преобразована к виду

```
(#$isa #$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))
```

Здесь все вхождения переменной `?M` заменены на автоматически сгенерированную сколемовскую функцию `(#$SKF-8675309 ?A)`.

2.7. Составные термы

Составные термы (non-atomic terms – NAT) представляют собой способ определения термов как функции от других термов. Составной терм представляет собой функцию, в которой другие термы выступают в качестве аргументов. Рассмотрим, например, функцию `#$FruitFn` (фрукт), которая берет аргумент типа «растение» (plant) и возвращает класс Фрукт (коллекцию всех фруктов) для переданного объекта растения. Эту фукнцию можно использовать для определения термов следующего вида:

```
(#$FruitFn #$AppleTree)
(#$FruitFn #$PearTree)
(#$FruitFn #$WatermelonPlant)
```

В онтологии может иметься, а может и не иметься, именованная константа для коллекции фруктов (например, с именем `#$Apple` (яблоко) для растения `#$AppleTree` (яблоневое дерево). Иначе говоря, составные термы позволяют

[3] Это, конечно, не верно в общем случае. Есть люди, которые потеряли ноги в результате несчастных случаев. Есть также люди, которые родились без ног в результате генетической аномалии. В последнем случае, конечно, можно еще подискутировать, можно ли называть такую персону человеком.

[4] Автору приходилось видеть столы с одной, вкопанной в землю, ногой. Вероятно, количество ног у стола (и вообще, наличие ног), не является существенным признаком понятия «стол».

[5] Это также, конечно, не верно. Особенно, в наш бурный век, стремительно меняющий общественные отношения.

[6] Приведение высказываний в сколемовскую нормальную форму необходимо для работы алгоритма логического вывода методом резолюций [13]. Сколемовская нормальная форма представляет собой суждение, содержащее только операторы конъюнкции и не содержащее кванторов существования. Существует алгоритм (см. [13] гл. 4 пар. 4.2) автоматического преобразования любого суждения в сколемовскую форму. Для перехода формула сначала приводится к конъюнктивному виду, для чего используются законы Де Моргана. После этого, первая переменная, связанная квантором существования, заменяется на некоторую константу и все вхождения этой переменной в формуле заменяются на эту константу. Вхождения второй и последующих связанных квантором существования переменных заменяются на функции от тех переменных, которые встретились в исходной формуле до вхождения данной переменной в формулу. Например, формула $(x)(y)(z)(u)(v)(w)P(x,y,z,u,v,w)$ преобразуется к виду $(y)(z)(v)P(a,y,z,f(y,z),v,g(y,z,v))$. Константы и функции, используемые для замены переменных квантора существования, называются *сколемовскими функциями*. В языке CycL первое вхождение переменной также заменяется на функцию, замена на константу не используется.

говорить о типах, даже если таковые не имеют именованных констант в онтологии.

Составные термы могут использоваться в формулах вместо констант. Например:

```
(#$implies
  (#$isa ?APPLE (#$FruitFn #$AppleTree))
  (#$colorOfObject ?APPLE #$RedColor))
```

Функции. Ввиду того, что построение составных термов производится с помощью функций, представляется полезным подробнее ознакомиться с ними. Так же, как и предикаты, большинство функций имеют фиксированное число аргументов. Можно определять функции с числом аргументов от одного до пяти. Функции с числом аргументов больше пяти в языке CycL определять нельзя. В CycL имеются функции с переменным числом аргументов. Например, функция сложения `#$PlusFn` имеет переменное число аргументов. Составной терм `(#$PlusFn 2 3 4)` означает число 9, а `(#$PlusFn 12 1)` – число 13.

Функции с фиксированным числом аргументов определяются так же, как предикаты. Типы аргументов задаются с помощью предикатов вида `#$arg1Isa`, `#$arg2Isa` и т.д. Если функция имеет нефиксированное число аргументов, то все ее аргументы должны иметь один и тот же тип, который задается с помощью предиката `#$argsIsa`. В отличие от предикатов функции возвращают некоторый терм в качестве результата, тип этого терма задается посредством предиката `#$resultIsa`. Рассмотрим, например, как определяется функция `#$GovernmentFn` (правительство):

```
   (#$arity #$GovernmentFn 1)
   (#$arg1Isa #$GovernmentFn
#$GeopoliticalEntity)
   (#$resultIsa #$GovernmentFn
#$RegionalGovernment)
```

Здесь сначала задается число аргументов функции, затем определяется тип единственного аргумента, как `#$GeopoliticalEntity` (геополитический субъект) и тип результата как `#$RegionalGovernment` (региональное правительство).

Функцию `#$GovernmentFn` можно использовать в любой формуле, где в качестве параметра выступает тип `#$RegionalGovernment`. Например, формула

```
   (#$isa (#$GovernmentFn
#$UnitedStatesOfAmerica)
          #$RegionalGovernment)
```

обязательно должна быть истинной, так как терм `(#$GovernmentFn #$UnitedStatesOfAmerica)` (правительство США) обязательно должен являться экземпляром класса `#$RegionalGovernment`.

Большинство функций являются экземплярами классов[7] `#$IndividualDenotingFunction` (функция, обозначающая индивидуальный объект) и `#$CollectionDenotingFunction` (функция, обозначающая коллекцию). Функция `#$GovernmentFn` из примера выше является экземпляром класса `#$IndividualDenotingFunction` так как составной терм `(#$GovernmentFn #$UnitedStatesOfAmerica)` обозначает единственное правительство, а не коллекцию правительств. С другой стороны, определенная выше функция `#$FruitFn` является экземпляром класса `#$CollectionDenotingFunction` ввиду того, что составной терм `(#$FruitFn #$AppleTree)` обозначает коллекцию всех яблок яблоневого дерева[8].

Для функций-элементов класса `#$CollectionDenotingFunction` необходимо, кроме типов аргументов и результата, определить еще класс, который будет являть суперклассом (родительским классом) каждого результата данной функции. Ввиду того, что каждая такая функция возвращает коллекцию объектов (т.е. класс), составной терм, задаваемый этой функцией, порождает новый класс в базе знаний Cyc и его необходимо встроить в онтологию Cyc. Для этого необходимо определить данный класс как подкласс какого-то существующего в онтологии класса. Делается это с помощью предиката `#$resultGenl`[9]. Это двуместный предикат, первым аргументов которого является функция, а вторым – суперкласс класса результата. Например, функцию

[7] В терминологии языка CycL классы называются *коллекциями* (collections). Автор посчитал удобным для читателя использовать более привычное название «классы».

[8] Об индивидуальных объектах и коллекциях мы подробнее поговорим ниже

[9] Genl"— это сокращение английского «general» (общий). Иначе говоря, термин обозначает обобщение некоторого понятия. Таким образом, данный предикат можно прочитать как «обобщение класса, возвращаемого функцией в качестве результата».

#$LeftPairMemberFn (левый элемент пары обуви) можно определить следующим образом:

```
   (#$isa #$LeftPairMemberFn
#$CollectionDenotingFunction)
   (#$arity #$LeftPairMemberFn 1)
   (#$arg1Isa #$LeftPairMemberFn
#$SymmetricalPartType)
   (#$resultIsa #$LeftPairMemberFn
#$ExistingObjectType)
   (#$resultGenl #$LeftPairMemberFn
#$LeftObject)
```

Здесь в качестве родительского класса каждой коллекции, возвращаемой функцией #$LeftPairMemberFn, выступает класс #$LeftObject. Родительские классы следует отличать от типов результата функции. Любой класс, возвращаемый функцией-экземпляром класса #$CollectionDenotingFunction, сам является экземпляром класса, заданного как тип результата данной функции. В примере выше каждый результат функции #$LeftPairMemberFn есть экземпляр коллекции #$ExistingObjectType.

Напрямую отношения класс-экземпляр и класс-суперкласс задаются предикатами #$isa и #$genls, соответственно. Об отношениях, задаваемых на объектах онтологии Cyc, мы подробно поговорим ниже.

Функции материализации. В Cyc часто используются функции-экземпляры класса #$ReifiableFunction. Каждый раз, когда такая функция вызывается с новым набором аргументов, в онтологию Cyc добавляется составной терм, построенный этой функцией с данным набором аргументов, т.е. составной терм «материализуется» (reified). Материализация означает резервирование места под данный терм в онтологии Cyc. Позже, если пользователь пожелает, на это место можно будет сослаться с помощью константы, но пока для ссылки на терм используется данная функция с данным набором аргументов.

При материализации терма создается следующее отношение:

```
(#$termOfUnit NAT-CONSTANT NAT-EXPRESSION)
```

Здесь NAT-CONSTANT означает константу, используемую для обозначения терма, а NAT-EXPRESSION означает составной терм, используемый для материализации данного терма. Интересно, что для функций-экземпляров класса #$ReifiableFunction оба аргумента часто совпадают, хотя означают разные вещи. Например,

```
(#$termOfUnit (#$GovernmentFn #$Canada)
              (#$GovernmentFn #$Canada))
```

Первый аргумент – это константа (#$GovernmentFn #$Canada), выглядящая как составной терм, а второй аргумент (#$GovernmentFn #$Canada) – это уже составной терм, использующийся для создания данной константы.

Вероятно требуется некоторое время, чтобы усвоить это различие. Наверное, лучше сказать проще: для именования составных термов, материализуемых вызовами функций-экземпляров класса #$ReifiableFunction, система Cyc использует строку, представляющую собой запись составного терма, использованного для материализации.

Материализованный таким образом терм можно потом как-нибудь назвать. Для этого используется все тот же предикат #$termOfUnit. Например,

```
   (#$termOfUnit #$TheYear1996 (#$YearFn 1996))
   (#$termOfUnit #$Apple (#$FruitFn
#$AppleTree))
```

Все сколемовские функции являются функциями материализации, контрпримером которым является функция #$PlusFn. Эта функция не материализует термы в онтологии Cyc, так как результатом ее вызова является число, представляющее сумму ее аргументов. На это число можно сослаться напрямую, поэтому надобности материализовывать терм не имеется. Другим контрпримером является функция #$UnitOfMeasure (единица измерения).

Использование кванторов в составных термах. Если функция представляет собой экземпляр класса #$ReifiableFunction, то составные термы, ею порождаемые, можно использовать в выражениях с кванторами. Это бывает полезно, когда в правилах имеются составные термы, в которые в качестве одного или нескольких аргументов передается переменная. Кванторы в составных термах можно использовать одним из следующих способов:

1. Можно использовать переменные ? ARG1, ? ARG2, ? ARG3 и т.д. для обозначения аргументов функции.

2. Ввести свободную переменную для обозначения результата функции данного составного терма.

3. Добавить в посылку правила суждение с предикатом #$equals с тем, чтобы сказать системе Cyc, что данная переменная является составным термом.

Приведем пример:

```
(#$implies
  (#$and
    (#$equals ?U (#$PreviouslyOwnedFn ?ARG1))
    (#$isa ?X ?U))
  (#$hasAttributes ?X #$Used))
```

2.8. Суждения

До сих пор описывался исключительно синтаксис языка CycL. Этот синтаксис может быть использован различными внешними приложениями, чтобы добавлять новые суждения в базу знаний системы Cyc или отправлять запросы к ее онтологии. Сейчас сосредоточим свое внимание на структуре онтологии Cyc.

Онтология Cyc состоит из большого множества суждений. Когда формула добавляется в онтологию (это происходит только тогда, когда формула правильно простроена), то она добавляется как одно или несколько суждений. С суждениями связаны следующие элементы:

- формулы;
- микротеории;
- истинностные значения;
- направления (directions);
- основания (supports).

Рассмотрим каждый из этих элементов подробнее.

Формулы представляют собой выражения на формальном языке, использующиеся для определения суждений о мире. В этом смысле формулы представляют собой аналог повествовательных предложений русского языка. Правильно построенные формулы в языке CycL называются *Cyc-формулами* (CycFormulas). Основным объектом изучения в разделе, посвященном синтаксису языка CycL, были формулы.

Микротеории. Каждое суждение содержится по крайней мере в одной микротеории. Если это необходимо, суждение может содержаться сразу в нескольких микротеориях. Имеются даже суждения, которые содержаться в каждой микротеории. Наибольшее число суждения содержится в микротеории под названием `#$BaseKB`.

Микротеории являются экземплярами класса `#$Microtheory` и, как и между любыми классами онтологии Cyc, между микротеориями можно устанавливать отношения вида класс-подкласс (отношение обобщения в терминологии Cyc). Если суждение принадлежит некоторой микротеории, то это суждение также должно принадлежать и всем микротеориям, представляющим собой подклассы данной микротеории. Микротеории также называются *контекстами*. Для детальной информации о контекстах [7].

Истинностные значения. К каждому суждению присоединено истинностное значение, показывающее «степень истинности» этого суждения. Для выражения степени истинности суждений в CycL используется пять возможных истинностных значений. Наиболее часто из них используются два:

- Постоянно истинные (monotonically true) суждения являются аналогом тавтологий в логических исчислениях. Такие суждения должны быть истинными всегда, т.е. для всех констант, заменяющих универсально квантифицированные переменные в этих суждениях. Если при каком-то наборе констант суждение становится ложным, то система генерирует ошибку.
- Истинные по умолчанию (default true) суждения, в отличие от постоянно истинных суждений, могут быть переопределены для каких-то конкретных значений аргументов. Если для каких-то значений аргументов данное суждение является ложным, то ошибка не генерируется, но запускается процесс так называемой *аргументации* (argumentation) для оценки степени истинности данного суждения.

По умолчанию все базовые атомарные формулы, начинающиеся с предикатов `#$isa` и `#$genls`, являются постоянно истинными, а все остальные суждения, включая правила, – истинными по умолчанию.

Направление (direction) представляет собой привязываемое к каждому суждению значение, которое задает, в каком сорте логического вывода данное суждение будет задействовано. Существует три значения для направления: прямое (forward), обратное (back) и код (code). Процедура логического вывода задействует суждения с прямыми направлениями, когда добавляет суждения в базу знаний Cyc, а сужде-

ния с обратными направлениями – когда строит запросы к базе знаний, причем эти запросы строятся с помощью процедуры *обратного вывода* (backward inference). По умолчанию все базовые атомарные формулы имеют прямое направление, а правила – обратное.

Суждения, направление для которых имеет значение «код», вообще не используются в обычных процедурах логического вывода. В системе Cyc имеется возможность добавлять в систему пользовательские модули логического вывода – так называемые модули *эвристического уровня* (heuristic level (HL))[10]. Если суждение имеет направление «код», то оно задействуется только при работе соответствующего модуля.

Основания (supports) представляют собой один из видов обоснований (justification) суждения. Обоснования говорят о том, *почему* данное суждение имеет то или иное истинностное значение. В этом смысле обоснования похожи на аргументы защиты или обвинения в судебном процессе. Основания говорят о том, каким образом данное суждение попало в онтологию. В абсолютном большинстве случаев, основание суждения имеет значение «локальное» (local), которое показывает, что данное суждение было добавлено в базу знаний из внешнего источника (обычно, это KEer[11]). Иногда основания имеют значения «источник» (source), показывающее, что суждение было добавлено в онтологию в результате работы процедуры логического вывода.

Заключение

Как читатель имел возможность убедиться в процессе чтения статьи, система Cyc довольно объемна. За 25 лет разработки в системе накоплено большое количество фактов, имеется также разветвленная структура их систематизации и хорошо разработанные инструменты для добавления новых фактов в онтологии системы. В данной статье, ввиду ограничений, налагаемых на объем текста, представлено только описание языка CycL, специально разработанного для описания суждений, содержащихся в онтологии. Описание других аспектов системы: структуры онтологии и систем логического вывода, будет дано во второй части работы.

[10] В системе Cyc имеются также модули *эпистемологического уровня* (epistemological level (EL)). Эти модули используются для представления знаний пользователям и при взаимодействии с другими программами

[11] KEer представляет собой сокращение от Knowledge Enterer – персоны, осуществляющей добавление знаний в онтологию Cyc.

Литература

1. Алексеева М.А. Обзор системы Cyc. // М.:РГГУ (2008). http://ezop-project.wiki.sourceforge.net/Alekseeva_Cyc.
2. Гаврилова Т.А., Хорошевский В.Ф. Базы знаний интеллектуальных систем. // СПб.: Питер, 2001.
3. Клини С. Математическая логика. // М.: Мир, 1973.
4. Лондон Дж. Морской волк. // М.: АСТ, 2006.
5. Сайт компании Cycorp. // http://www.cyc.com.
6. Сайт проекта OpenCyc. // http://opencyc.org.
7. Страница описания контекстов Cyc. // http://www.cyc.com/cycdoc/course/contexts-basic-module.html.
8. Страница проекта Cyc в английской Википедии. // http://en.wikipedia.org/wiki/Cyc.
9. Страница проекта Semantic Web на сайте консорциума W3. // http://www.w3.org/2001/sw.
10. Страница описания онтологии Cyc. // http://www.cyc.com/cyc/technology/whatiscyc_dir/whatdoescycknow.
11. Страница описания языка CycL на сайте проекта Cyc. // http://www.cyc.com/cycdoc/ref/cycl-syntax.html.
12. Страница Cyc-NL. // http://www.cyc.com/cyc/technology/whatiscyc_dir/cycrandd/areasofrandd_dir/nlu.
13. Чень Ч, Ли Р. Математическая логика и автоматическое доказательство теорем. // М.: Наука, 1983.
14. Хорошевский В.Ф. Пространства знаний в сети Интернет и Semantic Web. Часть 2. // Искусственный интеллект и принятие решений, № 4, 2009.
15. Хювёнен Э., Септянен И. Мир Лиспа. Том 1. Введение в язык Лисп и функциональное программирование. // М.: Мир, 1990.
16. Davidson C. Common Sense and the Computer. // New Scientist. April 2, 1994.
17. Guha R., Lenat D., Pittman K., Pratt D., Shepherd M. Cyc: a midterm report. // Communications of the ACM, August 1990/ Vol 33. No 8.
18. Lenat D., Guha R. Building Large Knowledge Based Systems. // Addison Wesley, Reading Mass, 1990.
19. Lenat D., Guha R. Enabling Agents to Work Together. // Communications of the ACM, July 1994.
20. Lenat D. Steps to Sharing Knowledge. In Toward Very Large Knowledge Bases. // Edited by N.J.I. Mars. IOS Press, 1995.
21. Lenat D. Artificial Intelligence. // Scientific American, September 1995.

Лапшин Владимир Анатольевич. Ведущий разработчик ООО «Инфовотч». Окончил Киргизский Государственный национальный университет в 1996 году. Кандидат физико-математических наук. 15 печатных работ. Область научных интересов: теория формальных языков, онтологии и их применение. E-mail: mefrill@yandex.ru