

Московский государственный институт электроники и математики
(технический университет)

На правах рукописи

Лебедев Андрей Сергеевич

РАЗРАБОТКА И ИССЛЕДОВАНИЕ СИСТЕМЫ
КОНЦЕПТУАЛЬНОГО ПРОГРАММИРОВАНИЯ С ИСПОЛЬЗОВАНИЕМ
ЛИНГВИСТИЧЕСКОГО ПРОЦЕССОРА

Специальность 05.13.11. – «Математическое и программное
обеспечение вычислительных машин, комплексов и компьютерных сетей»

Диссертация на соискание ученой степени кандидата технических наук

Научный руководитель:
д.т.н., профессор Г.П. Путилов

Москва – 2011

Оглавление

Основные обозначения и сокращения.....	6
Введение.....	7
Глава 1. Обзор литературы.....	17
1.1. Обзор методов и средств морфологического анализа.....	17
1.1.1. Парсер Mystem.....	17
1.1.2. Компонент морфологического анализа системы ДИАЛИНГ	20
1.2.3. ABBYY Retrieval & Morphology (ARM) Engine.....	21
1.2.4. Универсальный морфосемантический словарь.....	22
1.2. Обзор методов и средств синтаксического анализа.....	24
1.2.1. Система LinkParser.....	24
1.2.2. Синтаксический анализатор системы ЭТАП-3.....	28
1.2.3. Компонент синтаксического анализа системы ДИАЛИНГ	29
1.2.4. Расширенные сети переходов.....	31
1.3. Обзор методов и средств семантического анализа.....	38
1.3.1. Система ПОЛИТЕКСТ.....	38
1.3.2. Компонент семантического анализа системы ДИАЛИНГ	44
1.4. Системы извлечения знаний из текстов.....	47
1.4.1. Лингвистический процессор Semantix.....	47
1.4.2. Система извлечения знаний из документов «Аналитический курьер».....	50
1.5. Концептуальное программирование.....	52
1.6. Языко-ориентированное программирование.....	55
1.7. Естественнo-языковое программирование.....	62

1.7.1. Система ПЕГАС.....	62
1.7.2. Обзор существующих концепций использования объектно-ориентированного программирования для лингвистического моделирования.....	67
1.8. Онтология. Язык OWL.....	71
1.9 Средства редактирования расширенных сетей переходов.....	74
1.9.1. Augmented Syntax Diagram (ASD) Editor and Parser.....	74
1.9.2. Linguistic Tree Constructor.....	75
1.9.3. TreeForm Syntax Tree Drawing Software.....	76
1.10 Выводы.....	76
Глава 2. Структура программного комплекса NALAPS.....	79
2.1. Общая концепция системы.....	79
2.2. Лингвистический процессор.....	81
2.2.1. Компонент морфологического анализа.....	81
2.2.2. Компонент синтаксико-семантического анализа.....	85
2.3. Концептуальный процессор.....	95
2.4. Естественнo-языковой спецификатор языка Java.....	98
2.5. Кодогенератор.....	103
2.6. Динамическая объектная система как способ представления смысла текста.....	106
2.7. Обратный концептуальный и обратный лингвистический процессоры.	109
2.8. Редактор расширенных сетей переходов.....	112
2.9. Выводы.....	119

Глава 3.....	121
3.1. Общие положения.....	121
3.2. Работа морфологического процессора.....	121
3.3. Работа синтактико-семантического анализатора.....	122
3.4. Проблема интерпретации формализованного представления ЕЯ- текста на языке программирования.....	124
3.5. Формальная модель концептуального графа.....	127
3.6. Операция сравнения фреймовых структур f_m	131
3.7. Построение проекции из CG_{Java} в CG_{NL}	132
Глава 4.....	135
4.1. Описание проекта в системе Nalaps.....	135
4.2. Основное окно программы.....	137
4.3. Модульная интеграция со средой разработки IntelliJ IDEA.....	140
4.3.1. Инструмент естественно-языковой спецификации.....	141
4.3.2. Организация связи между концептами в тексте и соответствующими им классами.....	144
4.4. Построение синтактико-семантического и концептуального графов..	145
Глава 5.....	147
5.1. Создание автоматических тестов на основе текстовых пользовательских сообщений.....	147
5.2. Учебно-игровой аспект: пример реализации виртуальной трехмерной комнаты с объемными примитивами и имитацией гравитации.....	157
5.3. Автоматическое построение генеалогического древа по текстовому описанию родословных отношений.....	159

Заключение.....	163
Список литературы.....	166

Основные обозначения и сокращения.

АПН – Автоматическое понимание текста

АОТ – Автоматическая обработка текста

ЕЯ - Естественно-языковой

ОО – Объектно-ориентированный

ООЯП – Объектно-ориентированный язык программирования

МИЭМ – Московский Институт Электроники и Математики

ЭВМ – электронно-вычислительная машина

ЯПЗ – язык представления знаний

ЛП – лингвистический процессор

РСП – расширенные сети переходов

СинП – синтаксическое представление

СемП – семантическое представление

СинАн – синтаксический анализ

СемАн – семантический анализ

СемОт – семантическое отношение

ПО – программное обеспечение

Введение.

Актуальность проблемы.

Разработка современного программного обеспечения (ПО) носит, как правило, циклический характер, обусловленный потребностью заказчика в постоянном улучшении программного продукта: добавлении новой функциональности, а также изменением внутренней логики бизнес-процессов. Практика разработки коммерческого ПО показывает, что характер поддержки и доработки такого ПО после внедрения не затрагивает архитектуры системы и носит преимущественно проблемно-ориентированный характер. Современные тенденции решения данной задачи заключаются в разработке так называемых *малых языков* или *языков предметной области* (domain specific language – DSL) (М. Mernik, 2005). Между тем, необходимо отметить сходство концепции «малых языков» с идеями *концептуального программирования* – подходом, изложенным Э.Х. Тыугу еще в 80-х годах XX века (Тыугу, 1984). Данный подход в силу ряда причин не получил к настоящему моменту развития и широкого применения. Этому, по всей видимости, способствовал уровень развития вычислительной техники и компьютерной лингвистики в тот исторический период. Однако в настоящее время идеи концептуального программирования вновь набирают все большую популярность, в том числе и на Западе (например, язык программирования XL (Dinechin, 2011)). К этим идеям относятся следующие аспекты:

- программирование в терминах предметной области решаемых задач;
- использование ЭВМ на этапе постановки задач.

Близость подхода, применяемого в DSL и концептуальном программировании, заключается также в тенденции приблизить язык

программирования (или, говоря в терминах концептуального программирования, язык *описания задачи*) к *естественному языку*. Это позволяет, в конечном счете, не только максимально облегчить труд программиста, но и привлечь к такому «программированию» бизнес-аналитиков, незнакомых с обычными языками программирования и средствами разработки.

Таким образом, актуальность данной работы обусловлена двумя факторами. Во-первых, использование естественного языка в качестве языка программирования предметной области позволяет существенно упростить поддержку уже внедренного ПО и повысить ее эффективность. Во-вторых, ввиду сложности разработки предметно-ориентированных языков существует необходимость исследования и разработки системы, позволяющей создавать, поддерживать и встраивать такие *естественные языки предметной области* в существующие программные продукты.

Цели и задачи настоящего исследования.

Целью работы является разработка системы концептуального программирования с использованием лингвистического процессора, назначение которой - существенно упростить задачу создания и встраивания поддержки естественных предметно-ориентированных языков в программное обеспечение.

Для достижения поставленной цели были найдены решения следующих задач:

- анализ существующих моделей автоматического понимания текстов (АПТ); моделей естественно-языкового взаимодействия «человек-компьютер» и естественно-языкового программирования; языков и средств описания онтологий;

- выбор и синтез наиболее удачных подходов к автоматической обработке текстов (АОТ); выбор модели формализованного представления «смысла» текста;

- разработка концепции естественно-языкового спецификатора объектно-ориентированного языка программирования;

- разработка программного комплекса, включающего в себя лингвистический процессор и средства его настройки на предметную область; компоненты построения концептуального графа, кодогенерации, а также инструментарий естественно-языковой спецификации классов объектно-ориентированного языка программирования.

Методы исследования.

Методы исследования основаны на использовании технологии объектно-ориентированного программирования, технологии построения интерпретаторов языков программирования, принципов модели концептуальной зависимости и модели «мягкого понимания текстов», теории графов и теории множеств.

Научная новизна.

1. Предложен способ описания онтологии с помощью существующих объектно-ориентированных языков программирования; разработаны и реализованы соответствующие программные средства естественно-языкового спецификатора, что является дальнейшим развитием концептоцентрического подхода.

2. Предложен подход к представлению смысла текста в виде динамической объектной системы, состоящей из объектов – экземпляров классов языка Java.

3. Показана возможность обратного перехода от предложенной динамической объектной системы к концептуальному графу путем распознавания в объектной системе новых отношений и концептов, и переход от концептуального графа к естественно-языковому тексту.

4. Разработан визуальный редактор расширенных сетей переходов, имеющий встроенный морфологический анализатор, что позволяет производить поиск по имеющимся сетям и строить новые на основе вводимой фразы-шаблона.

5. В результате проведенной работы создана система, имеющая возможность модульной интеграции с существующей средой разработки программ на языке Java – IntelliJ IDEA, что позволяет использовать все современные средства интенционального программирования. Показана практическая возможность разработки естественных языков предметной области в системе Nalaps.

6. В качестве примера использования системы рассмотрен аспект автоматической генерации программных тестов для верификации исправленных ошибок в программных продуктах на основе предоставляемых в текстовом виде записей из систем отслеживания ошибок, а также система поддержки естественно-языковой настройки бизнес правил в банковском ПО.

Основные положения, выносимые на защиту.

1. Подход к представлению онтологической базы знаний, построенный на использовании классов объектно-ориентированного языка программирования и карт настройки классов на предметную область, описываемых на языке XML. Предложенный подход позволяет эффективным образом сочетать знания о понятиях предметной области с правилами и алгоритмами их обработки, выполняющимися автоматически при автоматической обработке текста.

2. Способ представления «смысла» исходного естественно-языкового запроса (текста) в виде динамической объектной системы, а также возможность обратного перехода от данного представления к концептуальному графу. Предложенный подход позволяет в реальном времени отслеживать состояние объектов автоматизации и организовывать естественно-языковой диалог с системой.

3. Техническая реализация предложенных подходов и построенных методов в программном комплексе Nalaps – системе концептуального программирования, которая предназначена для разработки базы знаний предметной области, программирования лингвистического процессора и непосредственной автоматической обработки текста, что позволяет встраивать в существующие программные продукты поддержку естественных языков предметной области.

Теоретическая ценность

Теоретическая ценность работы заключается в создании методических основ для построения базы знаний предметной области на основе классов объектно-ориентированных языков программирования. Описана математическая модель предложенного подхода.

Практическая ценность

Практическая ценность работы заключается в следующем:

1. Разработанная типовая структура программного обеспечения для организации связей классов языка Java с предметной областью, позволяет при необходимости создавать по предложенной методике аналогичные системы для других объектно-ориентированных языков программирования (C++, C#.net и др.)

2. Проведен ряд экспериментов, доказывающих практическую ценность созданного программного комплекса, а именно: создан ряд учебно-

игровых и практически применимых примеров использования системы Nalaps. Среди них:

- учебно-игровая среда управления трехмерными объектами путем подачи естественно-языковых текстовых команд;
- среда построения генеалогической базы данных и родословного древа путем текстового описания родственных связей между персонами;
- система автоматической генерации программных текстов на основе поступающих естественно-языковых описания ошибок и доработок в программном продукте.

3. Для эффективного использования системы программистами, она может быть встроена как программный модуль в среду разработки IntelliJ IDEA, что позволяет полностью интегрировать процесс создания базы знаний предметной области в процесс написания программных средств обработки этих знаний.

4. В качестве важных областей применения системы следует выделить следующие:

- создание естественно-языковых интерфейсов общения пользователя с программными средствами;
- создание естественно-языковых языков предметной области для внутренних нужд разработки;
- автоматическая обработка потоков текстовой информации на предмет регистрации определенных информационных сигналов (событий) и построение автоматической реакции на них;
- учебно-игровой аспект обучения объектно-ориентированному программированию.

5. Метод автоматической обработки текстов, разработанный в ходе диссертационных исследований, нашел свое практическое применение и в

других научно-исследовательских работах, в частности, в системах семантического поиска.

6. Создан инструмент программирования расширенных сетей переходов – визуальный редактор со встроенным морфологическим анализатором и внутренним скриптовым языком для описания семантических знаний.

Публикации и апробации работы.

По материалам диссертационной работы опубликовано 1 статья, 8 тезисов на Всероссийских и международных конференциях, получено 3 свидетельства на регистрацию программ для ЭВМ, в которых были использованы методы, разработанные в рамках диссертационной работы. Результаты настоящего исследования были представлены на следующих научных конференциях:

1. Научно-техническая конференция студентов, аспирантов и молодых специалистов МИЭМ в 2008 г.
2. Научно-техническая конференция студентов, аспирантов и молодых специалистов МИЭМ в 2009 г.
3. Международная конференция «Диалог 2009» (Бекасово, 27-31 мая 2009 г.)
4. Пятая азиатская международная школа-семинар «Проблемы оптимизации сложных систем» ИВМиМГ СОРАН в 2009 г.
5. Международная научная конференция Megaling 21-26 сентября 2009 г., Украина, Киев.
6. Научно-техническая конференция студентов, аспирантов и молодых специалистов МИЭМ в 2010 г.
7. Международная конференция «Диалог 2010».

8. 11th Annual Conference for Software Testing Professionals "Discover the Tester in You" (July 6, 2010, University of Latvia, Riga).

Структура диссертации и объем работы.

Диссертация изложена на 173 страницах машинописного текста и состоит из введения, обзора литературы, обсуждения результатов, экспериментальной части, выводов и списка литературы, включающего 73 источника. Диссертация иллюстрирована 44 рисунками и содержит и 4 таблицы.

Во введении обоснована актуальность работы, сформулирована цель и задачи диссертационного исследования, новизна и практическое значение полученных результатов, области и методы исследования, а также положения, выносимые на защиту, обоснованность и достоверность, реализация и апробация работы. Кратко изложено содержание диссертации по разделам.

В первой главе проводится обзор существующих средств морфологического анализа: парсера Mystem, компонента морфологического анализа системы ДИАЛИНГ, инструментария ABBYY Retrieval & Morphology Engine. Дано также описание методов и средств синтаксического анализа, таких как: система LinkParser, компонентов синтаксического анализа систем ЭТАП-3 и ДИАЛИНГ, аппарата расширенных сетей переходов. Проводится обзор подходов к семантическому анализу: систем ПОЛИТЕКСТ и ДИАЛИНГ. Рассматриваются существующие модели автоматического понимания текстов, обосновывается выбор модели мягкого «понимания» текста для поставленных задач. Проводится обзор таких близких подходов в программировании, как литературное программирование, языко-ориентированное программирование и концептуальное программирование, а также существующих программных

реализаций упомянутых парадигм. Дается обзор моделей и построенных на их основе программных средств извлечения знаний из текста (процессора Semantix и системы «Автоматический курьер»). В конце главы формулируются выводы. На основе проведенного обзора существующих методов, моделей и подходов, определяется и обосновывается выбор модели концептуальной зависимости в качестве семантической модели языка и концептоцентрического подхода для конструирования базы знаний.

Вторая глава посвящена подробному описанию архитектуры системы концептуального программирования Nalaps, разработанной в рамках диссертационного исследования. В данной главе приведены описания методик построения концептуальной базы знаний, естественно-языкового спецификатора языка Java, алгоритмов автоматической генерации программы на языке Java из концептуального графа и описание динамической объектной системы, служащей представлением смысла исходного текста.

В третьей главе предлагается такая структура базы концептуальных знаний, где в качестве *концепта* предлагается рассматривать описание класса объектно-ориентированного языка программирования, а настройку на предметную область и конкретный естественный язык производить с помощью конфигурационных файлов на языке XML. Обоснованием предложенного подхода является близость понятий класса из объектно-ориентированной модели языка программирования и лингвистического понятия концепта. С позиции концептуального подхода взаимосвязь между понятиями внеязыковой действительности выражается в классах объектно-ориентированных языков программирования. Данная взаимосвязь определяется в виде наличия типизированных полей, методов, типизированных по параметрам и возвращаемым значениям. Связь с

естественно-языковыми терминами, описывающими концептуальные знания, представляется на языке XML.

Работа концептуального процессора описывается с приведением примеров его работы. Основополагающие моменты, такие как автоматический переход от формализованного представления текста к его программной интерпретации на языке Java, оформлены в виде математической модели в основу которой положен аппарат теории множеств и теории графов.

В четвертой главе дается подробное описание интерфейсов системы Nalaps: основного окна программы; модуля, подключаемого к среде разработки IntelliJ IDEA, естественно-языкового спецификатора.

Пятая глава посвящена описанию основных направлений применения системы Nalaps. Выбранные области и разработанные для них проекты в системе Nalaps иллюстрируются на простых примерах.

Глава 1. Обзор литературы.

Поскольку одним из основных компонентов построенной в рамках данного исследования системы Nalaps является лингвистический процессор, то представляется целесообразным провести предварительный обзор существующих методов и средств построения составных частей такого компонента. Классическая полная модель лингвистического процессора (Ножов, 2003) состоит из четырех основных анализаторов: графематического (внешнее представление текста), морфологического, синтаксического и семантического. Обзор методов графематического анализа не представляется существенным в контексте данного исследования, поэтому первая часть обзора будет посвящена морфологическим анализаторам.

1.1. Обзор методов и средств морфологического анализа

В данном разделе приводится обзор существующих средств морфологического анализа. Поскольку морфологический анализ является наиболее тщательно исследованной областью компьютерной лингвистики, при проектировании системы Nalaps было принято решение об использовании в качестве морфологического компонента уже существующего программного обеспечения. Обоснование выбора такого программного средства приводится в рамках проведенного обзора средств.

1.1.1. Парсер Mystem

Обзор средств морфологического анализа позволительно начать с программы Mystem, которая и была выбрана в качестве соответствующего компонента лингвистического процессора системы Nalaps¹.

Программа Mystem разработана компанией Яндекс и представляет собой так называемый *стеммер*, т.е. средство, осуществляющее процесс

¹ Официальное разрешение от компании Yandex на включение парсера «Mystem» в состав сборки системы Nalaps получено 2 апр. 2010 года.

стемминга. Алгоритм стемминга представляет собой приведение всех слов с одинаковым корнем к общей форме, обычно путем лишения слова его формообразовательных и склоняемых суффиксов (Lovins, 1968). Иными словами, процесс стемминга – это нахождение основы слова для заданной словоформы. Однако Mystem не является просто стеммером, он умеет возвращать еще и информацию о всех лексемах, к которым принадлежит введенная словоформа, множество грамматических признаков данной лексемы и множество грамматических признаков данной словоформы. Кроме того, для неизвестных слов Mystem порождает гипотезы и выдает информацию о нестандартности грамматической формы.

Теоретические принципы работы парсера Mystem описаны в (Segalovich, 2003). Решения, принятые при организации словаря парсера, в основном опираются на морфологическую модель, представленную в «Грамматическом словаре» А.А. Зализняка (Зализняк, 1977).

Технически парсер Mystem выполнен в виде консольного приложения, т.е. работа с ним возможна только путем передачи во входной текстовый поток программы параметров (словоформ) и получения из текстового потока вывода формализованного результата работы.

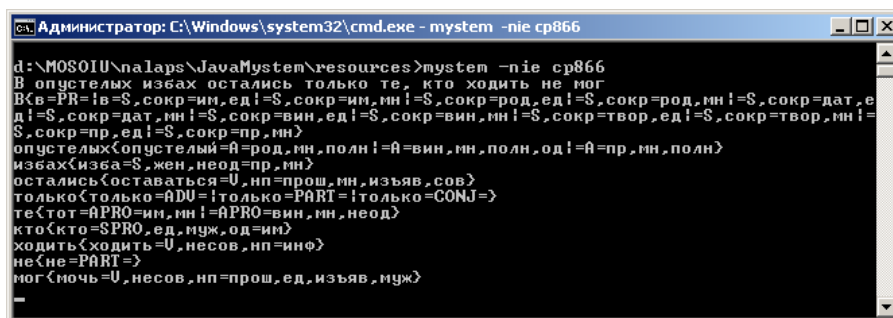


Рисунок 1. Пример экрана с результатом работы парсера Mystem

Результат анализа каждой словоформы парсер возвращает в виде строки, носящей аббревиатурный формализованный характер. Так, например,

для словоформы «избах» результат работы парсера выглядит следующим образом:

избах{*изба*=*S*,*жен*,*неод*=*пр*,*мн*}

где *S* обозначает часть речи (существительное), *жен* – женский род, *неод* – неодушевленность, *пр* – предложный падеж словоформы, *мн* – множественное число словоформы.

Отметим преимущества парсера Mystem:

- высокая скорость работы;
- бесплатная версия программы для некоммерческих проектов;
- возможность получения лексем для словоформ, а также определение наличия дополнительных признаков таких, как одушевленность, географическое название, имя собственное, отчество, фамилия и пр.;

- сравнительно большой объем внутреннего словаря программы;
- реализация программы для нескольких операционных систем: Windows, Linux, FreeBSD и Mac OS X 10.5.

Между тем, программа обладает и некоторым недостатками:

- единственно возможный способ взаимодействия с программой – консольный режим. В связи с чем при разработке системы Nalaps было необходимо создать программную обертку для вызова Mystem-a. Такой способ взаимодействия несколько уменьшает производительность системы в целом;

- отсутствие четкой документации о правилах формирования строки результатов грамматического разбора. Так, путем эмпирического анализа в рамках проведенного исследования удалось установить наборы правил, формирующих строки разбора и реализовать в системе Nalaps процесс преобразования строки разбора Mystem-a в формализованный вид;

- отсутствие поддержки иных языков кроме русского в последней версии. Данный недостаток, однако, не существенен, так как в настоящее

исследование базировалось на обработке естественно-языковых текстов на русском языке;

- невозможность вносить изменения в словарь парсера (изменять старые, добавлять новые входы и статьи).

При всех перечисленных недостатках парсер Mystem является все же наиболее оптимальным и удобным в использовании средством морфологического анализа из существующих в открытом доступе инструментов, почему он и был выбран в качестве компонента морфологического анализа в системе Nalaps.

1.1.2. Компонент морфологического анализа системы ДИАЛИНГ

Русский морфологический словарь системы ДИАЛИНГ, на котором построен компонент морфологического анализа, также как Mystem, базируется на грамматическом словаре А.А.Зализняка (Зализняк, 1977) и включает на данный момент 161 тыс. лемм.

Описание принципов работы морфологического модуля дано в (Сокирко, 2004). Отметим основные моменты, отличающие данную разработку:

- модуль обладает словарями достаточно большого объема, доступен развитый интерфейс пополнения словарей;
- при поиске в словаре программа использует конечный автомат, что позволяет находить слово за линейное от его длины время;
- имеет в распоряжении русский, немецкий и английский лексиконы;
- распространяется бесплатно под лицензией LGPL в исходных кодах;
- реализован компонент предсказания морфологических признаков для ненайденных в словаре слов.

Единственным «недостатком» описываемого компонента является его реализация на языке C++ в виде DLL библиотек. Разумеется, «недостатком» это можно назвать только с точки зрения интеграции данного компонента в рамках настоящего исследования с разработанной системой Nalaps: система Nalaps реализована на языке Java, а потому данный аспект исключает возможность кросс-платформенной интеграции с морфологическим компонентом системы Диалинг. Такая интеграция ограничивала бы платформу разработки только операционной системой Windows.

1.2.3. ABBYY Retrieval & Morphology (ARM) Engine

ABBYY Retrieval & Morphology (ARM) Engine (Анисимович К., 2000) – это инструментарий разработчика программного обеспечения, предназначенный для интеграции технологий полнотекстового поиска и морфологического анализа компании ABBYY Software House в локальные и клиент-серверные приложения разработчика через 32-битный COM (Component Object Model) API (Application Programming Interface).

Данный программный продукт, относящийся к классу "engines", то есть, предоставляет пользователю некоторые технологии не в составе прикладных программных продуктов, а в "чистом" виде. Технологии полнотекстового поиска и морфологического анализа ABBYY Retrieval & Morphology Engine могут быть встроены в любые программные системы и комплексы как готовые решения.

Morphology API ARM Engine дает разработчику доступ к функциям морфологического анализа для 30 языков (с разным объемом интерфейса для разных групп языков). Функции Morphology API включают:

- проверка орфографии (функции Check и Suggest);
- лемматизация (определение словарной формы по произвольной форме);
- определение словоизменительного типа несловарного слова;

- генерация парадигмы по произвольной словоформе;
- грамматическая атрибуция произвольной словоформы;
- расширение словаря.

Однако данная сравнительно старая разработка компании ABBYY больше официально не поддерживается. К сожалению, в последних версиях продукта присутствует большое количество программных ошибок, что является основным недостатком данного средства.

Кроме того, подобно морфологическому компоненту системы Диалинг, API данного продукта ориентирован на разработку исключительно под операционные системы семейства Windows, что ограничило бы применение системы Nalaps с данным компонентом.

Отметим также, что ABBYY Retrieval & Morphology (ARM) Engine является коммерческим платным продуктом, что делает затруднительным его применение в бесплатных научно-исследовательских целях.

1.2.4. Универсальный морфосемантический словарь.

В данном подпункте описывается концепция универсального морфосемантического словаря, программная реализация которого предоставляет возможность морфологического анализа на весьма высоком уровне скорости – около 200000 слов в секунду (Елкин, и др., 2003).

Для достижения упомянутой скорости авторы системы использовали метод цифрового поиска, заключающийся в представлении ключа в виде последовательности символов, принадлежащих рассматриваемому алфавиту, а структуры хранения данных в виде М-арного дерева, где М равно количеству символов в алфавите. Каждый узел уровня L является набором всех ключей, начинающихся с определенной последовательности L символов. Узел определяет разветвление на М путей в зависимости от (L+1)-го символа. При высокой скорости работы данного метода ($\log_M N$, где N – кол-во ключей – основ или постфиксов, а М – размер вектора указателей) у

него есть и существенный недостаток, указываемый самим авторами системы, - это огромный объем памяти требуемый для его работы. В целях экономии памяти применяются такие методики, как сокращение размера алфавита и создание таблицы перекодировки символов.

Существенно сократить расход ресурсов памяти позволил метод перехода от М-арного дерева к бинарному при выполнении некоторых условий. При построении бинарного дерева учитываются частоты появления символов в позиции $L+1$ для каждого множества ключей, начинающихся с рассматриваемого префикса. По простым вероятностным правилам вычисляется *цена дерева*, и оптимальным считается дерево с минимальным значением такой цены.

Семантическая компонента словаря заключается в закреплении семантической информации за отдельными словами. Так, одно понятие может соответствовать нескольким словам и наоборот. Определив, какому слову из морфологического словаря соответствует данная строка, можно определить набор понятий, связанных с данным словом. Целью такой семантической компоненты является повышение производительности при обработке текстов за счет сокращения временных затрат на поиск семантической информации в тезаурусе. За счет получаемой семантической информации появляется возможность выявления глубинных падежей еще на этапе синтаксического анализа, что позволяет повысить качество и скорость перевода.

Программная реализация данной системы осуществлена на языке C++ и предоставляется в виде интерфейса COM-объектов или отдельного DLL. Данный факт в сочетании с особенностями набора выходных параметров и частей речи делает, к сожалению, невозможным применение данного компонента в системе Nalaps.

1.2. Обзор методов и средств синтаксического анализа

Синтаксический анализ является, вероятно, вторым после морфологического анализа наиболее исследованным направлением в компьютерной лингвистике. Однако практическая сторона исследования показала целесообразность создания собственных средств синтаксического анализа, базируясь на уже разработанных в лингвистике моделях и методах. В связи с этим в данном разделе будет произведен анализ существующих компонентов на предмет используемых в них методов для реализации обсуждаемой задачи.

1.2.1. Система LinkParser

Проект Link Grammar Parser (LinkParser) создан группой американских ученых (Davy Temperley, Daniel Sleator, John Lafferty) в Университете Карнеги – Меллон (Carnegie Mellon University, CMU). Будучи разработан в начале 90-х годов, LinkParser развивается до сих пор (последнее обновление на официальном сервере проекта на момент написания данной работы датировался августом 2009 г.). Используемая в данной системе модель анализа основана на контекстно-свободной грамматике (Daniel D. K. Sleator, Davy Temperley, 1991).

Название системы «Грамматика Связей²» (Link Grammar) содержит в себе основную, заложенную в нее идею. Грамматика связей состоит из набора слов (терминальных символов грамматики), каждый из которых состоит из т.н. *потребностей связываний*. Цепочка слов определяется как *предложение* языка определенное грамматикой, если существует способ выстроить связи среди слов так, чтобы были выполнены следующие условия:

- 1) *Плоскостность* (planarity): связи не должны пересекаться, когда они изображаются в виде дуг над словами.

² Некоторыми исследователями предложен перевод «Грамматика Соединений» (Ножов, 2003)

2) *Связность*: связей должно быть достаточно, чтобы соединить все слова в предложении вместе.

3) *Удовлетворенность* (satisfaction): связи должны удовлетворять *потребности связываний* всех слов в предложении.

Наборы таких потребностей связываний для каждого слова хранятся в словарях и схематично изображаются, как показано на рис. 2.

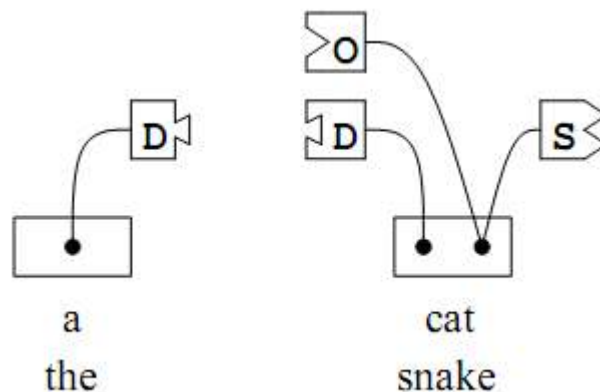


Рисунок 2. Схематичное изображение потребностей связываний в LinkParser

Каждая *необходимость связывания* содержит несколько видов *коннекторов* (на рис. 2. Изображены в виде разнородных многоугольников с латинскими буквами внутри). Коннекторы могут соединяться только с их комплиментарными частями, где комплиментарность условно определяется тем же набором символов внутри и схематично изображается как подходящие друг к другу выпуклые и вогнутые части многоугольников. Набор латинских букв определяет тип связи (S – субъект, O – объект и т.д.). Словарь системы, таким образом, содержит для каждой словоформы формулу, которая выражает *потребность связываний* и записана в удобном для обработки компьютером виде. Язык данных формул имеет три оператора: конъюнкции – «&», дизъюнкции – «or» и оператор неограниченного повторения – «@»; в языке также присутствуют фигурные скобки «{}» для обозначения необязательных (факультативных) связей.

Кроме того, знаки «+» и «-» стоящие после имени связи определяют вектор направления соединения – право-направленный и лево-направленный коннектор, соответственно. Для примера рассмотрим фрагмент словаря системы LinkParser. Словоформы, обладающие одинаковой формулой, записываются в одной ячейке колонки «Словоформа» и разделяются пробелами:

Словоформа	Формула
the a	D+
cat dog	D- & {O- or S+}
chased	S- & O+

На основе этих словарных статей результат разбора предложения «*The dog chased a cat*» («Собака преследовала кошку») будет выглядеть следующим образом (рис. 3):

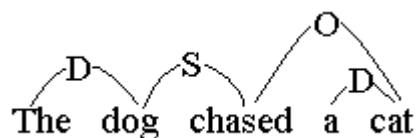


Рисунок 3. Пример разбора предложения

Так, словарная статья для слова «the» говорит о том, что у данного слова присутствует обязательная право-направленная связь «D», и артикль «the» таким образом употребляться сам по себе без существительного не может. Аналогичным образом у слова «dog» присутствует обязательная лево-направленная связь «D», требующая наличия артикля. При этом у того же слова «dog» связи типа Субъект (S) и Объект (O) являются факультативными и каждая из них может отсутствовать.

Оригинальность Грамматики Связей заключается в том, что авторы системы используют формализм для описания структуры английского языка

идеологически близкий к теории зависимостей, описанный в работах И. Мельчука. Бинарные связи, осуществляемые соединением однотипных коннекторов, не содержат вершин и не имеют направления, в отличие от формы представления в виде деревьев зависимостей.

В силу ряда причин использование системы LinkParser, также модели грамматики, на которой она основана, для русского языка представляется невозможным. Среди таких причин исследователями выделены следующие (Ножов, 2003):

- Использование право- и лево-направленных коннекторов не имеет смысла для русского языка, в силу относительно свободного направления связей (например, в глагольных группах). Так, невозможное в английском *~«chased the cat a dog»* вполне допустимо в русском: *«преследовала кошка собаку»*. «Спасти положение» не удастся даже путем маркировки всевозможных типов коннекторов с разным направлением – это значительно замедлит работу анализатора в целом и сведет целесообразность его использования к нулю.

- В силу морфологической развитости флективного языка словарь системы будет испытывать избыточность и иметь значительный объем: каждая морфологическая форма описывается отдельной формулой, где нижний индекс входящего в нее коннектора должен будет быть согласован, что усложнит процедуру составления коннекторов и приведет к увеличению их общего числа в грамматике. Данный аспект также существенно скажется на производительности системы.

Добавим к вышеприведенному списку и технический аспект: программный комплекс Link Grammar написан на языке ANSI-C, что сделало бы процесс его интеграции с системой Nalaps трудоемким и малоэффективным.

1.2.2. Синтаксический анализатор системы ЭТАП-3

Система ЭТАП-3 представляет собой многоцелевой лингвистический процессор, разработанный в лаборатории компьютерной лингвистики Института проблем передачи информации РАН (см. о процессоре и его приложениях (Апреснян, и др., 1989)).

Синтаксическая структура фразы в процессоре ЭТАП-3 строится в виде дерева зависимостей. В основе анализа лежит использование специальных бинарных правил (синтагм) число которых для каждого из рабочих языков системы (русского и английского) оценивается в несколько сотен. Бинарность правил подразумевает, что любая синтагма позволяет связать некоторым синтаксическим отношением два слова фразы, если все условия этой синтагмы, описывающие контекст данной пары слов во фразе, выполнены. Омонимы слов фразы могут связываться независимо друг от друга: синтагма связывает синтаксическим отношением пару омонимов некоторых слов, если они представлены в начале синтаксического анализа несколькими (морфологическими и/или лексическими) омонимами (Дружкин, и др., 2008).

Интересной особенностью процессора ЭТАП-3 является возможность построения альтернативного синтаксического разбора, что достигается за счет полноты корпуса синтагм. Предполагается, что при достаточно большом числе итераций для большинства фраз рано или поздно появляется правильная структура. Для того чтобы правильная структура выдавалась системой в качестве первого и основного варианта в процессоре реализовано ранжирование синтаксических гипотез. Ранжирование задается на основе эмпирической статистики, вносимой в систему лингвистом-экспертом.

Отметим, что использование данного компонента в системе Nalaps невозможно в силу закрытости синтаксического анализатора системы

ЭТАП-3 - он не доступен для отдельного свободного использования в сторонних приложениях.

1.2.3. Компонент синтаксического анализа системы ДИАЛИНГ

Данный компонент предоставляется рабочей группой Aot.ru (Сокирко) в свободном доступе бесплатно в составе ознакомительной версии программы «Рабочее место лингвиста» (РМЛ) и основан на компоненте синтаксического анализа системы ДИАЛИНГ.

Целью синтаксического анализа, производимого описываемым компонентом, является построение синтаксических групп на одном морфологическом варианте одного простого предложения в составе сложного, называемого *клаузой*. Группы строятся с помощью синтаксических правил. Клауза в составе другой клаузы именуется *подклаузой*. С клаузой всегда связан некоторый набор вариантов вершины клаузы, где вершина клаузы - это некоторый омоним возможного слова данной клаузы. (Сокирко)

Под морфологическим вариантом клаузы подразумевается набор *юнитов*, идущих друг за другом в данной клаузе, где *юнит* – это либо морфологический омоним одного слова, либо вариант подклаузы данной клаузы.

Синтаксическая группа в РМЛ определяется следующими параметрами:

1. номер первого и последнего *юнита*;
2. тип группы (строка);
3. главная подгруппа;
4. граммеы группы.

Основная идея алгоритма синтаксического анализа в РМЛ заключается в использовании *синтаксических правил* строящих группу определенного типа. Всего таких правил в системе около 40 штук. Принцип работы синтаксического правила заключается в получении на входе слова, от

которого правило пытается построить новую группу определенного типа, соблюдая принцип *проективности*. На данный момент все правила пытаются объединить входную группу только с группами, находящимися от нее справа. Все правила упорядочены, поэтому синтаксическая омонимия игнорируется, т.е. строится всегда только один вариант.

В системе ДИАЛИНГ обеспечены многие параметры промышленных систем, таковой она все же не считается в силу того, что синтаксический и семантический анализ в данной системе нуждается в серьезной доработке и с содержательной и с технической точек зрения (Леонтьева, 2006).

Данный компонент также не был принят в качестве реализации синтаксического анализатора в системе Nalaps в силу тех же причин, почему не был принят морфологический компонент системы Диалинг: технические особенности ее реализации, не позволяющие гибким и эффективным образом интегрировать ее в Java-приложение и обеспечить кроссплатформенность.

1.2.4. Расширенные сети переходов

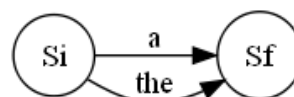
В завершении рассмотрения средств синтаксического разбора рассмотрим формализм *расширенных сетей переходов* (augmented transition network), который взят за основу реализации синтаксико-семантического анализа в системе Nalaps.

Расширенные сети переходов описываются в (Люгер, 2005) как полезные и мощные средства работы с естественным языком. В их основу расширенных положены обычные *сети переходов*. Каждая сеть соответствует одному нетерминальному элементу грамматики (например, *предложению*, или *группе сказуемого*). Сеть переходов представляет собой направленный граф, где дуги помечаются терминальными или нетерминальными символами. Пути от начального состояния к конечному соответствуют некоторому правилу для нетерминальных элементов. С помощью сетей переходов можно довольно наглядно описать простую грамматику английского языка, представленную на рис. 4.

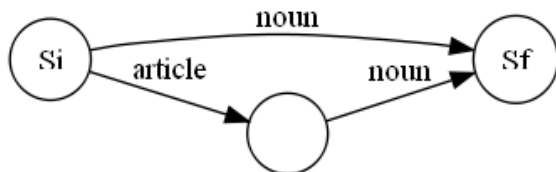
sentence:



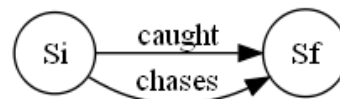
article:



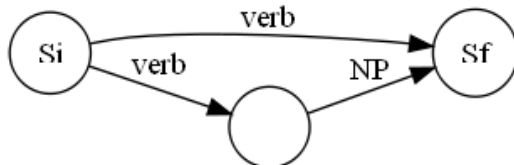
NP (noun phrase):



verb:



VP (verb phrase):



noun:

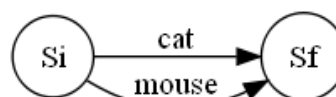


Рисунок 4. Определение сети переходов простой грамматики английского языка

Разбором предложения занимается анализатор, который представляет собой грамматику в виде набора конечных автоматов – сетей переходов. В начальном состоянии анализатор находится в узле S_i сети. Для прохода по дуге к следующему узлу анализатор проверяет ее метку. Для терминальных символов выполняется проверка совпадения очередного слова из входного потока с данным терминальным символом. Если терминальный символ соответствует, то переход осуществляется, в противном случае рассматриваются другие дуги, исходящие из текущего узла. Для нетерминальных символов, стоящих в качестве метки дуги, производится рекурсивный переход к сети, соответствующей данному нетерминальному символу и пытается найти путь в этой сети. Если путь найден, то происходит переход по метке в родительской сети, в противном случае переход также невозможен и происходит анализ следующей дуги и т.д. В конечном счете, если путь в сети *sentence* будет найден, то входной поток представляет собой корректное в данной грамматике предложение.

Работу анализатора можно продемонстрировать на простом примере разбора предложения “*The cat chases a mouse*” (“Кошка преследует мышку”):

- 1) Анализатор открывает сеть *sentence* и пытается осуществить переход по единственной дуге, выходящей из узла S_i – *NP* (noun phrase - именная конструкция).
- 2) Для этого анализатор открывает сеть *NP*. Первая дуга, выходящая из S_i данной сети – *noun*. Это также нетерминальный символ, поэтому:
- 3) Анализатор открывает сеть *noun* и не может найти ни одной подходящей дуги, =>
- 4) Анализатор переходит к рассмотрению следующей дуги в сети *NP* – *article*. Это опять же нетерминальный символ, поэтому:

5) Анализатор открывает сеть *article* и успешно находит по ней путь (дуга «the») => проход по дуге «article» в сети NP также возможен.

6) и т.д.

В конечном счете порядок прохождения будет выглядеть следующим образом:

Sentence [NP [article [**the**] -> noun [**cat**]] -> VP [verb [**chases**] -> NP [article [**a**] -> noun [**mouse**]]]].

Таким образом, простые сети переходов позволяют определять *контекстно-независимые языки (context-free language)* по иерархии Хомского (Chomsky, 1965), т.е. такие, которые основаны на правила вывода, где в левой части может располагаться только один нетерминальный символ. Разумеется, такая ограниченность не позволяет построить сколько-нибудь серьезный анализ естественного языка. Нас будут интересовать *контекстно-зависимые языки (context-sensitive language)*, составляющие супермножество контекстно-независимых языков. Определяющие их *контекстно-зависимые грамматики* позволяют использовать несколько символов в левой части правила для определения *контекста* применения этого правила. Контекстно-зависимые грамматики, однако, обладают рядом сложностей, появляющихся при реализации анализаторов на их основе, это:

- необходимость большого числа правил и нетерминальных символов для описания естественного языка;
- размывается структура фраз языка, более ясно представляемая с помощью контекстно-независимых правил;
- контекстно-зависимые грамматики сами по себе не позволяют строить семантическое представление смысла текста.

Решить вышеперечисленные проблемы позволяет формализм *расширенных сетей переходов*. Данный формализм сохраняет простоту структуры правил контекстно-независимой грамматики, но добавляет при

этом *расширение* этих правил за счет добавления специальных процедур, выполняющих необходимые контекстуальные проверки. Эти процедуры ассоциируются с дугами сети и выполняются АТN-анализатором при проходе по ним. Данные процедуры могут присваивать значения или выполнять проверки, запрещающие переход при невыполнении определенных критериев (например, наличия определенного падежа или рода у имен существительных).

Терминальные и нетерминальные символы представляются в данной концепции с помощью фреймоподобных структур с именами ячеек и их значениями, где в качестве значений указываются конкретные процедуры или указатели на другие структуры. У Д. Люгера (Люгер, 2005) отдельные слова предлагается определять также с помощью фреймов, содержащихся в словаре. Данные фреймы описывают набор морфологических и грамматических признаков слова (часть речи, неопределенная форма и присущие данной части речи грамматические свойства). Отметим, что данное понимание словаря не относится органически к теории АТN-анализаторов и его следует отнести к области морфологического анализа, выполняемого на более раннем этапе.

Для нетерминальных символов указанные процедуры описывают способ заполнения определенных ячеек фреймовой структуры на основании значений свойств конечных терминальных символов. Например, для именной конструкции данный фрейм может иметь вид:

Именная конструкция
Determiner:
Noun:
Number:

Тогда, записанная на псевдо-коде процедура **np1** для определения поля “Determiner” будет выглядеть как:

процедура np1;

начало

ARTICLE := фрейм определения для следующего входного слова;

Если ARTICLE.PART_OF_SPEACH = article, **то**

NOUN_PHRASE.DETERMINER := ARTICLE

иначе ОШИБКА

конец.

Данная процедура будет ассоциирована с дугой Si-> [article], см. рис. 5:

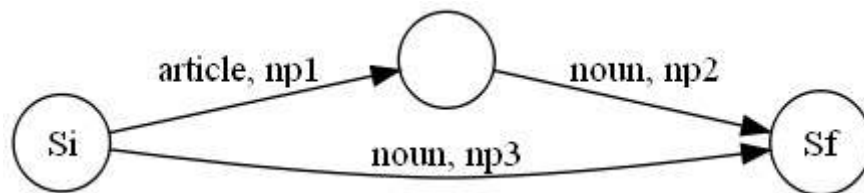


Рисунок 5. Пример расширенной сети переходов Noun phrase

В книге Д. Люгера также предлагается важное для данного диссертационного исследования положение о возможности объединения синтаксиса и знаний в АТН-анализаторах. В предложенной модели выделяются *объекты* и *действия*, описываемые в базе знаний. Важная роль отводится глаголам, определяющим отношения между подлежащими, дополнением и другими компонентами предложения. Каждый глагол представляется с помощью *надежного фрейма* (case frame), определяющего:

- 1) лингвистические отношения (объект, агент, инструмент и т.п.);
- 2) ограничения на смысловое употребление глаголов (например, что человек *не* может покусать собаку);
- 3) значения по умолчанию для компонентов фрейма, например, инструмент некоторого действия (например, «зубы» при укусе собакой).

Процедуры, генерирующие понятия и падежные фреймы также привязываются к дугам графа и вызываются при проходе по ним. В конечном счете, результатом семантического анализа становится концептуальный граф, где в качестве узлов сети используются понятия-объекты и понятия-отношения, заполняемые в соответствии с выделенными падежными фреймами.

Отметим, что, упомянутый в работах Д. Люгера *падежный фрейм* относится к методу *падежной грамматики*, автором которой является Ч. Филлмор (см. (Филмор, 1981)). В целом, данный метод описывает семантику предложения как систему семантических валентностей, диктуемыми вершинным глаголом. Данный подход получил свое развитие в теории «Смысл <-> Текст» И. Мельчука, где семантические валентности приписываются вообще каждому слову (см. (Мельчук, 1999), (Мельчук, 1995)).

В работе Пономаревой С.М. (Пономарева, 1999) описывается система анализа проблемно-ориентированных естественно-языковых текстов на основе эксплицитного представления знаний о предметной области. В данной системе предлагается реализация модуля поверхностного синтаксического анализа, построенная с помощью метаязыка расширенных сетей переходов. РСП представляется как граф с помеченными вершинами и дугами. Предлагается простейший подход к описанию такого графа в виде множества кустов. При этом расширенная форма Бэкуса-Наура (РБНФ), задающая метаязык РСП, определяется как:

```

<расширенная сеть переходов>::=(NET <описание сети>)
<описание сети>::=(<описание куста>) { <описание куста> }
<описание куста>::=(<состояние><дуга>{<дуга>})
<состояние>::=<идентификатор>

```

В качестве меток дуг рассматриваются терминальные символы (CAT-дуги), пустые символы (TST-дуги) и нетерминальные символы (PUSH-дуги). Заключительное состояние обозначается POP-дугами. На основании данной классификации даются следующие формировки для понятия «дуга»:

$$\begin{aligned} \langle \text{дуга} \rangle ::= & (\text{CAT} \langle \text{категория} \rangle \langle \text{условие} \rangle \{ \langle \text{действие} \rangle \} \langle \text{заключительное} \\ & \text{действие} \rangle) | (\text{TST} \langle \text{условие} \rangle \{ \langle \text{действие} \rangle \} \langle \text{заключительное} \quad \text{действие} \rangle) | \\ & (\text{PUSH} \langle \text{состояние} \rangle \langle \text{условие} \rangle \{ \langle \text{действие} \rangle \} \langle \text{заключительное} \quad \text{действие} \rangle) | \\ & (\text{POP} \langle \text{форма} \rangle \langle \text{условие} \rangle) \end{aligned}$$

Остальные компоненты определяются как:

$$\begin{aligned} \langle \text{условие} \rangle ::= & (\text{AND} \langle \text{условие} \rangle \{ \langle \text{условие} \rangle \}) | \\ & (\text{OR} \langle \text{условие} \rangle \{ \langle \text{условие} \rangle \}) | (\text{NOT} \langle \text{условие} \rangle) | (\text{EQ} \langle \text{форма} \rangle \langle \text{форма} \rangle) | \\ & (\text{EQOF} \langle \text{форма} \rangle \langle \text{форма} \rangle) | \text{T} | \text{NIL}, \text{ где EQ – проверка равенства, EQOF –} \\ & \text{проверка непустоты пересечения значений, T – истина, NIL – ложь.} \end{aligned}$$

$$\begin{aligned} \langle \text{действие} \rangle ::= & (\text{SETR} \langle \text{регистр} \rangle \langle \text{форма} \rangle) | (\text{SENR} \langle \text{регистр} \rangle \langle \text{форма} \rangle) \\ & | (\text{LIFTR} \langle \text{регистр} \rangle \langle \text{форма} \rangle) \end{aligned}$$

$$\begin{aligned} \langle \text{заключительное} \quad \text{действие} \rangle ::= & (\text{TO} \langle \text{состояние} \rangle) | (\text{OT} \langle \text{состояние} \rangle) | \\ & (\text{JUMP} \langle \text{состояние} \rangle) \end{aligned}$$

Формы определяются через набор функций от содержимого регистров.

$$\langle \text{фрагмент} \rangle ::= ([\langle \text{идентификатор} \rangle] + \{ [\langle \text{идентификатор} \rangle] [+] \})$$

$$\langle \text{регистр} \rangle ::= \langle \text{идентификатор} \rangle$$

Для трансляции данного метаязыка используется специальный ATN-конвертор, преобразующий конструкции ATN в конструкцию языка Рефал Плюс (Гурин Р.Ф., 1991): каждый ATN-куст конвертируется в одноименную Рефал-функцию, в которой каждой дуге соответствует одно Рефал-предложение. Такой способ описания ATN-сетей на метаязыке вместе с библиотекой ATN-Рефал-функций предлагается автором обсуждаемой работы для алгоритмов поверхностного синтаксического анализа, как не

требующий знаний об особенностях программной реализации системы в целом.

Между тем, отметим, что изучение такого метаязыка представляется несколько неудобным для экспертов-лингвистов и потребует от них определенных навыков программирования, пусть и не на полноценном языке программирования. Некоторый набор функций данного метаязыка, а также их характер, представляет собой, с нашей точки зрения, все же некоторую претензию на способности пользователей к алгоритмическому программированию, что не дает в конечном счете существенной оптимизации по сравнению с получаемым на выходе кодом программы на языке Рефал Плюс.

1.3. Обзор методов и средств семантического анализа

1.3.1. Система ПОЛИТЕКСТ

В книге Леонтьевой Н.Н. «Автоматическое понимание текстов: системы, модели, ресурсы» представлен авторский взгляд на состав и роль семантического компонента. Он рассматривается в контексте аспектов разработки экспериментальной лингвистической системы ПОЛИТЕКСТ, осуществляющей гибкое соединение лингвистических и предметных знаний.

На данном материале мы должны остановиться подробнее, т.к. модель *мягкого понимания текста*, представленная в данном источнике, была положена в основу метода, разработанного в настоящем диссертационном исследовании.

Проект ПОЛИТЕКСТ разрабатывался с начала 1990-х гг. в рамках Центра информационных исследований (ЦИИ) в Институте США и Канады РАН и был задуман, как единая многоканальная система понимания текстов и автоматического извлечения информации из текстов. Упомянутая выше

система ДИАЛИНГ была создана на основе лингвистической ветви проекта ПОЛИТЕКСТ.

В рассмотрение вводятся несколько типов текстовых структур в системах АПТ:

- Структуры типа А (лингвистические структуры предложений текста) – фиксируют результат локального, ограниченного пределами предложения, понимания текста. Главное достоинство таких структур в детальности их анализа, отражаемая в форме синтаксического и в идеале семантического деревьев, являющимися представлениями предложений (СинП и СемП соответственно). С точки зрения прикладного понимания такие лингвистические структуры не дают возможность понимания с обобщениями, они не допускают характерного для естественного процесса понимания выборочного подхода к интересующим человека частям содержания.
- Структуры типа Б (семантическая сеть целого текста). В чисто лингвистических системах реальный СемП целого текста имеет вид последовательности *синтаксико-семантических представлений* (СинСемП) всех предложений текста подряд (т.е. представлений, основанных на синтаксических деревьях, имеющих семантические узлы или связи). Получение структуры типа Б осуществляется за счет фиксации в СемП сведений о теме-реме или хотя бы референтных связей между структурами соседних предложений.
- Структуры типа В (информационные структуры целого текста) фиксируют результат глобального понимания текста и потоков текстов в единицах терминологии выбранной предметной области.
- Структуры типа Г (структуры баз данных и знаний) – специализированные, экстралингвистические структуры, отображающие

часть действительности. Они привлекаются при анализе естественного текста как дополнительное знание.

- Структуры типа Д (структуры систем машинного перевода) используются в системах и моделях автоматического перевода.

Все перечисленные структуры сочетаются в процессе восприятия и понимания текста человеком:

- а) пофразное, буквальное, поэлементное чтение и понимание ЕТ;
- б) столь же подробное понимание, но в масштабе текста и с отождествлением сущностей, упоминаемых в разных фразах;
- в) беглое чтение, выбирающее из ЕТ нужные терминологические элементы;
- г) чтение глазами специалиста, выхватывающее из ЕТ лишь те сюжеты, которые его интересуют;
- д) понимание иноязычного текста в единицах своего языка.

Модель «мягкого понимания» текста должна сочетать все эти аспекты, а также сочетать *информационный* и *лингвистический* подходы. В задачах настоящего диссертационного исследования мы будем рассматривать аспекты а) – г), т.е. все, кроме аспекта машинного перевода.

Предлагаемый семантический компонент (СемКомп) должен согласовывать три разных «языка»:

- а) язык построенных системой лингвистических структур, которые он получает на входе;
- б) язык той предметной области, к которой относится текст и термины;
- в) язык пользователя, для которого система АПТ должна получить Информацию.

В рассмотрение вводятся термины: *внутренние единицы* текста – единицы, строяемые лингвистическим анализом и *внешние единицы* – задаваемые системой понятий предметной области, заданной семантической

сетью или единицами индивидуальных знаний того, кто формулирует вопрос к тексту.

Окончательная структура текста рассматривается как граф, где в качестве вершин и ребер выступают *полноценные* единицы, т.е. наиболее значимые для содержания текста как целого и с учетом характера предметной области, задаваемой базой знаний.

Парадигма представляемого семантического компонента состоит из следующих пунктов (отметим только важные для нашего рассмотрения):

- 1) Метаязык для описания семантических структур.
- 2) Общая стратегия семантического анализа, которую задает управляющий алгоритм.
- 3) На вход СемКомп должен поступать результат хотя бы частичного синтаксического анализа – синтаксическое представление (СинП).
- 4) Правила интерпретации входных структур – правила построения и укрупнения единиц семантического представления (СемП).
- 5) Вычисление степени «полноценности» собираемых единиц СемП.

Итак, под метаязыком, называемым также *информационным языком-посредником (ИЯП)* понимается следующая концепция. Основной единицей языка является формула **$P(A, B)$** – описывающая, фактически, бинарное отношение P в котором находится A по отношению к B . В отличие от модели СемП «Смысл $\leftrightarrow$ Текст» (Мельчук, 1999) в данной модели используется язык не элементарных смыслов, а сложных семантических единиц – понятий, имеющих внутреннюю структуру. К элементарным смыслам в ИЯП относятся смысловые отношения типа *ПРИЧИНА, ЛОКАЛИЗАЦИЯ, ВРЕМЯ* и т.п.

При подстановке в место P, A, B в формулу семантического языка лексических выражений получается понятие *элементарной ситуации* ($\mathcal{E}Cum$).

В качестве *минимальной единицы* семантического анализа рассматривается выражение, удовлетворяющее вышеприведенной формуле. Под *максимальной единицей* понимается семантический граф, представляющий целый текст.

Семантический анализ текста делится на несколько этапов:

1) «Прямая» семантическая интерпретация СинП. Она заключается в построении СемП каждого отдельного предложения, заключающаяся в прямой и наиболее простой интерпретации построенного до этого синтаксического представления. Получаемый таким образом СемАн1 интерпретирует узлы и связи СинП1. Результат интерпретации узлов - это слова-значения, а интерпретация связей – это смысловые отношения между узлами СемП, типа *ВРЕМЯ* (...,...), *ЛОК*(...,...), и др.

Первичный СемАн строит *синтаксико-семантическое представление* (узлы и связи СинП получают семантические имена).

Выделяются тривиальные правила интерпретации единиц СинП:

а) каждое синтаксическое отношение переходит в какое-либо семантическое отношение, например:

подлежащее (а,в) => АГЕНС (А,В);

определение (а,в) => ПРИЗНАК(А,В);

и т.д.

б) синтаксические группы (именная или глагольная) становятся первичными семантическими узлами (СемУ).

в) главное слово синтаксической группы объявляется лексическим ядром (ЛЯ) семантического узла.

В рассмотрение вводятся *сильные* связи, а именно – те СемО, которые входят в описание значений лексем в качестве их валентностей в семантическом словаре. Данный семантический словарь вводит в рассмотрение семантические категории и смысловые валентности единиц, образующих словарные входы. Лексему, имеющую валентности, принято называть *предикатным словом*. Лексемы, заполняющие СемО, носят название *актантов*, а совокупность актантов – *актантной структуры*. Наибольшим числом валентностей обладают слова, описывающие ситуации (СИТ). Таким образом, интерпретирующие алгоритмы достраивают СинП до СинСемП со всеми возможными вариантами интерпретации узлов и связей.

Слабые связи определяются как те, которые нельзя определить механизмом поиска *сильных связей*. Слабыми оказываются такие синтаксические группы, которые остались невостребованными в качестве зависимых при заполнении валентностей однозначных лексем. Инструментом интерпретации слабых связей является словарь отношений – грамматика слабых связей, входом которого является пара наборов характеристик слов А и В, а выходом – смысловые отношения между А и В.

Следующий уровень семантического анализа текста – глобальный, который позволяет строить узлы структуры из материала разных предложений, получая, таким образом, единый связный граф. Такой анализ сопровождается обычно *сжатием* лексического материала, для получения избыточного представления. Отметим основные моменты данного уровня анализа:

- Проблема референции – построение отношений между двумя текстовыми единицами – *референтной связи*. Это, в частности, нахождение антецедентов и постцедентов местоимений.

- Использование *фреймов* – части схемы/сцены предметной области, словозначение вместе с набором смысловых валентностей, задающих роли участников какой-то абстрактной связной ситуации (слоты).
- Построение более крупных единиц семантического пространства, чем семантические узлы – объекты, ситуации и *текстовые факты*. Под *ситуацией* понимается структура, репрезентирующая содержание минимального текстового высказывания в терминах и в соответствии с грамматикой заданного семантического языка.

1.3.2. Компонент семантического анализа системы ДИАЛИНГ

В системе ДИАЛИНГ применяется алгоритм *поверхностного семантического анализа* (Сокирко, 2001). Первичный семантический анализ в данном компоненте представляет собой семантический анализ отдельных предложений, результатом которого является семантическая структура. Семантическая структура состоит из семантических узлов, понятие которых базируется на работах Н.Н. Леонтьевой, и семантических отношений. Под семантическим узлом понимается такой объект текстовой семантики, у которого заполнены все валентности, как эксплицитно выраженные в тексте, так и имплицитные – те, которые получаются из экстралингвистических источников.

На вход семантическому анализатору системы ДИАЛИНГ поступает синтаксическое представление одного предложения на русском языке и все семантические словари и тезаурусы, реализованные в системе ДИАЛИНГ, а именно: словарь оборотов, словарь групп времен (TimeRoss), локативный тезаурус, финансовый тезаурус, компьютерный тезаурус, словарь устойчивых словосочетаний.

На выходе, каждому узлу получаемой семантической структуры, приписывается множество атрибутов, таких как: набор графематический слов, из которых состоит данный узел; номер семантически главного слова в

узле; грамматическая интерпретация узла; номер клаузы, которому принадлежит узел; предлог, который в синтаксисе управлял этим узлом; ссылка на словарную статью в семантических словаря, которая является интерпретацией этого узла; ссылка на словарную статью открытого словосочетания и номер элемента в поле СОСТАВ этого словосочетания и ряд флагов.

Семантические отношения также наделяются рядом параметров, а именно: ссылка на словарную статью, откуда было взято отношение и номер валентности в этой статье; перечень русских слов, которые являются лексическими реализациями этого отношения; русское синтаксическое отношение, являющееся реализацией этого сем. отношения, ряд флагов и условная длина.

Непосредственно алгоритм *поверхностного семантического анализа* выглядит следующим образом.

1. Для каждого синтаксического варианта фрагмента производится ряд операций:
 - 1.1. Инициализация семантических узлов,
 - 1.2. Построение множества словарных интерпретаций узлов,
 - 1.3. Построение групп времени,
 - 1.4. Построение узлов из словосочетаний в кавычках,
 - 1.5. Построение узлов типа «Друг друга»,
 - 1.6. Подключение операторов типа «не», «только»,
 - 1.7. Для каждого лексического варианта для данного фрагмента в свою очередь проводятся следующие операции:
 - 1.7.1. Построение устойчивых словосочетаний,
 - 1.7.2. Построение лексических функций-параметров,
 - 1.7.3. Установление отношений между локативными узлами
 - 1.7.4. Интерпретация тезаурусных должностей и организаций,

- 1.7.5. Построение графа гипотетических связей,
- 1.7.6. Построение множественных актантов,
- 1.7.7. Для каждого варианта деревьев для данного лексического варианта:
 - 1.7.7.1. Проверка по семантическим характеристикам,
 - 1.7.7.2. Проверка проективности,
 - 1.7.7.3. Другие структурные оценки,
 - 1.7.7.4. Построение отношений по умолчанию.
- 1.8. Для каждого варианта приписывания фрагментам лучших деревьев:
 - 1.8.1. Построение межклаузных связей,
 - 1.8.2. Построение анафорических связей,
 - 1.8.3. Оценка.

Пример результата работы семантического компонента по анализу предложения «Дом Пашкова стоял на пригорке» представлен на рис. XX.

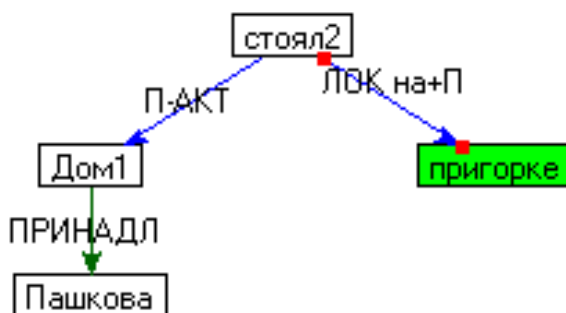


Рисунок 6. "Дом Пашкова стоял на пригорке"

В заключение рассмотрения данного семантического анализатора отметим:

- 1. Идеи, на которых построен данный семантический компонент, близки к концепции построения семантического компонента в системе ПОЛИТЕКСТ, т.к. базируются на работах Леонтьевой Н.Н.

2. В системе используется большой пакет словарей для выявления семантических отношений и заполнения валентностей слов.
3. Задачей семантического компонента в целом состоит в том, чтобы по возможности разрешить морфологическую, лексическую и древесную (синтаксического дерева) неоднозначность.

1.4. Системы извлечения знаний из текстов

Актуальность задачи извлечения знаний из текстов обусловлена тем, что огромные порции важной для людей информации существуют только в виде естественных текстов. В рамках настоящего исследования обзор данной задачи также важен, так как одной из ключевых возможностей разработанной системы Nalaps является именно *Information Extraction*.

1.4.1. Лингвистический процессор Semantix

Лингвистический процессор Semantix предназначен для областей, где требуется автоматическая обработка потоков текстов на естественном языке (ЕЯ): резюме, сообщения СМИ, информационно-рекламные материалы, почтовые сообщения, сводки происшествий, справки по уголовным делам, архивные материалы и др. (Кузнецов И.П., 2008) Процессор Semantix выделяет из текстовых документов интересующие пользователя объекты, их связи, участие объектов в действиях или событиях, которые также рассматриваются как комплексные объекты со свойствами и связями. В результате анализа документа строится семантическая сеть, которую можно экспортировать в XML-файл. Данные XML-файлы представляются как основа для составления досье, обзоров, отчетов, автоматического заполнения реляционных БД. С их помощью в системе формируется собственная База Знаний для организации поиска объектов, в том числе, различных видов семантического поиска.

Процессор Semantix состоит из следующих компонентов:

1) Блок лексического и морфологического анализа. Результатом его работы является пространственная структура текста, отображающая последовательность слов, их основные признаки, начало предложений и наличие пробельных строк. В блоке используется набор некоторых тематических словарей, таких как словарь стран, регионов России, имен, видов оружия и др. для группирования слов и придания им дополнительных семантических признаков.

2) Блок синтактико-семантического анализа. Данный блок преобразует пространственную структуру текста в семантическую структуру текста путем выделения объектов и их связей. В блоке используется массив лингвистических знаний, которые определяют процесс анализа текста. Основными задачами данного блока являются:

- Извлечение из потока ЕЯ-документов информационных объектов: лиц, организаций, действий, их места и времени, и т.п.
- Выявление связей объектов.
- Анализ глагольных форм, причастных и деепричастных оборотов с выявлением фактов участия объектов в соответствующих действиях.
- Выявление связей действий с объектами типа место или время (где и когда имело место данное действие или событие).
- Анализ причинно-следственных и временных связей между действиями и событиями.

3) Третий блок - Экспертные системы. На основе семантической структуры текста формируют новые знания - в виде дополнительных фрагментов расширенной семантической сети.

4) Блок обратного лингвистического процессора, генерирующий XML представление полученной семантической структуры текста и сохраняющий его в файл. При этом осуществляются необходимые замены

символов, служебных слов (имен объектов), выставляются метки начала и конца объектов, действий, предложений.

5) База лингвистических и экспертных знаний (БЗ) содержит правила анализа текста и экспертных решений, которые определяют работу лингвистического процессора. В состав системы входит несколько таких баз, которые активизируются в зависимости от предметной области и задач пользователя.

В системе существует общий набор типов выделяемых из текста информационных объектов и связей, которые носят характер служебной аналитической информации, большей частью ориентированной на обработку криминальных хроник, описаний катастроф, террористических актов и др.

Процессор Semantix обладает возможностью настройки на предметную область (Кузнецов И.П., 2009). Основными принципами данной настройки являются:

- 1) Создание каталога объектов и связей, присущих данной предметной области.
- 2) Описание правил выделения объектов. Для описания правил применяется собственный скриптовый язык, называемый *формализмом РСС* (расширенной семантической сети) и имеет вид выражения с левой и правой частью, где левая часть описывает условие применения правила, а правая – действие. Пример такого правила, взятый из (Кузнецов И.П., 2009), относится к предметной области *Памятники и места их расположения*, и описывает способ выделения объектов типа статуя императора Николая I, Фигура Петра Великого и т.д. :

MUSTBE(MONUM~2,1)

STR_OR(БЮСТ,СТАТУЯ,ФИГУРА,СОБОР,ЦЕРКОВЬ,ХРАМ/1+)

STR_OR(WORK_K,NAT_K,ИМПЕРАТОР,ЦЕСАРЕВИЧ,ГРАФ,КНЯЗЬ,ГЕРЦОГ,.../2+)

STR_OR(КОГО,КВЧ,MONUM_K,FIO,FAM/3+)

CONTEXT(1-,2-,3-/MONUM~2)

P_P(MONUM~2,MON~2L)

MONUM_(1,2,3/MON~2L) MON~2L(MONUM,ADD_)

MAYBE(MONUM~2,2)

Отметим, что система Semantix не представлена в свободном доступе, а официальный web-сайт, посвященный демо-версии данного продукта, на момент написания данного исследования не функционировал в должном режиме или вовсе был наполнен содержанием стороннего характера в течение более чем одного года.

Помимо закрытости системы, к ее недостаткам можно отнести следующие положения:

- сложный и специфичный способ описания правил выделения информационных объектов;
- невозможность настройки параллельной обработки выделенных ситуаций, объектов и связей по заданным алгоритмам (только построение графических связей и экспорт в XML формат).

1.4.2. Система извлечения знаний из документов «Аналитический курьер».

Система «Аналитический курьер» разработана компанией i-Тесо и является коммерческой системой аналитической обработки текстовой информации. Данная система имеет многослойную архитектуру с Web-интерфейсом, реализованным на Windows-платформе .NET. "Аналитический курьер" позволяет быстро погружаться в новые предметные области, которым посвящены тексты фонда, структурировать проблематику, готовить отчёты и информационно-аналитические материалы.

Классические методы тематического (кластерного) анализа, основанные на мерах близости между документами, разделяют поисковую выдачу на группы сходных документов – т.н. кластеры, в которых документы попарно похожи друг на друга, но «существо» сходства может отличаться у разных пар из одного кластера. Для повышения достоверности кластерного анализа в системе «Аналитический курьер» использован метод *бикластеризации* (объектно-признаковой, концептуальной кластеризации), в котором сходство объединяемых в один кластер документов выражается через общие структурные признаки (сущности, темы), выделенные из документов. Преимуществом метода является наличие всех тем кластера в каждом документе. Данный метод хорошо работает при небольшом числе документов в выборке. Для получения тематического дерева формируется решётка формальных понятий в виде двумерной матрицы, строкам которой соответствуют документы, а столбцам – сущности, извлеченные из них. Если документ содержит конкретную сущность, то на пересечении столбца и строки стоит частота ее появления в документе. Таким образом, данная матрица содержит всю информацию о взаимозависимостях, существующих между документами и сущностями. Визуальным представлением выявленных в решетке формальных понятий зависимостей является дерево кластеров документов. По оценке разработчиков быстроедействие метода бикластеризации практически не зависит от объема выборки документов. Скорость визуализации дерева зависит только от числа параллельно работающих пользователей.

Система также обладает возможностью автоматической аннотации, формируемой из фрагментов анализируемого документа. Объем такой аннотации зависит от главных тем документа и настроек. В аннотацию, генерируемую по выделенным объектам или проблемам могут включаться анафорические предложения документа. Имеется также компонент создания

общей аннотации на основе взаимосвязей тем и семантической сети подборки документов. (Николай Ильин, 2006)

1.5. Концептуальное программирование

Термин «Концептуальное программирование» появился в середине 80-х годов в работах знаменитого эстонского ученого Э.Х. Тыугу (Тыугу, 1984). Концептуальное программирование представляет собой способ решения задач на ЭВМ, заключающийся в описании понятий, достаточных для выражения смысла задачи, затем в терминах этих понятий описывается задача. По описанию задачи ЭВМ строит программу ее решения и выполняет вычисления.

Концептуальному программированию свойственно:

- программирование в терминах предметной области решаемых задач;
- использование ЭВМ уже на этапе постановки задачи;
- использование семантической памяти для накопления решаемых задач.

(Тыугу, 1984)

Концептоцентрический подход, вообще говоря, подразумевает, что в основу семантической классификации единиц естественного языка положена база знаний о классификации предметов и явлений внеязыковой действительности (ТАММ Б.Г., 1988).

Одной из первых реализаций языков концептуального программирования явился язык УТОПИСТ. Язык УТОПИСТ позволяет задавать условия задачи в виде совокупности описаний. Каждое описание задает абстрактный объект, который является абстракцией некоторой ситуации, отношения или действия. В ЭВМ абстрактный объект представляется вычислительной моделью, на которой можно решать задачи. Абстрактным объектом может быть все, что представимо вычислительной

моделью. Некоторые из абстрактных объектов имеют очень простое описание, например:

скорость: вещь.

Но большое число операций имеют сложную форму, как показывает пример на рис. 7. Внутри скобок задается все, что относится к данному объекту.

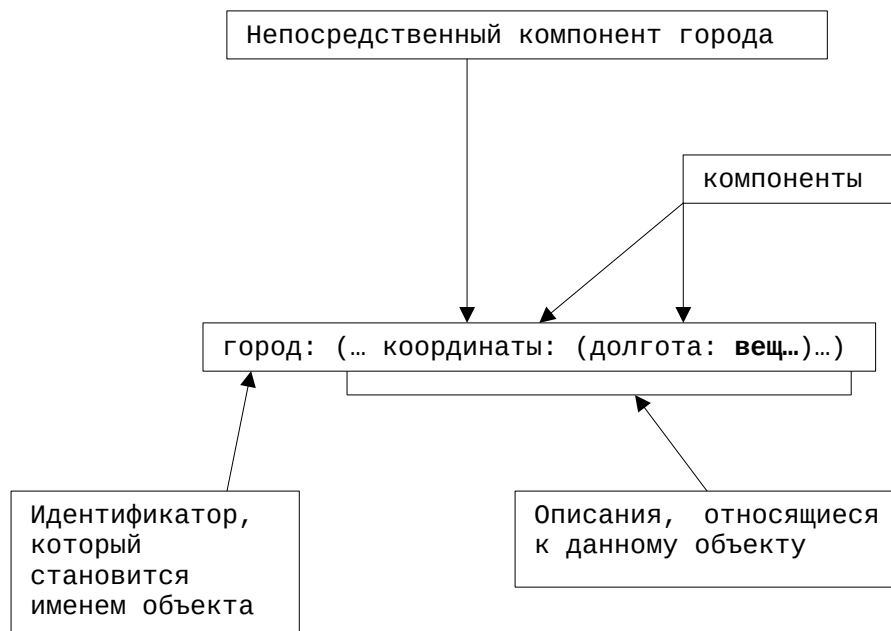


Рисунок 7. Структура абстрактного объекта.

В частности, там снова могут быть определены абстрактные объекты – компоненты первого объекта. На рис.7. координаты и долгота – компоненты объекта город.

Программа на языке УТОПИСТ состоит из абзацев, которыми бывают:

- описания обстановки;
- действия.

Описание обстановки начинается словом пусть, за которым следуют описания и отношения, которые представляют условия задачи. Действия

начинаются словом действия, за которым следуют операторы и управляющие предложения. Описания, отношения, операторы и управляющие предложения являются предложениями языка. Описания обстановки и действия могут чередоваться в программе в любом порядке.

Поддержка языка УТОПИСТ была произведена в системе ПРИЗ (Кахро, et al., 1988).

Синтез кода программы в системе ПРИЗ происходит на основе автоматически составляемого алгоритма решения задачи на так называемой вычислительной модели. Данная вычислительная модель использует дедуктивный подход к синтезу программ. При этом путь от задачи к программе состоит из следующих шагов: задача $\rightarrow$ теорема существования решения $\rightarrow$ доказательство теоремы $\rightarrow$ программа. В сущности, вычислительная модель представляет собой модель предметной области и формируется заранее. Она состоит из совокупности переменных, соответствующих понятиям предметной области, и отношений вычислимости. Каждое из отношений вычислимости связывает значения нескольких переменных модели и интерпретируется так: по известным значениям одних переменных (являющихся входными) можно вычислить значения других (выходных) переменных. Задача на вычислительной модели состоит в нахождении значений некоторых переменных по известным значениям других переменных на основе отношений вычислимости.

Кроме того, в системе ПРИЗ применен ряд подходов из области искусственного интеллекта, а именно:

1. Семантика входного языка описывается семантическими моделями, содержащими объекты и отношения. Обработка входного текста – перевод его в семантическое представление и работа с семантическими моделями.

2. Расширение входного языка осуществляется с помощью описания новых понятий через существующие (Кахро, et al., 1988).

Идеи, заложенные в систему ПРИЗ и язык УТОПИСТ несомненно опередили свое время и заложили основы принципов концептуального программирования. В системе был реализован принцип ветвления правил логики, лежащих в основе построения циклических и рекурсивных программ, что стало важным шагом в автоматизации синтеза программ. Однако, необходимо отметить, что система ПРИЗ, к сожалению, не получила распространения и прямого развития. К ее недостаткам исследователи подходов к автоматическому синтезу программ, относят тот факт, что для работы синтеза значительная часть работы, а именно, описание вычислительной модели, должна быть проделана заранее. В то же время, при решении задач из другой области, дополнительная информация снова добавляется в систему, что затрудняет ее расширение (Корухова, 2005).

1.6. Языко-ориентированное программирование

Парадигма «языко-ориентированного программирования» является своего рода синтезом идей «концептуального программирования» и «литературного программирования» (Дмитриев, 2005). Идею «литературного программирования», выдвинутую Дональдом Кнутом, можно охарактеризовать словами самого автора концепции: *“Давайте изменим традиционные приоритеты в создании программ: вместо представления о нашей задаче как о создании инструкций "Что делать?" для компьютера сконцентрируемся на объяснении другим людям описаний нашего видения того, что под управлением программы должен делать компьютер.”* (Knuth, 1992)

Идея языко-ориентированного программирования исходит из следующих соображений. Программиста ограничивает зависимость от инфраструктуры, которую нельзя изменить — в частности, языков

программирования и сред разработки. Свободу программиста ограничивают зависимости от авторов языка программирования и производителей сред разработки (IDE). Написание собственных компиляторов или интегрированных сред разработки отнимает *огромное* количество времени и сил и совершенно непрактично.

Чтобы увеличить степень свободы, нужно уменьшить степень зависимости. Например, одна из главных целей Java – уменьшить зависимость от операционной системы и дать разработчикам свободу распространения программ для любых ОС. Таким образом для освобождения от языков и сред необходимо уменьшить зависимость программиста от них.

Проблема заключается в том, что любой язык общего назначения (будь то Java или C++) в теории дает возможность сделать с компьютером что угодно. Однако языки общего назначения с точки зрения языко-ориентированного подхода недостаточно продуктивны. Как альтернатива, существуют языки, специфичные для предметной области (DSL, они же “малые языки”), чья цель – максимальная продуктивность в своей области – как, например, SQL для баз данных. Сила этих языков – специфичность для предметной области – в то же время и их слабость, ведь реальные программы работают с множеством предметных областей.

Методику современного преобладающего способа программирования можно описать следующими последовательными шагами:

1. **Обдумывание:** есть задача, которую надо запрограммировать; программист формирует в своем сознании концептуальную модель решения.
2. **Выбор:** программист выбирает некий язык программирования общего назначения (например, Java или C++), на котором будет написано решение.

3. Непосредственное программирование: программист создает программное решение, выполняя сложнейшее отображение своей концептуальной модели на выбранный язык.

Шаг «*непосредственного программирования*» рассматривается в языко-ориентированном подходе как узкое место, так как отображение концептуальной модели решения на язык программирования выполняется в большинстве случаев не естественным образом (рис.3). Концепция языко-ориентированного программирования предлагает следующую последовательность шагов решения задач:

1. Обдумывание: есть задача, которую надо запрограммировать; программист формирует в своем сознании концептуальную модель решения.
2. Выбор: программист выбирает некоторые специальные DSL, на которых будет написано решение.
3. Создание DSL: если нужных DSL еще не существует, программист их создает.
4. Непосредственно программирование: программист пишет решение, являющееся относительно прямым отображением концептуальной модели на DSL.

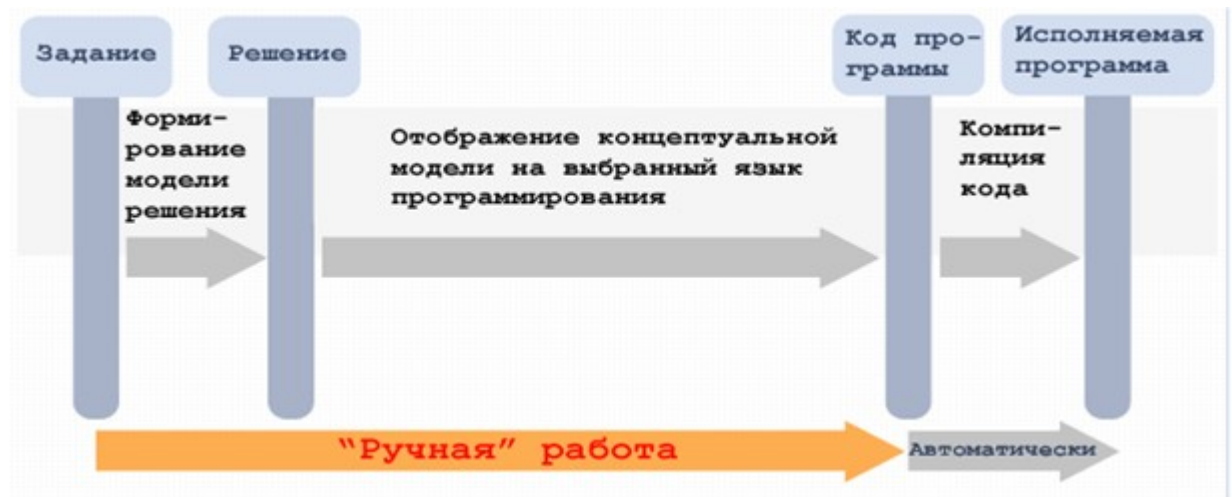


Рисунок 8. Программирование на языке общего назначения.

В этом случае шаг «непосредственного программирования» в существенно меньшей степени является «узким местом», так как DSL облегчают превращение концепции в нечто, понятное компьютеру (рис. 8). Совмещение подходящих инструментов и «самоприменимости» ЯОП существенно облегчает этот шаг.



Рисунок 9. Языково-ориентированное программирование на языке предметной области.

Мотивация ЯОП заключается в следующем: нужна возможность работать в терминах концепций и понятий проблемы, которую решает

программист, вместо того чтобы переводить свои мысли в нотацию языка программирования общего назначения (классы, методы, циклы, ветвления и т.п.). Для этого нужен язык, специфичный для предметной области. Языко-ориентированный подход предлагает программисту создать такой язык самому.

Языко-ориентированное программирование - вещь далеко не новая, и программисты уже давным-давно используют его в работе. На сегодняшний день существует несколько стилей языко-ориентированного программирования; некоторые из них:

- "Малые языки" UNIX являются наиболее очевидными представителями мира DSL. Эти языки представляют собой внешние DSL-системы, которые, как правило, используют для трансляции встроенные инструменты Unix.
- Lisp представляет собой, пожалуй, самый яркий пример существования различных DSL прямо в самом языке. Символьная обработка включена и в название, и в повседневную работу "лисперов". Этому способствуют минималистический синтаксис Lisp, его closures и макросы. Все это является отличной закваской для создания DSL.
- Конфигурационные файлы XML: В любом современном проекте написанном на Java, содержится огромное количество кода на XML. Крупные системы задействуют сразу несколько программных каркасов, большинство из которых используют для конфигурации XML файлы. По сути, эти файлы представляют собой DSL.
- Инструменты для построения пользовательских интерфейсов (GUI builders): Вместе с появлением графических пользовательских интерфейсов появились инструменты, позволяющие создавать такие интерфейсы путем перетаскивания элементов управления. Хотя

внешне такой стиль работы с графическим интерфейсом выглядит весьма заманчиво, у него есть определенные ограничения.

Инструменты для построения пользовательских интерфейсов представляют собой разновидность DSL, однако здесь редактирование кода существенно отличается от редактирования кода в текстовых программных языках, к которым мы так привыкли. Сами разработчики таких инструментов часто не рассматривают созданные им приложения для построения интерфейсов как языки программирования. Некоторые считают, что в этом и состоит основная проблема всех подобных инструментов.

Одна из тем, которая ставится в парадигме языко-ориентированного программировании - дать возможность непрофессионалам самим писать код. Всевозможные эксперты в различных предметных областях могли бы программировать, используя соответствующие DSL.

Один из способов решения этой проблемы заключается в выделении некой части проблемы (не очень сложной и имеющей четкие границы) и разрешении пользователям программировать именно в этой части. В последствие такие области можно превратить в DSL. Подобные DSL могут быть весьма сложными, например, MatLab.

Хороший DSL позволяет работать более продуктивно и профессиональному программисту, даже если этим DSL не смогут пользоваться непрофессионалы. Но он должен быть изначально писан профессиональным программистом, а затем уточнен и пересмотрен экспертами в предметной области.

Итак, самое большое достоинство языко-ориентированного программирования - это возможность использования DSL, а недостаток - необходимость создавать средства для их поддержки.

Система MPS, разработанная компанией JetBrains, является попыткой реализовать парадигму языко-ориентированного программирования.

Этапы разработки в данной системе представляются в следующем виде.

Как необходимый минимум, необходимо определить «структуру» нового языка. Это позволит писать «точно определенные» программы. Структура языка вовсе не означает его текстовую грамматику, язык может вообще не иметь текстового представления, а только графовое.

В большинстве случаев, используя ЯОП, программист работает на двух «уровнях» программирования – на метауровне и на программном уровне. Определив язык на метауровне, программист пишет программу на программном уровне. Определяя структуру нового языка, он использует DSL языковой структуры для определения нового языка, который будет использован на программном уровне.

В MPS каждый узел на программном уровне имеет «тип», который является просто связью с другим узлом метауровня. Узел программного уровня – это как бы «экземпляр» типа. Узел «тип» метауровня определяет свойства экземпляра и отношения, допустимые для него. Язык описания языка метауровня называется просто Языком Структуры.

Для определения абстрактного синтаксиса языка при помощи *языка структуры* достаточно перечислить все типы в языке. Типы представляют собой возможности (концепции), поддерживаемые языком. Каждая концепция определяется именем, внутренними свойствами экземпляров и отношениями (просто связи), в которых экземпляры могут находиться с другими узлами

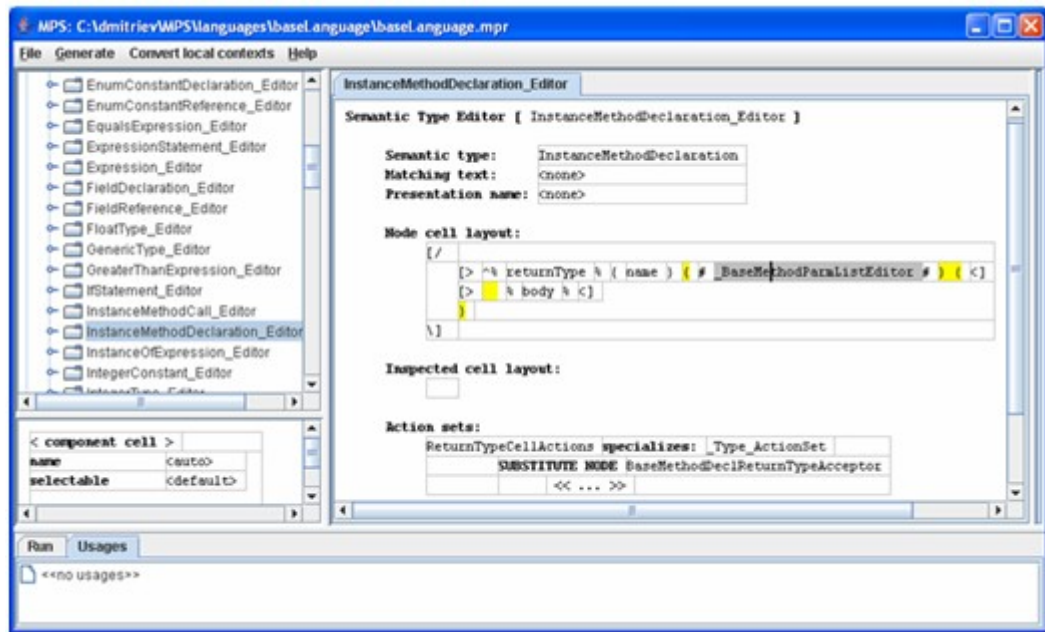


Рисунок 10. Снимок экрана системы MPS от Jet Brains.

MPS, по словам самого ее автора, еще не является полноценной системой, но близка к тому. В целом, интерфейс системы оставляет желать лучшего. Интуитивно разобраться в необходимых действиях для написания программы в данной системе не представляется возможным. В настоящий момент отсутствует полноценная документация к системе. Разработка новых DSL-языков в ней представляется процессом кропотливым и трудоемким.

1.7. Естественно-языковое программирование

1.7.1. Система ПЕГАС

Система «Пегас» представляет собой наиболее близкий по своей тематике проект, работы над которым ведутся в Техническом Университете Дармштадта (Германия). Данная разработка позиционируется как система естественно-языкового программирования. Эффективность данного метода подкрепляется некоторыми философскими рассуждениями создателей системы на тему проблем современных техник и методик программирования. Система Пегас призвана помочь программистам и избавить их от

необходимости изучать новые языки программирования, но излагать алгоритмическое описание решения задачи на естественном языке (Roman Knöll, 2006).

В настоящий момент Пегас поддерживает два языка: немецкий и английский. Важной особенностью системы является возможность перевода текста программы, написанной на естественном языке, на другой естественный язык, поддерживаемый «Пегасом». В текущем состоянии поддерживается перевод с немецкого языка на английский и соответственно наоборот.

Основываясь на некоторой теории *идей*, которая представляет знания в сознании человека как *сети идей* (фактически – концептуальные графы), авторы представляют архитектуру системы Пегас, как состоящую из трех компонентов:

- чтение естественного языка,
- генерация программного кода,
- выражение естественного языка.

Данная совокупность компонентов названа авторами системы Пегас «мозгом» системы (brain). В него также включаются такие компоненты, как «мышление» (mind), долгосрочная и краткосрочная память (long term memory и short term memory) (см. рис. 11)

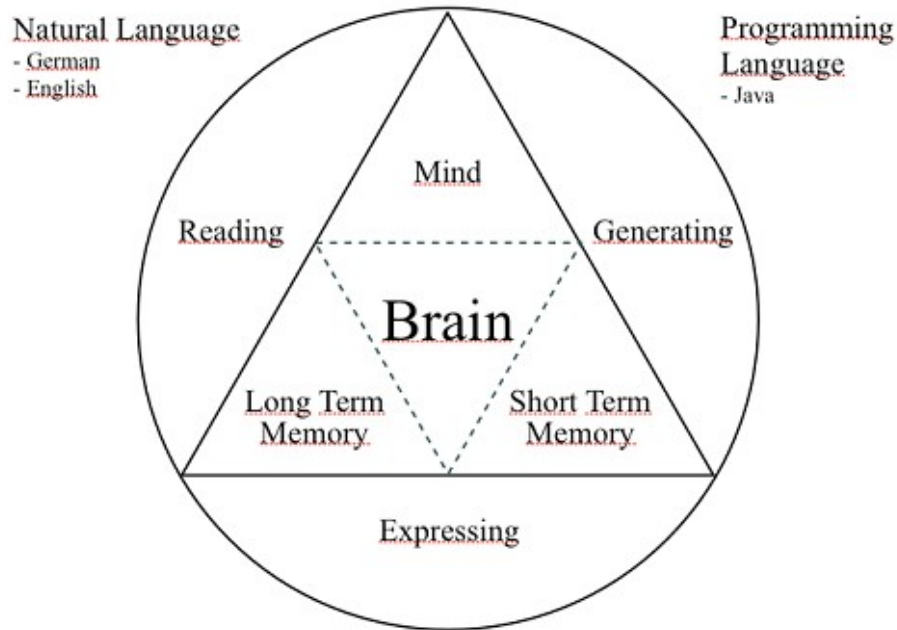


Рисунок 11. Архитектура системы Пегас.

Вся информация внутри «мозга» представлена с использованием нотации идей. Задача компонента «мышление» заключается в *понимании* прочитанного выражения. Долгосрочная память служит как источник информации для чтения и понимания высказывания, включая *значение* в форме *лексического* и *семантического* знаний. Краткосрочная память представляет текущий контекст: она позволяет связывать значение с имплицитными ссылками.

На сложность возможного входного естественно-языкового выражения накладываются некоторые ограничения. Система Пегас способна понимать фразы, составленные из командных предложений и нескольких различных логических сочетаний утверждений. Командные предложения носят характер наиболее общеупотребительных в английском (и, вероятно, немецком) языках конструкций: *субъект + глагол + объекты*. Так, например, система понимает фразу «*Move the cursor quickly into the corner of the right window on the screen*» («*Быстро передвинь курсор в угол правого окна на экране*»).

Для выражения *лексического знания* в долгосрочной памяти «мозга» системы Пегас реализован словарь, где для каждой *идеи* (концепта) хранятся описания сущности, действий и свойств данного концепта в каждом из поддерживаемых языков (см. таблицу 1).

Таблица 1. Пример хранения лексических знаний в словаре Пегаса.

	Сущность	Действие	Свойство
Немецкий язык	<u>Kleinheit</u> der Kleinheit der Kleinheit der Kleinheit die Kleinheit	<u>verkleinern</u> Verkleinere Verkleinerst Verkleinern Verkleinert Verkleinern 1 Object, Akkusativ	<u>klein</u> kleiner am kleinsten
Английский язык	<u>smallness</u>	<u>to downsized</u> downsized downsize 1 Object, direct	<u>small</u> smaller smallest

С помощью такого словаря Пегас распознает, что, например, «smaller» (меньше) это форма сравнения прилагательного «small» (маленький), будучи свойством концепта «smallness» (малость).

В словаре системы Пегас также содержатся т.н. *шаблоны идей*. Фактически, шаблонны идей описывают некоторую синтаксическую языковую конструкцию команды. Каждому шаблону сопоставляются фрагменты программного кода, где могут быть подставлены соответствующие составляющие шаблона. Пример:

(((??), statement), ((??), command)), condition)
Generator.println(" if (" + statement.get() + "); Generator.openBlock(); command.execute(); Generator.closeBlock();

Получая на входе программу на естественном языке, Пегас генерирует программу на языке Java. Фактически, такая генерация осуществляется с использованием шаблонов идей, представляющих собой некоторую таксономию конструкций предложений, характерных для используемых в системе языков. Полученные фрагменты программного кода склеиваются в единую программу, которая затем подвергается компиляции.

Отметим, что система фактически предлагает программисту записывать алгоритм на естественном языке, т.е. делает синтаксис и семантику языка программирования максимально приближенным к естественному человеческому языку. Помимо этой задачи достигается возможность перевода текста «программы» с одного естественного языка на другой.

Главными отличиями данной разработки от методик, предлагаемых в настоящем диссертационном исследовании, являются:

1. В системе Пегас связь естественного языка с языком программирования осуществляется на уровне синтаксических языковых конструкций, которым сопоставляются фрагменты исходного кода на языке Java.
2. Онтологическая модель описания концептов (*идей*) в словаре Пегас осуществляется, по всей видимости, на уровне внутренних способов представления данных, где жестко фиксируются три категории «сущность», «действие» и «свойства» для каждой словарной статьи.

Таким образом, представляется, что система Пегас не может использоваться для естественно-языкового анализа отвлеченных от программирования текстов, таких как новостные сводки, описания событий, мест, явлений, выражаемых в относительно свободном (не командном) стиле. Кроме того, можно предположить, что разработчикам придется решать серьезные проблемы при интеграции в систему русского языка, которая заложена в планах по развитию Пегаса.

1.7.2. Обзор существующих концепций использования объектно-ориентированного программирования для лингвистического моделирования.

Одним из источников развития объектно-ориентированного программирования (ООП) являлось использование средств представления знаний. В настоящее время объектно-ориентированный анализ (ООА) относится к сфере индустриальных методов, позволяющих разрабатывать сложное программное обеспечение. Однако применение этой парадигмы к задачам искусственного интеллекта по-прежнему сохраняет свою специфику с теоретической точки зрения. Традиционно ООА применяется для решения задачи в некоторой предметной области из реального мира. В результате, понятия этой области интерпретируются как объекты, которые принадлежат к некоторой иерархии классов, обладают некоторыми свойствами и могут быть идентифицированы от других объектов. В этом случае мы имеем две размерности: объекты реального мира и наше представление этих объектов. Однако если мы попытаемся применить ООА для задачи понимания естественно-языковых описаний из некоторой предметной области, то результаты применения этого анализа качественно меняются, поскольку вводится еще одно измерение, а именно представление того, как объекты реального мира и взаимодействия между ними выражаются на естественном языке. (Шаров, 1995)

Одной из первых концепций использования ООП для лингвистического моделирования явилась система SNOOP (System with Networks and Object – Oriented Production rules). (Sharoff, 1993) Эта система обеспечивает возможность интеграции и использования различных средств представления знаний: основной структурой данных является сеть объектов, которые связаны бинарными отношениями и принадлежат классам, описанным в разделе дефиниций данного приложения. Для навигации по сети и ее модификации используется механизм сопоставления с образцом, основанный на продукционных правилах. Свойства, которыми обладают классы, описываются с помощью групп продукционных правил. Благодаря тому факту, что система SNOOP предназначена для создания лингвистических приложений, была разработана библиотека классов, ориентированная на обработку естественного языка, а также словарная подсистема со встроенной поддержкой русской морфологии и средствами моделирования морфологии других флективных языков.

Поиск объектов, присутствующих в сети и обладающих определенными свойствами, осуществляется с помощью механизма наложения образца продукционного правила и вычисления условия на совпавшие узлы. Каждый образец состоит из ограничений на классы объектов и классы отношений, связывающих эти объекты. При этом если найдены объекты классов, наследующих классы, указанные в образце, и эти объекты связаны отношениями, наследующими отношения, указанные в образце, то такие объекты будут удовлетворять этому образцу.

Пример на рис. 12 иллюстрирует способ навигации по сети с помощью образцов, использующих иерархию классов. Связи DirectObj и IndirectObj наследуют более общий класс связи АктантнаяСвязь, а класс Местоимение наследует класс Существительное; прямоугольники и направленные стрелки обозначают узлы и отношения на сети.

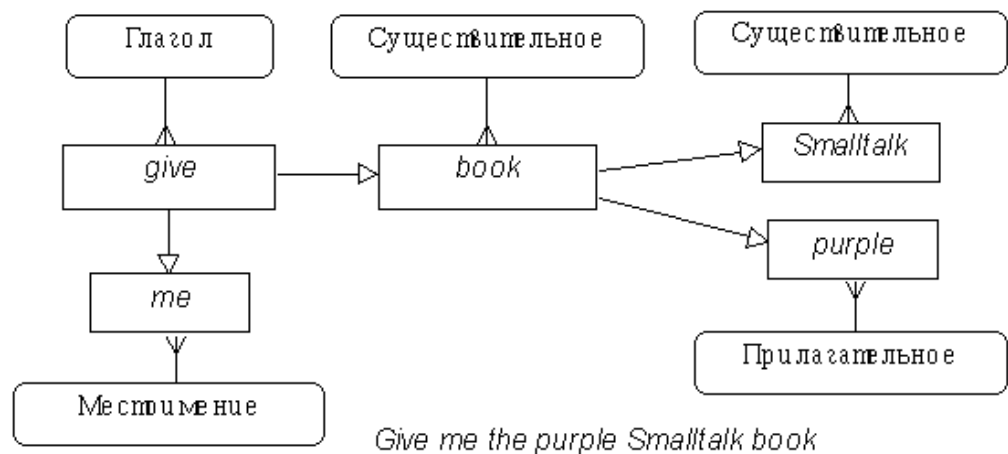


Рисунок 12. Способ навигации по сети с помощью объект.

Например, образец:

(X:Глагол, <АктантнаяСвязь>, Y:Существительное)

будет успешно сопоставлен на этой сети с двумя отношениями:

(give, <DirectObj>, book)

(give, <IndirectObj>, me).

Если образец задан как

(X:Глагол, <АктантнаяСвязь>, Y:Местоимение) ,

то класс связи будет удовлетворять обоим отношениям, но класс второго узла из образца может быть сопоставлен только с узлом из второго отношения. Еще одна специфическая черта этого формализма заключается в механизме модуляризации продукционных правил, разделенных на свойства, которые ассоциированы с объектами соответствующих классов и состоят из набора продукционных правил. Этот формализм был применен в задачах анализа текстов в узкой предметной области.

Существенным недостатком данной системы является то, что образцов ее реализации в свободном доступе не имеется, и работы по ней, судя по отсутствию документации, были прекращены на рубеже 1995-1996 гг.

Данная система получила свое развитие в коммерческой системе InBASE - естественно-языковой оболочке для распространенных СУБД и

Интернета. Ключевым для технологии InBASE является его основной принцип - семантически-ориентированный подход к анализу ЕЯ-текста. Он предполагает качественно иной способ формализации содержания текста ЕЯ-запроса, чем уровневые синтаксически-ориентированные анализаторы, неэффективность которых для решения задач информационных технологий подтверждена в ходе разработок трех последних десятилетий (Кра).

Технология InBASE предназначена для обеспечения:

(а) надежного и эффективного автоматического понимания ЕЯ-запросов к реляционным базам данных;

(б) простоты настройки системы на различные базы, содержащих различные данные и использующих различные языки. Для второй задачи в технологии разделяется знание о семантическом поле, содержащем логические и коммуникативно-ориентированные значения, обслуживающие область ЕЯ-запросов к базам данных вообще, от семантических полей конкретных предметных областей. Это позволяет быстро и эффективно настраивать результаты анализа запроса на конкретную базу данных.

InBASE строит промежуточное формальное представление запроса (Q-язык), схожее по своей парадигме с языком объектных запросов (Object Query Language – OQL), с использованием задаваемой для каждого ЕЯ-интерфейса модели предметной области (МПО) на основе диаграммы классов. Дальнейший перевод Q-запроса в SQL или любой другой формальный язык запросов происходит с учетом строения каждой конкретной базы данных на основе ее реляционной модели (МБД) и отделено от процесса понимания запроса. Результат после получения из базы данных может быть преобразован в т.н. QR-представление, соответствующее Q-представлению запроса и представляющее результат в терминах предметной области. Q-запрос, МПО, МБД и QR-результат имеют свои XML-представления.

Система InBASE имеет в своем составе следующие компоненты:

- лингвистический процессор, который переводит текст запроса из естественно- языковой формы в промежуточное формальное представление - Q-запрос;
- модель предметной области (МПО), модель базы данных (МБД), прокачка (сканирование полей базы данных и извлечение лексики в словарь);
- словарь
- БД-зависимая часть (генерация SQL, обработка “сырого” результата из базы данных в QR-представление результата).
- монитор настройки для адаптации словаря, МПО и МБД к конкретной базе данных
- монитор обработки запросов, передающий запросы Л-процессору и отображающий ответы.

Ядром системы InBASE является Л-процессор, который обеспечивает основную функцию системы – переход от естественного языкового к формальному представлению текста запроса. (Кра)

Минусом данной системы является ее сугубо коммерческое приложение, отсутствие открытого исходного кода, а также узкая область применения.

1.8. Онтология. Язык OWL

OWL (англ. Web Ontology Language) – язык описания онтологий для семантической паутины. Под *семантической паутиной* (Semantic Web) понимается такое расширение всемирной Паутины (World Wide Web), когда используются универсальные спецификаторы ресурсов (URI), онтологии и языки описания метаданных. С точки зрения представления знаний функция семантической паутины – предоставить компьютерам доступ к структурированным совокупностям информации и наборам правил, которые они могут использовать для управления автоматизированными

«рассуждениями». Семантическая паутина призвана дать возможность машинам «понимать» семантические документы и данные, а не человеческую речь или письмо (Berners-Lee, et al., 2001). Термин *онтология* взят из философии, где он обозначает теорию о природе бытия, об описании видов сущностей в мире и как они связаны между собой. Онтология языка OWL может включать в себя описание *классов*, *свойств* и их экземпляров. Представленная таким образом онтология описывается семантикой языка OWL. Она определяет, как выводить ее логические заключения, т.е. те факты, которые дословно не представлены в онтологии, но которые влечет за собой ее семантика (Smith, et al., 2010).

Различаются три диалекта OWL:

- OWL Lite – самый простой диалект, используется в первую очередь для классификации иерархии и простых особенностей налагаемых связей.
- OWL DL используется теми пользователями, которым нужна максимальная выразительность без потери вычислительной полноты и разрешимости (все вычисления должны завершиться в конечное время).
- OWL Full задуман для максимальной выразительности и синтаксической свободы, присущей RDF без гарантий вычислительности. Например, в OWL Full класс может обрабатываться одновременно как собрание индивидов и как один индивид в своем собственном значении. OWL Full позволяет такие онтологии, которые расширяют состав предопределенного (RDF или OWL) словаря (OWL10).

Каждый из этих диалектов - расширение его более простого предшественника, и в том, что касается выразительных возможностей, и в том, что касается возможностей производимых заключений.

RDF - это спецификация, которая определяет модель представления мира и синтаксис для сериализации и обмена этой модели. Консорциум

всемирной сети (World Wide Web Consortium, сокр. W3C) разработал XML-сериализацию для RDF. RDF XML - это стандартный формат обмена для RDF в семантической сети. RDF обеспечивает последовательный стандартный способ описания и работы практически с любыми интернет-ресурсами: от текстовых страниц и графиков до аудио-файлов и видео-клипов. Он предлагает синтаксические возможности для взаимодействия сетей и формирует базовый слой для создания семантической сети. RDF определяет управляемые графы связей, представленные тройками объект-атрибут-значение. Например, объект О имеет атрибут А со значением V. OWL улучшает возможности автоматической интерпретации содержимого интернета по сравнению с теми, что могут обеспечить XML, RDF и схема RDF. Это происходит благодаря тому, что OWL предоставляет дополнительные словарные возможности наряду с формальной семантикой (Бэлэни, 2007).

Итак, выделяются следующие компоненты языка OWL:

Классы - это основные блоки онтологии OWL. Класс - это концепция в домене. Классы обычно образуют таксономическую иерархию (т.е. систему подкласс-надкласс).

Свойства включают две основные категории:

- свойства объекта (Object properties), которые связывают индивидуальные элементы между собой;
- свойства типов данных (Datatype properties), которые связывают индивидуальные элементы со значениями типов данных, такими как целые числа, числа с плавающей запятой и строки. Для определения типов данных OWL использует схему XML.

Индивидуальные элементы - это элементы классов; свойства могут связывать их друг с другом. Например, индивидуальный элемент Smith может быть описан как элемент, принадлежащий классу Person

(индивидуум). Свойство `hasEmployer` (имеет работодателя) может связывать его с другим индивидуальным элементом - `Webify Solutions`, указывая, таким образом, что `Smith` работает в компании `Webify Solutions`.

В заключение рассмотрения OWL отметим, что данный язык является мощным инструментом для описания онтологий, но он не является алгоритмическим языком в полном смысле этого слова. На нем нельзя описать алгоритмы обработки знаний, нельзя описать алгоритмы взаимодействия определенных объектов между собой.

1.9 Средства редактирования расширенных сетей переходов

В заключении обзора рассмотрим существующий средства создания и редактирования расширенных сетей переходов. Наличие таких средств представляется весьма важным фактором, влияющим на эффективность программирования лингвистического процессора, построенного с использованием АТН-анализатора.

1.91. Augmented Syntax Diagram (ASD) Editor and Parser

Augmented Syntax Diagram (ASD) Editor and Parser - редактор Расширенных Синтаксических Диаграмм, предложенный профессором Джеймсом А. Масоном (James A.Mason). Сравнение ASD и АТН сетей произведено проф. Д. Масоном в (Mason). Особенности данного редактора являются:

- использование числовых индексов, или номеров этапов, в узловых метках для различения между разными узлами, которые помечены такими же словарными элементами;
- графический интерфейс, позволяющий конструировать графы на экране, а также изменять местоположение элементов с помощью мыши.

К недостаткам данного ASD-редактора можно отнести:

- недостаточно понятный интерфейс визуального отображения сетей и функций их редактирования;
- отсутствие функций поиска и автоматического построения заготовок сети;
- система адаптирована большей частью для обработки английского языка.

В целом, можно отметить, что данный ASD-редактор отражает в большей степени видение модели автором (проф. Д. Масоном), что накладывает некоторые ограничения на его широкое применение.

-

1. 9.2. Linguistic Tree Constructor

Linguistic Tree Constructor – визуальный редактор для анализа текстов с помощью синтаксических деревьев (Lin10). Программа имеет платформо-независимую реализацию, поддерживает любые языки для описания синтаксических деревьев.

К недостаткам системы можно отнести:

- слишком упрощенную модель отображения деревьев;
- обилие англоязычных сокращений-терминов;
- неудобный графический инструментарий;
- собственные форматы файлов данных, что затрудняет использование их в стороннем ПО.

1.9.3. TreeForm Syntax Tree Drawing Software

TreeForm Syntax Tree Drawing Software – мощный инструмент для создания и редактирования синтаксических деревьев (Tre10). Программа

имеет платформу-независимую реализацию, поддерживает различные языки при конструировании деревьев.

Можно выделить следующие преимущества системы:

- оригинальный и удобный интерфейс для построения синтаксических деревьев;
- возможность настройки цветовых схем интерфейса пользователя;
- экспорт деревьев в графических форматах (JPEG и PNG);

К недостаткам системы TreeForm можно отнести следующее:

- обилие англоязычных сокращений-терминов;
- отсутствие функции поиска;
- отсутствие нумерации узлов сети.

1.10 Выводы

В результате рассмотрения совокупности средств и методов построения лингвистических процессоров, методов использования объектно-ориентированных подходов при проектировании лингвистических процессоров, парадигм концептуального и языко-ориентированного программирования, а также визуальных средств работы с расширенными сетями переходов, были сделаны следующие выводы:

1) При проектировании лингвистического процессора целесообразно использовать существующие программные средства морфологического анализа. Для целей настоящего исследования самым подходящим выбран парсер Mystem компании ООО Яндекс. Получено официальное разрешение правообладателя на включение программы Mystem в состав разработанной в рамках диссертационного исследования системы Nalaps.

2) В качестве средств синтаксического и семантического анализа найдено эффективным осуществить разработку собственных алгоритмов и

собственной реализации модуля синтаксико-семантического анализа, построенного на использовании формализма расширенных сетей переходов (ATN). Расширенные сети переходов выбраны, как самый, на наш взгляд, простой и прозрачный способ программирования лингвистического процессора.

3) В связи с выбором в качестве основы синтаксико-семантического анализатора расширенных сетей переходов, проанализированы существующие средства конструирования РСП. Обнаружено, что ни одно из представленных в свободном доступе средств не обеспечивает нужного для целей исследования функционала, в связи с чем признана необходимым разработка собственного визуального редактора расширенных сетей переходов.

4) «Мягкая модель понимания текста» выбрана за теоретическую основу при проектировании лингвистического процессора. В ходе исследований наряду с мягкой моделью понимания текста был применен концептоцентрический подход, получивший свое развитие в онтологической модели построения базы знаний.

5) Языко-ориентированный подход показал актуальность задачи построения языков программирования предметной области (DSL). Данная парадигма подразумевает перенесение задачи в сферу автоматической работы компьютера, когда на вход системы поступает лишь описание задачи на DSL языке. Развитием данной концепции представляется в виде использования естественного языка для формулирования широкого класса задач, и представление таким образом естественного языка как языка «программирования» в узких предметных областях.

6) Языки описания знаний, такие как OWL, дают мощные возможности для описания смысловых взаимосвязей между объектами и их свойствами, однако не дают возможностей описания алгоритмов обработки и

взаимодействия этих объектов. В связи с этим в рамках настоящего исследования предложено использование онтологической модели и ее синтеза с уже существующими языками программирования (с языком Java). При этом XML-описание онтологии предлагается генерировать автоматически на этапе естественно-языковой спецификации классов языка Java.

Глава 2. Структура программного комплекса NALAPS.

2.1. Общая концепция системы.

Система Nalaps представляет собой программный комплекс, состоящий из следующих основных компонентов:

1. Лингвистический процессор, который получает на вход естественно-языковой текст. Текст проходит несколько этапов лингвистического анализа, в результате которого на выходе генерируется синтаксико-семантическое представление текста.

2. Концептуального процессора, осуществляющего генерацию концептуального графа на основе полученного семантического представления.

3. Кодогенератор, который сопоставляет узлы и отношения концептуального графа со специфицированными классами языка Java и их методами строит, таким образом, программный код на языке Java. Автоматически генерируемый Java код выполняется после генерации каждого оператора, что создает в памяти ЭВМ динамическую объектную среду, являющуюся отображением «смысла» обработанного текста.

4. Обратный концептуальный процессор, способный в любой момент времени преобразовывать «снимок» динамической объектной модели обратно в концептуальный граф с анализом новых объектов, новых образовавшихся связей и новых значений свойств.

5. Прототип обратного лингвистического процессора, который строит текст на основе концептуального графа, что позволяет налаживать естественно-языковое общение оператора с ЭВМ.

6. Отдельным компонентом является **спецификатор языка Java**, представляющий собой модуль, интегрируемый в существующие средства разработки программ на языке Java. С одной стороны данный модуль

позволяет программистам эффективно использовать современные средства интенционального программирования (Simonui, 1995) для спецификации уже существующих классов. С другой, данный модуль позволяет строить классы языка Java, используя все те же принципы интенционального программирования, на основе анализа естественно-языковых предложений.

7. Редактор расширенных сетей переходов с графическим интерфейсом пользователя.

Общая структура системы NALAPS представлена на рис. 13.

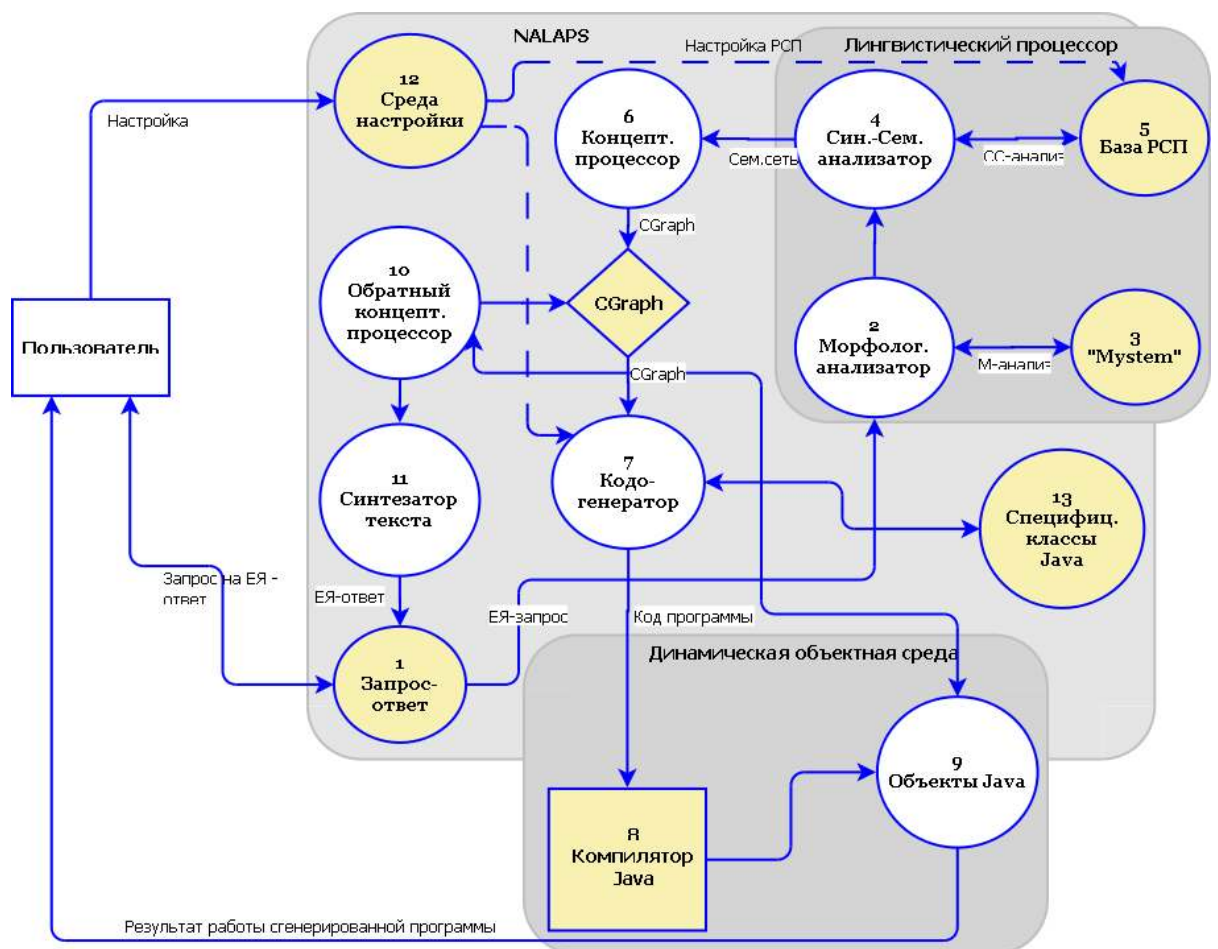


Рисунок 13. Общая структура системы NALAPS.

Каждый из перечисленных компонентов будет рассмотрен подробнее в следующих разделах данной главы.

2.2. Лингвистический процессор.

За теоретическую основу проектирования лингвистического процессора была выбрана модель «мягкого понимания текста». В разработанном лингвистическом процессоре реализованы аспекты а) и б) указанной модели, а именно:

а) пофразное чтение и «понимание» естественного текста

и

б) «понимание» текста в масштабах всего документа, установка связей между семантическими представлениями отдельных предложений.

Общая схема работы лингвистического процессора является классической и представлена на рис. 14.

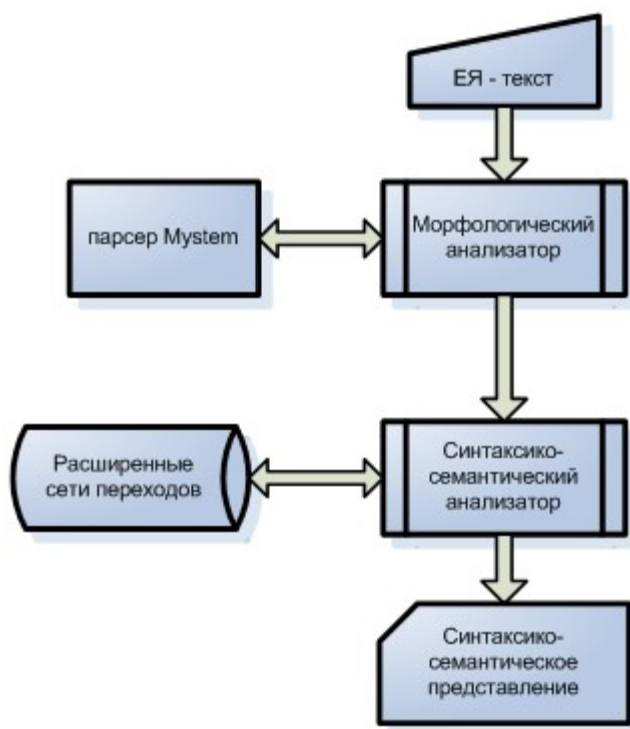


Рисунок 14. Общая схема работы лингвистического процессора.

2.2.1. Компонент морфологического анализа

В качестве компонента морфологического анализа (МорфАн) используется парсер Mystem. Ввиду особенностей технической реализации

программы Mystem – он является консольным приложением - для обработки результатов его работы было необходимо реализовать специальную программную оболочку. Кроме того, морфологический анализатор Mystem не умеет анализировать такие символы и их последовательности, как знаки препинания, цифры и числа: программа Mystem их игнорирует и в результатах анализа опускает их упоминание. Ввиду этого в упомянутую оболочку были встроены простейшие алгоритмы обработки вышеуказанных последовательностей символов.

Кроме того, отметим некоторое несовпадение набора частей речи, используемых в результатах анализа парсера Mystem и внутри морфологического анализатора системы Nalaps. Данное несоответствие и способ сопоставления приведены в таблице 2.

Таблица 2. Соответствие частей речи парсера Mystem в системе Nalaps

Часть речи, используемая в результатах анализа парсера Mystem	Соответствующее ей обобщение в системе Nalaps
прилагательное	прилагательное
наречие	наречие
местоименное наречие	местоименное_наречие
порядковое числительное	числительное
местоименное прилагательное	местоимение
часть композита (первая часть сложных слов)	- (не используется)
союз	союз
междометие	- (не используется)
числительное	числительное
частица	частица

предлог	предлог
существительное	существительное
местоимение	местоимение
глагол	глагол
- (отсутствует)	знак_препинания
- (отсутствует)	число

В качестве морфологических признаков парсером *mystem* выделяются следующие группы:

- Время (глаголов): настоящее, непрошедшее, прошедшее.
- Падеж: именительный, родительный, дательный, винительный, творительный, предложный, партитив (второй родительный), местный (второй предложный), звательный.
- Число: единственное, множественное.
- Репрезентация и наклонение глагола: деепричастие, инфинитив, причастие, изъявительное наклонение, повелительное наклонение.
- Форма прилагательных: краткая, полная, притяжательная.
- Степень сравнения: превосходная, сравнительная.
- Лицо глагола: 1-е лицо, 2-е лицо, 3-е лицо.
- Род: мужской, женский, средний.
- Вид (аспект) глагола: совершенный, несовершенный.
- Залог: действительный, страдательный.
- Одушевленность: одушевленное, неодушевленное.
- Переходность: переходный глагол, непереходный глагол.
- Особые свойства: вводное слово (не используется в системе *Nalaps*), географическое название, образование формы затруднено (не используется в системе *Nalaps*), имя собственное, искаженная форма (не используется в системе *Nalaps*), общая форма мужского и женского рода (в

системе Nalaps разбивается на два значения признака: мужской и женский), обценная лексика (не используется в системе Nalaps), отчество, предикатив (не используется в системе Nalaps), разговорная форма (не используется), редко встречающееся слово (не используется), сокращение, устаревшая форма (не используется), фамилия.

Для каждой словоформы парсер Mystem выдает список лексем, к которым может относиться данная словоформа. Таким образом, на этапе морфологического анализа система сталкивается с проблемой многозначности. Данная проблема в системе Nalaps решается путем представления результата морфологического разбора в виде двумерного массива фреймовых структур (таблица 2), где в качестве его элементов выступают фреймы с описанием признаков. Например, для фразы «*Растопил печь докрасна*», структурированный результат разбора представлен в таблице 3:

Таблица 3. Структурированный результат разбора примера

<i>растопил</i>	<i>печь(1)</i>	<i>докрасна</i>
Словоформа: <i>растопил</i>	Словоформа: <i>печь</i>	Словоформа: <i>докрасна</i>
Число: <i>единственное</i>	Число: <i>единственное</i>	Неопределенная форма: <i>докрасна</i>
Род: <i>мужской</i>	Род: <i>женский</i>	Часть речи: <i>наречие</i>
Наклонение: <i>изъявительное</i>	Падеж: <i>именительный, винительный</i>	
Неопределенная форма: <i>растопливать</i>	Неопределенная форма: <i>печь</i>	
Время: <i>прошедшее</i>	Одушевленность: <i>неодушевленное</i>	
Вид: <i>совершенный</i>		
Часть речи: <u><i>глагол</i></u>	Часть речи:	

	<u>существительное</u>	
	<p>печь(2)</p> <p>Словоформа: <i>печь</i></p> <p>Наклонение:</p> <p><i>инфинитив</i></p> <p>Неопределенная форма: <i>печь</i></p> <p>Вид: <i>несовершенный</i></p> <p>Часть речи: <u>глагол</u></p>	

2.2.2. Компонент синтаксико-семантического анализа.

Синтаксический и семантический компоненты анализа объединены в реализации анализатора расширенных сетей переходов (ATN-анализатор). В формализм расширенных сетей переходов введен ряд особенностей и нововведений, призванный повысить эффективность и удобство описания синтаксиса и семантики.

Каждая расширенная сеть переходов в системе Nalaps представляет собой ориентированный однонаправленный граф (см. рис. 13). Каждый узел графа имеет числовой уникальный номер. Особенностью предложенного формализма является возможность задания произвольного числа конечных вершин графа. Под конечной вершиной подразумевается такая, на которой обход данного графа прекращается.

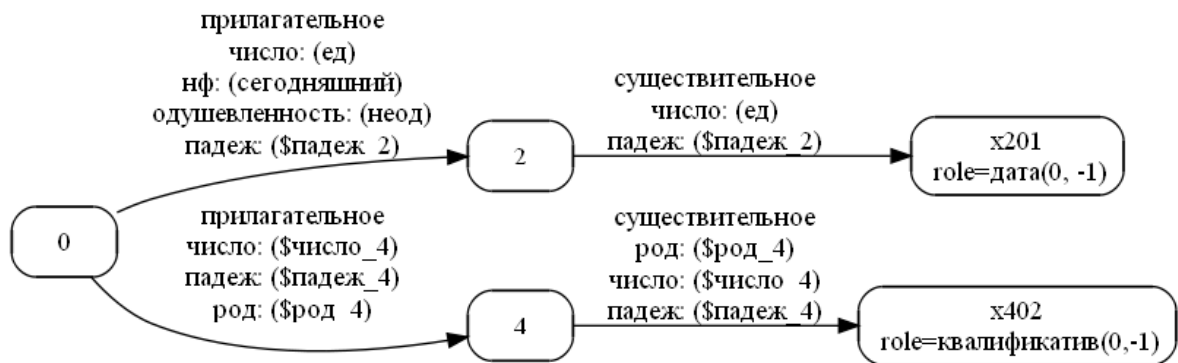


Рисунок 15. Пример расширенной сети переходов «Прилагательное».

Каждая дуга графа помечена набором произвольного числа меток, которые определяют набор свойств, которыми *достаточно* обладать очередному фрейму морфологического разбора, соответствующему очередному слову из входного потока текста для перехода по данной дуге. Метка является совокупностью двух ячеек: *свойства* и *значения*.

В узлах графа также могут быть помещены метки, которые могут:

а) быть внутренними командами АТН-анализатора. К ним относятся:

1) Команда экстренного прекращения разбора ***deprecated:yes***.

Используется, в случае попадания разбора на «запрещенную» ветку, указывающую на то, что, например, данный вариант снятия омонимии некорректен в данном контексте (в данной конструкции предложения в данной предметной области).

2) Команда создания или изменения значения внутренней переменной, например: ***\$main:-2***. Внутренние переменные могут содержать в себе ссылки на конкретные фреймовые структуры морфологического разбора (иными словами на конкретные слова из конкретных предложений). Данной командой можно изменять значение переменных, путем указания либо номера узла, в котором был произведен разбор нужного слова, либо указанием другой переменной, значение которой необходимо скопировать.

б) либо командами выделения синтаксико-семантических отношений:
role: \$asRole(\$main,\$a). В качестве параметров отношений могут быть указаны как конкретные номера узлов, так и динамически вычисляемые номера узлов, путем указания смещения в кол-ве шагов, которые требуется сделать назад в истории разбора для получения конкретного узла, либо указание конкретной внутренней переменной. В первых двух случаях в соответствующую позицию отношения записывается соответствующая узлу, а во втором – переменной, фреймовая структура морфологического разбора.

Таким образом, записанный в РБНФ внутренний язык узлов РСП, выглядит следующим образом (терминалы заключены в двойные кавычки и выделены *полужирным курсивом*):

Команда ::= Запрещение | Присваивание | Отношение;

Запрещение ::= ***“deprecated:yes”***;

Присваивание ::= Идентификатор ***“:”*** Адрес;

Адрес ::= (Идентификатор | узел | смещение);

Узел ::= Цифра {Цифра};

Смещение ::= ***“-”*** Цифра {Цифра};

Отношение ::= ***“role:”*** (Имя | Идентификатор) ***“(”*** Адрес {***“,”*** Адрес} ***“)”***;

Идентификатор ::= ***“\$”*** Буква{Буква|Цифра|***”_”***};

Имя ::= Буква{Буква};

Цифра ::= ***“0”|“1”|“2”|“3”|“4”|“5”|“6”|“7”|“8”|“9”***;

Буква ::= ***“A”|“B”|“C”|“D”|“E”|“F”|“G”|“H”|“I”|“J”|“K”|“L”|“M”|“N”|“O”|“P”|“Q”|“R”|“S”|“T”|“U”|“V”|“W”|“X”|“Y”|“Z”|“a”|“b”|“c”|“d”|“e”|“f”|“g”|“h”|“i”|“j”|“k”|“l”|“m”|“n”|“o”|“p”|“q”|“r”|“s”|“t”|“u”|“v”|“w”|“x”|“y”|“z”|“A”|“Б”|“В”|“Г”|“Д”|“Е”|“Ё”|“Ж”|“З”|“И”|“Й”|“К”|“Л”|“М”|“Н”|“О”|“П”|“Р”|“С”|“Т”|“У”|“Ф”|“Х”|“Ц”|“Ч”|“Ш”|“Щ”***

| "Ъ" | "Ы" | "Ь" | "Э" | "Ю" | "Я" | "а" | "б" | "в" | "г" | "д" | "е" | "ё" |
 "ж" | "з" | "и" | "й" | "к" | "л" | "м" | "н" | "о" | "п" | "р" | "с" | "т" | "у" |
 "ф" | "х" | "ц" | "ч" | "ш" | "щ" | "ъ" | "ы" | "ь" | "э" | "ю" | "я";

Алгоритм работы предложенного ATN-анализатора описывается следующими шагами:

1. Предложения разбираются поочередно.
2. Из входного потока поступает следующее слово и фрейм его морфологического разбора (последовательный перебор слов и знаков препинания слева на право).
3. Если нет открытых РСП или нет подходящих дуг для перехода, то открыть РСП соответствующее части речи текущей словоформы.
4. Если у текущего слова из входного потока МорфАн-ом выделено более одной лексемы, т.е. присутствует омонимия, то процесс анализа в п.3. распараллеливается для каждой лексемы.
5. У текущего узла РСП анализируется набор дуг. Если есть дуги с терминальными метками, то переход к шагу 6, иначе к шагу 7.
6. Анализируется число совпадений грамматических признаков текущей словоформы и грамматических признаков дуг, выходящих из текущего узла сети. Совпадение считается равным «-1», если полностью не совпадает хотя бы один грамматический признак (например, у дуги прописан признак «Падеж: им., вин.», а у входного слова однозначно определен предложный падеж). Далее переход на шаг 8.
7. Производится обход всех нетерминалов до нахождения подходящего. При заходе в РСП соответствующего нетерминала осуществляется переход на шаг 5.
8. Выбирается дуга с наибольшим числом совпадений, отличных от «0» и происходит переход по ней к следующему узлу. В новом узле выполняются все описанные в нем команды.

9. Если в п.6 не найдено ни одной дуги, то переход к п.3.
10. Если узел помечен признаком конца, то текущая РСП закрывается.
11. Переход к п.3.

Пример работы предлагаемого алгоритма, рассмотрим на следующем примере:

Если просроченная задолженность по ипотеке клиента больше 100000 рублей, то карточке клиента установить стадию «Hard». (1).

Для разбора примера (1) необходимо построить следующие расширенные сети переходов (РСП):

1. РСП «Sentence » (предложение):

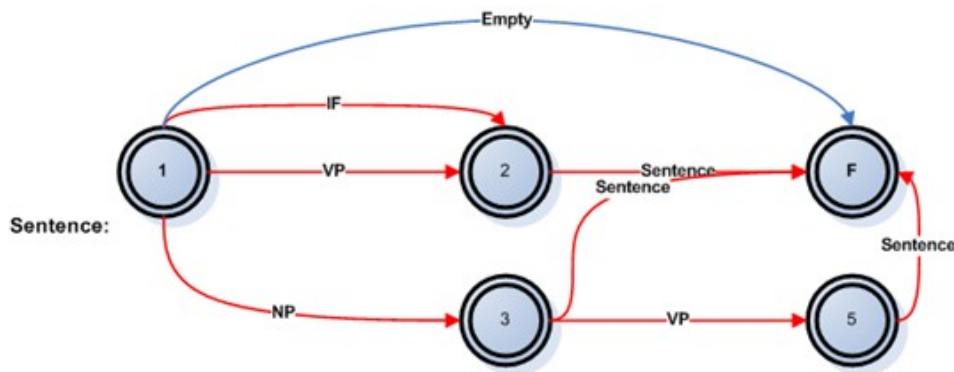


Рисунок 16. Расширенная сеть переходов "Sentence"

Это начальная сеть для разбора предложений. Из узла «1» имеется 4 выхода:

- «Empty» - для окончания разбора предложения (терминал);
- «IF» - для разбора простых предложений типа «Если...то» (нетерминал);
- «VP» - для разбора группы сказуемого (нетерминал);
- «NP» - для разбора группы подлежащего (нетерминал).

При разборе примера (1) правильным переходом будет переход к узлу «2» по дуге «IF».

2. РСП «NP»:

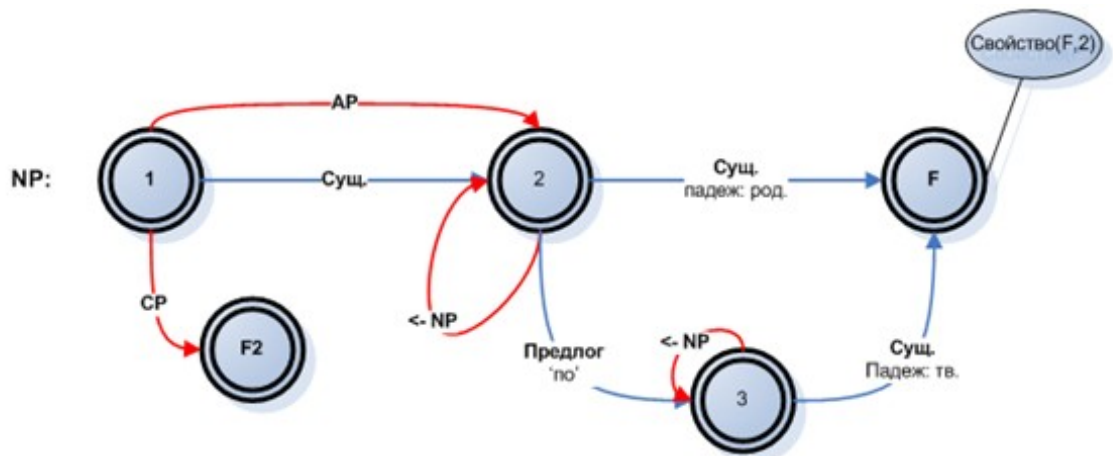


Рисунок 17. Расширенная сеть переходов "NP"

Данная сеть служит для разбора группы подлежащего – простого предложения с главным словом – существительным.

3. РСП «AP»:

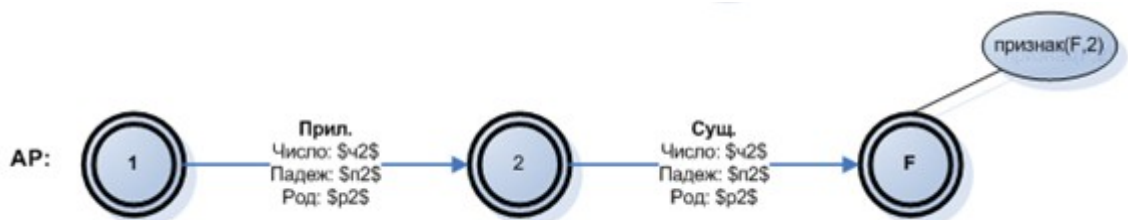


Рисунок 18. Расширенная сеть переходов "AP"

Данная сеть предназначена для разбора простых словосочетаний типа «прилагательное» + «существительное». В дугах «1-2» и «2-F» указаны переменные, используемые для согласования числа, падежа и рода словоформ. Так, при первой обработке переменной $ч2$ в нее будет помещено значение признака «число» и прилагательного из входного потока. На ребре «2-F» будет произведено сравнение данного значения со значением

признака «число» у следующего слова из входного потока – существительного.

У конечного узла «F» в данной сети указана метка «*признак(F,2)*», обработка которой создает синтактико-семантическое отношение «признак».

4. PCП «IF»:

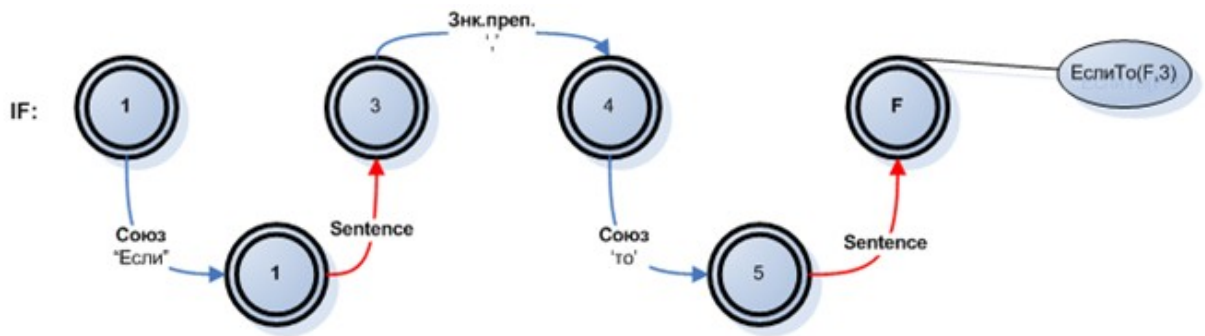


Рисунок 19. Расширенная сеть переходов "IF".

На рис. 19 представлена расширенная сеть переходов для обработки словосочетаний типа «Если ... то ...». В узле «F» выделяется отношение «*ЕслиТо(F,3)*», в качестве аргументов которого передаются не конкретные лексемы, а целые семантические сети, полученные на этапе обработки нетерминалов *sentence* из дуги «1-3» и «5-F».

5. PCП «VP»

Данная сеть, представленная на рис. 20 необходима для осуществления разбора *группы сказуемого*. Здесь необходимо отметить наличие двух альтернативных цепочек и, соответственно, финальных узлов «F» и «F2».

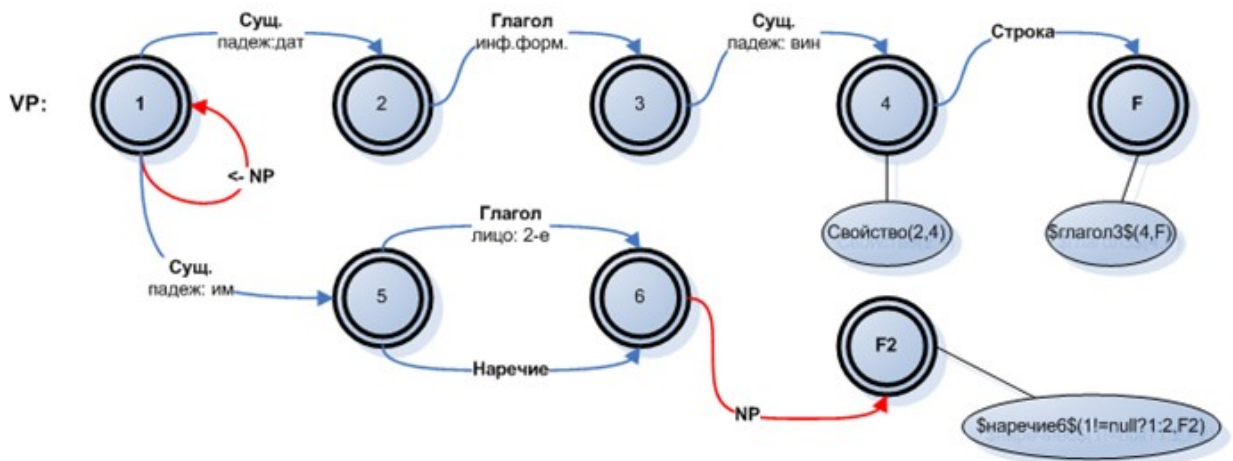


Рисунок 20. Расширенная сеть переходов "VP".

Конечный путь обхода представленных на рис. 16-20 сетей для примера (1) выглядит, как:

Sentence[IF[“Если”+Sentence[NP[AP[прил.+сущ.]+”по”+NP[сущ.
+сущ.]]+VP[сущ.+наречие+NP[CP[числ.+сущ.]]]]
+','+’to’+Sentence[VP[NP[сущ.+сущ.]+глагол+сущ.+строка]]]]

Полученное в результате работы ATN-анализатора с вышеописанными сетями синтактико-семантическое представление примера (1) имеет вид, представленный на рис.21.

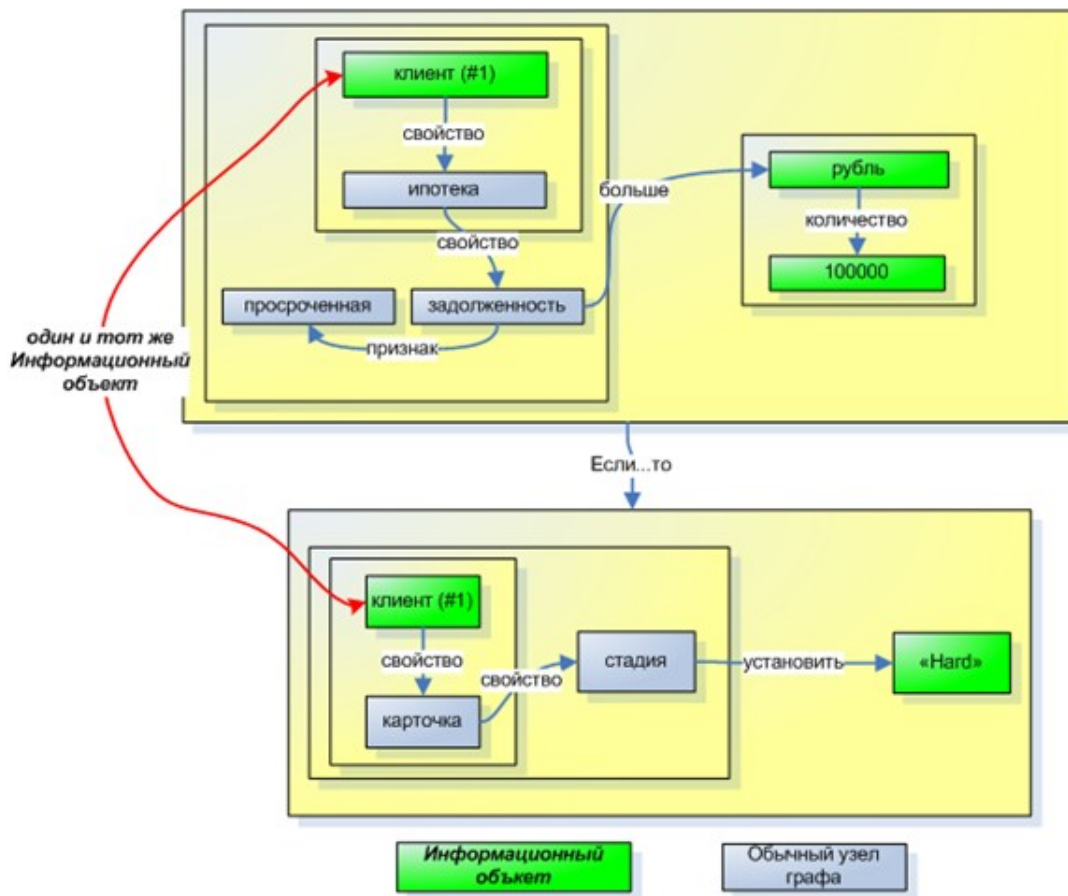


Рисунок 21. Синтактико-семантическое представление примера (1)

Таким образом, синтактико-семантическое представление каждого предложения – это семантический граф (Пономарева, 1999), определяемый как:

$$\Pi = \langle Q, U \rangle,$$

где Q – конечное множество вершин графа, содержащих в себе единицы предложения – конкретные фреймовые структуры с морфологическими признаками, ассоциированные со словами из входного предложения; и U – множество дуг, описывающие эксплицитно выраженные в тексте синтактико-семантические связи (семантические отношения) между словами.

Семантические отношения выражаются формулой:

$$P(A, B),$$

где P – имя (а точнее – класс или тип) отношения, в котором A находится по отношению к B (Леонтьева, 2006); т.е. задает фактически бинарные отношения. Отметим, что в предложенной модели допускается наличие *унарных* отношений $P(A)$, являющихся вырожденным случаем бинарных отношений, когда второй аспект отношения заполнен пустым объектом. Такие отношения возможны для предложений вроде:

Бомба взорвалась (2) - Взрываться(Бомба).

Отметим, однако, что в системе допускается описание в таком виде и многоместных предикатов, но использование двухместных является более предпочтительным, в то время как любой многоместный предикат представим с помощью суперпозиции двухместных (Мельчук, 1999).

Каждый узел семантического графа Π , и, соответственно, аспекты семантических отношений сопоставляются с определенными конкретными фреймовыми структурами, имеющими конкретные уникальные идентификаторы. Так, при анализе двух предложений:

*...Тут же стояла большая серебряная **чаша** с вином, растворенным водою (3.1). Вокруг этой **чаши** стояли кубки в виде звериных и птичьих голов (3.2) (Лесков)*

Фреймовые структуры для слова «чаша» из (3.1) и для слова «чаши» из (3.2) будут независимыми сущностями с уникальными внутрисистемными номерами. Принцип сопоставления этих сущностей будет описан ниже.

Важным компонентом лингвистического процессора является *стек отношений* в который предполагается осуществлять запись отношений в том порядке, который отражает временную или причинно-следственную связь событий в тексте. Фактически, порядок корректности выделения таких связей полностью зависит от правильности программирования расширенных сетей переходов, обработка которых позволяет сохранять такой порядок. Отметим, однако, что создание такого стека – всего лишь попытка создать

компонент, который будет способен отражать последовательность событий/ситуаций в тексте. Фактически такой порядок удастся выделить лишь на уровне отдельных предложений. Серьезный и глубокий анализ этой проблемы – тема отдельного исследования. Описанный стек будет использоваться в кодогенерирующем компоненте, о чем подробнее будет сказано ниже.

Работа лингвистического процессора тесно связана работой концептуального процессора. Для сопоставления семантических представлений отдельных предложений между собой для получения представления всего текста и концептуального графа используется единый механизм генерации *информационных объектов*. Информационные объекты являются связующими узлами конечного графа, которые «склеивают» семантические представления отдельных предложений, представляя собой своего рода укрупненные узлы. Поэтому дальнейшее рассмотрение работы системы следует продолжить с рассмотрения работы концептуального процессора.

2.3. Концептуальный процессор.

Предложенное выше понятие «*информационных объектов*» схоже, с одноименными сущностями, такими как *информационный объект* в терминологии И.П. Кузнецова (Кузнецов И.П., 2009) и *текстовый факт* у Н.Н. Леонтьевой (Леонтьева, 2006). Наиболее близким к пониманию *информационного объекта* в системе Nalaps являются эквивалентные представления об объектах: «чтобы указать на какой-либо объект, необходимо знать его уникальное имя или же описать его свойства и отношения, по которым он может быть однозначно выделен. При этом может существовать множество способов описания, однозначно определяющих объект» (Кузнецов, 1986). Однако, в отличие от предлагаемых у Кузнецова *граф-продукций*, фактически тот же самый анализ проводится с применением

элементарных операций сравнения множеств. Математическая модель данных преобразований будет дана в главе 3. Отметим здесь общие принципы построения информационных объектов.

Способом выражения понятия в лексике является *имя* – единица языка, соответствующая со стороны семантики отражению отдельного предмета, объекта или явления объективного мира, а со стороны синтаксиса – субъекту или объекту высказывания, предметной постоянной или переменной в пропозициональной функции, месту в предикате или в структурной схеме предложения (Степанов, 2007). Таким образом, при последовательном анализе синтаксико-семантических представлений отдельных предложений, каждое существительное рассматривается как потенциальный «кандидат» на роль информационного объекта. Для каждого такого «кандидата» составляется список синтаксико-семантических отношений, в которых он участвует, однако в данный список попадают не все отношения, а только относящиеся к следующему набору, предопределенных в системе типов. Предложенный ниже список основан на (Золотова, 2006) (Пономарева, 1999):

Таблица 4. Семантические роли.

Наименование роли	Описание роли
Квалитатив	Признак, обозначающий качество, свойство предмета
Квалификатив	Понятие, являющееся для субъекта классифицирующим, родо-видовым
Комитатив	Компонент, обозначающий сопровождающее действие, признак
Локатив	Компонент, характеризующий местоположение субъекта
Темпоратив	Признак, выражающий временную

	характеристику действия
Номинатив	Наименование

Данный список при необходимости может расширяться. Выбор именно таких отношений обусловлен тем, что они выражают определенные иерархические и межобъектные связи, которые легко интерпретируются в онтологических понятиях классов и свойств.

Получаемый список для каждого информационного объекта пополняется набором некоторых морфологических признаков слова, а именно: число, собственность, род, переходность, время, неопределенная форма. При анализе очередных «кандидатов» на включение или сопоставление с существующим информационным объектом, данные списки попарно сравниваются. Результатом сравнения является числовая характеристика отличная по модулю от нуля, выражаемая формулой:

$$S = \sum s_i * \left| \prod \|s_i\| \right|,$$

где s_i – результат сравнения одноименных пар-свойств. Пара-свойство имеет вид совокупности *имени_свойства* - значения, где *имя_свойство* является одним из типов морфологических признаков, добавляемых в список свойств объекта, или же типом «роль», который принимает значение одной из перечисленных выше ролей объекта. В формуле сумма таких результатов перемножается с модулем произведения нормированных по единице результатов сравнения. Таким образом, если хотя бы одно из свойств не совпадает, то результат его сравнения s_i становится равным 0, что обращает все выражение в 0.

Знак полученного значения S определяет, какой из двух объектов («кандидат» или текущий) следует считать более полным. «Меньший» объект включается в «большой», и все указатели на «меньший» объект во всех отношениях, в которых он участвует, заменяются на указатель

«большого» объекта. Тем самым происходит «слитие» разрозненных семантических графов в единое семантическое представление всего текста.

Создание концептуального графа ведется параллельно созданию семантического представления текста. Узлы графа, соответствующие словам из синтаксических групп, составляющих описание объекта, заменяются на узлы, сопоставленные с информационными объектами. К таким группам относятся те, которые описываются отношениями, представленными в таблице 3. Полученный концептуальный граф по числу узлов не превосходит свой родительский семантический граф. Данный концептуальный граф является выражением смысла исходного естественно-языкового текста.

2.4. Естественно-языковой спецификатор языка Java.

Прежде чем перейти к рассмотрению кодогенератора, необходимо рассмотреть такой компонент системы, как **естественно-языковой спецификатор языка Java**. Основное назначение данного компонента – связать классы языка Java, которые являются абстракциями некоторых объектов и явлений внеязыковой действительности некоторой предметной области, с естественно-языковым описанием этих сущностей. Данная операция осуществляется путем разметки уже существующих классов языка Java, а именно соотносением терминов предметной области с именами, полями и методами класса. Перечислим вначале общие принципы и правила данной естественно-языковой спецификации:

1) Каждый элемент, задающий шаблон спецификации, будем называть *меткой*.

```

public class Debtor {

    public String stage;

    private boolean dead;

    public Account mortgage;

    public void setStage(String stage) {
        this.stage=stage;
    }

    public boolean isDead() {
        return dead;
    }
}

```

Рисунок 22. Пример класса Java для спецификации.

2) В качестве метки выступают фреймовые структуры, содержащие набор признаков из числа возможных в таблице 1.

3) Каждый компонент класса может быть размечен произвольным числом меток.

Фактически процесс разметки выглядит следующим образом. Рассмотрим некий обобщенный пример класса языка Java, который требуется специфицировать для некоторой предметной области (рис. 22).

Разметка компонентов класса осуществляется путем использования механизма *аннотаций*. Аннотации в языке Java — возможность добавления в текст программы метаданных, не влияющих на выполнение кода, но допускающих использование для получения различных сведений о коде и его исполнении (Хорстман, и др., 2009). Для аннотирования классов естественно-языковой разметкой в систему Nalaps введен простой интерфейс *Semanticable*:

```

public @interface Semanticable {

```

```
String value();
}
```

Таким образом, разметка для класса, приведенного на рис. 20 может выглядеть следующим образом (рис. 23):

```
@Semanticable("Клиент, карточка клиента")
public class Debtor {

    @Semanticable("Стадия")
    public String stage;

    private boolean dead;

    @Semanticable("Ипотека")
    public Account mortgage;

    @Semanticable("Передать в")
    public void setStage(String stage) {
        this.stage=stage;
    }

    @Semanticable("Умер")
    public boolean isDead() {
        return dead;
    }
}
```

Рисунок 23. Пример естественно-языковой спецификации класса Java.

Введенное выше определение *метки* говорит о *шаблоне* спецификации. Заданный шаблоном термин служит отправной точкой для задания набора характеристик спецификации. Встроенный инструмент редактирования спецификации позволяет выбрать, какие из свойств метки следует оставить, а какие нет. Например, в классе, который является абстрактным описанием персоны:

```
public class Person {
    //...
}
```

следует использовать разметку `@Semanticable("Иванов, Иван, Иванович, Петрова Марья Петровна")`, однако, из свойств каждого шаблона

(«Иванов», «Иван» и т.д.) будет оставлен лишь набор, состоящий из таких признаков, как *род*, *число*, а также особые свойства, такие как *фамилия*, *имя* и *отчество*. Снимок экрана с примером настройки свойств приведен на рис.

24.

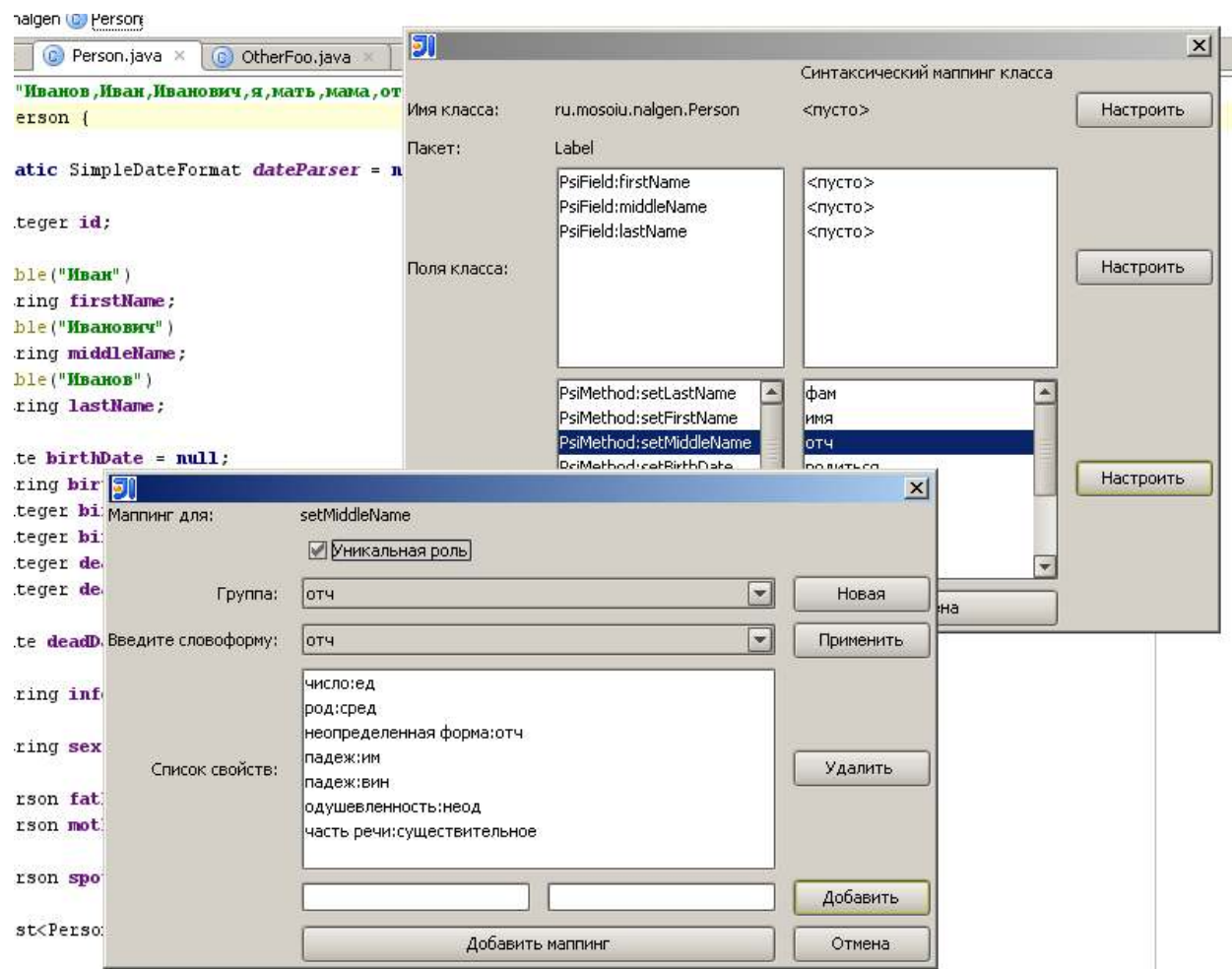


Рисунок 24. Снимок экрана работы инструмента настройки свойств.

Полученные в результате настройки фреймовые описания спецификации автоматически выгружаются в XML-файлы, представляя собой структуры, близкие по своему характеру к описанию онтологии на языке OWL (рис. 25):

```

<?xml version="1.0" encoding="UTF-8" ?>
} <Specification className="Debtor">
}   <SpecGroup group="className">
}     <Label>Клиент</Label>
}     <Properties>
}       <Property value="неопределенная форма">
}         клиент
}       </Property>
}     </Properties>
}   </SpecGroup>
}   <SpecGroup group="className">
}     <Label>Карточка клиента</Label>
}     <Properties>
}       <Property value="неопределенная форма">
}         клиент
}       </Property>
}     </Properties>
}     <Properties>
}       <Property value="неопределенная форма">
}         карточка
}       </Property>
}     </Properties>
}   </SpecGroup>
}   <SpecGroup group="field" name="stage">
}     <Label>Стадия</Label>
}     <Properties>
}       <Property value="неопределенная форма">
}         стадия
}       </Property>
}       <Property value="род">
}         жен
}       </Property>
}     </Properties>
}   </SpecGroup>
} </Specification>

```

Рисунок 25. Пример описания ЕЯ-разметки на языке XML.

Данные XML-файлы мы назовем *картами настройки классов на предметную область*, т.к. фактически содержат в себе описание интерфейса класса и его естественно-языковую спецификацию. Карты настройки сохраняются в тех же пакетах, что и классы, которые они описывают и имеют тоже имя, что и имя описываемого класса.

2.5. Кодогенератор.

Рассмотрев принцип спецификации классов и способ их хранения, можно приступить к описанию кодогенератора. Данный компонент осуществляет интерпретацию поступающего ему на вход концептуального графа с помощью классов Java, подключенных к данной предметной области. Каждая предметная область в системе Nalaps описывается как *проект*, содержащий в себе библиотеки языка Java со специфицированными классами и пакеты расширенных сетей переходов. Таким образом кодогенератор обрабатывает только те классы, которые подключены к проекту.

В начале работы кодогенератор создает в памяти компьютера каталог специфицированных классов и их *карт настройки* для быстрого поиска и доступа.

Общий принцип интерпретации концептуального графа состоит из следующих положений:

а) каждый узел (а, следовательно, соответствующий ему *информационный объект*) интерпретируется как объект соответствующего ему класса Java;

б) внутреннее описание каждого *информационного объекта*, а именно все его признаки, интерпретируются как значения соответствующих полей объекта Java, поставленного в соответствие с данным *информационным объектом*;

в) каждая дуга графа, т.е. отношение между двумя *информационными объектами* (в вырожденном случае это унарное отношение, т.е. *петля графа*) интерпретируется как соответствующий метод объекта, выступающего в данном отношении в роли агенса. Пациент передается в метод как тело сообщения, т.е. как параметр (в случае унарного отношения подразумевается метод с пустым набором аргументов);

г) создание всех объектов в памяти компьютера предшествует вызову методов, соответствующих дугам графа; последующие заполнение значений и вызов его методов производится средствами рефлексии в Java.

На вызове методов следует остановиться подробнее. Отдельными особенностями является алгоритм вызова перегруженных методов, т.е. имеющих одинаковые имена, а также ограничения на порядок вызова этих методов. Особенностью спецификации методов является то, что специфицируется не сам метод как отдельная сущность с присущими ему параметрами, а фактически его имя в конкретном классе. Таким образом, с одной стороны это накладывает некоторые ограничения на способ написания классов программистом, но с другой стороны это позволяет реализовать более гибкое описание различных семантических валентностей одной роли. Например, рассмотрим предикат *родился* в примере:

Иван Иванович родился в 1899 году в Пензе.

У глагола «родиться» («рождаться») можно выделить, по крайней мере, 2 валентности:

- 1) Родился (кто?, когда?)
- 2) Родился (кто?, где?)

Соответственно, в классе *Person*, описывающим данные о человеке, целесообразно описать два одноименных метода:

- 1) `public void born (Date birthDate)` - метод для введения данных о дате рождения;
- 2) `public void born(String birthPlace)` – метод для введения данных о месте рождения.

Спецификацию имени метода *born* мы опишем как `@Semanticable(“рождаться, родиться”)`.

Результатом работы синтаксико-семантического компонента, положим, будет набор следующих семантических отношений:

родиться(Иван, 1899),
 родиться (Иван, Пенза).

Кодогенератор выбирает метод класса агенса для интерпретации семантического отношения, исходя из типа класса, который соответствует пациенсу. Так, в первом случае, *1899 год* будет сопоставлен с классом *Date*, описывающим дату. Соответствующее значение года будет передано экземпляру данного объекта. Для слова *Пенза* может отсутствовать особым образом специфицированный класс, тогда используется общий для таких случаев класс *String*, для описания строковых объектов. Итак, зная типы, соответствующие второму члену отношения, кодогенератор вызовет для первого отношения метод *born(1899) (1)* и *born(Penza) (2)* для второго соответственно.

Заметим, что при таком подходе нет смысла выделять семантические отношения на более глубинном уровне, т.е. выделять, например, отношение *локатив* по отношению к действию *рождаться*. Достоинством предложенного метода как раз и является отсутствие необходимости в большинстве случаев производить глубинный анализ, и наоборот достаточность поверхностного анализа. Глубинная семантика переносится на уровень описания ее в неявном (вообще говоря, для *не-программиста*) виде с помощью языка программирования.

Упомянутое выше ограничение на порядок вызова методов обусловлено тем, что сам кодогенератор не производит анализа: в каком порядке следует интерпретировать дуги концептуального графа и вызывать соответствующие методы. При описании лингвистического процессора был упомянут *стек отношений* который способен хранить порядок причинно-следственных связей. Ограничения именно в этом последнем компоненте накладывают ограничения на корректность порядка вызова методов. В свою очередь эти ограничения накладываются и на стиль исходного ЕЯ-текста. Можно

допустить, что большую часть технических текстов, диалоговый командный режим, простые описательные тексты и тому подобное, что не содержит художественных приемов, связанных с описанием причины *после* следствия, предложенная методика, несмотря на имеющиеся ограничения, способна анализировать успешным образом.

Важные следствия и достоинства получаемой в результате работы кодогенератора динамической объектной среды мы рассмотрим далее.

2.6. Динамическая объектная система как способ представления смысла текста.

Динамическая объектная система, получаемая в результате создания кодогенератором объектов специфицированных классов, обладает рядом важных свойств.

Отметим в начале, что фактически каждый объект (экземпляр) класса языка Java, который был обработан естественно-языковым спецификатором, является *сигнификатом*, *концептом* или *интенсионалом* – т.е. тем, что в современной лингвистике именуется *смыслом* (Степанов, 2007) – по отношению к реальной вещи (явлению, процессу) объективной действительности. С этой же точки зрения, говоря об интенсionaле отдельных предложений и текста в целом, мы можем утверждать, что система таких объектов, когда связи между ними соответствуют связям в исходном тексте, также является *смыслом исходного текста*. Здесь, однако, наличествует ряд тонких моментов. Во-первых, в классическом понимании *смысла*, один и тот же интенсional можно записать разными способами, применяя трансформацию в экстенсionalной сфере. Например, у предложений:

Иван Петрович родился в 1899 году

и

Иван Петрович появился на свет, когда шел 1899 год по Р.Х.

очевидно одинаковый смысл, который можно записать, например, в виде семантического графа в некотором атомарном виде, как, например, у И. Мельчука:

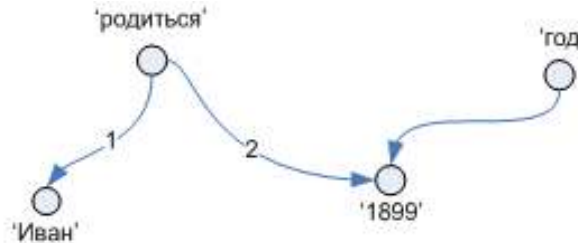


Рисунок 26. «Атомарный» смысл выражения.

При этом, например, у Мельчука декларируется, что от такого семантического представления можно с помощью системы перефразирования получить, в конечном счете, все возможные (правильные или допустимые) предложения (через промежуточные *глубинно-синтаксические структуры*) (Мельчук, 1999). Такой подход, в частности, реализован в системе ЭТАП-3 (Иомдин Л.Л., 2009). Система Nalaps не обладает таким компонентом *перефразирования*. Кроме того, такой переход от динамической объектной системы в общем случае невозможен в принципе. О таком переходе можно вести речь лишь на уровне концептуального графа. Но при этом динамическая система обладает другими уникальными особенностями. Рассмотрим еще один пример. Можно ли утверждать, что у следующих двух предложений (высказываний) одинаковый смысл?

а) *Иван – отец Петра.*

б) *Петр Иванович*

В случае б) подразумевается, что отчество содержит информацию об имени отца человека, т.е. фактически ту же информацию, что и предложение (а). Вопрос можно поставить и по-другому: можно ли осуществить

построение фразы (б) на основании атомарного семантического графа, полученного после анализа (а) и наоборот? Между тем, реализовать такой переход с помощью динамической объектной системы не представляет практической сложности. Для этого достаточно запрограммировать в метод назначения ссылки на отца в классе *Person* простейший алгоритм вычисления отчества по имени отца, а для обратного перехода – определение имени из отчества и создание нового объекта класса *Person*, для описания отца. Ниже схематично представлен процесс перехода от (а) к (б) (рис. 24).

У объекта *Person* (1) вызывается метод «УстановитьОтца()» с параметром *Person* (2):

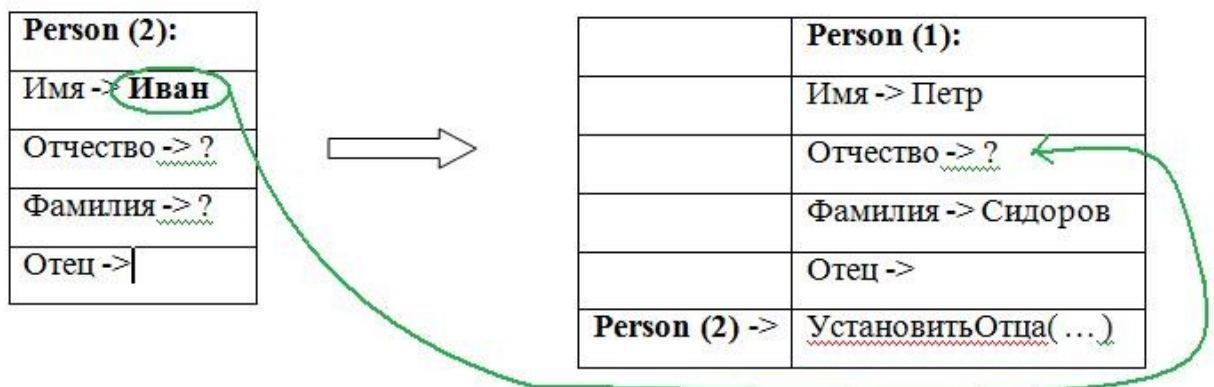


Рисунок 27. Переход от (а) к (б).

Итак, описанную на данном примере особенность динамической объектной системы следует сформулировать в обобщенном виде: динамическая объектная система позволяет выявлять скрытые связи в ЕЯ-тексте, настолько глубоко, насколько глубоко и полно описана абстракция соответствующих детонантов объективной действительности на языке Java в специфицированных классах. Это позволяет уйти от проблемы получения некоего низкоуровневого «атомарного» семантического представления на этапе работы лингвистического процессора, и выделять только те глубинные связи, которые характерны и актуальны именно в данной конкретной предметной области.

Следующая особенность динамической объектной системы вытекает из технического характера возможностей реализации специфицируемых классов, а именно в виде независимых потоков. Применение данной методике можно найти само разнообразное. Например, расширить трехмерную комнату геометрических примитивов, реализованную Т. Виноградом в его системе SHRDLU, с помощью подключения к системе физического процессора, т.е. эмуляции физических законов. Данная игровая система была реализована с помощью системы Nalaps (об этом подробнее см. гл. 4) (Путилов Г.П., 2009). Вытекающие отсюда следствия: независимые потоки взаимодействуют друг с другом, в результате у одних объектов меняются свойства, другие объекты уничтожаются, третьи наоборот появляются. Динамическая объектная система действительно становится *динамической* и *независимо* динамической, т.е. ее динамика становится обусловлена не только командами, посылаемыми кодогенератором, но и за счет внутренних независимых механизмов.

2.7. Обратный концептуальный и обратный лингвистический процессоры.

Обратный концептуальный и обратный лингвистический процессоры, точнее говоря, их прототипы, целесообразно рассмотреть вместе. Назначение обратного концептуального процессора вытекает из рассмотренных выше особенностей динамической объектной системы: он призван отслеживать такие изменения и преобразовывать их в новые (при необходимости) узлы и связи, или удалять существующие. Данный механизм имеет свои ограничения и особенности.

Во-первых, при отслеживании свойств (значений полей) объектов исследуется только специфицированные компоненты классов, т.к. их значения можно соответствующим образом интерпретировать в обратную сторону.

Во-вторых, вызов специфицированных методов, производимый внутри классов не по командам кодогенератора, а по внутренним алгоритмам, не отслеживается; следовательно, извлечь подобного рода информацию и интерпретировать в настоящий момент система не может. Однако, техническая реализация такой задачи представляется возможной, и исследования по данной проблеме запланированы как будущие этапы развития проекта.

В-третьих, для обратного концептуального процессора заложен механизм, позволяющий заполнять пустые вакантные места в семантических валентностях, если соответствующая естественно-языковая разметка присутствует в классах. Поясним данный механизм на примере. В упомянутой выше системе эмуляции трехмерных примитивов с подключенным физическим процессором, реализуется поведение объектов при их столкновении, падении и пр. Так, при ударе одного объекта, падающего с высоты на другую группу объектов, может измениться их взаимное расположение в пространстве, а также относительно угла зрения наблюдателя (сцену можно вращать), см. рис. 28. Чтобы узнать новое взаимное расположение объектов оператор системы может задать вопрос, например: *«где находится синий кубик?»*. Лингвистический процессор при этом должен создать группу не полностью заполненных предикатов, вида:

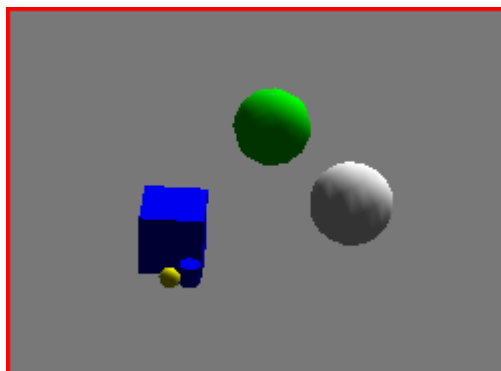


Рисунок 28. Снимок экрана сцены трехмерных примитивов.

справа(синий_кубик, ?)

слева(синий_кубик, ?)

перед(синий_кубик, ?)

... и т.д.

Данные отношения затем будут проверены обратным концептуальным процессором путем вызова у объекта, описывающего *синий кубик* специальных методов, имеющих соответствующую спецификацию и возвращающих ссылки на объекты, соответствующие данному взаимному расположению. Например, некоторый метод *__before()* может вернуть ссылку на объекты *желтый_кубик* и *синий_цилиндр*. Обратный концептуальный процессор интерпретирует эти ссылки и заполнит соответствующие пустые (знак «?» в предикатах) места. В случае, если подобные методы возвращают пустое значение (*NULL*), данные предикаты удаляются.

Разумеется, описанный механизм требует от программиста придерживаться определенного интерфейса написания методов специфицируемых классов. Так, при описании метода, который устанавливает взаимное расположение двух фигур в пространстве в соответствии с углом зрения наблюдателя как «*некоторая фигура слева от другой фигуры*», программист должен реализовать два метода наподобие следующих:

- 1) `public void left(Object3D otherObj);` - для установки соответствующего взаимного расположения, и
- 2) `public Object3D __left();` - для получения ссылки на объект, который располагается в данный момент слева от данного объекта.

При этом спецификация данных методов будет **одна**. Т.е. не требуется вводить естественно-языковую разметку для каждого из методов `left(Object3D)` и `__left()`.

2.8. Редактор расширенных сетей переходов.

Последний компонент системы, который будет рассмотрен в данной главе – это редактор расширенных сетей переходов с графическим интерфейсом пользователя (Лебедев, 2009)(Путилов Г.П., 2009).

Редактор представляет собой оконное приложение, написанное на языке Java 1.5. В качестве морфологического анализатора теоретически может использоваться любой, практически же один из рассмотренных выше (программа *mystem* или морфологический анализатор *Abbyy*). Практическая заменимость морфологического анализатора достигается за счет его реализации в отдельном пакете.

Интерфейс редактора представляет собой окно, позволяющие отобразить в отдельных вкладках несколько расширенных сетей в виде графов (см. рис. 30).

Конструирование сети производится с помощью мыши. Правым щелчком мыши у выбранного узла сети создаются новые ребра. Нумерация узлов производится автоматически. Признаком конца сети является латинская буква «x» в начале имени узла. Морфологические свойства, которыми обладают ребра, могут быть заданы путем выбора нужного ребра, двойным щелчком мыши на выбранном ребре и последующим редактированием окна ввода свойств ребра (см. рис. 30). Во вкладке «Собственные свойства» задаются парные морфологические свойства в виде «свойство - значение», например: «падеж – вин».

Аналогичным образом осуществляется редактирование свойств узла (см. рис. 29). Под свойствами узла также понимаются парные совокупности, типа «свойство – значение». Название свойств не регламентировано, что позволяет при разработке ATN-анализатора не привязываться к фиксированным именам-константам: например, в качестве имени свойства, описывающего семантическое отношение в использованной реализации

АТН-анализатора, выбрано имя «role». Семантическая роль описывается как заключенная в круглые скобки последовательность числовых значений, выражающих смещение влево

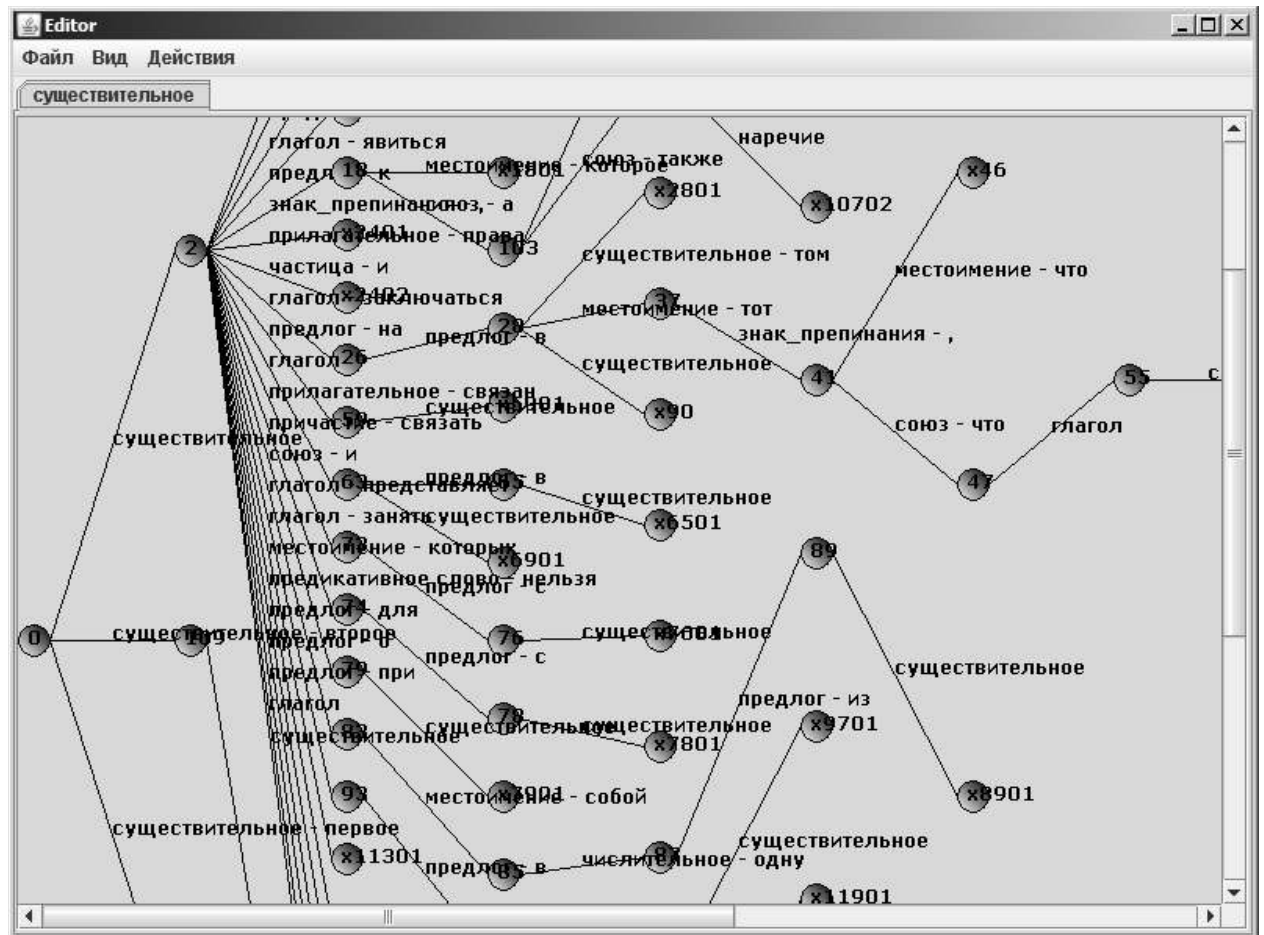


Рисунок 29. Снимок экрана визуального редактора расширенных сетей переходов.

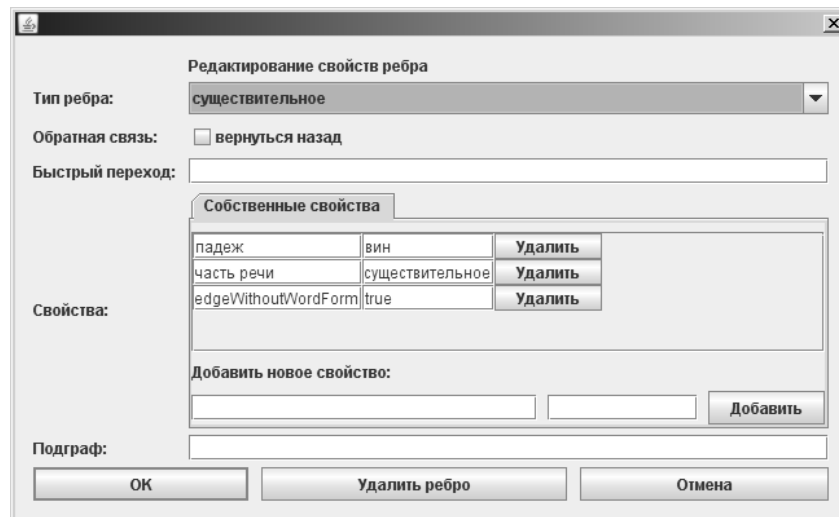


Рисунок 30. Снимок экрана редактирования свойств ребра графа.

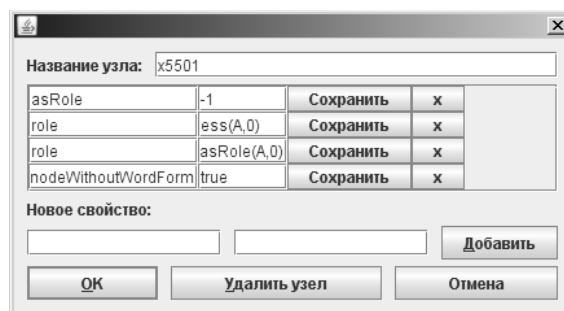


Рисунок 31. Снимок экрана редактирования свойств узла графа.

относительно текущего узла сети. Например, если в семантическое соотношение нужно включить слово, связанное с текущим ребром (т.е. ребром, «правым» концом которого является текущий узел), то его смещение будет равно «0», предыдущее ребро – «-1» и т.д. Порядок указания смещений соответствует порядку в описании семантической роли. Пользователь может выбрать узлы, участвующие в создаваемой роли, путем щелчка мыши по нужным узлам в соответствующем порядке. Затем достаточно выбрать узел для создания роли и указать ее название.

Редактор предоставляет пользователю широкие возможности по визуальному конфигурированию графа на экране, в том числе автоматическое выстраивание узлов сети, обеспечивающие удобное

зрительное восприятие картинки, а именно, исключает пересечения ребер графа и представляет его в виде дерева. В редакторе реализована функция прокрутки экрана, что позволяет редактировать сети практически неограниченного объема.

Для ускорения конструирования сетей в редакторе представлена возможность автоматического построения ветвей графа по введенному предложению. Данный процесс выглядит следующим образом: пользователь вводит предложение, которое разбивается на отдельные лексические единицы (слова, знаки препинания, цифры), и каждая единица обрабатывается встроенным морфологическим анализатором. В строгой последовательности, в соответствии с введенным предложением, система строит ветвь сети, где каждому ребру приписывается набор морфологических признаков, взятых у соответствующего слова. Пользователю редактора остается только удалить лишние морфологические свойства и добавить нужным узлам сети свойства или семантические роли.

При большом числе вершин графа картинка может оказаться весьма громоздкой и трудной для восприятия. Для устранения этого эффекта предлагается использование фильтров, в частности:

- «только со словоформами» (т.е. отображаются только те дуги, чьи фреймовые структуры содержат конкретные словоформы)
- «только универсальные» (т.е. отображаются только те дуги, чьи фреймовые структуры не содержат конкретных словоформ).

В функционал программы включен также поиск по сетям. Экран окна поиска представлен на рис. 32.

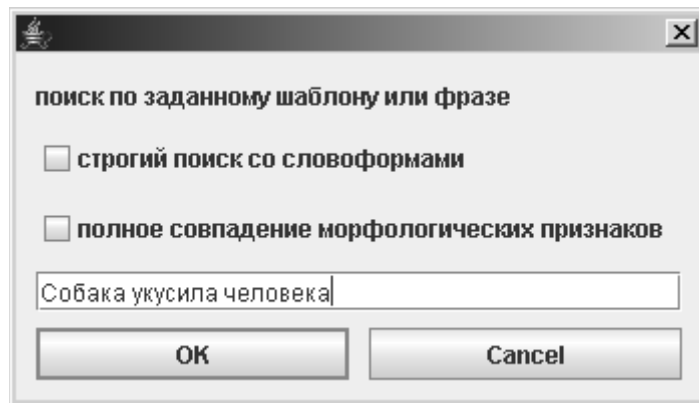


Рисунок 32. Снимок экрана окна поиска.

Поиск можно производить в обычном режиме, а также в комбинации режимов «строгий поиск со словоформами» и «полное совпадение морфологических признаков».

Функция поиска также использует морфологический анализатор. В обычном режиме будут найдены все пути графа, позволяющие произвести разбор данного предложения. В режиме строго поиска отобразятся те пути графа для разбора данного предложения, дуги которых содержат точное совпадение по словоформам введенной строки поиска. В режиме полного совпадения морфологических признаков отобразятся те пути графа, чьи дуги содержат морфологические признаки, которые совпадают с соответствующими признаками слов строки поиска, однако совпадение словоформ в данном режиме не обязательно.

В системе также реализован механизм внутреннего поиска, который запускается каждый раз при вызове автоматической перестройки графа. Данный алгоритм позволяет автоматически отслеживать одинаковые фрагменты сети, расположенные в разных частях графов и при возможности их объединять. Под двумя одинаковыми фрагментами в данном случае понимаются такие, которые имеют началом либо корневой узел графа, либо оканчиваются терминальными узлами (помеченными префиксом «х»), и имеют совпадающий по длине и соответствующим свойствам ребер и узлов

некоторый путь. В частности, данный механизм удобен в случае использования автоматического построения «скелета» сети по шаблону-строке, когда после окончания редактирования свойств ребер требуется перестроить граф, объединив одинаковые фрагменты.

Сконструированные сети сохраняются в виде XML файлов, что упрощает применение созданных файлов в стороннем программном обеспечении благодаря легкости обработки языка XML. Классы, описывающие объекты графов, предоставляются в открытом доступе; таким образом, их можно использовать непосредственно при программировании на языке Java для использования сконструированных расширенных сетей переходов. Ниже приведен фрагмент XML файла, описывающий один узел сети:

```
<void property="node1">
  <object id="GraphXNode0" class="g.GraphXNode">
    <void property="name">
      <string>0</string>
    </void>
    <void property="parent">
      <object idref="GraphX0"/>
    </void>
    <void property="properties">
      <void method="add">
        <object class="g.GraphXProperty">
          <void property="name">
            <string>editor_x</string>
          </void>
          <void property="value">
```

```

    <string>0</string>
  </void>
</object>
</void>
<void method="add">
  <object class="g.GraphXProperty">
    <void property="name">
      <string>editor_y</string>
    </void>
    <void property="value">
      <string>320</string>
    </void>
  </object>
</void>
</void>
</object>
</void>

```

Созданный визуальный редактор расширенных сетей переходов позволяет облегчить работу эксперта-программиста лингвистического процессора, основанного на использовании АТН-анализатора. Он включает в себя все необходимые функции, связанные с обработкой графов, назначением свойств ребер и вершин. Включение в состав редактора морфологического анализатора позволяет строить «скелеты» сетей по заданному предложению-шаблону, а алгоритм внутреннего поиска отслеживает наличие подобных структур в уже имеющейся базе знаний, что упрощает создание и расширение существующих сетей.

На использовании морфологического анализатора также основана функция поиска, которая позволяет эксперту быстро находить нужные фрагменты сетей для дальнейшего анализа или редактирования.

Фильтрация отображения позволяет отделить на визуальном уровне универсальные части сети от частей, предназначенных для обработки определенных особых словоформ и конструкций.

В настоящий момент разработанный редактор в совокупности представляемых им набором функций и удобным пользовательским интерфейсом является уникальным инструментом подобного типа.

2.9. Выводы

Предложенная архитектура системы обладает рядом преимуществ, обусловленных ее модульностью. Так, лингвистический процессор и его внутренние компоненты (морфологического и синтактико-семантического анализа) могут быть при необходимости заменены другими модулями, например, в целях введения поддержки иных языков или применения альтернативных и более эффективных подходов к семантическому анализу. Благодаря разработанной архитектуре, система обладает возможностью прямого взаимодействия с виртуальной машиной Java (JVM), что позволяет организовать работу обратного концептуального процессора.

Лингвистический процессор, описанный в данной главе, а также средства его настройки (визуальный редактор расширенных сетей переходов), являются отдельными независимыми программными продуктами, которые могут применяться независимо от системы Nalaps. Такое применение этих компонентов было найдено в ряде научно-

исследовательских работ (ГУ "НИИ МЭИИТ МИЭМ"; Руководитель Путилов Г.П.; Тумковский С.Р., Белялетдинов И.И., Игнатовский А.Н., Смирнова Е.В., Галыгин А.Н., Михейкин И.В., Лебедев А.С., Зонтов Ю.В., Бобин А.В.).

Глава 3.

В данной главе предлагается математическое описание предложенного метода перехода от формализованного представления текста к концептуальному графу, и от концептуального графа к коду программы на языке Java.

3.1. Общие положения.

Пусть:

T – связный текст,

S_i – i -ое предложение текста T .

Под связным текстом понимается цельный текст, где наличествуют связи между его предложениями, и между группами предложений (сверхфразовыми единицами) (Каменская, 1990).

Далее работа системы рассматривается по компонентам.

3.2. Работа морфологического процессора.

Пусть $S = x_1 x_2 x_3 x_4 x_5 \dots x_n$ – входная строка, где x_i – символы алфавита $A = \{ A..Za..zA..Яа..я0..9.,!?" \}$.

Функция $St(S) = \{ st_i | i = \overline{1..N} \}$ разбивает строку S на предложения st_i (членение по символам «.!?»).

Функция $Wd(st) = \{ w_i | i = \overline{1..M} \}$ разбивает предложение st на слова и знаки пунктуации (членение по пробелам).

Каждому слову w отображение Mr ставит в соответствие фреймовую структуру f_m . Например, для словоформы «Петра» (см. рис. 33):

L атрибутов

Атрибут (p _i)	Множество значений (V _i)
Неопределенная форма	Петр
Падеж	род., вин.
Число	ед.
...	...

Рисунок 33. Пример разбора словоформы "Петра"

$$f_m = \text{Петр}$$

Множество всех возможных фреймовых структур задается в виде:

$$Fm = \{f_{m_k} | k = \overline{1..M}\}$$

3.3. Работа синтактико-семантического анализатора.

Синтактико-семантический анализатор построен на формализме расширенных сетей переходов.

Определение 1. Расширенная сеть переходов *atn* определяется, как связанный ориентированный граф, задаваемый четверкой: *atn* = (*Nd*, *T*, *label_{Nd}*, *label_T*), где:

Nd – упорядоченный набор вершин графа,

T – упорядоченный набор ребер графа.

Отображение $label_{Nd}(nd \in Nd) = \{relation_i(nd_k, nd_l) | i = \overline{1..N}\}$ задает метки вершин графа.

Отображение $label_T(t \in T) = \{f_{m_i} | i = \overline{1..N}\}$ или Петр , где *V_N* – словарь нетерминалов, задает метки ребер графа.

Обработка расширенной сети переходов осуществляется с помощью *машины сети переходов*, определяемой следующим образом:

Определение 2. Машина сети переходов M_{ATN} задается как пятерка:

$M_{ATN} = (V_T, V_N, S, M, I)$, где:

- V_T - словарь терминалов, совпадает со множеством Fm , т.е. это конечные словоформы, или шаблоны словоформ;
- V_N – словарь нетерминалов, т.е. определенных компонентов предложения, таких как *группа подлежащего*, *группа сказуемого* и т.д.;
- S – начальное состояние машины сети переходов;
- M – набор конечных автоматов со входным словарем $V = V_T \cap V_N$;
- I – функция индексирования, назначающая каждому нетерминалу $v_{N_i} \in V_T$ уникальный автомат из M .

Описанная таким образом машина сети переходов начинает свою работу из начального состояния S . Переход в графе сети переходов осуществляется по наиболее подходящей текущему слову из входного потока дуге графа. При попадании на дугу, помеченную нетерминальным символом, осуществляется назначение уникального конечного автомата для обработки данного нетерминала, т.е. фактически осуществляется вызов машины обработки сети переходов, соответствующей данному нетерминалу.

Результатом работы машины сети переходов M_{ATN} является путь обхода имеющихся в базе данных сетей. Применяя суперпозицию описанных функций морфологического и синтактико-семантического анализа к входной строке, может быть получен набор пар значений, записанных в виде: «<идентификатор сети переходов>.<идентификатор узла>»:

$$M_{ATN} \circ Mr \circ Wd \circ St (S) = \textcolor{red}{i} \textcolor{red}{i}$$

Синтактико-семантическое представление **SSP** - это ориентированный граф с вершинами $atn_{k_i}.nd_{l_j}$ и ребрами, инцидентность которых вычисляется

путем анализа меток $label_{Nd}(nd \in Nd) = \{relation_i(nd_k, nd_l) | i = \overline{1..N}\}$.

Данные метки представляют собой информацию о типе семантического отношения и о направлении конструируемой дуги графа: *relation* – имя отношения, nd_k - начальный узел дуги, nd_l - конечный узел.

Каждому узлу $atn.nd$ графа **SSP** ставится в соответствие f_m , полученная путем морфологического анализа терминала (слова) из входного потока.

Каждому ребру, где *relation* также описывается через терминал, ставится в соответствие также f_m , соответствующая лексеме из входного потока, описывающей имя отношения.

3.4. Проблема интерпретации формализованного представления ЕЯ-текста на языке программирования.

Естественная метаонтология $Ont(nat)$ представляется в виде:

$Ont(nat) = (\text{объект}, \text{свойство}, \text{отношение})$ (Kulba, et al., 2009).

Она может быть рассмотрена в разрезе *естественного языка* (на примере русского языка) и в разрезе машины обработки данных (на примере Java):

$\text{Ont}(\text{nat}) = (\text{Объект}, \text{свойство}, \text{отношение})$

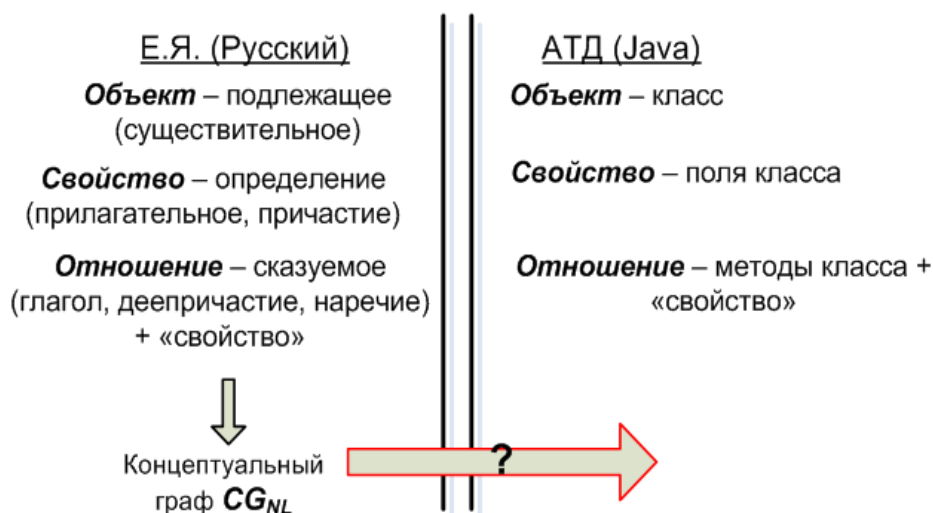


Рисунок 34. Сравнение онтологии естественного языка и языка Java

Как показано на рис. 34, в естественном языке, в предложении, в качестве *объекта* может быть рассмотрено *подлежащее*, в качестве *свойства* – *определение*, и в качестве *отношения* – *сказуемое*. Получаемое на основе данной онтологии формализованное представление – концептуальный граф (CG_{NL}). Такой граф назовем *естественно-языковым концептуальным графом*. В абстрактном типе данных (класс Java) прослеживается аналогичная естественная онтология, где в роли *объекта* может быть рассмотрено *имя класса*, в качестве *свойства* – *поля класса*, а в роли *отношения* – *методы класса* и в том числе его *свойства* (отношение типа «свойство»). В то же время, такая структура (класса Java) не всегда имеет семантически адекватные имена компонентов: зачастую они написаны на английском языке и в форме акронимов, сокращений и пр. По этой причине явный переход от естественно-языкового графа к классу Java является в общем случае невозможным (на рис. показано стрелкой со знаком вопроса). Однако данная проблема может быть решена путем введения в класс *естественно-языковой спецификации*, которая позволяет семантизировать абстрактный тип данных, см. рис. 35.

Ont(nat) = (Объект, свойство, отношение)

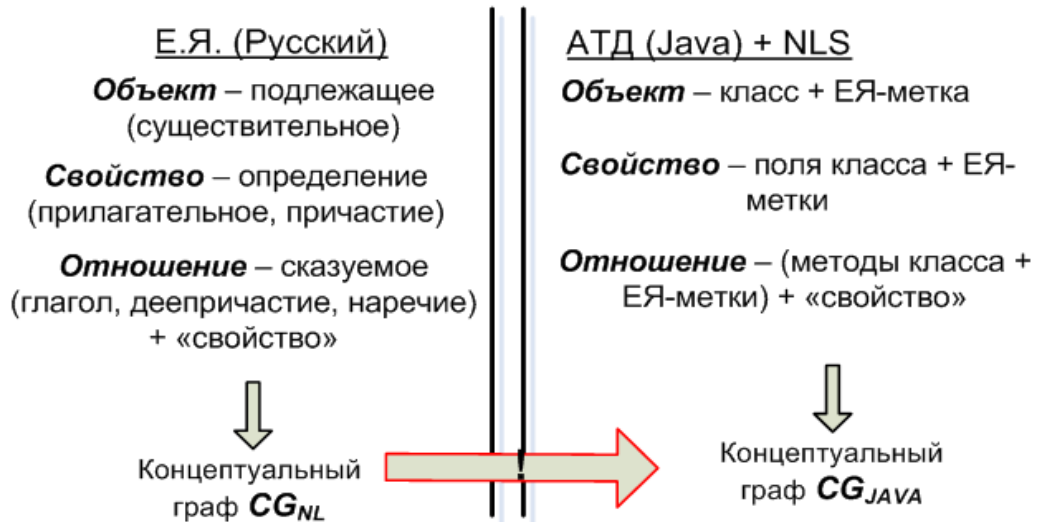


Рисунок 35. Сравнение онтологий ЕЯ и специфицированных классов Java

Описание CG_{Java} - концептуального графа класса Java будет дано позже. Итак, в общем случае, задача состоит в построении проекции Π из CG_{Java} в CG_{NL} , такой, что $CG_{NL} \leq CG_{Java}$. Требуемое отношение между концептуальными графами обусловлено следующими соображениями: специфицированный абстрактный тип данных может описывать понятие или ситуацию *полнее*, чем она представлена в исходном тексте (запросе). По этой причине концептуальный граф класса будет всегда равен или превосходить соответствующий ему подграф естественно-языкового концептуального графа. Для пояснения и формализации данного соответствия следует рассмотреть разложение естественно-языкового концептуального графа на подграфы отдельных *объектов*.

Пусть CG_{Ob} - подграф CG_{NL} для отдельного объекта естественно-языковой онтологии, CG_{Java} – концептуальный граф отдельного специфицированного класса Java. Тогда:

$$\begin{aligned}
CG_{NL} &= CG_{Ob_1} + CG_{Ob_2} + \dots + CG_{Ob_N} = \Pi(CG_{Java}) = \\
&= \Pi(CG_{Class_{K_1}} + CG_{Class_{K_2}} + \dots + CG_{Class_{K_N}}) = \\
&= \Pi(CG_{Class_{K_1}}) + \Pi(CG_{Class_{K_2}}) + \dots + \Pi(CG_{Class_{K_N}})
\end{aligned}$$

Таким образом, с каждым $Ob_{EЯ}$ необходимо сопоставить специфицированный класс $Class_{Java}$, каждое отношение типа «свойство» проинтерпретировать как поля класса, а остальные отношения – как методы класса.

3.5. Формальная модель концептуального графа

Определение 3. Концептуальный граф CG задается как ориентированный граф, представляемый следующей тройкой:

$CG = (C, U, label)$, где:

C – множество концептов, причем $C \neq \emptyset$, задающее вершины графа.

U – множество ребер, задающее отношение между концептами.

Каждая вершина имеет метку, определенную отображением

$label(c) = \text{л\textsubscript{е}к\textsubscript{с}е\textsubscript{м}\textsubscript{а}},$ которое сопоставляет с вершиной конкретную лексему и ее фреймовую структуру.

Каждое ребро u имеет метку:

$label(u) = \{ \text{с\textsubscript{в}\textsubscript{о\textsubscript{й}\textsubscript{с}\textsubscript{т}\textsubscript{в}\textsubscript{о}} \} + \text{л\textsubscript{е}к\textsubscript{с}е\textsubscript{м}\textsubscript{а}}.$

Естественно-языковой концептуальный граф CG_{NL} задается с помощью семантической сети SSP , полученной на выходе из синтактико-семантического анализатора.

Необходимо отметить, что предложенная модель концептуального графа отличается от модели концептуального графа, предложенной Дж. Сова (Sowa, 2008). Однако, как будет показано далее, существует биективное

отображение между двумя моделями, а различие заключается в способе представления отношений между концептами.

Для пояснения указанного факта можно рассмотреть падежный фрейм Ч. Филмора (Филмор, 1981), представленный в схематичном виде на рис. 36.

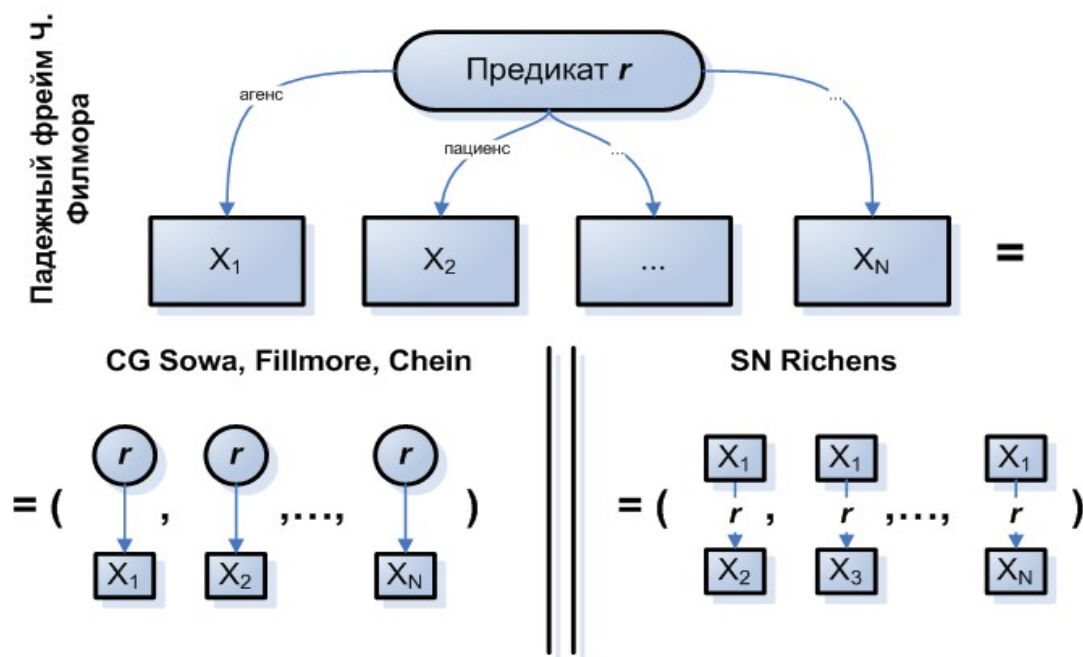


Рисунок 36. Схема представления падежного фрейма

В верхней части рисунка изображен предикат r с набором семантических ролей $x_1, x_2, \dots, x_n$, первые два из которых обозначены как *агенса* и *пациенса* соответственно. В нижней части рисунка изображены две параллельных схемы представления падежного фрейма в концептуальном графе Совы (слева) и в семантической сети Риченса (Richens, 1956) (справа). Способ представления падежного фрейма в предложенной в данной работе модели концептуального графа соответствует правой схеме рисунка (сеть Риченча).

Утверждение 1: Существует взаимно однозначное отображение B для концептуального графа Совы (CG_{Sowa}) и концептуального графа CG :

$$\exists B_r(CG_{Sowa}) = CG, \exists B_r^{-1}(CG) = CG_{Sowa}$$

Для доказательства утверждения данное отображение будет построено в прямую и обратную сторону.

Для перевода CG_{Sowa} в CG отображение строится следующим образом. На первом шаге рассматриваются все отношения r_i , соединяющие предикат r и соответствующий ему узел с объектом x_i . На втором шаге среди таких r_i выбирается семантическая роль *агенса* – r_a . На третьем шаге для каждого r_i кроме r_a в конструируемом CG строится отношение r_i' , связывающее соответствующий r_a узел x_a с соответствующим r_i узлом объекта x_i :

$$B_r = \begin{cases} 1. r: \{r_1, r_2, \dots, r_n\}; r_i = (r, x_i) \\ 2. \exists r_a: r_a = \text{ital агентс} \quad \exists r_i' = (x_a, x_i) \end{cases}$$

Построение обратного отображения также состоит из трех шагов. На первом шаге рассматриваются все отношения в CG – это дуги, инцидентность которых задается в виде (x_a, x_i) . Для каждого такого r_i' в CG_{Sowa} конструируется отношение r_i – дуга графа, соединяющая вершину предиката r и вершину x_i . На третьем шаге строится последняя дуга графа, отражающая семантическую роль *агенса*, инцидентность которой задается парой (r, x_a) :

$$B_r^{-1} = \begin{cases} 1. r_i' = (x_a, x_i) \\ 2. \forall r_i' \exists r_i = (r, x_i) \\ 3. r_a = (r, x_a) \end{cases}$$

Таким образом построено взаимно однозначное соответствие между CG_{Sowa} и CG . Наличие такой биекции важно потому, что для концептуальных графов J. Sowa определен ряд операций. Пусть G – это концептуальный граф CG_{Sowa} , тогда концептуальный граф $(CG_{Sowa}) H$ может быть выведен из G путем:

- Копирования

- Удаления (любого концепта, при сохранении связности графа и отсутствии «висячих» связей)
- Разделения (удаления любой связи, при сохранении связности графа)
- Обобщение
- Уточнение $c_i \in G, c_j \in H: \text{type}(c_i \in G) \subseteq \text{type}(c_j \in H) \wedge \text{label}(c_i) = \text{label}(c_j) \Rightarrow$
- Соединение $\Rightarrow c_i \leftrightarrow c_j$

В силу построенной биекции эти операции применимы и к концептуальному графу **CG**. Операции *удаления+разделения* и *соединения*

делают возможным разбиение: $CG_{NL} = CG_{Ob_1} + CG_{Ob_2} + \dots + CG_{Ob_N}$.

При этом концептуальный граф отдельного объекта **CG_{Ob}** определим как: $CG_{Ob} = (\{c_{Ob}\} + \{c_i | \exists u_k = (c_{Ob}, c_i)\}, \{u_j | u_j = (c_{Ob}, c_l)\})$,

т.е. такой граф строится из вершины самого объекта **Ob** (c_{Ob}), всех дуг, исходящих из данной вершины с соответствующими им конечными вершинами.

Рассмотрим модель класса Java, специфицированного естественным языком. Такой класс можно описать четверкой: $class_{nl_s} = (name, Fl, Mt, label)$, где:

name – имя класса,

Fl – упорядоченный набор его полей (атрибутов),

Mt – упорядоченный набор методов.

label – отображение, задающее естественно-языковую разметку.

Совокупность таких классов образует набор терминологических аксиом T_{CJava} .

$\forall fl \in Fl: fl = (name_{fl}, cl_{fl} \in T_{cjava})$, где cl_{fl} задает тип поля, а $name_{fl}$ – его имя.

$\forall mt \in Mt: mt = (name_{mt}, \{0\} \cup \{arg_{mt} \in T_{cjava}\})$, где arg_{mt} – описывает тип аргумента метода.

Описав таким образом модель специфицированного класса языка Java можно дать определение концептуальному графу класса языка Java.

Определение 4. Концептуальный граф CG_{java} класса $class_{nls}$ описывается тройкой: $CG_{java} = (C_J, U_J, label)$, где:

C_J – упорядоченный набор специфицированных классов, являющихся вершинами графа:

$$C_J = \{class_{nls}\} + \{arg_{mt} | mt \in class_{nls}.Mt\} + \{cl_{fl} | fl \in class_{nls}.Fl\}$$

U_J – упорядоченный набор ребер графа,

$$U_J = \{свойство\} + \{name_{mt} | mt \in class_{nls}.Mt\},$$

$$\forall u \in U_J: u = (class_{nls}, arg_{mt_i}) \text{ или } u = (class_{nls}, cl_{fl_j})$$

Отображение $label$ задает естественно-языковую разметку (маркеры) для каждой вершины и ребра графа в соответствии с разметкой класса:

$$label(u) = \{свойство\} + \textcolor{red}{\text{!}} \textcolor{red}{\text{!}}.$$

3.6. Операция сравнения фреймовых структур f_m

Для построения искомой проекции Π необходимо также описать операцию сравнения фреймовых структур f_m , полученных в результате работы морфологического процессора.

Для сравнения элементов на множестве Fm вводится функция σ :

$$f_{m_i} < f_{m_j} \Leftrightarrow \sigma(f_{m_i}, f_{m_j}) < 0,$$

$$f_{m_i} = f_{m_j} \Leftrightarrow \sigma(f_{m_i}, f_{m_j}) = |f_{m_i}|,$$

$$f_{m_i} <> f_{m_j} \Leftrightarrow \sigma(f_{m_i}, f_{m_j}) = 0$$

Последнее условие описывает случай, когда две фреймовые структуры f_m не сравнимы между собой. Саму функцию σ опишем как:

$$\sigma(f_{m_i}, f_{m_j}) = \sum_k \chi(p_{i_k}, p_{j_k}) \cdot \prod_k \|\chi(p_{i_k}, p_{j_k})\|, \text{ где}$$

p_i - элементы множества морфологических признаков, задающих имена слотов в структуре f_m . Вспомогательная функция χ описывается как:

$$\chi(p_i, p_j) = \begin{cases} 0, & V_i \cap V_j = \emptyset \\ 1, & V_i \supseteq V_j \\ -1, & V_i \subset V_j \\ \frac{1}{2}, & |V_i| \geq |V_j|, V_i \cap V_j \neq \emptyset \\ -\frac{1}{2}, & |V_i| < |V_j|, V_i \cap V_j \neq \emptyset \end{cases}$$

т.е. функция χ описывает способ сравнения набора значений для двух одноименных морфологических признаков p_i и p_j : результат их сравнения считается равным 0, если для данного признака p у двух фреймовых структур нет ни одного совпадающего значения. Значение ± 1 устанавливается в случае полного вхождения одного набора в другой. Значение $\pm \frac{1}{2}$ устанавливается в случае пересечения набора значений (есть общие значения), но отсутствия полного совпадения или вхождения одного набора в другой.

3.7. Построение проекции из CG_{Java} в CG_{NL} .

Фактически, конструируемая операция проекции – это графовый морфизм, сохраняющий порядок ребер и разрешающий ограничение меток вершин.

Определение 5. Проекция Π из $CG_{java} = (C_J, U_J, label_J)$ в $CG_{NL} = (C_N, U_N, label_N)$ – это упорядоченная пара $\Pi = (f, g)$ отображений, f из U_J в U_N , и g из C_J в C_N , таких, что:

1. $\forall u \in U_J, f(u) \in U_N$, и если $c = CG_{J_i}(u)$, то $g(c) = CG_{NL_i}(f(u))$
2. $\forall u \in U_J, lab_J(u) \geq lab_N(f(u)) \Leftrightarrow \sigma(f_{m_u}, f_{m_{f(u)}}) > 0$
3. $\forall c \in C_J, lab_J(c) \geq lab_N(g(c)) \Leftrightarrow \sigma(f_{m_c}, f_{m_{g(c)}}) > 0$

Иными словами, условие (1) сохраняет ребра и порядок на ребрах графа.

Таким образом, предложено решение проблемы программной интерпретации формализованного представления текстов путем построения проекции на концептуальном графе CG_{NL} , а также операция сравнения графов ($CG_{NL} \leq CG_{Java}$). Следует отметить, что теоретические обоснования применимости данных операций (проекции и сравнения) к графам имеются также в работах М. Chein и M.L. Mugnier (Chein, et al., 1995), где с применением дескриптивной логики описываются операции концептуальных графов, а также их морфизм. В (Chein, и др., 1995) приводится доказательство теоремы, согласно которой, если G и H – концептуальные графы, и $G \leq H$, если существует проекция из H в G . Это же утверждение верно и в обратную сторону, что было доказано J. Sowa в (Sowa, 1984): если G и H – концептуальные графы, и $G \leq H$, то существует проекция из H в G .

В предложенной модели под *информационными объектами* следует понимать те объекты естественно-языковой онтологии, которые являются вершинами концептуального графа CG_{NL} . При анализе текста, для описания одних и тех же объектов действительности используются одни и те же *информационные объекты*. Т.е. узлы концептуального графа CG_{NL} содержат фактически ссылки на экземпляры объектов, что позволяет одному и тому же объекту находится в нескольких концептуальных графах (отдельных предложений) сразу.

Глава 4.

В данной главе дается описание системы Nalaps с технической точки зрения ее реализации. Здесь также даются описания пользовательских интерфейсов системы (за исключением редактора расширенных сетей переходов).

4.1. Описание проекта в системе Nalaps

Проект в системе Nalaps – это фактически XML-конфигурационный файл, содержащий настройки предметной области и общие параметры. Экран с настройками проекта представлен на рис. 37.

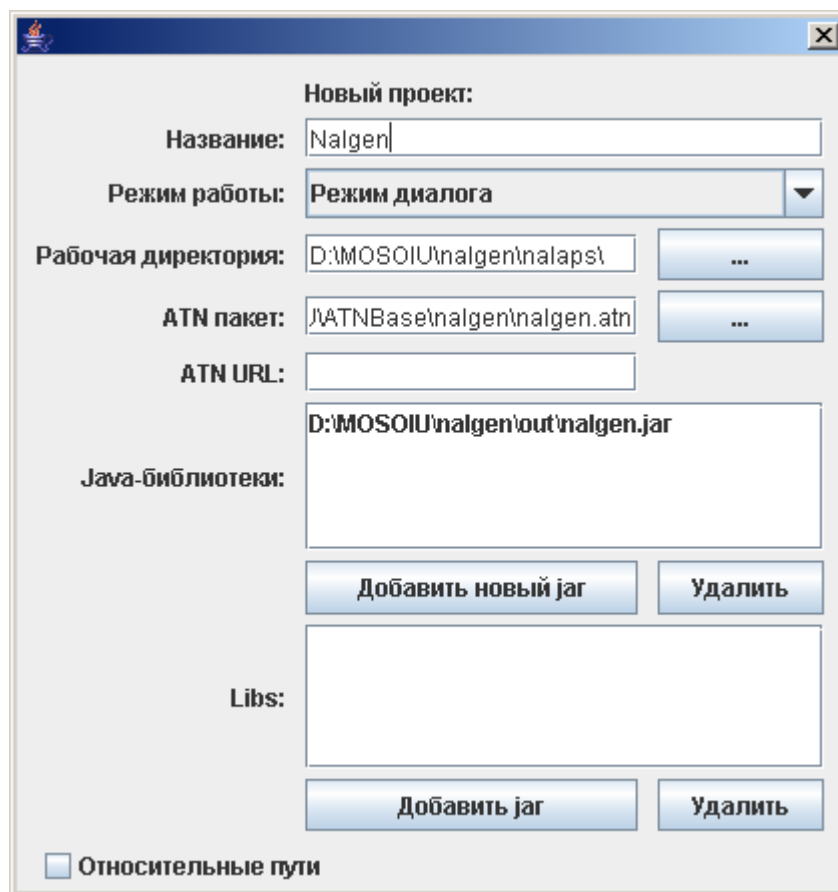


Рисунок 37. Снимок экрана с настройками проекта

Название проекта используется в качестве имени файла, в котором хранятся настройки проекта.

Режим работы – список выбора из двух элементов:

- *Текстовый режим* – обработка текста целиком.
- *Режим диалога* – последовательный ввод команд (фраз, предложений), с возможностью задавать системе вопросы и получать ответы.

Рабочая директория – задает каталог на диске, в котором система будет хранить временные файлы, нужные для работы.

ATN-пакет - путь к файлу-архиву, содержащему пакет файлов-описаний расширенных сетей переходов, настроенных для данной предметной области.

ATN URL – альтернативный способ указания пути к ATN-пакету – в виде HTTP-адреса (некоторые ATN-пакеты для тестовых и демонстрационных примеров хранятся на сервере системы – <http://www.nalaps.ru>).

Java-библиотеки – пути к java-библиотекам, содержащим специфицированные классы, т.е. XML-файлы карт настроек на предметную область.

Libs – пути к вспомогательным java-библиотекам, которые следует подключить к проекту для работы, но которые не содержат в себе специфицированных классов.

При включении опции «*относительные пути*» все пути в настройках проекта (за исключением ATN URL) интерпретируются как относительные относительно директории, из которой запущен проект. Данная опция используется при передаче файлов проекта другим пользователям.

Данный экран системы используется как при создании нового проекта, так и при редактировании старого. При каждом запуске системы по умолчанию открывается последний проект, с которым пользователь работал последний раз до закрытия приложения, и открывается окно настроек данного проекта.

4.2. Основное окно программы.

Основное окно программы состоит из меню и окна ввода текста (см. рис. 38).

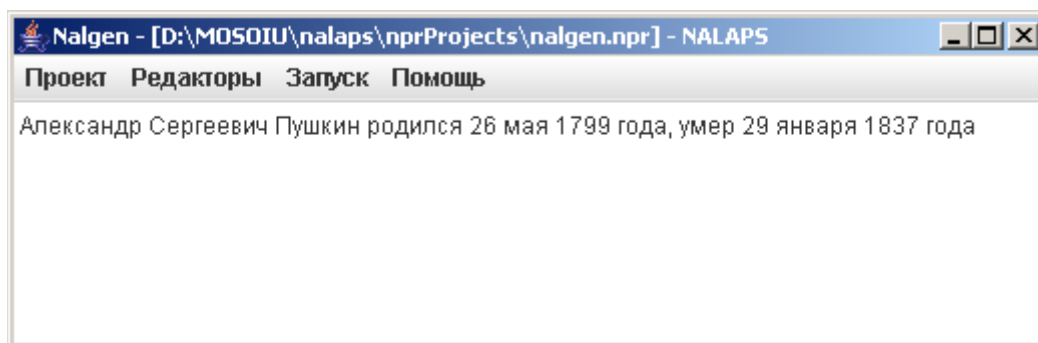


Рисунок 38. Снимок экрана основного окна программы

Рассмотрим каждый пункт меню в отдельности.

1. *Проект* – содержит специфичные для работы с настройками проекта:

- *Новый проект* – вызывает окно настроек для создания нового проекта.
- *Свойства проекта* – открывает окно настроек текущего проекта (опция меню не активно, если ни один проект не открыт).
- *Открыть проект* – вызывает стандартное окно браузера дисков для поиска и открытия файла проекта.
- *Сохранить как...* – вызывает стандартное окно браузера дисков для выбора директории и имени файла, для сохранения проекта (неактивно, если нет открытых проектов).
- *Сохранить (F2)* – сохраняет текущий проект, т.е. его настройки в файл на диске (неактивно, если нет открытых проектов). Может быть также вызвано нажатием клавиши *F2*.
- *Заккрыть проект* – закрывает текущий проект (неактивно, если нет открытых проектов).

2. *Редакторы* - содержит всего одну опцию:

- *Редактор ATN (F4)* – вызывает открытие редактора расширенных сетей переходов. Может быть также вызвано нажатием клавиши *F4*. По умолчанию редактор открывает последний раз редактировавшийся ATN пакет.

3. *Запуск* – содержит специфичные команды, связанные с обработкой текста в основном окне:

- *Синтактико-семантический граф* – вызывает построение синтактико-семантического графа обработанного текста и показывает его в отдельном окне.

- *Концептуальный граф* – вызывает построение концептуального графа обработанного текста и показывает его в отдельном окне.

- *Скомпилировать (F5)* – вызывает обработку текста или включает режим диалога (в зависимости от типа проекта), при которой происходит параллельное исполнение генерируемого кода программы на языке Java. Может быть также вызвано нажатием клавиши *F5*.

4. *Помощь – о программе* – показывает окно с краткой информацией о системе Nalaps.

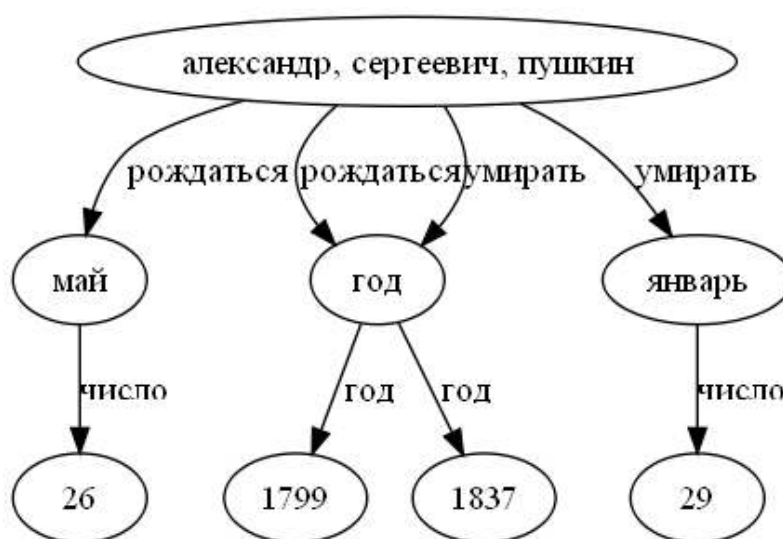


Рисунок 39. Пример отображения концептуального графа в системе Nalaps

В основном окне программы в поле для ввода текста предусмотрена графическая текстовая разметка для разобранных предложений. Так, **полужирным** цветом выделяются слова, для которых найдены концепты в базе знаний, *курсивом* выделяются предикаторы (Степанов, 2007), **красным полужирным** – нераспознанные концепты.

При наведении курсора мыши на распознанные концепты, при удерживании в нажатом состоянии клавиши “Ctrl” над словом всплывает информационное сообщение с идентификатором сопоставленного с данным концептом объекта Java. На рис. 40 представлен пример цветовой разметки разобранного предложения, когда в режиме нажатой клавиши “Ctrl” при наведении курсора мыши на слово «Сергеевич» оно подсвечивается синим цветом, а поверх него появляется окно с идентификатором «ru.mosoiu.nalgen.Person@a6e0a9».

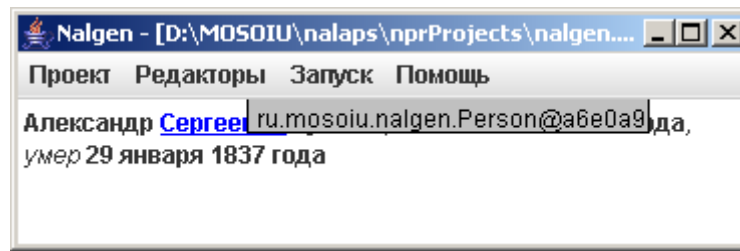


Рисунок 40. Пример цветовой разметки разобранного предложения

4.3. Модульная интеграция со средой разработки IntelliJ IDEA.

Целью создания модуля, интегрированного в среду разработки программ на Java, является необходимость наличия удобного и простого инструмента для естественно-языковой спецификации классов языка Java. Первым шагом к созданию такого инструмента является выбор среды разработки для его интеграции.

Среда разработки IntelliJ IDEA является мощным и удобным инструментом для создания приложений на языке Java (Duane, et al., 2006) (Bes). Это явилось одной из главных причин выбора данной системы для разработки встраиваемого модуля естественно-языковой спецификации. Кроме того, разработчиками IDE IntelliJ IDEA выпущен и поддерживается большой набор открытых библиотек для создания собственных модулей (plugin). Данные библиотеки дают возможность полного доступа ко всем механизмам интенционального программирования, вплоть до написания собственных парсеров и грамматик языков.

Модуль Nalaps для IntelliJ IDEA состоит из двух основных компонентов:

- 1) инструменты для естественно-языковой спецификации, и
- 2) организация связи на уровне графического интерфейса между найденными в тексте концептами и соответствующими им классами и компонентами классов.

Данные компоненты будут рассмотрены ниже подробнее.

4.3.1. Инструмент естественно-языковой спецификации.

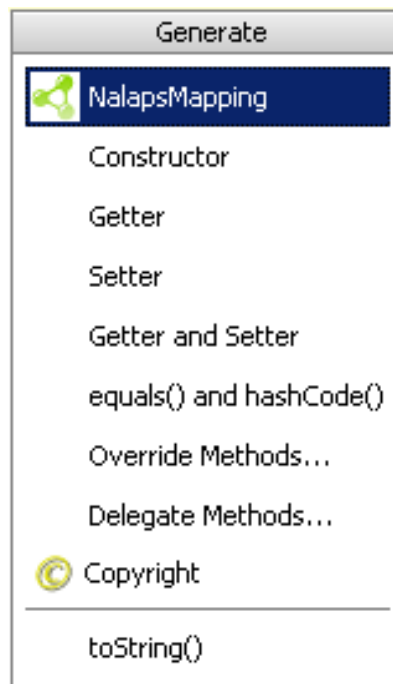


Рисунок 41. Меню "Generate"

Как было отмечено выше, шаблон спецификации задается программистом с помощью *аннотаций Java*. Данные шаблоны служат отправной точкой для конфигурации списка свойств для каждого размеченного компонента класса. Непосредственно инструмент конфигурации вызывается из меню “Generate”, выпадающего при использовании стандартной для IDEA комбинации клавиш [Alt+Ins] (рис. 41).

Данный инструмент имеет вид окна, состоящего из трех блоков, соответствующих описанию класса: имени класса, его полей и его методов (рис. 42). Первый блок, описывающий имя класса, содержит текстовую метку (label) с именем класса и текстовую метку с шаблоном-разметкой разметки. Второй блок содержит в левой колонке список полей класса и также список шаблонов разметки. Аналогичным образом третий блок содержит список методов и соответствующий им список шаблонных меток.

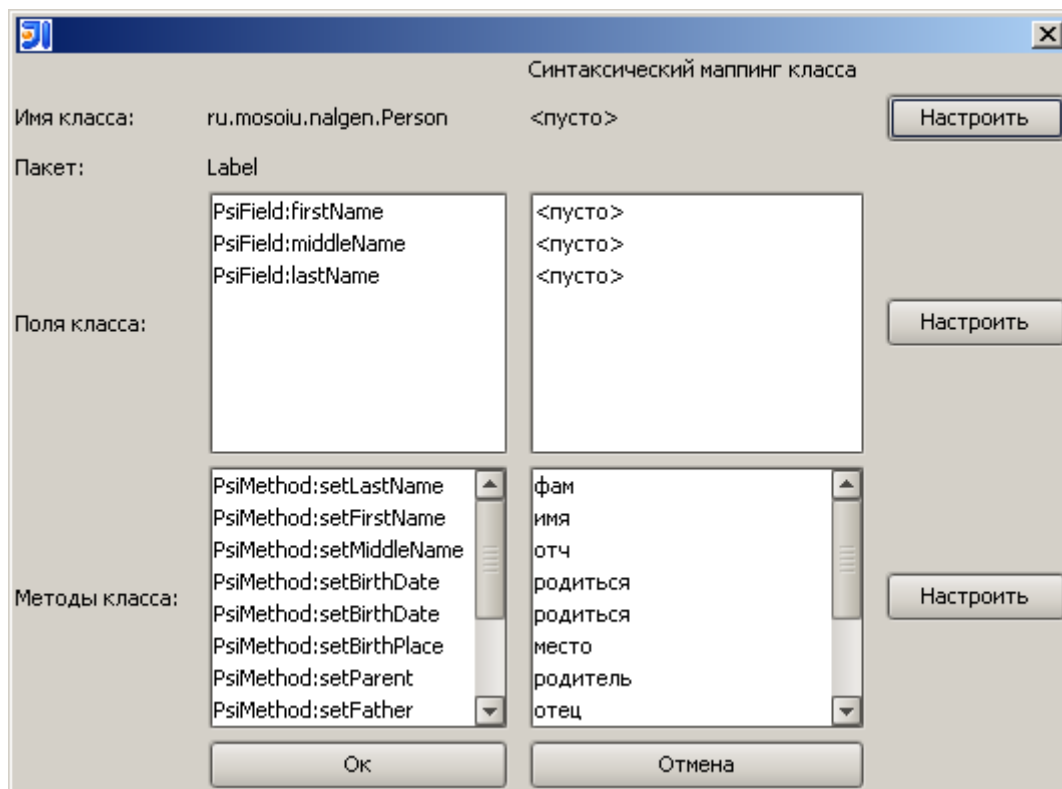


Рисунок 42. Снимок экрана инструмента естественно-языковой спецификации класса

Нажатие кнопки «Настроить» для каждого из блоков приводит к открытию окна с конфигурацией списка свойств выбранного поля/метода или имени класса. Интерфейс данного окна представлен на рис. 43.

Первая строка в окне «Маппинг для:» содержит имя компонента класса для которого конфигурируется естественно-языковая метка.

Будучи включенным, флаг «Уникальная роль» означает, что семантическая роль, сопоставленная, с данным полем/методом будет в данном информационном объекте уникальной. Данная уникальность влияет на анализатор ролей-дубликатов: после объединения информационных объектов роли-дубликаты, относящиеся к одному компоненту сопоставленного класса с флагом «уникальности», объединяются в одну роль (с самыми актуальными аргументами). Напротив, неуникальные роли с разными аргументами могут накапливаться. Например, такие отношения, как

«Имя», «Отчество», «Фамилия», «отец», «мать» в понятии, описывающем человека, следует сделать уникальными, т.к. у человека (в общем случае) может быть только одна мать, один отец, одно отчество и т.п. Такие отношения как, например, «дочь», «сын», напротив, не могут быть уникальными, иначе данное свойство будет ограничивать число дочерей и сыновей у персоны единственным числом.

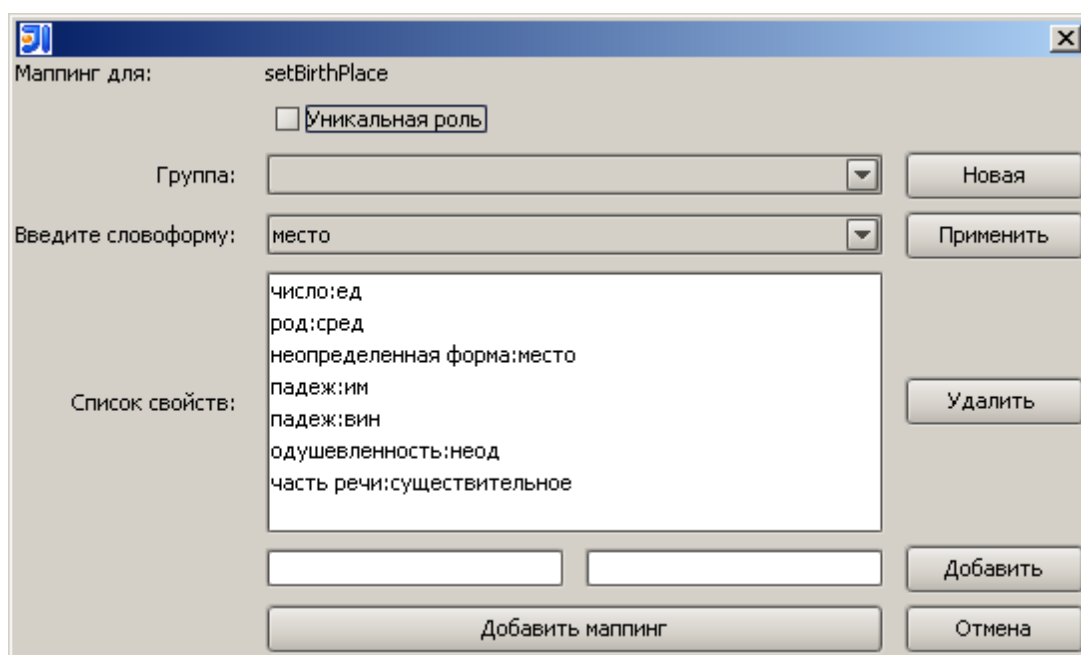


Рисунок 43. Снимок экрана окна конфигурации свойств метки

Выпадающий список «Группа» и рядом расположенная кнопка «Новая» позволяют осуществлять управление *семантическими группами*, сопоставленными с данным классом. Понятие *семантической группы* в данном контексте расширяет понятие «уникальной роли». Подразумевается, что у информационного объекта одновременно может присутствовать только одно отношение из числа входящих в данную группу. Примером использования данного функционала может быть описание объекта, относительного его положения по отношению к другому объекту для текущего наблюдателя. Данный список будет включать такие отношения, как: *перед, слева, справа, сзади, сверху, снизу* – все их следует включить в

одну группу «положение». Очевидно, например, что одновременно *слева* и *справа* относительно текущего угла наблюдателя один объект не может находиться по отношению к другому. Таким образом, анализатор ролей-дубликатов оставит только самое актуальное (последнее выделенное) отношение из данной группы.

Список выбора «Введите словоформу» позволяет выбрать конкретную метку-шаблон из числа метой данного компонента и произвести настройку его свойств в следующем блоке окна «Список свойств». По умолчанию в данном блоке отображаются все свойства, выделенные у выбранной словоформы. Таким образом, конфигуратор использует морфологический анализатор для создания предварительного списка свойств. Данные свойства можно как удалять, так и добавлять новые.

4.3.2. Организация связи между концептами в тексте и соответствующими им классами.

Данный второй компонент модуля Nalaps позволяет осуществлять навигацию по элементам разобранных предложений анализируемого текста. Описываемая навигация работает аналогично навигации по операторам и объектам при обычном программировании на Java в среде IDEA. Навигация работает в двух режимах:

1. Из основного окна программы.

Когда основное окно программы вызвано непосредственно из модуля для IDEA, в нем включается навигация по словам разобранных предложений. При удержании в нажатом положении клавиши “Ctrl”, при нажатии левой кнопки мыши на разобранном слове происходит открытие окна в среде IDEA с исходным текстом класса, сопоставленного с информационным объектом, который сопоставлен с данным словом текста.

2. При редактировании текста в файлах типа NPR.

В модуль Nalaps для IDEA также включена поддержка файлов с расширением “.nlp”. Эти текстовые файлы, будучи открыты в среде IDEA при включенном модуле Nalaps, используются вместо основного окна программы, т.е. текст введенный в них автоматически распознается в рамках текущего настроенного проекта Nalaps. Текст в таком режиме подвергается более богатой цветовой и стилистической разметке, что достигается благодаря использованию богатых возможностей системы IDEA для синтаксического разбора и представлений его результатов. Пример работы описываемого режима представлен на рис. 44.

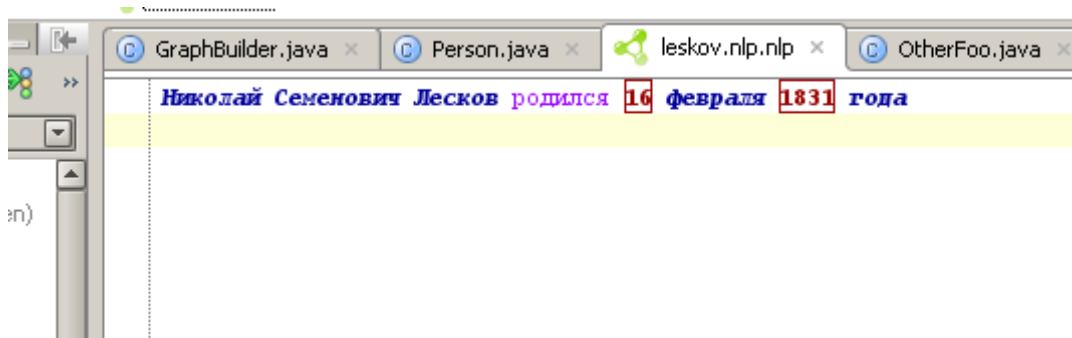


Рисунок 44. Пример разбора текста в файле типа .nlp

Каждый распознанный элемент в таком представлении также обладает навигацией, по нему можно перейти на ассоциированный с ним компонент класса.

4.4. Построение синтактико-семантического и концептуального графов.

Для построения графического отображения графов используется внешняя программа **dot** из комплекса программ Graphviz – Graph Visualization Software (Elson, et al., 2003)(Gra10). Программа dot принимает на вход текстовое формализованное описание генерируемого графа на внутреннем скриптовом языке DOT Language. Основными выражениями, используемыми в языке DOT являются:

node – для описания узлов графа,

edge – для описания ребер графа, и

graph – для описания графа в целом.

Пример программы на языке DOT для создания концептуального графа, представленного на рис. 35, приведен ниже:

```
digraph {
    obj0 [label="александр, сергеевич, пушкин"];
    obj1 [label="май"];
    obj2 [label="26"];
    obj3 [label="год"];
    obj4 [label="1799"];
    obj5 [label="январь"];
    obj6 [label="29"];
    obj7 [label="1837"];
    obj1->obj2 [label="число"];
    obj0->obj1 [label="рождаться"];
    obj3->obj4 [label="год"];
    obj0->obj3 [label="рождаться"];
    obj5->obj6 [label="число"];
    obj0->obj5 [label="умирать"];
    obj3->obj7 [label="год"];
    obj0->obj3 [label="умирать"];
}
```

Глава 5.

В данной главе описываются области и способы применения разработанной системы Nalaps:

- 1) Создание автоматических тестов на основе текстовых пользовательских сообщений об ошибках в программе (**Лебедев, 2010**).
- 2) Учебно-игровой аспект: пример реализации виртуальной трехмерной комнаты с объемными примитивами и имитацией законов гравитации.
- 3) Автоматическое построение генеалогических деревьев по текстовому описанию родословных отношений.

5.1. Создание автоматических тестов на основе текстовых пользовательских сообщений.

Автоматические тесты (АТ) являются неотъемлемой частью цикла разработки крупных коммерческих и промышленных программных комплексов. Под АТ-ом понимается программа, написанная обычно на скриптовом языке или на том же языке программирования, что и сам программный комплекс. Автоматическое тестирование таким образом – это разработка планов тестирования и программирование самих тестов. (Дастин, и др., 2003). Цель написания и работы автоматического теста – проверка критической ситуации, конкретных модулей системы или отдельных участков кода для выявления потенциальных сбоев системы и проверки корректности ее работы в эмулируемых ситуациях. Необходимость создания автоматических тестов обусловлена сложностью современных программ, что влечет за собой побочные и практически неизбежные эффекты разработки, например, когда исправление одной ошибки приводит к возникновению

новой в уже ранее отлаженном блоке. Автоматически обнаружить такие ошибки позволяет комплекс мер, называемый системой непрерывного автоматического тестирования, где перед каждой сборкой проекта производится запуск всех АТ-ов.

Вместе с вышесказанным следует указать и на другую, практическую сторону процесса написания АТ-а. Ввиду их специфики они являются с одной стороны довольно простыми по логической структуре, но достаточно трудоемкими по созданию с другой, что обусловлено необходимостью ручного внесения в код программы большого количества шаблонных тестовых данных, а также описание рутинных действий пользователя системы, эмулирующих заданную для проверки ситуацию. Такая задача отнимает у программистов много времени, являясь тяжелым и рутинным занятием, в особенности, если речь идет о тестировании графического пользовательского интерфейса (GUI) (Калинов А.Я.). Однако если фундаментальный набор тестов, покрывающий явно очевидные критичные блоки программы, необходимо создавать в начале цикла разработки программы, то написание программ для тестирования уникальных ситуаций, в которых могла бы возникнуть (или возникала ранее) ошибка, можно было бы автоматизировать, низведя до уровня простого конструктора объектов метаданных. Данное предположение обосновывается опытом, полученным при написании ряда промышленных и коммерческих программных продуктов – в том числе и опытом поддержки систем, эксплуатируемых в промышленном режиме.

В действительности, большинство замеченных в системе ошибок пользователи описывают в виде обычного текста на естественном языке (ЕЯ), предоставляя такие описания в различные системы для ведения списка дефектов. Иного удобного способа подачи заявок на исправление дефектов кроме текстового описания, в системе общения «пользователь<-

>разработчик» пока не придумано, или, по крайней мере, не выявлено в качестве реальной альтернативы. Полученная заявка на исправление ошибки проходит проверку: является ли описанная ситуация ошибкой и воспроизводится ли она при описанных условиях? В том случае, если ошибка подтверждается, ее устраняют и создают тест, проверяющий, что данный дефект при описанных условиях больше не проявляется.

Предложенный подход генерации тестов с помощью системы Nalaps позволяет сократить временные затраты аналитиков и разработчиков на написание тестов для ситуаций, имеющих достаточно формализованное (в терминах предметной области) текстовое описание, что достигается за счет автоматизации данного процесса целиком или его основных этапов. При наличии достаточно полной базы знаний предметной области (т.е. той предметной области, к которой относится тестируемый программный комплекс) представляется теоретическая возможность полной автоматизации процесса создания тестов. База знаний в свою очередь накапливается в процессе работы разработчиков с предложенным инструментом.

Поскольку А-тест также является программой, то и его генерация возможна в системе Nalaps. Основной сложностью в данной задаче является «понимание» системой текстового описания поставленной задачи, на основе которой необходимо сгенерировать программу. Предполагается, что вводимые в систему тексты являются довольно хорошо формализованными, что обусловлено спецификой проблемы: предметная область достаточно узка и поддается описанию для конкретного тестируемого программного комплекса.

Рассмотрим перечисленные выше этапы с привлечением простейшего примера для иллюстрации.

В примере (1) информационными объектами будут выступать существительные, связанные со своими определениями. В данном случае, это:

Карточка клиента

Закладка «счета»

Поле «Дата»

Сегодняшний день

Красный цвет

Третий этап заключается в создании текста программы на основе полученного концептуального графа. Заметим, что генерация предполагает параллельное создание, как текста программы, так и пошаговое выполнение созданных команд, т.е. система Nalaps выступает также как интерпретатор. Отметим, что описание того, что поле подсвечивается красным цветом, избыточно в рамках решаемой задачи, поэтому в базе знаний нет необходимости определять сущность «цвет».

Приведем сокращенный код классов, описывающих понятия предметной области.

Для понятия «клиент»:

```
@Semanticable("Клиент")
public class Debtor {
    @Semanticable("Карточка")
    private DebtorCard card;
}
```

Для понятия «карточка»:

```
@Semanticable("карточка")
public class DebtorCard {
    @Semanticable("закладка")
```

```

    private Bookmark activeBookmark;
}

```

Для понятия «закладка»:

```

@Semanticable("закладка")
public class Bookmark {
    private String name;
    @Semanticable("поле")
    private FormField activeField;
    @Semanticable("имя")
    public void setName(String name) {
        this.name = name;
    }
}

```

Для понятия «поле»:

```

@Semanticable("поле")
public class FormField {
    private String name;
    @Semanticable("имя")
    public void setName(String name) {
        this.name = name;
    }
    @Semanticable("указывать")
    public void setValue(Object value) {
        ...
    }
}

```

Для понятия «день»

```
@Semanticable("день")
public class DayT extends DateT {
    @Semanticable("датавремя")
    public void setDatetime(String datetime) {
        if (datetime.equalsIgnoreCase("сегодняшний"))
            setTime(new Date().getTime());
    }
}
```

Как видно из кода, в нем присутствует разметка, выполненная с помощью инструмента аннотаций (@Semanticable). Указанная разметка служит вспомогательным инструментом для описания карты настройки класса на предметную область. Подобно описанию дуг в графах РСП, каждому размеченному таким образом элементу класса ставится в соответствие набор грамматических признаков слов (чаще всего с неопределенной формой), которые позволяют сопоставлять выделенные в концептуальном графе понятия с нужными классами.

Основной сложностью при написании тестов в данном примере является необходимость тестирования пользовательского web-интерфейса, т.к. все доступные операции в системе выполняются непосредственно из него. Для этих целей используется система Selenium HQ, представляющая собой набор инструментов для тестирования web-приложений (9). Указанная специфика обуславливает необходимость создания классов-оберток, описывающих способы доступа к тем или иным элементам интерфейса, описываемым в пользовательских сообщениях об ошибках. Не будем останавливаться на программной реализации сценариев их работы с web-интерфейсом. Отметим, что данная операция носит примитивный характер и

закljučается в перечислении действий в браузере, необходимых для получения страницы с требуемым элементом. В приведенном примере данные сценарии опущены.

Пример работы предлагаемого механизма автоматического тестирования и работы ЛП, в частности, рассмотрим на следующем примере:

В карточке клиента на закладке “счета” в поле “дата платежа” невозможно указать сегодняшний день. Оно подсвечивается красным цветом. (1).

В примере (1) информационными объектами будут выступать существительные, связанные со своими определениями. В данном случае, это:

Карточка клиента

Закладка «счета»

Поле «Дата»

Сегодняшний день

Красный цвет

Третий этап заключается в создании текста программы на основе полученного концептуального графа. Заметим, что генерация предполагает параллельное создание, как текста программы, так и пошаговое выполнение созданных команд, т.е. система Nalaps выступает также как интерпретатор. Отметим, что описание того, что поле подсвечивается красным цветом, избыточно в рамках решаемой задачи, поэтому в базе знаний нет необходимости определять сущность «цвет».

Приведем сокращенный код классов, описывающих понятия предметной области.

Для понятия «клиент»:

@Semanticable("Клиент")

public class Debtor {

```

    @Semanticable("Карточка")
    private DebtorCard card;
}

```

Для понятия «карточка»:

```

@Semanticable("карточка")
public class DebtorCard {
    @Semanticable("закладка")
    private Bookmark activeBookmark;
}

```

Для понятия «закладка»:

```

@Semanticable("закладка")
public class Bookmark {
    private String name;
    @Semanticable("поле")
    private FormField activeField;
    @Semanticable("имя")
    public void setName(String name) {
        this.name = name;
    }
}

```

Для понятия «поле»:

```

@Semanticable("поле")
public class FormField {
    private String name;
    @Semanticable("имя")

```

```

public void setName(String name) {
    this.name = name;
}
@Semanticable("указывать")
public void setValue(Object value) {
    ...
}
}

```

Для понятия «день»

```

@Semanticable("день")
public class DayT extends DateT {
    @Semanticable("датавремя")
    public void setDatetime(String datetime) {
        if (datetime.equalsIgnoreCase("сегодняшний"))
            setTime(new Date().getTime());
    }
}

```

Как видно из кода, в нем присутствует разметка, выполненная с помощью инструмента аннотаций (@Semanticable). Указанная разметка служит вспомогательным инструментом для описания карты настройки класса на предметную область. Подобно описанию дуг в графах РСП, каждому размеченному таким образом элементу класса ставится в соответствие набор грамматических признаков слов (чаще всего с неопределенной формой), которые позволяют сопоставлять выделенные в концептуальном графе понятия с нужными классами.

Основной сложностью при написании тестов в данном примере является необходимость тестирования пользовательского web-интерфейса,

т.к. все доступные операции в системе выполняются непосредственно из него. Для этих целей используется система Selenium HQ, представляющая собой набор инструментов для тестирования web-приложений (9). Указанная специфика обуславливает необходимость создания классов-оберток, описывающих способы доступа к тем или иным элементам интерфейса, описываемым в пользовательских сообщениях об ошибках. Не будем останавливаться на программной реализации сценариев их работы с web-интерфейсом. Отметим, что данная операция носит примитивный характер и заключается в перечислении действий в браузере, необходимых для получения страницы с требуемым элементом. В приведенном примере данные сценарии опущены.

Первый этап создания текста программы – автоматическая генерация объектов языка Java. Для каждого объекта система создает латинский акроним, полученный путем транслитерации неопределенных форм соответствующих данному информационному объекту лексем.

создаем объект, представляющий сущность «Клиент»

```
autotests.Debtor klient_1 = new autotests.Debtor();
```

для понятия «карточка клиента»

```
autotests.DebtorCard kartochka_1 = new autotests. DebtorCard ();
```

и т.д.:

```
autotests.Bookmark zakladka_1 = new autotests.Bookmark(“счета”);
```

```
autotests.FormField pole_1 = new autotests.FormField(“дата платежа”);
```

```
autotests.DayT den_segodnyashnii = new autotests.DayT();
```

Затем система генерирует код вызова методов на основании связей в концептуальном графе:

```
klient_1.setDebtorCard(kartochka_1);
```

```
kartochka_1.setActiveBookmark(zakladka_1);
```

```

zakladka_1.setActiveField(pole_1);
den_segodnyashnii.setDatetime("сегодняшний");
pole_1.setValue(den_segodnyashnii);

```

Полученный текст программы может быть включен в цикл непрерывного тестирования – в случае возникновения ошибки, тест сгенерирует исключение (exception).

5.2. Учебно-игровой аспект: пример реализации виртуальной трехмерной комнаты с объемными примитивами и имитацией гравитации.

В качестве иллюстративного примера в системе Nalaps реализован учебно-игровой пример реализации виртуальной трехмерной комнаты, в которой естественно-языковыми текстовыми командами осуществляется управление объемными примитивами: кубами, шарами, конусами, цилиндрами и пирамидами (Путилов Г.П., 2009). Объемные примитивы наделены виртуальной массой, а в виртуальном пространстве реализована эмуляция физических законов, за счет чего виртуальная комната наделена динамикой. Объекты отталкиваются при столкновении, подвержены перемещениям за счет виртуальной гравитации. Пользователь может произвольным образом изменять угол и точку зрения, т.е. изменять свое виртуальное положение в пространстве.

Фактически данный пример является расширенной реализацией работы системы SHRDLU, разработанной Терри Виноградом в MIT в 1968 – 1970 годах (Winograd, 1971). Однако, в данной системе для общения пользователя с программой используется русский язык.

В качестве иллюстрации работы лингвистического процессора рассмотрим тривиальный пример из «мира блоков» (1). После обработки команды:

Создай синий кубик. Поставь его слева от красного кубика. Создай желтый кубик и помести его над синим.(2)

на выходе из ЛП получим набор ИнфОбов (3):

- 1) кубик_1
- 2) кубик_2
- 3) кубик_3

И набор семантических отношений (СемОт) (4):

- 1) цвет(кубик_1,синий)
- 2) цвет(кубик_2,красный)
- 3) цвет(кубик_3,желтый)
- 4) слева(кубик_1, кубик_2)
- 5) сверху(кубик_3, кубик_1)

После получения семантического представления следующим этапом обработки текста, поступившего на вход, служит этап генерации программы на языке Java.

В соответствии со списком ИнфОб-ов (3) получим следующие операторы языка Java (5):

```
Cubic kubik_1 = new Cubic();
Cubic kubik_2 = new Cubic();
Cubic kubik_3 = new Cubic();
```

Список СемОт-ов (4) будет интерпретирован следующим образом (6):

```
kubik_1.setColor("синий");
kubik_2.setColor("красный");
kubik_3.setColor("желтый");
kubik_1.putLeft(kubik_2);
kubik_3.putOn(kubik_1);
```

Пользователь виртуальной комнаты также может задавать ЭВМ вопросы о месте нахождения определенных объектов, относительно своего текущего местоположения и угла зрения, а также относительно других объектов: система формирует ответ с указанием местоположения

обозначенного примитива относительно других объектов (используя предлоги типа: перед, справа, слева и т.д.). Используя предикаты семантической связки понятий типа «X – это Y» пользователь может произвольным образом называть объекты в виртуальной комнате и при необходимости спрашивать систему о том, чем является то или иной объект:

Большой белый кубик – это дом.

Что такое дом?

> Дом номинатив большой белый кубик.

Ограниченностью системы является отсутствие в ней полноценного морфологического и синтаксического генераторов, т.е. невозможно построить ответ системы по всем правилам русского языка. Однако концептуальная основа для такого ответа в системе заложена. Ответ системы «Дом номинатив большой белый кубик» - это фактически распечатка найденного системой нужного для ответа на вопрос семантического отношения в концептуальном графе.

5.3. Автоматическое построение генеалогического древа по текстовому описанию родословных отношений.

Одной из интересных задач, которую могла бы решать система естественно-языкового программирования – автоматическая интерпретация текстового описания родословных отношений, вводимая пользователем, в виде генеалогического древа, автоматически генерируемого после анализа введенных данных. Такая задача была решена на определенном несложном наборе синтаксических конструкций, используемых для биографических описаний, в качестве проекта в системе Nalaps. Фактически данный пример иллюстрирует возможности системы по извлечению знаний из текста (Лебедев, 2009), т.к. при правильной настройке, система будет игнорировать информацию, не относящуюся к биографии человека, а регистрировать

только генеалогические сведения. Рассмотрим в качестве примера разбор системой двух простых предложений из заданной предметной области:

Сидоров Иван Петрович родился в 1980 году в Москве. В 2005 году он женился на Петровой Нине Федоровне.

Отметим, какие информационные объекты будут выделены (с учетом анализа анафорических местоимений):

1. Сидоров_Иван_Петрович
2. 1980_год
3. Москва
4. 2005_год
5. Петрова_Нина_Федоровна

Символ «_» при этом отражает внутреннюю связь между понятиями в информационном объекте.

Кроме того, будут выделены следующие семантические отношения (СемОт):

1. родиться (Сидоров_Иван_Петрович, 1980)
2. родиться(Сидоров_Иван_Петрович, Москва)
3. жениться(Сидоров_Иван_Петрович, Петрова_Нина_Федоровна)
4. жениться(Сидоров_Иван_Петрович, 2005)

Для описания персоны из базы данных генеалогических сведений создан класс:

@Semanticable(“Кустов, Георгий, Георгиевич, Кустова, Ольга, Петровна”)

```
public class GCPerson { // Класс, описывающий персону
```

```
    @Semanticable(“Георгий, Ольга”)
```

```
    private String firstname; // Имя
```

```
    @Semanticable(“Георгиевич, Петровна”)
```

```
    private String middlename; // Отчество
```

```

@Semanticable("Кустов, Кустова")
private String surname; // Фамилия
@Semanticable("1845")
private Date dBirth; // Дата рождения
@Semanticable("Самара")
private String bPlace; // Место рождения
@Semanticable("Кустова, Ольга, Петровна")
private GCPerson wife; // Ссылка на объект «супруга»
@Semanticable("Кустов, Георгий, Петрович")
private GCPerson husb; // Ссылка на объект «супруг»
public GCPerson(String ...) { ... } // Конструктор класса
@Semanticable("родился, родилась")
public void setDBirth(Integer year) { ... }
@Semanticable("родился, родилась")
public void setBPlace(String bPlace) { ... }
@Semanticable("женился")
public void setWife(GCPerson wife) { ... }
}

```

В классе присутствует интеллектуальный конструктор (его подробное описание здесь опущено), на вход которого поступает массив строковых данных. Его внутренний тривиальный алгоритм определяет, какие из входных строковых констант являются Фамилией, Именем и Отчеством соответственно. Метод *setDBirth* служит для заполнения даты рождения. На вход он получает целое число, рассматриваемое, как год рождения. Объект класса *java.lang.Date* формируется автоматически. Методом заполнения поля «место рождения» является *setBPlace*, и т.д.

После выполнения парсинга текста на выходе будет получен следующий код программы:

```

GCPerson sidorov_ivan_petrovich_1 = new GCPerson("Сидоров", "Иван",
"Петрович");
Integer god_2005_1 = new Integer("2005");
GCPerson      petrova_nina_fedorovna_1      =      new
GCPerson("Петрова", "Нина", "Федоровна");
Integer god_1980_1 = new Integer("1980");
String moskva_1 = new String("Москва");
Integer god_2005_1 = new Integer("2005");
Интерпретация семантических отношений:
sidorov_ivan_petrovich_1.setDBirth(god_1980_1);
sidorov_ivan_petrovich_1.setBPlace(moskva_1);
sidorov_ivan_petrovich_1.setWife(petrova_nina_fedorovna_1);

```

В данном примере СемОт, выражающий дату свадьбы, не был задействован, т.к. для него не была найдена соответствующая естественно-языковая метка в классе GCPerson.

Объекты GCPerson в свою очередь передают информацию в класс PersonManager, который анализирует связи между объектами GCPerson и строит на экране ЭВМ графическое отображение генеалогического древа. В программе также реализована возможность экспорта полученного древа в файл формата GedCom 5.5 (The10).

Заключение.

Система Nalaps является:

а) системой концептуального программирования, т.к. в ней применен концептоцентрический подход: каждый концепт предметной области имеет свое описание в виде специфицированного класса языка Java;

б) системой языко-ориентированного программирования, т.к. позволяет создавать собственные языки предметной области (DSL), используя, фактически, ограниченные естественные языки для программирования.

в) инструментом извлечения знаний из текстов, в том числе имплицитных смыслов, что достигается за счет представления смысла текста в виде динамической объектной модели.

Основные результаты работы:

1. Предложена методика описания единиц базы концептуальной знаний с помощью классов объектно-ориентированного программирования. Для этих целей создан естественно-языковой спецификатор языка Java, позволяющий осуществлять настройку класса на предметную область на основе лексематической разметки его компонентов.
2. Предложен подход к представлению смысла текста в виде динамической системы объектов языка Java, что позволяет автоматически выявлять глубинные связи в тексте, закладывая их описание на уровне алгоритмов на языке программирования.
3. Разработан визуальный редактор расширенных сетей переходов, имеющий встроенный морфологический анализатор, что позволяет производить поиск по базе расширенных сетей и строить их «скелеты» по введенному шаблону.
4. Дана математическая модель предложенного метода получения концептуального графа и связи его с концептами базы знаний.

5. Разработан программный комплекс Nalaps, позволяющий создавать и использовать предложенную модель базы концептуальных знаний. Система Nalaps позволяет разработчикам программного обеспечения создавать предметные базы знаний параллельно с разработкой с написанием кода программы, а также встраивать в разрабатываемые программные средства естественно-языковые интерфейсы.
6. Найдено применение системы Nalaps как средство генерации автоматических тестов для верификации работоспособности программного обеспечения в описываемых на естественном языке в текстовом виде сообщений об ошибках программы.
7. Созданный комплекс Nalaps может также применяться в учебных целях для демонстрации принципов объектно-ориентированного программирования на естественно-языковых примерах.
8. Система Nalaps может применяться в индустрии компьютерных игр для реализации естественно-языкового интерфейса пользователя.
9. Возможна обработка потоков текстовых данных и настройка системы на регистрацию определенных информационных сигналов с последующей их произвольной обработкой.
10. Методика построения лингвистического процессора, разработанная в рамках диссертационной работы, применена автором и в других научно-исследовательских работах, в частности, в модуле поиска авторефератов системы InfoNIOKR (Свидетельство о государственной регистрации программы для ЭВМ № 2009613349).
11. Система Nalaps и ее основные компоненты зарегистрирована в Роспатенте в сентябре 2010 года:
 - а. «Nalaps»: Свидетельство о государственной регистрации программы для ЭВМ №2010616060.

- б. Визуальный редактор расширенных сетей переходов:
Свидетельство о государственной регистрации программы для
ЭВМ №2010616061.

Список литературы

About Selenium [В Интернете] // Selenium HQ Web application testing system. - 29 01 2010 г.. - <http://seleniumhq.org/about/>.

Berners-Lee Tim, Hendler James и Lassila Ora The Semantic Web [Журнал] // Scientific American. - 17 may 2001 г..

Best Java IDE :: Do more high-quality code in less time with IntelliJ IDEA [В Интернете]. - <http://www.jetbrains.com/idea/>.

Chein M. и Mugnier M. L. Conceptual Graphs are also Graphs [Доклад] / LIRMM (CNRS and Universite Montpellier II). - 1995.

Chomsky N. Aspects of the Theory of Syntax [Книга]. - Cambridge, MA : MIT Press, 1965.

Daniel D. K. Sleator, Davy Temperley Parsing English with a Link Grammar [Конференция]. - Pittsburgh : Carnegie Mellon University, 1991. - CMU-CS-91-196.

Dinechin Christophe de The XL Programming Language [В Интернете] // XL Extensible Language. - 31 01 2011 г.. - 31 01 2011 г.. - <http://xlr.sourceforge.net/language>.

Duane K. Fields, Saunders S. и Belyaev E. IntelliJ IDEA in Action [Книга]. - [б.м.] : Manning Publications Co., 2006. - ISBN: 1932394443.

Elson J. [и др.] Graphviz and Dynagraph - Static and Dynamic Graph Drawing Tools [Раздел книги] // Graph Drawing Software / авт. книги Michael Junger Petra Mutzel, eds. - [б.м.] : Springer-Verlag, 2003.

Graphviz - Graph Visualization Software [В Интернете] // Graphviz. - 31 05 2010 г.. - <http://www.graphviz.org>.

Knuth Donald E. Literate Programming [Журнал]. - California : Stanford University Center for the Study of Language and Information, 1992 г.. - ISBN 978-0937073803.

Kulba V., Nikolsky S. и Zaikin O. Ontological approach to modelling of discrete event dynamic system [Статья] // Bulletin of the Polish Academy of sciences. Technical sciences. - 2009 г.. - No. 3 : T. Vol. 57.

Linguistic Tree Constructor: About LTC. [В Интернете] // Linguistic Tree Constructor: About LTC.. - 15 05 2010 г.. - <http://ltc.sourceforge.net/about.html>.

Lovins Julie Beth Development of a Stemming Algorithm [Статья] // Mechanical Translation and Computational Linguistics. - Cambridge, Massachusetts : [б.н.], 1968 г.. - vol.11, nos.1 and 2, March and June 1968.

M. Mernik J. Heering, A.M. Sloane When and how to develop domain-specific languages [Журнал] // ACM Computing Surveys (CSUR). - New York, NY, USA : [б.н.], 2005 г.. - T. Volume 37 Issue 4.

Mason James A. Augmented Syntax Diagram Grammars [В Интернете] // Augmented Syntax Diagram Grammars. - 15 05 2010 г.. - <http://www.yorku.ca/jmason/asdgram.htm>.

OWL, язык веб-онтологий. Краткий обзор [В Интернете]. - 24 05 2010 г.. - http://www.sherdim.rsu.ru/pts/semantic_web/REC-owl-features-20040210_ru.html.

Richens R. H. Preprogramming for Mechanical Translation [Статья] // Mechanical Translation. - 1956 г.. - T. Vol. 3.. - стр. p. 20.

Roman Knöll Mira Mezini Pegasus: First Steps Towards a Naturalistic Programming Language [Конференция] // OOPSLA 2006. - 2006.

Segalovich I. A fast morphological algorithm with unknown word guessing induced by a dictionary for a web search engine. [Статья] // MLMTA-2003. - 2003 г..

Sharoff S. Proc. of EWAIC'93 [Конференция] // SNOOP: A System for Development of Linguistic Processors.. - Moscow : [б.н.], 1993. - стр. 184-188.

Simonyi Charles The Death Of Computer Languages, The Birth of Intentional Programming [Доклад] / Microsoft. - 1995. - MSR-TR-95-52.

Smith Michael K. и Chris Welty Deborah L. McGuinness OWL Web Ontology Language Guide [В Интернете]. - W3C, 24 05 2010 г..

Sowa J.F. Conceptual Structures - Information Processing in Mind and Machine [Книга]. - [б.м.] : Addison-Wesley, 1984.

Sowa John F. Chapter 5. Conceptual Graphs [Раздел книги] // Handbook of Knowledge Representation / авт. книги F. van Harmelen V. Lifschitz, B. Porter. - [б.м.] : Elsevier, 2008.

The GEDCOM Standard Release 5.5 [В Интернете]. - 06 06 2010 г.. - <http://homepages.rootsweb.ancestry.com/~pmcbride/gedcom/55gctoc.htm>.

TreeForm Syntax Tree Drawing Software, Version 1.0.3. [В Интернете] // TreeForm Syntax Tree Drawing Software, Version 1.0.3. . - 15 05 2010 г.. - <http://www.ece.ubc.ca/~donaldd/treeform.htm>.

Winograd Terry Procedures as a Representation for Data in a Computer Program for Understanding Natural Language [Книга]. - [б.м.] : Massachusetts Institute of Technology, 1971.

Автоматическая генерация тестов на основе формального описания [В Интернете] // INTUIT.ru Интернет Университет Информационных технологий. - 29 01 2010 г.. - <http://www.intuit.ru/departments/se/testing/21/> .

Анисимович К. Балл В., Селегей В. ABBYY Retrieval & Morphology Engine [Конференция] // Материалы Международной конференции «Диалог 2000». - 2000.

Апреснян Ю. Д. [и др.] Лингвистическое обеспечение системы ЭТАП-2 [Книга]. - М. : Наука, 1989.

Бэлэни Нэвин Будущее Web - за семантикой [В Интернете]. - 25 06 2007 г.. - 24 05 2010 г.. - <http://www.ibm.com/developerworks/ru/library/wa-semweb/>.

ГУ "НИИ МЭИИТ МИЭМ"; Руководитель Путилов Г.П.; Тумковский С.Р., Беялетдинов И.И., Игнатовский А.Н., Смирнова Е.В., Галыгин А.Н., Михейкин И.В., Лебедев А.С., Зонтов Ю.В., Бобин А.В. Разработка

теоретических положений решения задачи семантического анализа результатов выполненных НИР и ОКР на примере ФЦП "Исследование и разработка по приоритетным направлениям развития научно-технического комплекса на 2007-2012 г. [Доклад] : Отчет о НИР (заключит.). - ГР №0120.0 808037; Инв.№02200900878.-М., 2008..

Гурин Р.Ф. Романенко С.А. Язык программирования Рефал Плюс [Книга]. - М. : Интертех, 1991.

Дастин Э., Рэшка Д. и Пол Д. Автоматизированное тестирование программного обеспечения. Внедрение, управление и эксплуатация [Книга]. - М. : Изд-во «Лори», 2003.

Дмитриев С. Языково-ориентированное программирование: следующая парадигма. [Журнал] // RSDN Magazine. - 2005 г.. - 5.

Дружкин К. Ю. и Цинман Л. Л. СИНТАКСИЧЕСКИЙ АНАЛИЗАТОР ЛИНГВИСТИЧЕСКОГО ПРОЦЕССОРА ЭТАП-3: ЭКСПЕРИМЕНТЫ ПО РАНЖИРОВАНИЮ СИНТАКСИЧЕСКИХ ГИПОТЕЗ [Конференция] // Компьютерная лингвистика и интеллектуальные технологии: По материалам ежегодной Международной конференции «Диалог». - М. : РГГУ, 2008.

Елкин С. В., Клышинский Э. С. и Стекляников С. Е. Сб. трудов Международных конференций IEEE AIS'03 и CAD-2003 [Конференция] // Проблемы создания универсального морфосемантического словаря. - Дивноморское : [б.н.], 2003. - Т. 1.

Зализняк А. А. Грамматический словарь русского языка [Книга]. - М. : [б.н.], 1977. - 4-е издание.

Золотова Г. А. Синтаксический словарь. Репертуар элементарных единиц русского синтаксиса [Книга]. - 2006. - ISBN 5-354-01147-7.

Иомдин Л.Л. Лобанов Б.М. Синтаксические корреляты просодически маркированных элементов предложения и их роль в задачах синтеза речи по тексту [Конференция] // Компьютерная лингвистика и интеллектуальные

технологии: По материалам ежегодной Международной конференции "Диалог 2009" (Бекасово, 27-31 мая 2009 г.). - М. : [б.н.], 2009. - стр. 136-143.

Калинов А.Я. Косачев А.С., Посыпкин М.А., Соколов А.А. Автоматическая генерация тестов для графического пользовательского интерфейса по UML диаграммам действий [В Интернете] // Труды Института Системного Программирования РАН. - 29 01 2010 г.. - http://www.citforum.ru/SE/testing/generation_uml/.

Каменская Л. О. Текст и коммуникация [Книга]. - М. : Высшая школа, 1990.

Кахро М. И., Калья А. П. и Тыгу Э. Х. Инструментальная система программирования ЕС ЭВМ (ПРИЗ) [Книга]. - М. : Финансы и статистика, 1988. - 2-е изд., перераб. и доп.. - ISBN5-279-00111-2.

Корухова Ю. С. Система автоматического синтеза функциональных программ [Книга]. - Москва : МГУ им. М.В. Ломоносова, 2005.

Краткое описание технологии InBASE [В Интернете]. - <http://www.inbase.artint.ru/proinfo/techpub.asp>.

Кузнецов И. П. Семантические представления [Книга]. - М. : Наука, 1986.

Кузнецов И.П. Ефимов Д.А. Особенности извлечения знаний из текстов семантико-ориентированным лингвистическим процессором Semantix [Конференция] // Компьютерная лингвистика и интеллектуальные технологии: По материалам ежегодной Международной конференции «Диалог» (Бекасово, 4–8 июня 2008 г.). - М. : РГГУ, 2008. - ISBN 978-5-7281-1022-4.

Кузнецов И.П. Ефимов Д.А. Средства настройки процессора Semantix на предметную область [Конференция] // Компьютерная лингвистика и интеллектуальные технологии: По материалам ежегодной Международной конференции «Диалог 2009» (Бекасово, 27-31 мая 2009 г.). Вып 8 (15). - М. : РГГУ, 2009. - стр. 262-270.

Лебедев А. С. АВТОМАТИЧЕСКА ГЕНЕРАЦИЯ ТЕСТОВ НА ОСНОВАНИИ ТЕКСТОВОГО ОПИСАНИЯ ОШИБКИ С ПОМОЩЬЮ СИСТЕМЫ NALAPS [Конференция] // Научно-техническая конференция студентов, аспирантов и молодых специалистов МИЭМ. Тезисы докладов.. - М. : МИЭМ, 2010. - стр. 200-201.

Лебедев А. С. Естественнo-языковoe программирование как средство извлечения знаний [Конференция] // Труды ИВМиМГ, Информатика, 9. - Новосибирск : [б.н.], 2009. - стр. 64-71.

Лебедев А. С. Редактор расширенных сетей переходов с графически интерфейсом пользователя [Конференция] // Компьютерная лингвистика и интеллектуальные технологии: По материалам ежегодной Международной конференции «Диалог 2009» (Бекасово, 27-31 мая 2009 г.). Вып. 8 (15). - М. : РГГУ, 2009. - стр. 284-290.

Леонтьева Н. Н. Автоматическое понимание текстов: системы, модели, ресурсы: учеб. пособие для студ. лингв. фак. вузов [Книга]. - М. : Издательский центр «Академия», 2006.

Люгер Джордж Ф. Искусственный интеллект: стратегии и методы решения сложных проблем, 4-е издание [Книга]. - М. : Издательский дом "Вильямс", 2005.

Мельчук И. А. Опыт теории лингвистических моделей "Смысл ↔ Текст" [Книга]. - М. : [б.н.], 1999. - 2-е издание.

Мельчук И. А. Русский язык в модели «Смысл ↔ Текст» [Книга]. - Москва-Вена : Школа "Языки русской культуры", Венский славистический альманах, 1995.

Николай Ильин Сергей Киселёв, Сергей Танков, Владислав Рябышкин Технологии извлечения знаний из текста [Журнал] // Открытые системы. - 2006 г.. - 6.

Ножов И.М. Реализация автоматической синтаксической сегментации русского предложения [Книга]. - Москва : [б.н.], 2003.

Пономарева С.М. Система формирования концептуального представления предметной области на основе анализа проблемно-ориентированных естественно-языковых текстов: модели, методы и программное обеспечение [Книга]. - Переславль-Залесский : [б.н.], 1999.

Путилов Г.П. Лебедев А.С. Визуальный редактор расширенных сетей переходов [Журнал] // Качество. Инновации. Образование. - май 2009 г.. - 5 (48). - стр. 42-48.

Путилов Г.П. Лебедев А.С. Система естественно-языкового концептуального программирования Nalaps [Конференция] // Материалы международной научной конференции Megaling 2009, 21-26 сентября 2009. - Киев : [б.н.], 2009. - стр. 65-66.

Сокирко А. АОТ :: Технологии :: Синтаксический анализ [В Интернете] // АОТ. - 14 05 2010 г.. - <http://aot.ru/docs/synan.html>.

Сокирко А. В. АОТ - Автоматическая Обработка Текста [В Интернете] // АОТ. - 14 05 2010 г.. - <http://aot.ru/index.html>.

Сокирко А. В. Морфологические модули на сайте www.aot.ru [Конференция] // Материалы международной конференции Диалог 2004. - М. : РГГУ, 2004.

Сокирко А. В. Семантические словари в автоматической обработке текста: по материалам системы ДИАЛИНГ [Книга]. - Москва : [б.н.], 2001.

Степанов Ю. С. Имена, предикаты, предложения. Семиологическая грамматика. [Книга]. - [б.м.] : Едиториал УРСС, 2007.

Тамм Б.Г. Тыугу Э.Х. Представление машиностроительных моделей в базах знаний и персональные САПР [Журнал] // Вестник РАН. Журнал №5. - 1988 г.. - стр. 39.

Технологии компании ПРОМТ [В Интернете] // Технологии компании ПРОМТ. - 07 06 2010 г.. - <http://www.promt.ru/company/technology/promt/>.

Тыгу Э. Х. Концептуальное программирование [Книга]. - М. : Наука. Главная редакция физико-математической литературы, 1984.

Филмор Ч. Дело о падеже [Статья] // Новое в зарубежной лингвистике. - М. : [б.н.], 1981 г.. - X.

Хорстман К. С. и Корнелл Г. Java 2. Библиотека профессионала, том 2. Тонкости программирования [Книга]. - [б.м.] : Вильямс, 2009. - ISBN 978-5-8459-1482-8, 978-01-3235479-0.

Шаров С. А. Использование объектно-ориентированного программирования для лингвистического моделирования. [Конференция] // Труды Международного Семинара ДИАЛОГ'95. - 1995. - стр. 332-339.

Шаров С. А. Труды Международного Семинара ДИАЛОГ'95 [Конференция] // Использование объектно-ориентированного программирования для лингвистического моделирования.. - 1995. - стр. 332-339.

Приложение 1.

РОССИЙСКАЯ ФЕДЕРАЦИЯ



СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

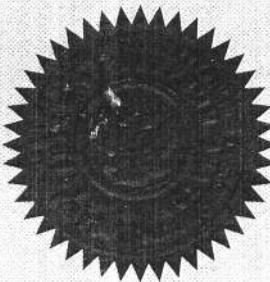
№ 2010616060

Nalaps

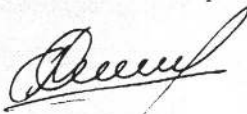
Правообладатель(ли): *Государственное образовательное учреждение высшего профессионального образования «Московский государственный институт электроники и математики (технический университет)» (RU)*

Автор(ы): *Лебедев Андрей Сергеевич, Путилов Георгий Петрович (RU)*

Заявка № **2010614373**
Дата поступления **20 июля 2010 г.**
Зарегистрировано в Реестре программ для ЭВМ
14 сентября 2010 г.



Руководитель Федеральной службы по интеллектуальной собственности, патентам и товарным знакам

 **Б.П. Симонов**

Приложение 2.

РОССИЙСКАЯ ФЕДЕРАЦИЯ



СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

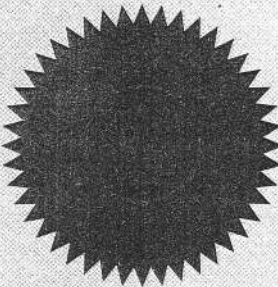
№ 2009613349

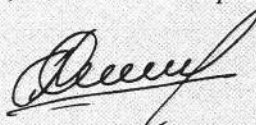
InfoNIOKR

Правообладатель(ли): *Государственное учреждение
«Научно-исследовательский институт микроэлектроники
и информационно-измерительной техники Московского
государственного института электроники
и математики (технического университета)» (RU)*

Автор(ы): *Путилов Георгий Петрович,
Беялетдинов Игорь Исхакович, Лебедев Андрей Сергеевич (RU)*

Заявка № **2009612046**
Дата поступления **5 мая 2009 г.**
Зарегистрировано в Реестре программ для ЭВМ
26 июня 2009 г.

 Руководитель Федеральной службы по интеллектуальной
собственности, патентам и товарным знакам

 **Б.П. Симонов**

Приложение 3.

РОССИЙСКАЯ ФЕДЕРАЦИЯ



СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2010616061

Визуальный редактор расширенных сетей переходов

Правообладатель(ли): *Государственное образовательное учреждение высшего профессионального образования «Московский государственный институт электроники и математики (технический университет)» (RU)*

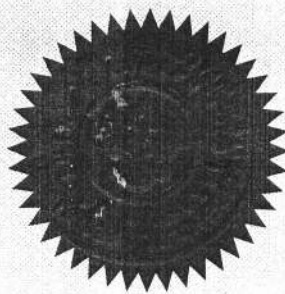
Автор(ы): *Лебедев Андрей Сергеевич, Путилов Георгий Петрович (RU)*

Заявка № 2010614374

Дата поступления 20 июля 2010 г.

Зарегистрировано в Реестре программ для ЭВМ

14 сентября 2010 г.



Руководитель Федеральной службы по интеллектуальной собственности, патентам и товарным знакам

Б.П. Симонов

Приложение 4.



АКТ

об использовании результатов
кандидатской диссертационной работы
Лебедева Андрея Сергеевича

Настоящий Акт свидетельствует, что следующие результаты диссертационной работы Лебедева А.С. на тему «Разработка и исследование системы концептуального программирования с использованием лингвистического процессора» использованы в процессе разработки программного обеспечения в ЗАО «МТЦ»: 1) способ проектирования и построения естественно-языкового спецификатора языка Java, 2) методика построения компонента автоматической обработки текста. Данные результаты были использованы:

- при создании естественно-языковых элементов интерфейса системы StrategyExpress;
- при разработке системы автоматического тестирования некоторых компонентов разрабатываемого программного обеспечения.

Компоненты естественно-языкового анализа позволили упростить процесс создания новых метаданных системы, а система автоматического тестирования снизила затраты на рутинные операции подготовки выпуска новых версий программного обеспечения.

Генеральный директор
ЗАО «МТЦ»



Новиков С.В.

Приложение 5.



МФД-ИнфоЦентр Плюс

информационные услуги и программные продукты
для инвесторов и профессиональных участников рынка России

АКТ

об использовании результатов
кандидатской диссертационной работы
Лебедева Андрея Сергеевича

Настоящий Акт свидетельствует, что следующие результаты диссертационной работы Лебедева А.С. на тему «Разработка и исследование системы концептуального программирования с использованием лингвистического процессора» использованы в процессе разработки программного обеспечения в «МФД-ИнфоЦентр Плюс»: 1) методика проектирования и построения естественно-языкового спецификатора языка Java, 2) методы автоматической генерации программных тестов. Эти результаты были внедрены в технологии проектирования и позволили:

1. Разработать ряд эффективных программных надстроек к интегрированной среде разработки IntelliJ IDEA, позволяющих существенным образом сократить временные затраты на создание автоматических программных тестов к разрабатываемым программным продуктам.
2. Оптимизировать и частично автоматизировать процесс обработки текстовых сообщений об ошибках в программном обеспечении, поступающих в службу технической поддержки.

Использование перечисленных результатов позволило: повысить качество предоставляемых услуг по поддержке программного обеспечения, разрабатываемого предприятием; сократить затраты на рутинные операции и обеспечить возможность анализа поступающих пользовательских сообщений об ошибках.

Генеральный директор
ООО «МФД-ИнфоЦентр Плюс»



Клычков А.Ю.

МФД-ИНФОЦЕНТР ПЛЮС

IFH-INFOCENTRE PLUS

103009, Россия, Москва, Большая Никитская 24/1, стр.5; Тел.: (495)921-15-50; Факс: (495)921-15-50; Телекс: LUVA-RU; E-mail: info@mfd.ru; Для почтовых отправлений: 123242, Москва, а/я №26. <http://mfd.ru>

24/1 Bolshaya Nikitskaya St. B5, 103009 Moscow, Russia; Tel.: (495)921-15-50; Fax: (495)921-15-50; Telex: LUVA-RU; E-mail: info@mfd.ru; 123242, Post Box 26, Moscow. <http://mfd.ru>