

Министерство образования и науки Российской Федерации

Севастопольский государственный университет

Введение в среду разработки Лисп-приложений Eclipse Cusp

Методические указания к лабораторным работам по дисциплине
“Методы и системы искусственного интеллекта”
для студентов дневной и заочной форм обучения
направления 09.03.02 — “Информационные системы и технологии”

**Севастополь
2015**

УДК 004.42 (075.8)

Введение в среду разработки Лисп-приложений Eclipse Cusp: методические указания к лабораторным работам по дисциплине “Методы и системы искусственного интеллекта” для студентов дневной и заочной формы обучения направления 09.03.02 — “Информационные системы и технологии”/ СевГУ; сост. **В. Н. Бондарев**. — Севастополь: Изд-во СевГУ, 2015. — 44 с.

Цель указаний: оказать методическую помощь студентам дневной и заочной форм обучения направления 09.03.02 — “Информационные системы и технологии” в освоении в интегрированной среды разработки Eclipse Cusp, используемой при выполнении лабораторных работ по дисциплине “Методы и системы искусственного интеллекта”

Методические указания составлены в соответствии с требованиями программы дисциплины “Методы и системы искусственного интеллекта” для студентов дневной и заочной форм обучения направления 09.03.02 — “Информационные системы и технологии” и утверждены на заседании кафедры Информационных систем (протокол № от 2015 г.)

Допущено учебно-методическим центром СевГУ в качестве методических указаний.

Рецензент: Кожаяев Е.А., к.т.н., доцент кафедры информационных технологий и компьютерных систем

СОДЕРЖАНИЕ

Введение.....	4
1. Платформа разработки приложений Eclipse.....	5
1.1. Проект Eclipse.....	5
1.2. Архитектура платформы Eclipse	6
2. Установка среды Eclipse и Cusp плагина... ..	8
2.1. Установка исполняющей среды Java.....	8
2.2. Установка Eclipse Classic IDE	9
2.3. Установка Cusp плагина и Лисп-компилятора SBCL	9
3. Практическая работа в среде Eclipse Cusp.....	10
3.1. Интерфейс пользователя Eclipse.....	10
3.2. Элементы проекции Lisp.....	12
3.3. Интерпретация Лисп программ	14
3.4. Простейший проект.....	17
3.5. Файл определения пакета defpackage.lisp.....	19
3.6. Создание исполняемого файла .exe.....	21
3.7. Сообщения компилятора и отладка программы	22
3.8. Свойства редактора Cusp	28
3.9.Добавление файлов в проект.....	33
Библиографический список.....	34
Приложение А. Использование библиотек.....	35
Приложение Б. Тестирование.....	37
Приложение В. Пример проекта.....	38
Приложение Г. Запись и чтение s-выражений из файлов.....	41
Приложение Д. Получение справочной информации из Интернет.....	42

ВВЕДЕНИЕ

Cusp — это среда разработки для языка программирования **Лисп**. При помощи языка Лисп можно разрабатывать все виды приложений, в том числе и столь популярные сегодня Web-приложения. Лисп был разработан в конце 50-х годов 20-го века при выполнении исследований в области искусственного интеллекта (ИИ). В основе языка Лисп лежит *функциональная модель* вычислений, ориентированная, прежде всего, на решение задач *символьной обработки*, которые характерны для ИИ. Название языка Лисп (**Lisp**) происходит от английских слов **LISt Processing** — обработка списков. Представление данных и программ в виде списочных структур одно из важнейших свойств Лиспа.

Имеется большое количество диалектов языка Лисп. В 1981 г. агентство DARPA при Министерстве обороны США инициировало работы по разработке стандарта языка. В результате этого в 1984 г. была разработана версия языка под названием Коммон Лисп (Common Lisp). **Коммон Лисп** содержит весь арсенал средств, необходимый для разработки больших программных систем. Его современные версии поддерживают технологию объектно-ориентированного программирования [1].

Рассматриваемая в методических указаниях среда **Cusp**, обеспечивающая поддержку разработки приложений на языке Коммон Лисп, представляет расширение (дополнение) для платформы **Eclipse**. В настоящее время эта платформа завоевывает все новые рубежи и становится, практически, стандартом для выполнения разработок на самых различных языках во многих компаниях [2,5]. Сочетание промышленного качества, кроссплатформенность, открытость кода и бесплатность, а также возможность дополнения Eclipse не только средствами разработки на языке Лисп, но и средствами разработки Пролог-приложений, делают эту среду универсальным инструментом при изучении языков и методов искусственного интеллекта.

Программирование задач искусственного интеллекта на языке Лисп в современной интегрированной среде поддержки разработок **Eclipse** должно обогатить студентов как освоением необычных для них принципов *функционального программирования*, свойственных Лиспу, так и знакомством с самой средой, которая обеспечивает необходимую интеграцию самых различных инструментов.

Применение единой среды программирования при изучении различных разделов дисциплины «Методы и системы искусственного интеллекта» методически оправдано, так как позволит сосредоточиться в большей степени на решении самих задач ИИ.

Основная цель методических указаний оказать помощь студентам в освоении интегрированной среды разработки **Eclipse Cusp**, используемой при выполнении лабораторных работ по дисциплине «Методы и системы искусственного интеллекта»

1. ПЛАТФОРМА РАЗРАБОТКИ ПРИЛОЖЕНИЙ ECLIPSE

1.1. Проект Eclipse

Eclipse ([i'klips] от англ. *затмение*) представляет собой основанную на **Java** интегрированную инструментальную платформу с открытым исходным кодом (OpenSource), предназначенную для разработки кроссплатформенных приложений. По сути, платформа Eclipse — это расширяемая среда разработки и набор сервисов для создания **программного обеспечения (ПО)** на основе встраиваемых компонентов (**плагинов**). Eclipse также включает в себя среду разработки плагинов (**PDE**), что дает разработчикам инструментарию возможность предложить свои расширения к Eclipse.

Изначально проект разрабатывался в **IBM** как корпоративный стандарт интегрированной среды разработки (**IDE**) для различных платформ. С 2004г. проект разрабатывается и поддерживается независимой некоммерческой организацией **Eclipse Foundation** (www.eclipse.org). Среди участников Eclipse Foundation следует назвать компании IBM, Nokia, Oracle, SAP, OBEI, Ericsson, Intel и др.

Хотя платформа Eclipse и написана на Java, её возможности не ограничиваются разработками на этом языке. Помимо Java, в Eclipse имеются расширения, поддерживающие разработку ПО на языках **C/C++**, **FORTRAN**, **COBOL**, **PHP**, **Perl**, **Python**, **Groovy**, **Erlang**, **Ruby** и др. Множество расширений дополняет Eclipse средствами для работы с базами данных, моделирования, разработки графических приложений, серверов приложений и др. В настоящее время имеется более 1000 плагинов, расширяющих возможности Eclipse (<http://marketplace.eclipse.org/>).

В силу бесплатности и промышленного качества Eclipse постепенно завоевывает позиции корпоративного стандарта для разработки самых различных приложений во многих организациях.

Последние версии Eclipse: Eclipse 3.3.2 (**Europa**), Eclipse 3.4 (**Ganymede**, 2008 г.), Eclipse 3.5 (**Galileo**, 2009г.), Eclipse 3.6 (**Helios**, 2010г.), Eclipse 3.7 (**Indigo**, 2011г.).

Выход следующей версии Eclipse 4.2 (**Juno**) запланирован на июнь 2012г. В рамках проекта Juno разрабатывается более 50 подпроектов.

Подпроекты Eclipse сгруппированы в пакеты в соответствии с основными потребностями разработчиков и размещены по адресу: <http://www.eclipse.org/downloads/> Более детальные сведения о проекте Eclipse и его подпроектах приведены в [2].

В настоящих методических указаниях рассматривается архитектура Eclipse, общие принципы организации интерфейса пользователя и инструментарий разработки Lisp-приложений **Cusp**.

1.2. Архитектура платформы Eclipse

Платформа Eclipse обеспечивает реализацию следующих возможностей [2,3,5]:

- поддержку конструирования разнообразных инструментов для разработки приложений;
- поддержку неограниченного числа поставщиков инструментариев, включая независимых поставщиков ПО;
- поддержку инструментов манипулирования данными произвольных типов (т.е., HTML, Java, C, JSP, EJB, XML и GIF).
- обеспечение "бесшовной" интеграции инструментов разных поставщиков для работы с содержимым различных типов;
- поддержку сред разработки приложений как с графическим интерфейсом пользователя (GUI), так и без него;
- поддержку широкого спектра операционных систем, включая Windows , Linux, Mac OS и др.;
- использование кроссплатформенного языка Java для написания инструментария.

Для интеграции необходимых пользователю инструментов в единую систему в Eclipse используется идея сборки платформы из *модулей с унифицированными интерфейсами подключения (плагинов)*. Для этого в плагинах имеются специальные *точки расширений (extension point)*. Благодаря этому платформа реализуется в виде слоев плагинов, в каждом из которых определяются *расширения* для подключения плагинов нижних уровней, а также определяются свои точки расширений для подключения плагинов верхних уровней. Такая модель позволяет разработчикам плагинов легко добавлять новые инструменты к платформе.

В качестве примера на рис.1.2 изображены основные компоненты платформы Eclipse, распространяемой в виде пакета Eclipse Classic [5]. Платформа построена в виде набора расширяемых подсистем, а не как единое приложение. Каждая из подсистем реализуется с помощью одного или нескольких плагинов.

В пакет Eclipse Classic входят три подсистемы, разрабатываемые более или менее независимо друг от друга — собственно **платформа** (Eclipse Platform), **Java-инструменты** (JDT — Java development tools) и **среда разработки плагинов** (PDE — Plug-in development environment). Платформа предоставляет базовые сервисы, JDT позволяет разрабатывать приложения Java, а PDE — новые компоненты Eclipse. Указанные подсистемы являются надстройкой над **средой выполнения платформы** (Platform runtime).

Платформа является ядром Eclipse. Возможности, которые обеспечивает платформа, позволяют определять основные объекты, с которыми будет работать пользователь, создавать пользовательские интерфейсы, поддерживать коллективную (командную) работу с сохранением версий разрабатываемого ПО, поддерживать работу с отладчиками, получать справочную информацию. Соответствующими компонентами платформы, реализующими эти возможности, являются: **рабочее пространство** (Workspace), **рабочий стол** (Workbench — базовый пользо-

вательский интерфейс Eclipse), **средства поддержки версий** (CVS— Concurrent Versions Systems), **подсистема помощи** (Help).

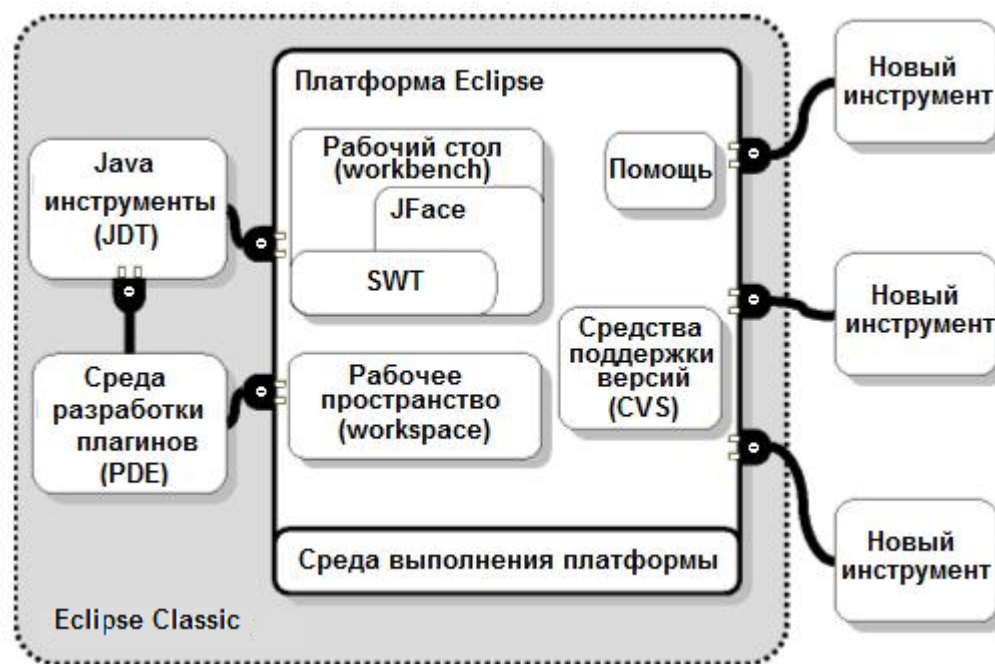


Рисунок 1.2— Основные компоненты платформы Eclipse

Среда выполнения платформы (Platform runtime) определяет основные плагины (`org.eclipse.osgi` и `org.eclipse.core.runtime`), которые используются всеми другими плагинами. Она динамически обнаруживает плагины и собирает информацию о плагинах и их точках расширения в реестре платформы. Она также отвечает за поиск и выполнение основного приложения.

Рабочее пространство (Workspace) определяет основные объекты, с которыми могут работать пользователи и приложения Eclipse, структурирует эти объекты и предоставляет возможность управления ими. Объекты, о которых идет речь — это *проект* (project), *папка* (folder) и *файл* (file). В терминологии Eclipse все эти объекты называются *ресурсами*. В техническом плане компонент Workspace обеспечивает унифицированный доступ ко всем указанным ресурсам.

Рабочий стол (Workbench) обеспечивает базовый пользовательский интерфейс (UI), а также определяет точки расширений для добавления собственных компонентов пользовательского интерфейса. Основные объекты интерфейса, с которыми работает Workbench, — это *редакторы* (editors), *представления* (views) и *перспективы* (perspectives).

Фундаментом, на котором построены все графические средства Eclipse, является входящий в Workbench, компонент **SWT** (Standard Widget Toolkit). SWT является основным средством создания пользовательского интерфейса для приложений на основе Eclipse. Для прорисовки графических элементов он использует средства оконного интерфейса соответствующей операционной системы (например, Win 32 API). Поэтому приложения, построенные на базе SWT, визуально не

отличаются от “родных” приложений используемой операционной системой. На основе SWT построен компонент **JFace**, который решает задачи построения пользовательского интерфейса более высокого уровня. JFace содержит средства для создания диалогов, страниц свойств, управления шрифтами и др.

JDT — интегрированная среда разработки для Java, содержащая компилятор Java, редакторы, средства для отладки, тестирования, навигации и рефакторинга исходного кода.

PDE обеспечивает поддержку разработки разнообразных плагинов, редакторов, представлений и страниц настроек. Именно PDE позволяет расширять сам Eclipse.

Рассматриваемый ниже пакет **Cusp** представляет собой одно из дополнений (плагин) платформы **Eclipse** для программирования на языке Коммон Лисп. **Cusp** предусматривает совместную работу с эффективным свободно распространяемым Лисп-компилятором **SBCL (Steel Bank Common Lisp)** и набором тестовых библиотек. **Eclipse** совместно с **Cusp** удовлетворяет всем основным требованиям, которые предъявляются к современным интегрированным средам разработки. В частности, обеспечивает [4]:

- подсветку кода;
- сворачивание различных частей кода и автозавершение конструкций языка;
- ассоциативную помощь;
- автоматическое создание проекта и отображение его структуры;
- встроенное тестирование и управление библиотеками;
- отладку с возможностью приостановки программы в заданных точках.

2. УСТАНОВКА СРЕДЫ ECLIPSE И CUSP ПЛАГИНА

2.1. Установка исполняющей среды Java

Перед тем как устанавливать Eclipse, необходимо убедиться, что на компьютере установлена виртуальная Java-машина (Java Virtual Machine — **JVM**), которую часто называют исполняющей средой **JRE (Java Runtime Environment)**. JRE может быть установлена либо отдельно, либо в составе комплекта разработки программного обеспечения на языке Java (Java Development Kit — **JDK**). Комплекты последних версий JRE и JDK Standard Edition (Java SE) можно свободно загружать с сайта:

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

Следует отметить, что комплект JDK дополнительно содержит средства, предназначенные для специальных отладочных режимов работы с Java-машиной.

Чтобы проверить версию исполняющей среды JRE, установленной на компьютере, введите в командной строке команду:

```
java -version
```

Рекомендуется установить последнюю из имеющихся версий. На момент написания данных методических указаний такой версия была JRE 1.7.0_02.

2.2. Установка Eclipse Classic IDE

Чтобы установить пакет Eclipse Classic IDE необходимо:

- 1) открыть веб-страницу www.eclipse.org/downloads/packages/ и выбрать для загрузки **Eclipse Classic 3.7.1** (174 Мб, Windows 32-bit) в результате будет загружен архив **eclipse-SDK-3.7.1-win32.zip**;
 - 2) перейти на страницу проекта **Babel Language Packs for Indigo**:
http://archive.eclipse.org/technology/babel/babel_language_packs/R0.9.0/indigo/indigo.php ;
 - 3) найти раздел **Language: Russian**. Загрузить архивы, частично русифицирующие устанавливаемую среду:
[BabelLanguagePack-eclipse-ru_3.7.0.v20110723043401.zip \(75.36%\)](#);
[BabelLanguagePack-mylyn-ru_3.7.0.v20110723043401.zip \(30.76%\)](#) ;
[BabelLanguagePack-rt.equinox-ru_3.7.0.v20110723043401.zip \(70.8%\)](#) ;
[BabelLanguagePack-rt.equinox.p2-ru_3.7.0.v20110723043401.zip \(32.75%\)](#) ;
 - 4) разархивировать все загруженные zip-архивы на диск **c:** с сохранением структуры папок архивов.
- В итоге будет создана папка **c:\eclipse**, в которой будут располагаться все файлы Eclipse. Среда Eclipse готова для запуска.

2.3. Установка Cusp плагина и Лисп-компилятора SBCL

Eclipse позволяет выполнять автоматическую загрузку и установку из Интернета различных дополнений с помощью специального менеджера обновлений и загрузки. Для этого после запуска Eclipse следует выбрать **Справка > Установить новое ПО (Install New Software)** и указать адрес соответствующего сайта, где находится необходимое дополнение.

Поскольку в нашем случае наряду с установкой **Cusp** плагина требуется выполнить установку и Лисп-компилятора, то рассмотрим «ручную» установку. Для этого необходимо выполнить следующее:

- 1) загрузить архив с **Cusp** плагином [cusp0.9.390.zip](#) (7,6 Мбайт), который находится по адресу <http://www.sergeykolos.com/cusp/archive/> , разархивировать указанный архив с сохранением его структуры в папку **c:\eclipse**;
- 2) перейти на страницу <http://www.sbcl.org/platform-table.html> и загрузить последний релиз двоичного установочного файла Steel Bank Common Lisp (SBCL) для ОС Windows — файл **sbcl-1.0.54-x86-windows-binary.msi** (10,1 Мбайт);
- 3) дважды щелкнуть мышью на файле **sbcl-1.0.54-x86-windows-binary.msi** и выполнить установку **SBCL** в папку, предлагаемую по умолчанию в процессе установки — **C:\Program Files\Steel Bank Common Lisp** (можно также установить SBCL в папку **C:\Eclipse\plugins\sbcl_win32_1.0.54\sbcl**; в таком случае следующий шаг установки можно пропустить);
- 4) проверить в разделе *переменных среды пользователя* (Пуск > Панель управления > Система > Дополнительные параметры системы > Переменные сре-

ды) значение переменной **SBCL_HOME**, если переменной нет, то создать её и установить для неё значение **C:\Program Files\Steel Bank Common Lisp\1.0.54**
Среда Eclipse с установленным плагином **Cusp** готова для запуска.

3. ПРАКТИЧЕСКАЯ РАБОТА В СРЕДЕ ECLIPSE CUSP

3.1. Интерфейс пользователя Eclipse

Выполним первый запуск Eclipse. Для этого в папке **c:\Eclipse** необходимо найти файл **eclipse.exe** и создать для него на рабочем столе ярлык «**Eclipse**». Дважды щелкнув по ярлыку, запустим Eclipse.

Рабочая область (workspace)

Первое окно, которое отобразится на экране (рис. 3.1), — это диалоговое окно выбора **рабочей области** (workspace). В простейшем случае *рабочая область* (*пространство*) — это каталог на диске, в котором хранятся проекты пользователя. *Проект* (project) представляет собой набор файлов с исходными и бинарными кодами, файлов сценариев компоновки и других дополнений. Всё, что находится внутри этого каталога, считается частью рабочей области. В методических указаниях рабочая область будет размещаться в каталоге **D:\Eclipse_Lisp**.

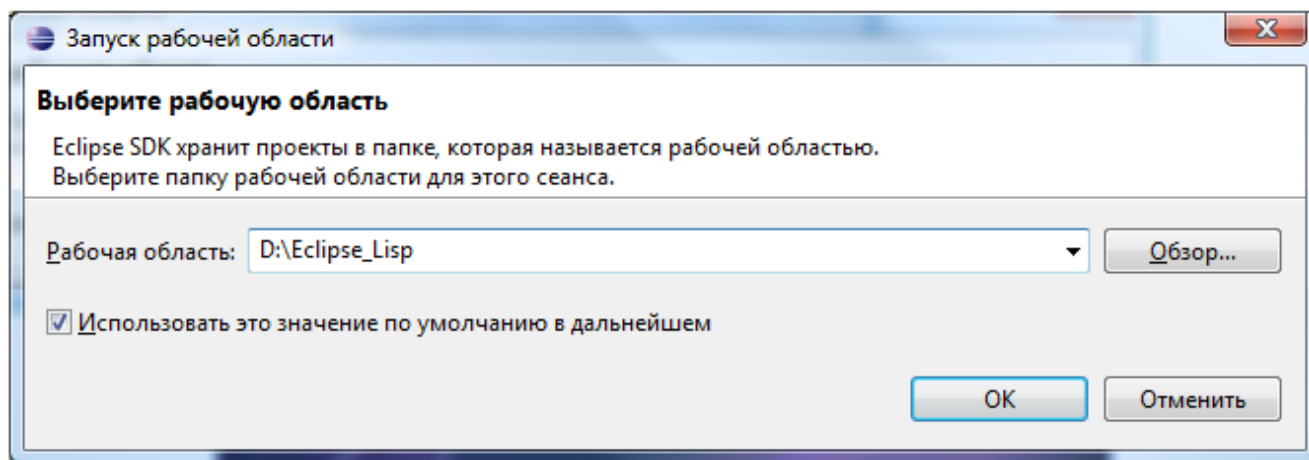


Рисунок 3.1 — Выбор рабочей области (пространства)

После ввода имени папки, в которой будут храниться файлы ваших Lisp проектов, следует установить флажок «**Использовать это значение по умолчанию в дальнейшем**». Установка флажка позволит использовать выбранную рабочую область по умолчанию. При последующих вызовах данное окно появляться не будет. В дальнейшем, чтобы сменить рабочую область, указанное окно можно открыть с помощью команд меню **Файл > Сменить рабочую область**.

Начальная страница (welcome)

Нажатие кнопки «**ОК**» в окне выбора рабочей области приведёт к появлению начальной страницы (страницы приветствия — welcome), на которой имеется 5 графических кнопок (рис. 3.2):

- Обзор — содержит ссылки на обучающие Интернет-ресурсы Eclipse;

- Новое — содержит обзор основных нововведений пакета;
- Примеры — изучение Eclipse на готовых примерах;
- Учебники — содержит ссылки на справочную систему Eclipse;
- Рабочая среда — это рабочий стол с основными инструментами среды.

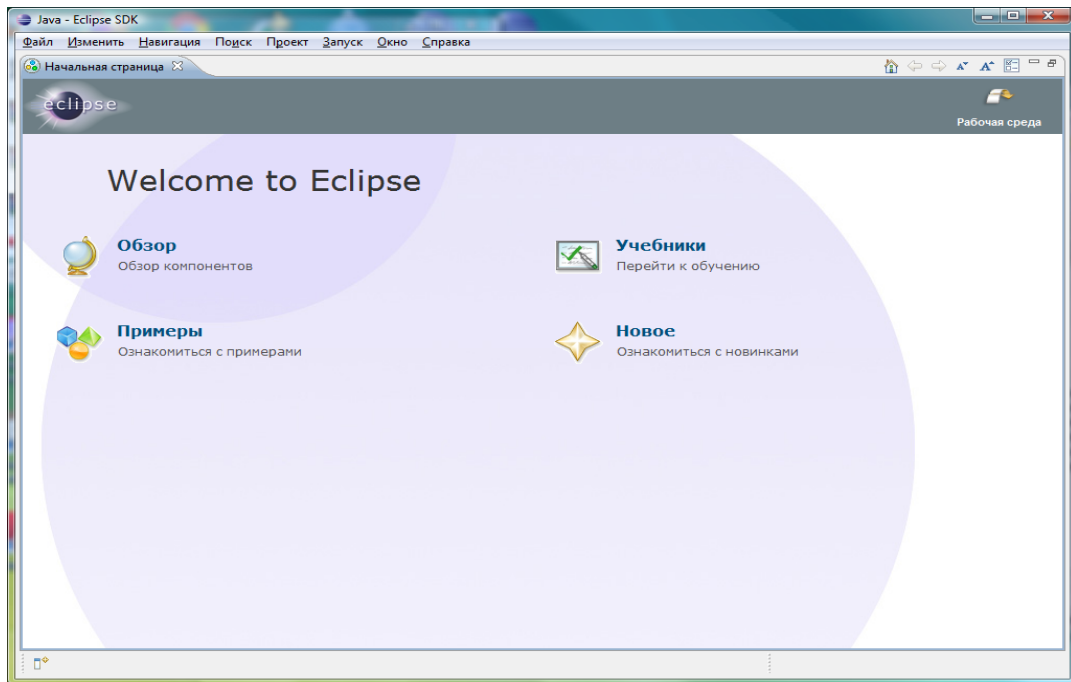


Рисунок 3.2 — Начальная страница

Рабочий стол (workbench)

Для того чтобы приступить к работе, нажмите кнопку **Рабочая среда**. По умолчанию откроется *рабочий стол (workbench)*, изображенный на рисунке 3.3.

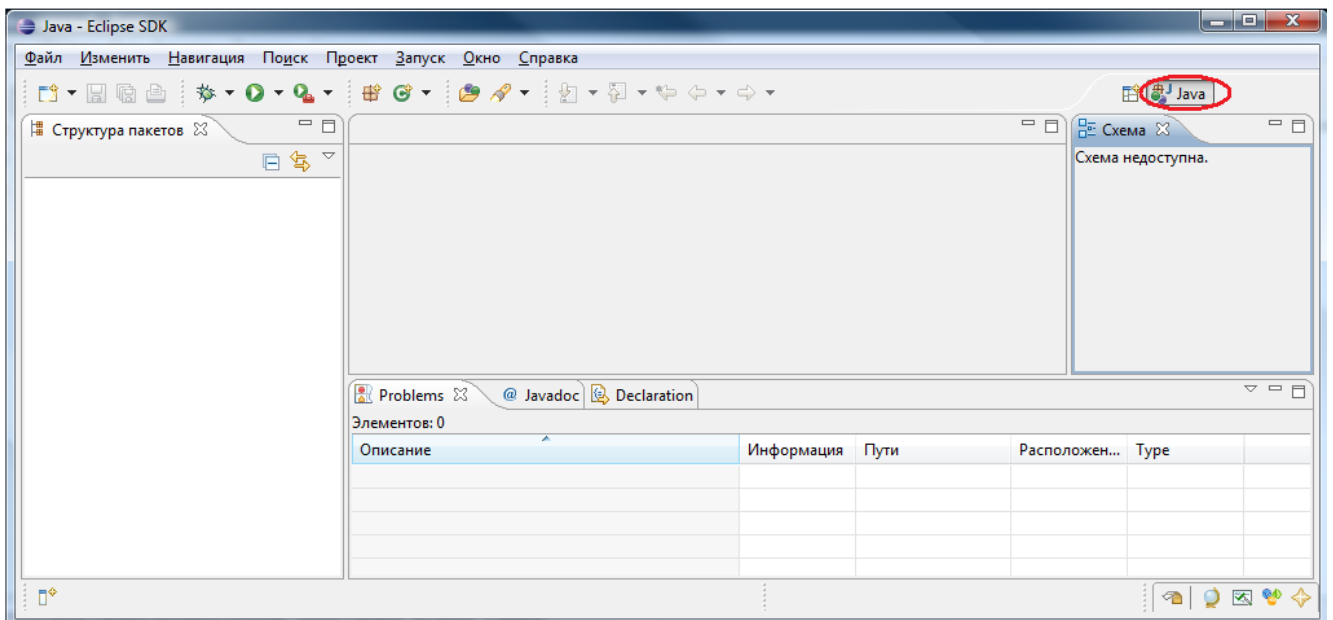


Рисунок 3.3 — Рабочий стол, открываемый по умолчанию

На рис. 3.3 в правом верхнем углу овалом выделена метка, которая отображает текущий режим рабочего стола. Рабочий стол — это, по существу, сама

платформа с набором инструментов. К инструментам рабочего стола относится набор соответствующих *редакторов (editors)* и *панелей (views)*, размещенных на нём.

Проекции (perspectives)

Для конкретной задачи набор определенных редакторов и панелей, расположенных на рабочем столе в определенном порядке, называют *проекцией (перспективой — perspective)* [3].

В каждой проекции присутствует свой набор инструментов. Некоторые проекции могут иметь общие наборы инструментов. В определенный момент времени активной может быть только одна проекция. Быстрое переключение между различными проекциями осуществляется нажатием клавиш **<Ctrl+F8>**. На рис. 3.3 изображена проекция Java.

Используя проекции можно настроить свой рабочий стол под определенный тип выполняемой задачи. Открыть соответствующую проекцию можно командой меню **Окно > Открыть проекцию (Window > Open Perspective)**. В дальнейшем вся работа будет выполняться с использованием проекции Lisp.

Редакторы (editors)

Редакторы позволяют создавать, открывать, редактировать, сохранять файлы различных типов.

Панели (views)

Панели (представления — views) являются дополнениями к редакторам и обеспечивают вывод дополнительной информации о файлах и проекте в целом. Открыть панель можно командой меню **Окно > Показать панель (Window > Show View)**.

3.2. Элементы проекции Lisp

На рис. 3.4. изображена проекция Lisp с открытым проектом. В этой проекции рабочего стола можно выделить несколько основных составляющих интерфейса пользователя, которые на рисунке заключены в прямоугольники (окна) и обозначены цифрами.

Прежде всего, это основные *элементы управления средой*:

- **Строка меню** (окно 1) — содержит пункты меню платформы Eclipse с набором функций для работы с проектами;
- **Панель инструментов** (окно 2) — содержит набор кнопок, которые обеспечивают быстрый выбор того или иного инструмента;
- **Открытие проекции** (окно 3) — кнопка, позволяющая выбрать необходимую проекцию (перспективу) из списка имеющихся проекций;
- **Текущая проекция** (окно 4) — отображает имя текущей активной проекции;
- **Показать панель как быструю панель** (окно 9) — кнопка, которая позволяет осуществить быстрое открытие одной из панелей (представлений), указанных ниже. В дальнейшем будем называть эту кнопку просто «**Быстрая панель**».

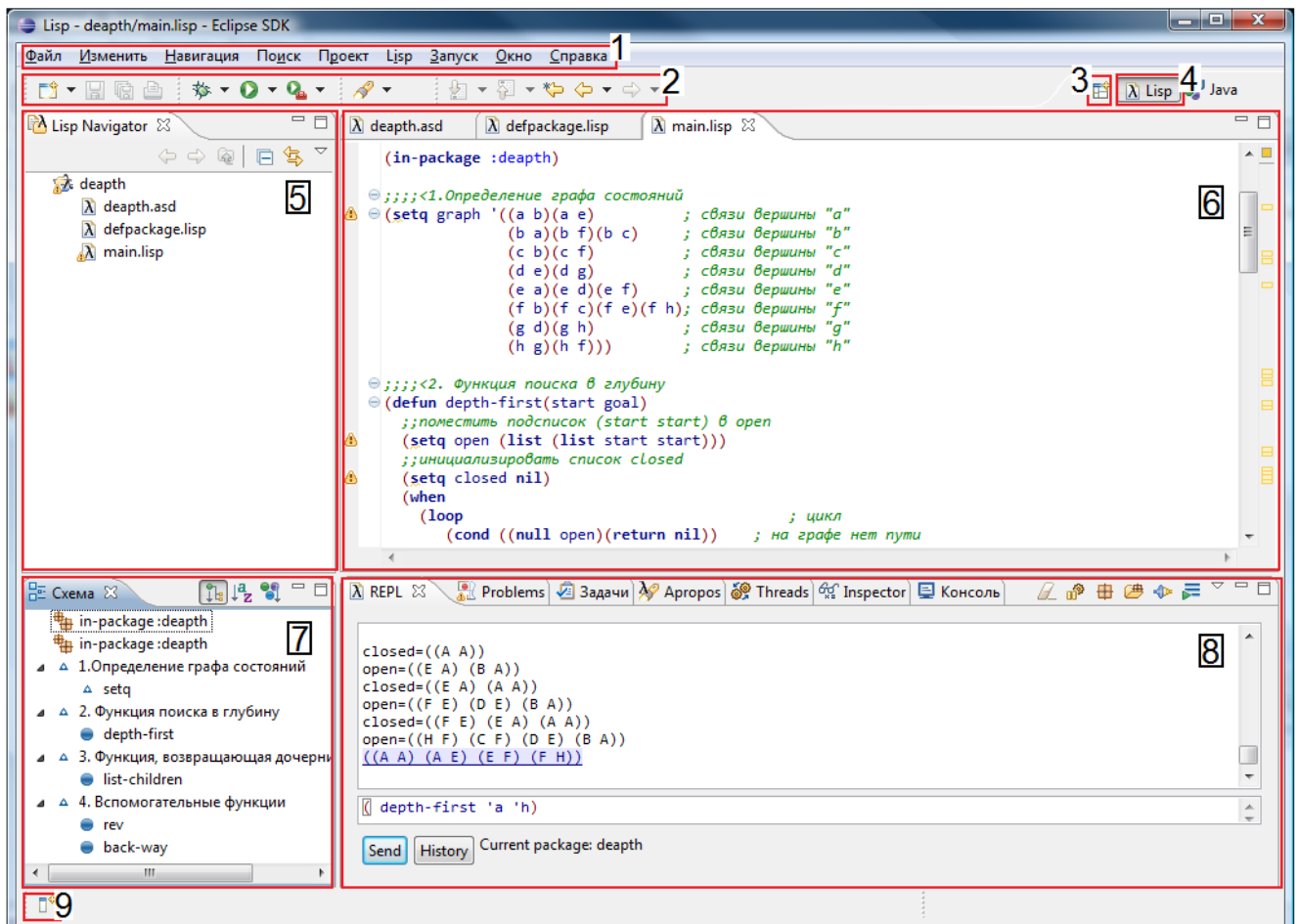


Рисунок 3.4 — Проекция Lisp

В окнах, обозначенных номерами 5,6,7и 8 (рис. 3.4), отображаются различные **панели** (представления), расположение которых, при необходимости, можно менять:

- **Lisp Navigator** (окно 5) — отображает структуру рабочей области в виде каталогов и файлов, входящих в проект;
- **Редактор** (Editor) (окно 6) — обеспечивает ввод и редактирование файлов проекта;
- **Схема** (Outline) (окно 7) — обеспечивает отображения структуры файла, который в данный момент открыт в окне редактора;
- **REPL (read, eval, print -loop)** (окно 8) — интерактивная панель для взаимодействия с SBCL, реализующая основной цикл интерпретатора Лисп: считывание s-выражения(*read*), оценку его значения (*evaluate*) и вывод результатов (*print*);
- **Неполадки (Problems)** (окно 8) — отображает предупреждения и ошибки компиляции;
- **Задачи** (Tasks) (окно 8) — отображает список задач, которые вы запланировали;
- **Threads** (окно 8) — обеспечивает работу с многопоточными Lisp-приложениями ;

- **Inspector**(окно 8) — отображает свойства объектов, выделяемых на панели REPL;
- **Консоль** (Console) (окно 8) — системная консоль, используемая для ввода-вывода данных программы, а также для вывода результатов работы компилятора;
- **Apropos** (от англ. *кстати*) (окно 8) — для введенной подстроки символов обеспечивает вызов функции *ассоциативного поиска* (apropos), которая возвращает все известные системе символы, в именах которых содержится введенная подстрока (рис. 3.5).

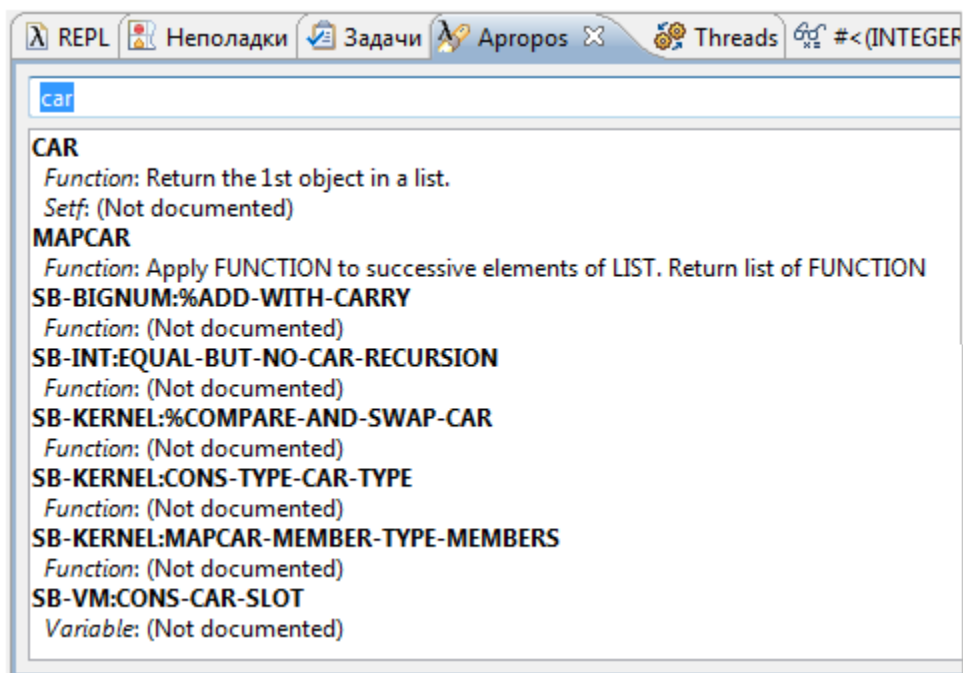


Рисунок 3.5 — Окно функции Apropos

Кнопка «**Быстрая панель**», находящаяся в левом нижнем углу рабочего стола, открывает список доступных панелей и добавляет выбранную панель поверх окон 5 и 7 (рис 3.4). При этом вновь открытая панель закрывает собой некоторую часть рабочего стола. Имеется возможность добавить открываемую панель в любое из окон 5,6,7 или 8. Для этого необходимо нажать мышью на закладке заголовка панели и перетащить её в необходимое окно. В процессе перетаскивания панели на рабочем столе будет появляться прямоугольная рамка, показывающая новое расположение панели.

3.3. Интерпретация Лисп программ

Лисп предназначен, прежде всего, для обработки символьной информации. Поэтому основным типом данных, используемым в языке, является *символьное выражение или s-выражение* [1]. С помощью s-выражений в Лиспе представляются как данные, так и программы. В простейшем случае s-выражение представляет собой список элементов, заключенных в круглые скобки, например:

(a b c d e)

Программа на языке Лисп состоит из s-выражений и её выполнение сводится к вычислению s-выражений. Однако не любое s-выражение может быть вычислено. S-выражение, которое может быть вычислено ЛИСП-интерпретатором, называется *формой*.

На рисунке 3.6 приведена обобщенная схема интерпретации s-выражений в Лисп системах [1]. Работу схемы можно представить в виде трех шагов.

На первом шаге происходит считывание s-выражения с помощью встроенной функции READ. По умолчанию s-выражение вводится с клавиатуры.

На втором шаге выполняется интерпретация s-выражения с помощью функции EVAL, которая является вызовом интерпретатора.

На третьем шаге функция PRINT осуществляет вывод значения s-выражения. Затем цикл **READ-EVAL-PRINT** (*REPL*) повторяется для другого s-выражения и т.д.

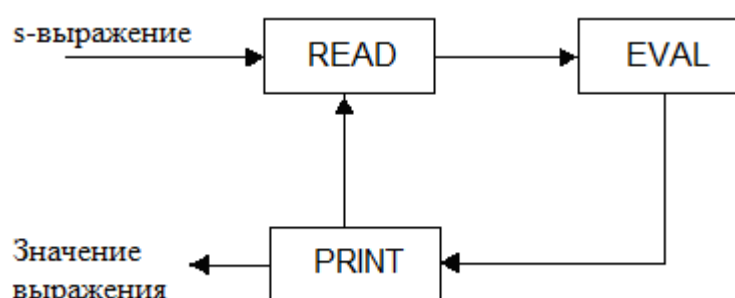


Рисунок 3.6 — Схема интерпретации s-выражений

Программы на языке Лисп состоят из форм и функций. Лисповская функция может рассматриваться как коллекция форм, которые вычисляются при вызове функции. В свою очередь, вызов функции выполняется по её имени из формы, записанной в виде списка:

(имя-функции *аргумент1* *аргумент2* ... *аргументN*)

Например, если требуется вычислить сумму чисел 5,4,3,2 и 1, то с клавиатуры вводится следующее s-выражение (+ 5 4 3 2 1). Здесь знак “+” — это имя функции, выполняющей сложение, а числа 5,4,3,2 и 1 — аргументы функции. В результате в цикле REPL будет получено значение 15, которое отобразится на экране.

Для того чтобы проверить эти возможности, необходимо выбрать **Окно > Открыть проекцию > Прочие > Lisp** или вверху справа нажать мышкой кнопку **Открыть проекцию** и в списке **Прочие** выбрать проекцию **Lisp**. Откроется Lisp проекция, изображенная на рис. 3.4.

При первом открытии Lisp проекции необходимо немного подождать, пока Cuspr выполнит компиляцию кода для соединения с Лисп процессом. При последующих запусках процесс открытия Lisp проекции будет происходить быстрее.

Панель REPL этой проекции содержит верхнюю и нижнюю части (рис. 3.7). Нижняя часть используется для ввода s-выражений, а верхняя — для отображения введенного выражения и результатов вычислений. После ввода s-выражения следует нажать кнопку **Send** (или **Ctrl+Enter**), чтобы передать выражение интерпре-

татору **EVAL**. В примерах, изображенных на рис. 3.7, были введены простейшие s-выражения, обеспечивающие вычисление суммы чисел и поиск максимального числа.

Кнопка **History** позволяет в отдельном окне просматривать список ранее введенных выражений и выбирать мышью любое из выражений для повторного вычисления. Для быстрого выбора предыдущего или следующего выражения из списка **History** (без открытия окна) можно использовать сочетание клавиш **Ctrl+P** и **Ctrl+N**, соответственно.

Панель REPL является удобным интерактивным средством для изучения функций языка Лисп. Просто вводите в нижней части панели s-выражение, содержащее вызов интересующей вас функции, нажимайте кнопку **Send** и анализируйте результаты.

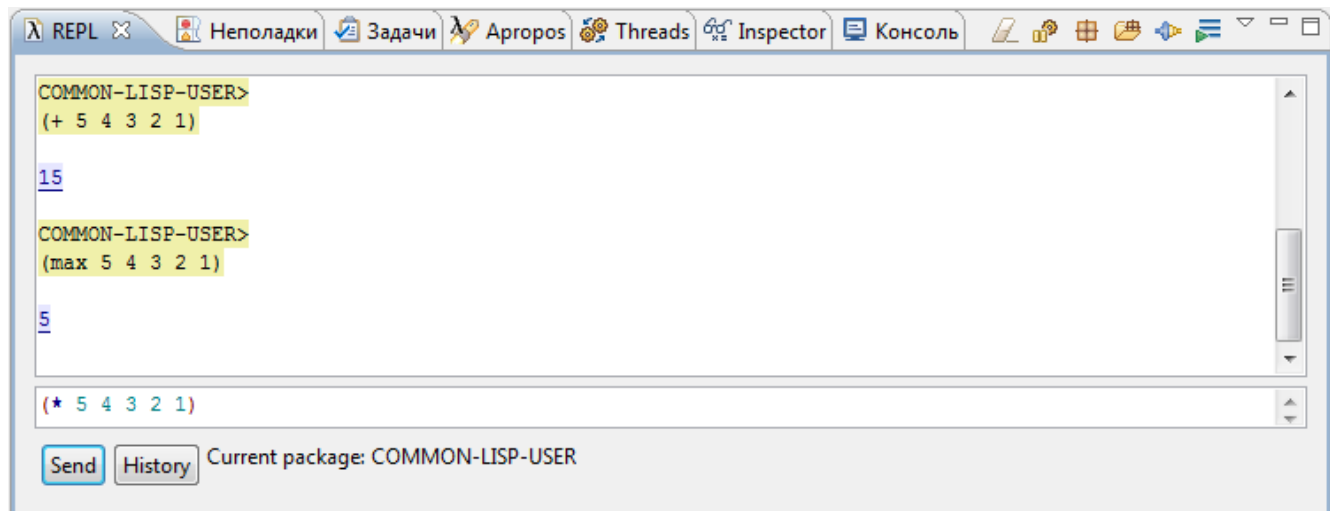


Рисунок 3.7 — Панель REPL

В общем случае s-выражения (т.е. вычисляемые формы) могут быть заданы не только списками, но также константами и переменными [1]. Форма, заданная в виде списочной структуры, может представлять: вызов функции, рассмотренный выше, вызовы специальных форм, макровыводы. *Константы* соответствуют *самоопределяемым формам*, которые представляют самих себя, имеют фиксированные значения и не требуют вычислений — числа, символы **T** (истина) и **NIL** (ложь или пустой список), строки и др. *Переменные* в Лиспе представляются *символами*. При вычислении формы, заданной символом, **EVAL** возвращает последнее значение, которое было *связано* с символом. Если символ не имеет значения, то выдается сообщение об ошибке. Символы в Лиспе, только внешне похожи на переменные. Например, символу одновременно может быть назначено не только значение, но и определение функции и др. Поэтому не следует отождествлять символы Лиспа с именами переменных алгоритмических языков.


Реализация цикла REPL в системе SBCL имеет свои особенности. SBCL, по сути, является *не интерпретатором, а компилятором*. Поэтому после ввода s-выражения **EVAL** преобразует его в так называемое *лямбда-выражение* (представляет собой код тела функции без имени), которое путём вызова функции **compile** компилируется “на лету” и затем полученный код вызывается для испол-

нения вызовом **funcall**. Вызов **EVAL** обеспечивает непосредственную интерпретацию только некоторых простых форм s-выражений, таких как связанные символы.

3.4. Простейший проект

Приступим к созданию и выполнению простейшей программы (приложения) «HelloWorld» с использованием Cusp. Эта программа будет выводить в консоль Eclipse фразу «Здравствуй, мир». Для того чтобы создать проект с такой программой, необходимо выполнить несколько шагов.

Шаг1. Создание проекта

Выбрать **Файл > Создать > Lisp Project** или нажать на панели инструментов стрелку рядом с кнопкой  — **Создать** и в выпадающем списке выбрать **Lisp Project**. В результате откроется окно создания нового Лисп-проекта (рис. 3.8).

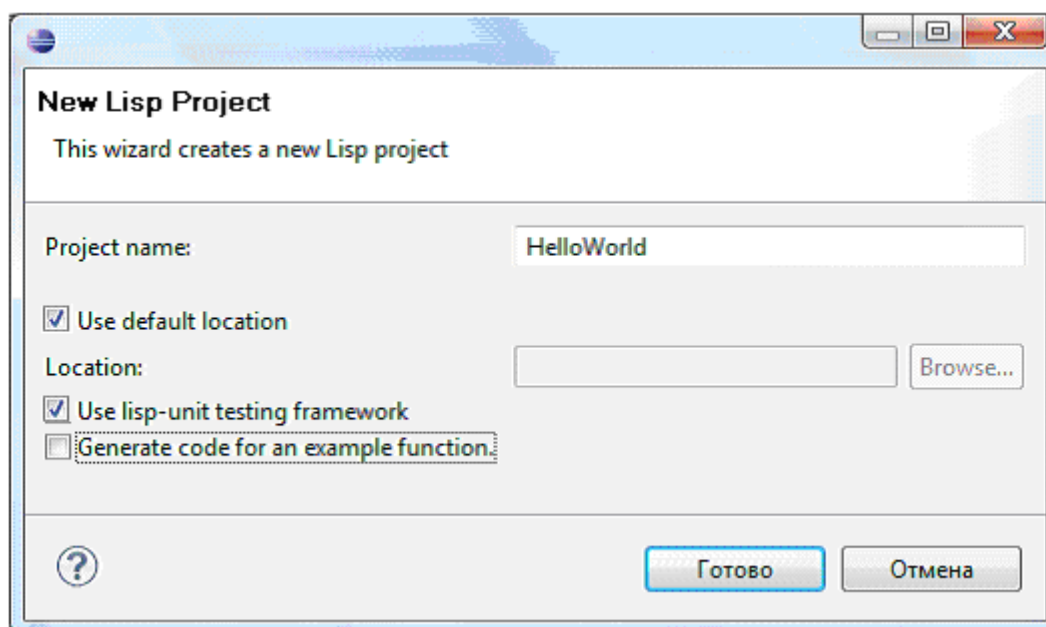


Рисунок 3.8 — Менеджер создания Лисп проекта

В окне следует ввести имя проекта, например **HelloWorld**, а затем снять отметку напротив строки **Generate code for an example function** (генерировать код примера функции). Нажмите кнопку **Готово**.

В результате будет создан новый проект. При этом в рабочей области создаётся папка проекта **HelloWorld**, которая отобразится в панели **Lisp Navigator**. Папка будет содержать четыре файла:

- **helloworld.asd** — файл проекта, используемый при компиляции и загрузке проекта, указывающий Лисп системе, какие библиотеки следует подключить к проекту и какие файлы образуют проект;
- **defpackage.lisp** — интерфейсный файл (наподобие заголовочных файлов в проектах C/C++), в котором объявляются символы языка Лисп, экспортируемые из пакета, либо импортируемые в пакет;
- **main.lisp** — файл с исходным кодом программы (пока пустой);

- **test.lisp** — файл, в который вводят тест исходного кода программы (если нет необходимости в его создании, то следует снять отметку напротив строки **Use lisp-unit testing framework**, изображенную на рис 3.8).

Указанные файлы будут автоматически открыты в окне редактора.

Будучи созданным, проект автоматически компилируется и загружается. При этом в панели REPL появятся сообщения об успешной компиляции, изображенные на рис. 3.9. В последней строке сообщения указано, что текущий пакет изменен на **helloworld**. Об этом также свидетельствует изменение записи внизу панели REPL в поле **Current package** с **COMMON-LISP-USER** (см. рис. 3.7) на **helloworld**.

```
; compiling file "D:/Eclipse_Lisp/HelloWorld/defpackage.lisp" (written 17 DEC 2011 09:18:53 PM):
; compiling (IN-PACKAGE :COMMON-LISP-USER)
; compiling (DEFPACKAGE :HELLOWORLD ...)

; C:/Users/admin/AppData/Local/common-lisp/cache/sbcl-1.0.54-win-
x86/D/Eclipse_Lisp/HelloWorld/ASDF-TMP-defpackage.fasl written
; compilation finished in 0:00:00.009
; compiling file "D:/Eclipse_Lisp/HelloWorld/main.lisp" (written 17 DEC 2011 09:18:53 PM):
; compiling (IN-PACKAGE :HELLOWORLD)

; C:/Users/admin/AppData/Local/common-lisp/cache/sbcl-1.0.54-win-
x86/D/Eclipse_Lisp/HelloWorld/ASDF-TMP-main.fasl written
; compilation finished in 0:00:00.008
; compiling file "D:/Eclipse_Lisp/HelloWorld/tests.lisp" (written 17 DEC 2011 09:18:53 PM):
; compiling (IN-PACKAGE :HELLOWORLD)

; C:/Users/admin/AppData/Local/common-lisp/cache/sbcl-1.0.54-win-
x86/D/Eclipse_Lisp/HelloWorld/ASDF-TMP-tests.fasl written
; compilation finished in 0:00:00.008
Loaded package D:/Eclipse_Lisp/HelloWorld/helloworld.asd
;Package changed to helloworld
```

Рисунок 3.9 — Сообщения об успешной компиляции

Шаг2. Редактирование исходного кода

Откройте в окне редактора файл **main.lisp** и введите после строки **(in-package :helloworld)** текст простейшей функции на языке Лисп, обеспечивающей вывод фразы "Здравствуй, мир":

```
; вывод фразы "Здравствуй, мир"
(defun hello ()
  "Здравствуй, мир ")
```

Сохраните файл, нажав **Ctrl+S** (или выбрав **Файл > Сохранить** или щелкнув правой кнопкой мыши на панели с открытым файлом и выбрав пункт меню **Сохранить**). Обратите внимание, что на панели **Схема** появилось имя введенной функции.

Шаг3. Компоновка проекта

По умолчанию компоновка (компиляция и объединение программных модулей) проекта выполняется автоматически при сохранении файла **main.lisp**. Об этом свидетельствуют сообщения, появившиеся в верхней части панели REPL:

```
; D:/s5a8..fasl written; compilation finished in 0:00:00.010.
```

Если вы внесете изменения в текст программы, то повторную компиляцию можно выполнять не только автоматически при сохранении файла (**Ctrl+S, рекомендуемый вариант для дальнейшей работы**), но и нажатием клавиш **Alt+C** (или из пункта меню **Lisp > Compile Top Level**) или **Alt+K** (или из пункта меню **Lisp > Compile File**). При этом повторная компиляция, выполняемая с помощью **Alt+C**, обеспечивает так называемую *инкрементальную компиляцию*, т.е. компиляцию только тех форм, в которые были внесены изменения в процессе редактирования. Повторная компиляция, выполняемая с помощью **Alt+K**, приводит к перекompilированию всего редактируемого файла.

Если при компиляции произойдет автоматический вызов отладчика (см. ниже), то чтобы выйти из режима отладки щелкните мышью в окне REPL и введите с клавиатуры литеру “**q**” (это соответствует команде **Quit debug** — выйти из отладки).

Шаг 4. Выполнение Lisp-приложения

Чтобы проверить работу функции **hello** введите в нижней части окна REPL (**hello**). Нажмите кнопку **Send** (или **Ctrl+Enter**). В верхней части окна REPL отобразятся результаты вызова функции (рис. 3.10).

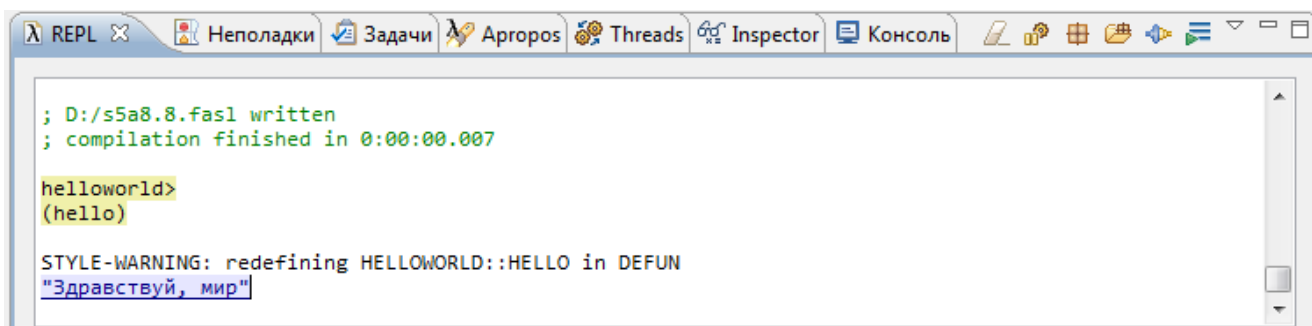


Рисунок 3.10 — Результаты вызова функции **hello**

Если вы завершите работу Eclipse, то при следующем сеансе работы с Eclipse проект следует сначала загрузить. Для этого щелкните правой кнопкой мыши на папке проекта в панели **Lisp navigator** и выберите **Load Project**. Это обеспечит необходимую компиляцию всех файлов вашего проекта и смену текущего пакета в окне REPL.

3.5. Файл определения пакета **defpackage.lisp**


Для контроля доступа к различным объектам программ в языке Коммон Лисп используются пакеты. Пакеты ограничивают область действия имен программных объектов (в Лиспе — символов) и позволяют при разработке программ применять модульное программирование.

В Коммон Лиспе определено несколько стандартных пакетов. Например [1]:

COMMON-LISP-USER — (сокращенное мнемоническое имя **CL-USER**) пакет, который становится текущим в момент запуска Лисп-системы;

COMMON-LISP — (сокращенное мнемоническое имя **CL**) содержит примитивы системы Коммон Лисп (за исключением ключевых слов), в нем определены такие внешние символы, как : **CAR**, **CDR**, ***PACKAGE*** и др;

KEYWORD — (мнемонического имени не имеет) содержит символы, представляющие ключевые слова.

В каждый момент времени только один пакет является текущим. Текущий пакет можно сменить. Например, чтобы сменить текущий пакет **helloworld** на пакет **CL-USER**, используемый по умолчанию, следует нажать на панели REPL кнопку  — **Change package** и выбрать в списке **CL-USER**, подтвердив **OK**.

Чтобы вызвать функцию из другого пакета следует перед её именем через “::” указать имя пакета, где она определена, например: **(helloworld::hello)**.

Имена объектов программы, содержащиеся в пакете, подразделяются на два класса: внешние и внутренние. *Внешние имена* представляют интерфейсную часть пакета и видимы из других пакетов. Имена становятся внешними, если они экспортируются за пределы пакета. *Внутренние имена* доступны внутри пакета и невидимы за его пределами.

Чтобы экспортировать имя (символ) за пределы пакета нужно в функцию, определяющую пакет, добавить экспортируемые символы. Функция **defpackage**, определяющая пакет **helloworld**, содержится в файле **defpackage.lisp**. Поэтому можно непосредственно в редакторе внести необходимые изменения в этот файл. Однако с целью демонстрации *навигационных возможностей* среды Cuspr выполним это иным способом (конечно для рассматриваемого простейшего примера программы эти действия излишни).

Щелкните мышью на панели **Схема** в строке **in-package :helloworld**, чтобы перейти на соответствующий участок исходного кода, который ссылается на определение пакета. Будет выделена строка **(in-package :helloworld)** в файле **main.lisp**. Нажмите клавишу **Ctrl** и щелкните мышью на слове **helloworld** (или нажмите **F3**) (рис. 3.11). В открывшемся окне **Definitions** необходимо выделить **(DEFPACKAGE :HELLOWORLD)** и подтвердить **OK**.

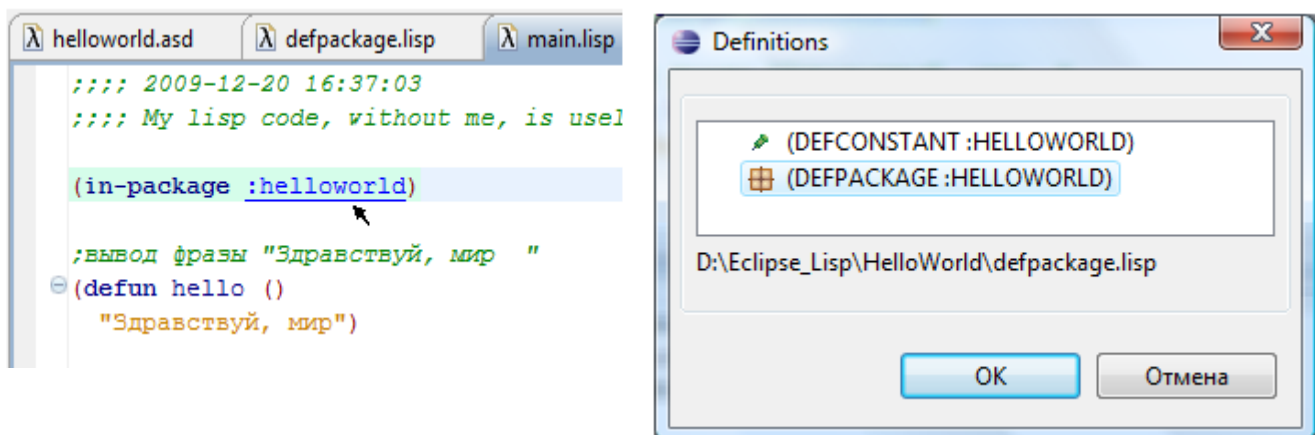


Рисунок 3.11 — Поиск определения пакета

В результате мы перейдем в файл **defpackage.lisp**. Этот файл содержит функцию **defpackage**, которая и определяет пакет **helloworld** с мнемоническим (nicknames) именем **helloworld**. Чтобы функция **hello** экспортировалась из пакета, добавьте по-

сле комментария “*;; Exported symbols go here*” (Экспортируемые символы следуют ниже) имя функции **hello**:

```
(defpackage :helloworld
  (:nicknames :helloworld)
  (:use :cl
    ;; Packages you want to import go here
  )
  (:export
    ;; Exported symbols go here
    hello
  ))
```

Сохраните файл **defpackage.lisp** (или нажмите **Ctrl+K**). Теперь функция **hello** экспортирована из пакета **helloworld** и обращаться к ней можно, записывая перед её именем через двоеточие префикс, представляющий имя пакета. Проверьте это, напечатав в нижней части окна REPL (**helloworld:hello**).

3.5. Создание исполняемого файла .exe

Система SBCL реализована в виде двух подсистем: подсистемы нижнего уровня, представляющей исполняющую среду, написанную на языке Си, и подсистемы верхнего уровня, которая написана на языке Коммон Лисп. Лисп-программы, создаваемые в Eclipse Cusp, исполняются в среде, которая обеспечивается программной системой SBCL. Поэтому при компиляции Лисп-программ они транслируются в некоторый внутренний код (двоичные файлы в формате **fasl**), который затем загружается и исполняется Лисп-средой (своего рода виртуальной машиной). Поэтому обычно создание исполняемых файлов для конкретной платформы не требуется.

Тем не менее, Cusp позволяет создавать отдельные исполняемые файлы. Для этого исходный код на языке Лисп должен содержать функцию (в общем случае форму) верхнего уровня. Эта функция не должна иметь параметров и должна возвращать в качестве результата 0. Данная функция будет первой получать управление при выполнении исполняемого кода.

Внесем следующие изменения в код функции **hello**:

```
(defun hello ()
  (princ "Hello, world")
  0 )
```

Здесь вызов функции **princ** обеспечивает вывод фразы "Hello, world". Выбор английского языка объясняется проблемами с выводом букв русского языка в консольном окне Windows. Ноль в конце — значение, возвращаемое функцией.

Для создания исполняемого файла перейдите на панель Lisp navigator, затем щелкните правой кнопкой мыши на папке проекта или любом файле проекта и выберите **Create exe**. В результате откроется окно, изображенное на рис.3.12. В поле **Top level form** (форма верхнего уровня) введите имя функции **hello** и подтвердите **Готово**. Просмотрите в окне **Консоль** протокол создания .exe файла.

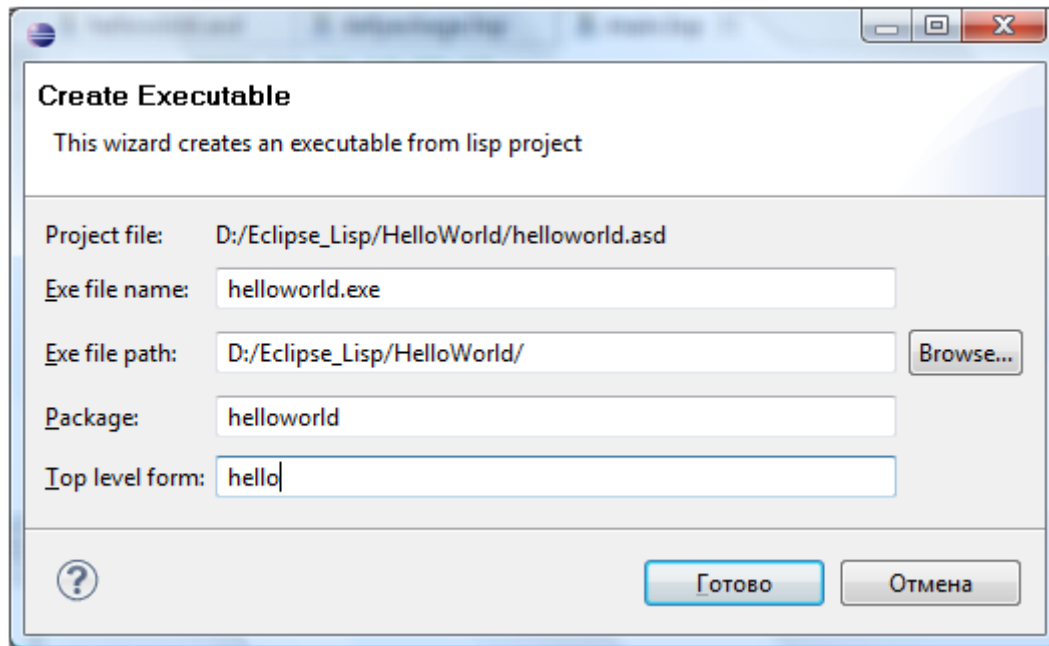


Рисунок 3.12 — Создание исполняемого файла

Используя командную строку Windows, запустите файл **helloworld.exe**. Обратите внимание, что размер файла **helloworld.exe** примерно равен 27Мбайт. Такой большой размер исполняемого файла объясняется тем, что он содержит Лисп-среду, включая компилятор и отладчик.

Менеджер проектов **Cusp** (см. рис. 3.8) содержит пример, обеспечивающий создание исполняемого файла, которому при вызове можно передавать аргументы из командной строки. Для создания проекта-примера сохраните отметку в поле **Generate code for an example function** (см. рис. 3.8). Изучите самостоятельно указанный пример проекта.

3.7. Сообщения компилятора и отладка программы

3.7.1. Диагностические сообщения компилятора

Диагностические сообщения компилятора подразделяются на четыре группы:

- 1) ошибки (errors);
- 2) предупреждения (warnings);
- 3) предупреждения стиля (style warnings);
- 4) замечания (note).

Краткое содержание диагностических сообщений отображается в окне **Неполадки**, а в окне редактора, слева от соответствующих строк исходного кода, появляются специальные метки. На рис. 3.13 представлен пример сообщения с ошибками компиляции. В данном случае в исходный код была внесена лишняя скобка после слова **princ**. При сохранении файла Cusp предпринял попытку вы-

полнить компиляцию и вывел диагностические сообщения в окне **Неполадки**. Обратите внимание, что компилятор диагностировал эту ошибку не в строке 8, а в строке 9 и выдал сообщение об отсутствии пары для закрывающейся скобки. Примеры меток предупреждений можно увидеть на рис. 3.4.

Полные диагностические сообщения компилятора могут выводиться в панели REPL при попытке загрузки проекта с ошибкой компиляции. Обычно они весьма объемные. При этом происходит автоматический вызов отладчика, который также выводит свои сообщения в панель REPL. Разобраться в этих сообщениях без определенного опыта сложно. Поэтому рекомендуется на первых этапах пользоваться информацией размещаемой в панели **Неполадки**, а работу автоматически вызываемого отладчика прерывать вводом литеры “q” (**Quit debug**).

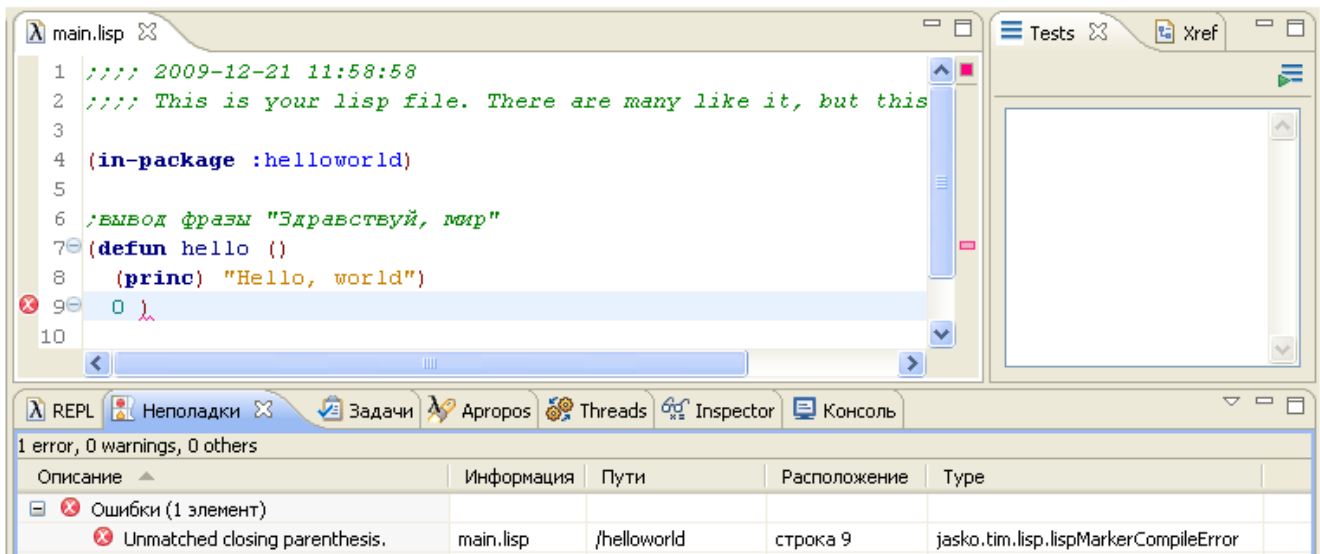


Рисунок 3.13 — Диагностические сообщения компилятора

3.7.2. Отладка программы

Отладка программы — это процесс выполнения программы с целью её проверки и выявления ошибок. Отладку выполняют с помощью отладчика, который может быть вызван различными способами:

- автоматически при обнаружении необрабатываемой ошибки;
- специально путем установки точек прерываний;
- непосредственно путем вызова функции `invoke-debugger`.

Рассмотрим случай *автоматического вызова отладчика* при обнаружении ошибки. Создайте новый проект, например `debug_example1`, и введите в файл `main.lisp` следующий код:

```
(defun print_power_2 (x)
  (print (power_2 x)))

(defun power_2 (x)
  (square x)
)
```

В данном примере определена функция `print_power_2`, которая печатает квадрат значения своего входного параметра “`x`”. Для вычисления квадрата “`x`” вызывается функция `power_2`, которая, в свою очередь, обращается к функции `square`, чтобы возвести “`x`” во вторую степень. Выполните компиляцию проекта, а затем в нижней части окна REPL введите s-выражение `(print_power_2 5)` и нажмите **Send**.

Из-за ошибок в программе будет автоматически вызван отладчик, сообщения которого будут выведены в панели REPL (рис. 3.14). Сообщения отладчика отображаются в трех условных областях, обозначенных на рис.3.14 прямоугольниками.

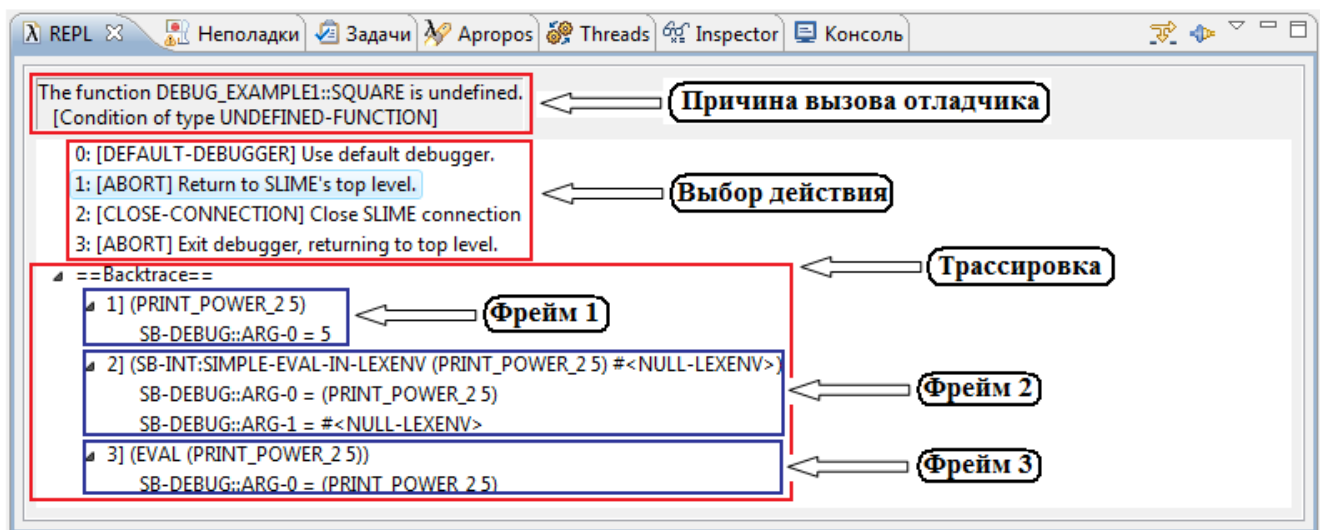


Рисунок 3.14 — Сообщения отладчика

В верхней области отображается сообщение ошибки, указывающее основную причину останова программы и вызова отладчика. В данном случае указывается, что функция `SQUARE` не определена.

В середине отображается область, в которой указываются возможные действия для продолжения работы. Пользователь может выбрать одно из предлагаемых действий, введя номер соответствующей строки или выполнив двойной щелчок мышью на требуемой строке. Действие, предлагаемое по умолчанию, выделяется цветом. Чтобы его подтвердить, достаточно нажать `Enter`. В рассматриваемом случае предлагается выбрать `1:[ABORT]` (или нажать “`q`”), чтобы прервать работу отладчика и вернуться в REPL.

В нижней области (рис. 3.14) отображается трассировка вызовов функций (`backtrace`). С помощью трассировки можно выяснить, какие вычисления выполняла Лисп-система до того, как произошла ошибка. Трассировка отображается в виде стека фреймов (кадров, блоков). Фрейм содержит информацию, которую Лисп-система сохраняет в стеке перед вызовом функции. Поэтому в области трассировки отображается столько фреймов, сколько было вызовов. Фреймы автоматически нумеруются. На рис. 3.14 изображена трассировка вызова функции

(`print_power_2 5`). В верхушке стека фреймов находится последний вызов, который собственно и привел к ошибке.

Отладчик позволяет выяснять свойства объектов фреймов. Для этого следует выделить интересующий объект фрейма и нажать **Enter** (либо дважды щелкнуть мышью). Свойства объекта будут представлены на панели **Inspector**.

Существует два основных варианта *специального вызова отладчика* с использованием точек прерываний, которые вызывают остановку программы и выводят сообщения о её состоянии.

Первый вариант (стандартный) заключается в вызове функции (`break` сообщение). Вызов этой функции следует разместить в том месте исходного кода, где требуется выполнить остановку программы. После перекомпиляции программы указанный вызов будет приводить к вызову отладчика в заданной точке программы.

Вторая возможность предоставляется средой Cusp. Выделите s-выражение, где требуется остановить программу. Для этого щелкните мышью перед скобкой, начинающей выражение, и нажмите **Shift+Enter**. Будет выполнено так называемое *расширенное выделение*, которое также можно выполнить обычным образом с помощью мыши. Затем установите точку прерывания, выбрав **Lisp > Toggle Breakpoint** или нажав **Alt+B**. Cusp создаст из выделенного s-выражения вызов макроса точки прерывания и отметит строку исходного кода, содержащего точку прерывания, меткой в виде литеры “i” (рис. 3.15). Если понадобится удалить эту точку прерывания, выполните повторно расширенное выделение выражения и нажмите **Alt+B**. Поведение такой точки прерывания аналогично вызову функции `break`, т.е. при её достижении автоматически вызывается отладчик. А далее управлять выполнением программы можно, используя команды отладчика.

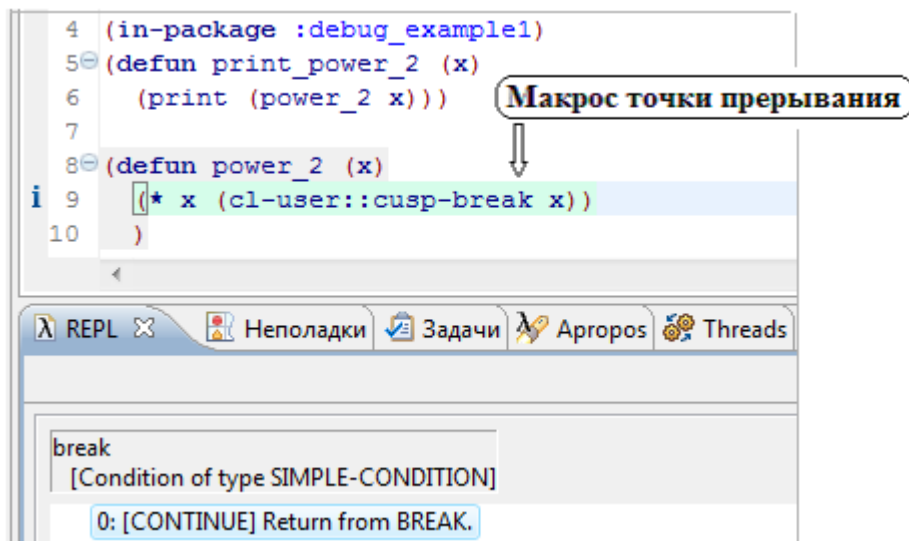


Рисунок 3.15 — Создание точки прерывания с помощью Cusp

Рассмотрим использование некоторых команд отладчика на примере. Пусть требуется вывести квадраты целых чисел в диапазоне от 0 до 9. Для этого переопределим функцию `power_2`, добавив в неё итерационный цикл `loop`:

```


(defun power_2 (x)                                ;заголовок функции
  (loop                                           ;цикл
    (if (>= x 10) (return "Выход"))              ;условие завершения цикла
    (print x)                                     ;печать x в новой строке
    (prin1 ( * x x))                             ;печать x*x
    (setq x (+ x 1))                             ;увеличить x на 1, т.е. x=x+1
  )
)

```

Здесь **loop** обозначает цикл, **if** — форма, соответствующая условному оператору, **print** и **prin1** — функции печати, вызов функции **setq** присваивает “**x**” новое значение, увеличенное на 1.

Создайте новый проект и внесите код функции **power_2** в файл **main.lisp**. Сохраните файл (**Ctrl+S**). В нижней части окна REPL введите выражение **(power_2 0)** и нажмите **Send**. В результате будут вычислены значения квадратов целых чисел в диапазоне от 0 до 9.

Проверим работу программы по шагам. Создадим точку прерывания с помощью **Cusp**: **(cl-user::cusp-break (print x))**. Сохраним файл, нажав **Ctrl+S**, а затем в окне REPL введем выражение **(power_2 0)**. Теперь при достижении установленной точки прерывания программа будет остановлена и управление будет передано отладчику (рис 3.16).

Далее можно перевести отладчик в *пошаговый режим* (в ходе которого можно вычислять поочередно s-выражения), нажав кнопку  — **Step** (рис. 3.16). При этом имеется возможность выбора команд пошаговой работы отладчика (рис. 3.17): **step-next** — шаг вперед без захода внутрь вызова очередной функции; **step-into** — шаг вперед с заходом внутрь вызова очередной функции; **step-out** — выход из текущей функции; **step-continue** — возобновить нормальное выполнение.

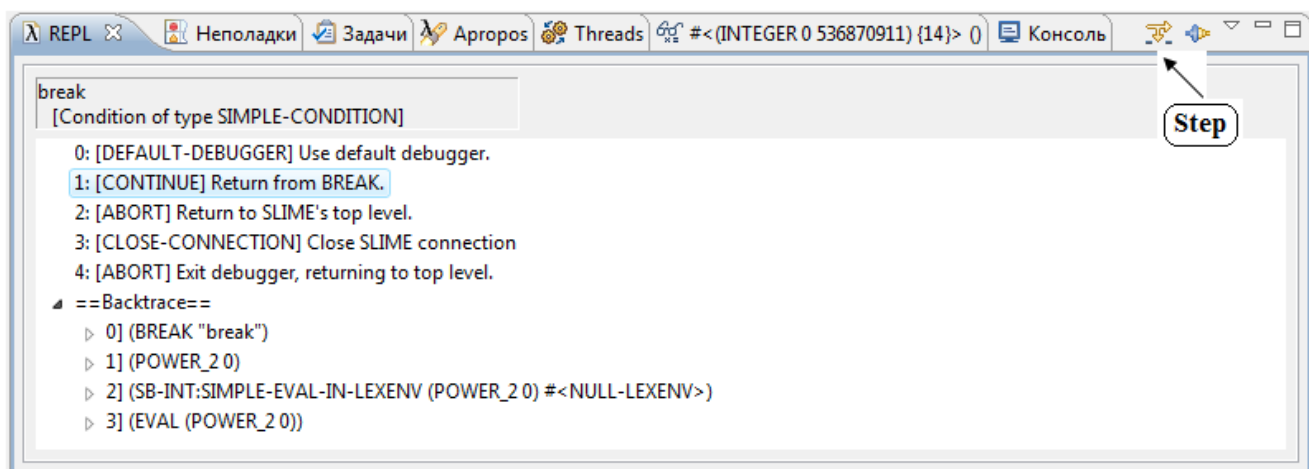


Рисунок 3.16 — Остановка программы в точке прерывания

Поскольку в этом случае в окне REPL при каждом нажатии кнопки **Step** отображаются в основном команды отладчика для текущего шага выполнения и

не видны результаты вычислений текущего и предыдущих шагов, то целесообразно направить вывод этих сведений на консоль. Для этого следует открыть окно настроек параметров Lisp проекции, выбрав **Окно > Параметры > Lisp** и перейти на страницу **Implementations**. Установить флажок напротив строки **Write Compiler Log To Console**. Теперь результаты выполнения каждого шага отладки будут доступны для просмотра в окне консоли.

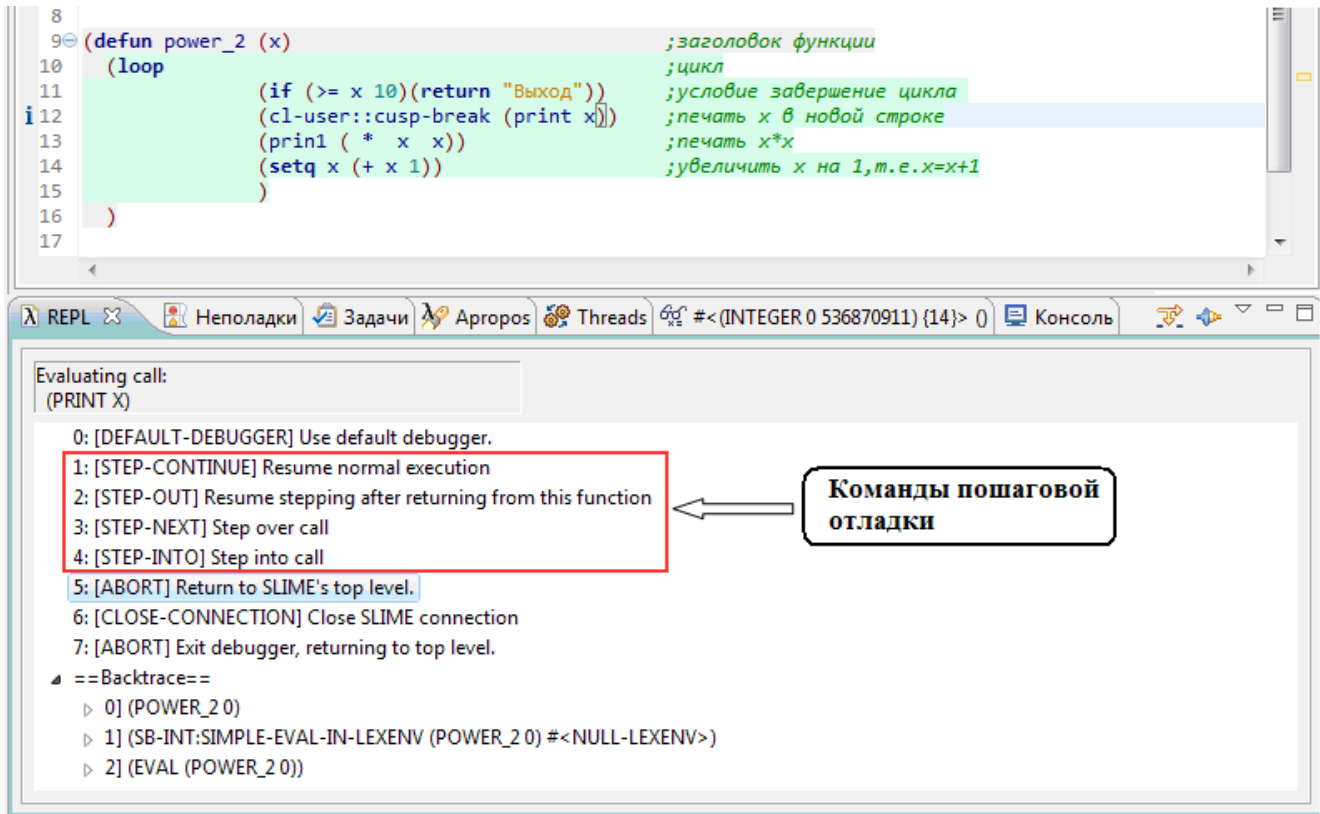



Рисунок 3.17 — Команды пошаговой отладки

Для завершения работы отладчика следует выбрать команду **[Abort] Return to Slime's top level** или ввести литеру "q" — команда выхода из отладчика (**Quit Debug**).

Если требуется прервать выполнение программы, вошедшей в бесконечный цикл, то следует нажать на панели REPL кнопку  — **Interrupt execution** (возможно несколько раз). Если программу не удаётся остановить таким способом, то воспользуйтесь **Диспетчером задач Windows**.

Если вам требуется проследить выполнение некоторой функции без остановки программы, то можно использовать макрос трассировки **trace** [1]. Кроме этого, Cusp предлагает собственный макрос для отслеживания значений выражений — **cusp-watch**, который создает *точку наблюдения*, маркируемую в исходном коде меткой "i". Для создания такой точки следует выполнить расширенное выделение требуемого выражения (**Shift+Enter**), а затем нажать сочетание клавиш **Alt+T** (или выбрать **Lisp > Toggle Watch**). Чтобы удалить точку наблюдения необходимо опять выделить выражение и повторно нажать **Alt+T**. Создадим точку наблюдения для выражения `(+ x 1)` (рис. 3.18), и после компиляции в нижней части окна REPL введем выражение `(power_2 0)`.

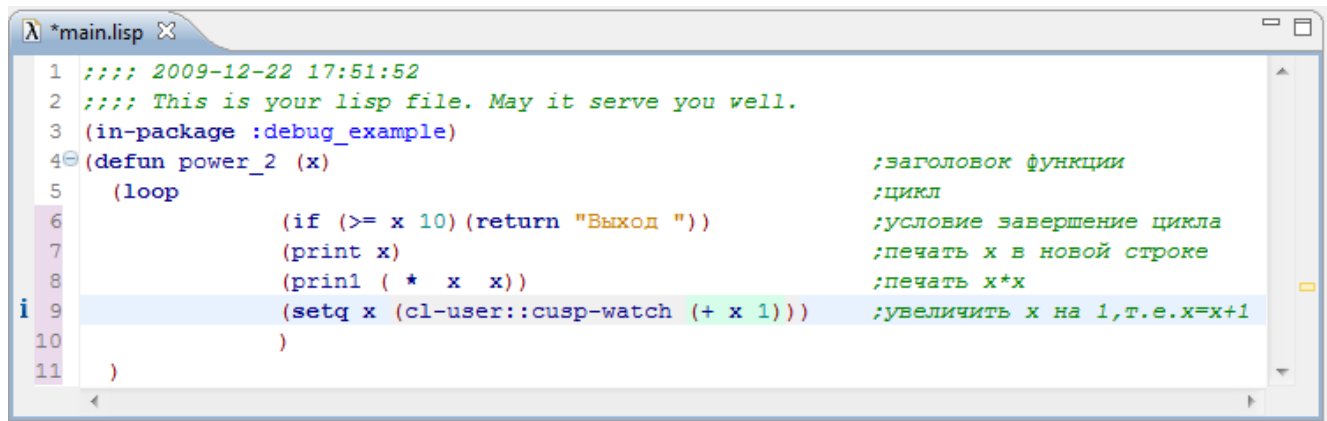


Рисунок 3.18 — Создание точки наблюдения

В результате в верхней части окна REPL получим следующий протокол:

```

debug_example> (power_2 0)
0 0(+ X 1) -> 1
1 1(+ X 1) -> 2
2 4(+ X 1) -> 3
3 9(+ X 1) -> 4
4 16(+ X 1) -> 5
5 25(+ X 1) -> 6
6 36(+ X 1) -> 7
7 49(+ X 1) -> 8
8 64(+ X 1) -> 9
9 81(+ X 1) -> 10

```

Как следует из этого протокола, точка наблюдения обеспечивает вывод, как самого выражения, так и его значений.

Следует помнить, что отладчик работает с двоичным кодом и поэтому доступная ему информация зависит от режимов оптимизации, используемых при компиляции. Чем выше требования к оптимизации программы при её компиляции, тем меньше дополнительных сведений сохраняется в двоичном коде, которые могут использоваться отладчиком. К сожалению, задаваемые режимы оптимизации при компиляции программ из среды Cusp плохо документированы. Поэтому в некоторых ситуациях (в частности, при выполнении компиляции с помощью ALT+K или в процессе загрузки проекта) не все команды отладчика будут доступны. В общем случае требования к режимам оптимизации можно изменить с помощью декларации `optimize` (см. документацию к SBCL <http://www.sbcl.org/platform-table.html>).

3.8. Свойства редактора Cusp

Лисп редактор, используемый в среде Eclipse Cusp, наряду со стандартными функциями, свойственными многим современным текстовым редакторам, предоставляет следующие дополнительные возможности:

- обеспечивает «подсветку» (выделение цветом) синтаксически различных участков исходного кода;
- выделяет метками строки кода с ошибками;

- может сворачивать некоторые участки кода для улучшения его обозримости;
- имеет режим всплывающих быстрых подсказок, а также предоставляет возможность автозавершения имен вводимых символов языка Лисп.

Возможности редактора, связанные с подсветкой участков кода и с выделением ошибок метками, демонстрировались ранее на рис. 3.4 и рис. 3.13, соответственно.

С целью улучшения обозримости текста программы редактор позволяет *сворачивать* некоторые участки исходного кода. Напротив участков кода, которые можно сворачивать либо разворачивать, видны специальные метки в виде кружочков со знаками плюс или минус (рис.3.19).

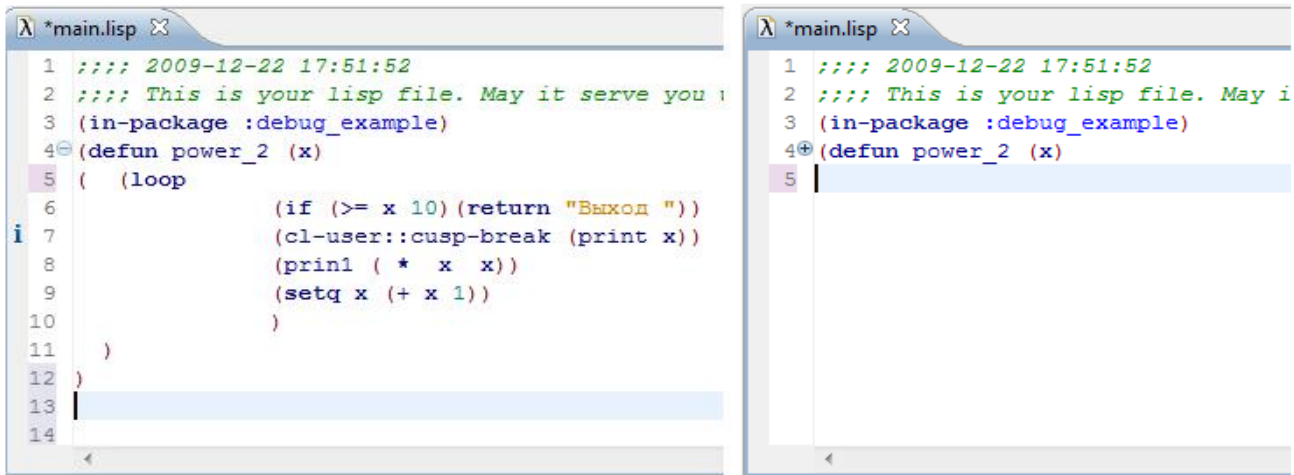


Рисунок 3.19 — Пример сворачивания исходного кода

Для того чтобы свернуть участок кода, отмеченный знаком минус следует щелкнуть мышью на этом знаке. И наоборот, чтобы его развернуть, следует щелкнуть мышью на знаке плюс.

При “зависании” указателя мыши над символом программы редактор по умолчанию отображает *быструю подсказку* (рис. 3.20)

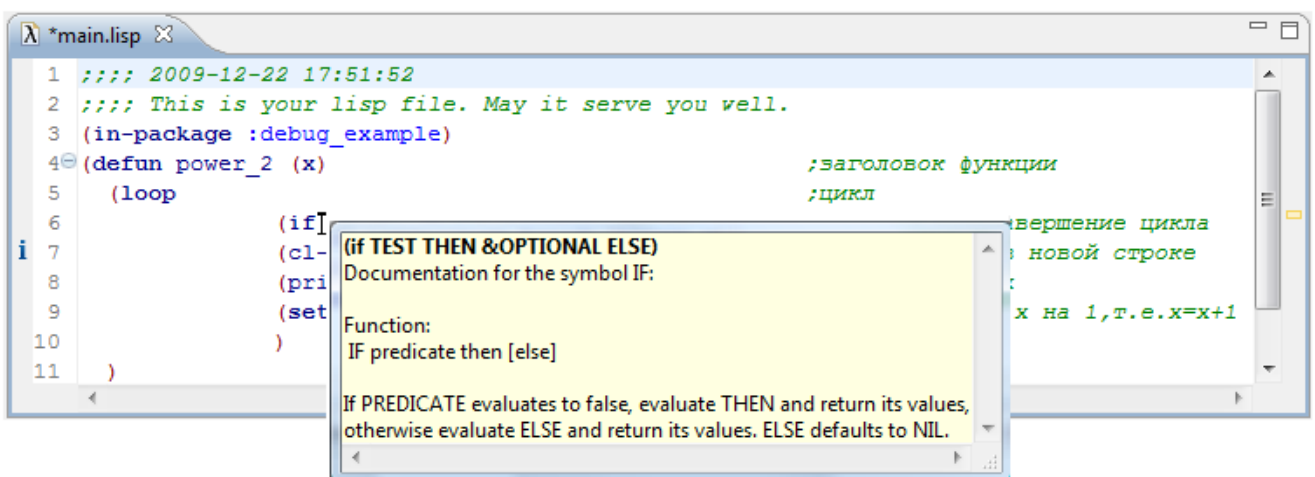


Рисунок 3.20 — Всплывающая подсказка

Возможность *автозавершения* вводимых имен базируется на том, что редактор выполняет ассоциативный поиск символов языка Лисп, имена которых начинаются с литер, введенных пользователем. Например, напечатав литеры “do”,

получим всплывающий список символов, которые начинаются этими литерами (рис.3.20), с поясняющим текстом быстрой подсказки в отдельном окне. Если выбрать символ из появившегося списка двойным щелчком мыши, то он будет вставлен в программу, т.е. можно выполнить автозавершение ввода имени символа, введя всего несколько литер.

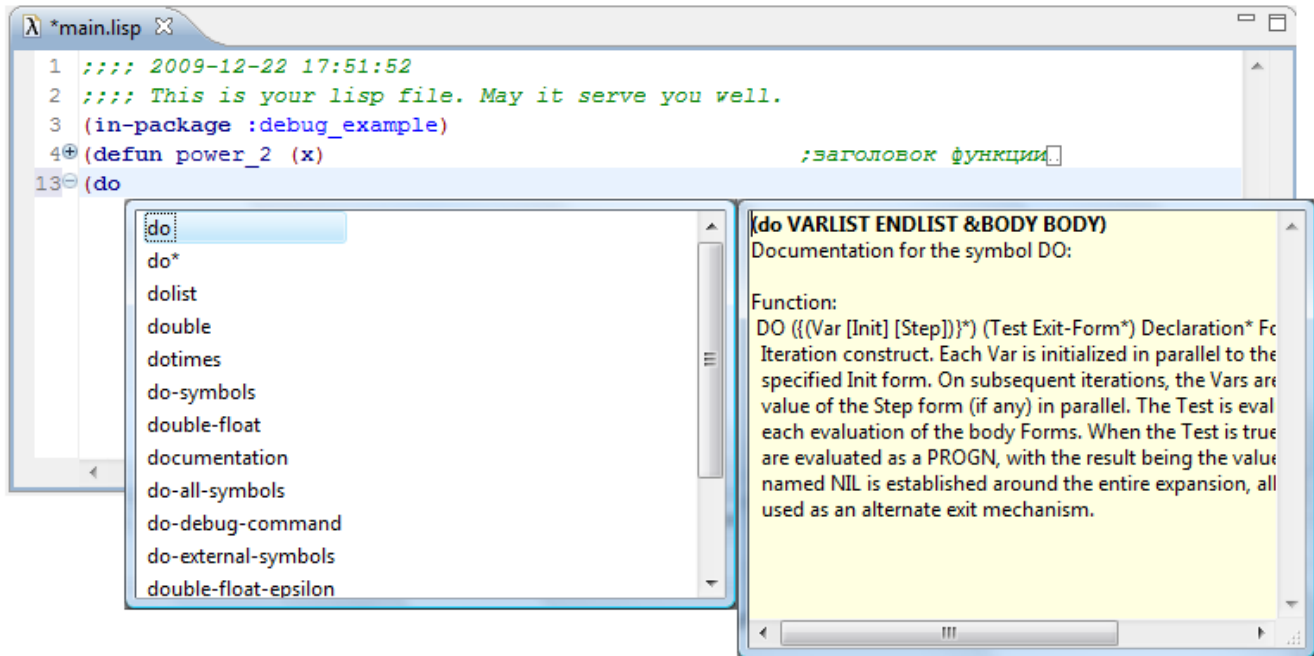


Рисунок 3.21 — Список автозавершения и текст быстрой подсказки

При необходимости автоматическое отображение списка автозавершения можно запретить, установив соответствующие свойства при настройке редактора (см. ниже). В таком случае этот список будет отображаться только при нажатии клавиш **Ctrl+Space**, когда он действительно необходим.

Свойства редактора устанавливаются на странице редактора в окне параметров (рис. 3.22): **Окно > Параметры > Lisp > Editing**.

На этой странице можно установить либо отменить следующие свойства редактора:

- автоматическое добавление закрывающейся круглой скобки при вводе открывающейся скобки (**Automatically close '('...**);
- автоматическое ограничение следующего за позицией курсора символа круглыми скобками при вводе знака квадратной скобки "[" (**'['encloses next sexp...**);
- автоматическое закрытие парных двойных апострофов (**Automatically close double quotes**);
- автоматическое закрытие комментария, начинающегося литерами "#|" (**Automatically close '#|'**);
- автоматическое отображение списка автозавершения имен символов или подсказки параметров (**Automatically show popup...**) (см. рис. 3.21);
- отображать подсказку ниже курсора (**Show documentation hints below...**);
- автоматически добавлять символ автозавершения, если он единственный в списке (**Automatically insert ...**);

- использовать нечеткий режим автозавершения (**Use fuzzy mode ...**);
- отображать совместно со списком автозавершения быструю подсказку (**Show quick documentation...**) (см. рис. 3.21).

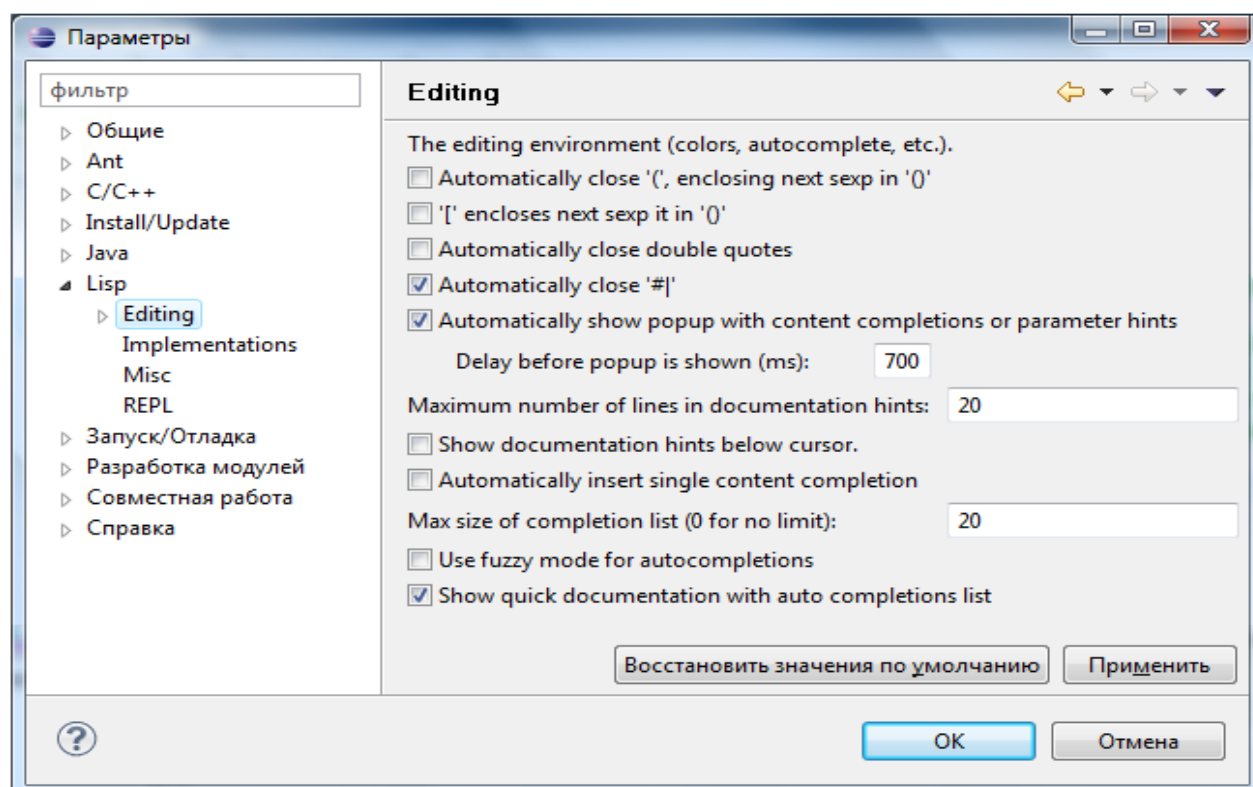


Рисунок 3.22 — Страница настроек редактора Лисп

Назначение каждого из упомянутых свойств не требует особых пояснений, за исключением нечеткого режима автозавершения. В этом режиме редактор выполняет поиск составных имен символов по первым литерам слов, образующих такие имена. Например, при вводе в окне редактора или в нижней части окна REPL последовательности символов (i-t-u) откроется список автозавершения, изображенный на рис. 3.23.

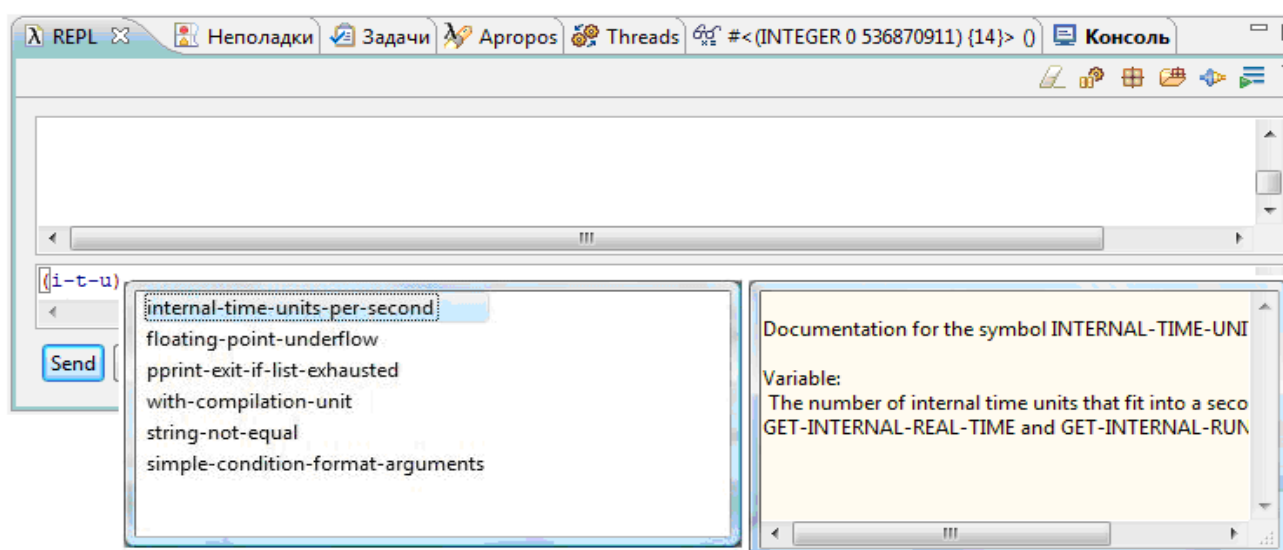


Рисунок 3.23 — Нечеткий режим автозавершения

В дополнение к сказанному редактор позволяет составлять список задач и разделять исходный код на секции, что весьма удобно в случае разработки объемных программ. Чтобы добавить задачу в список задач, вставьте в текст программы комментарий, который начинается с последовательности литер “**;TODO:**”. На панели задач отобразится добавленная задача (рис. 3.24).

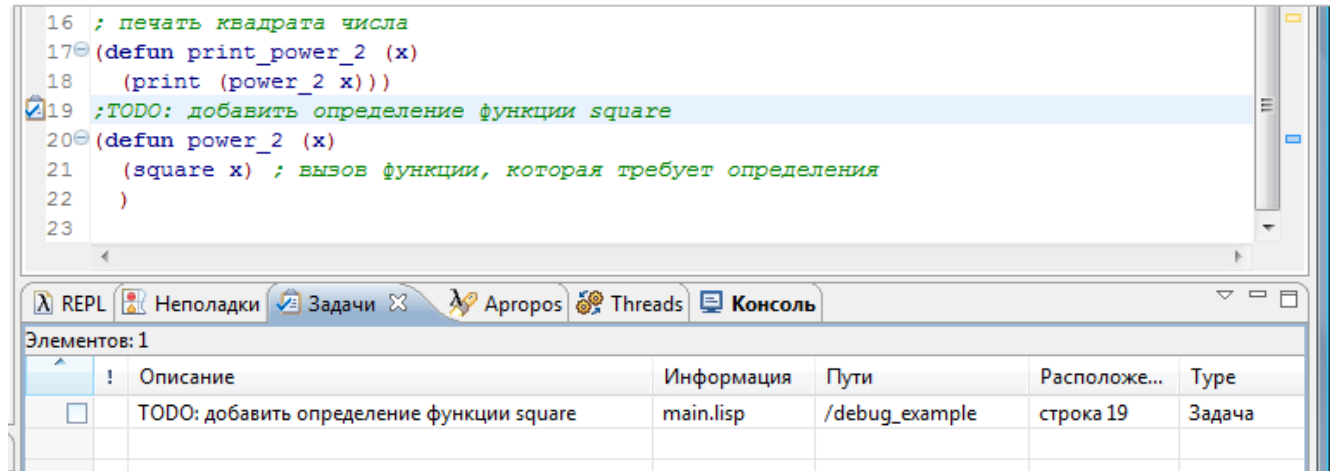


Рисунок 3.24 — Добавление задачи

Для разделения исходного кода на секции вставьте комментарий, начинающийся с литер “**;;;<**”. Например:

;;;<1. Основная функция поиска в ширину

Введенные таким образом секции можно сворачивать в окне редактора. Они также отображаются на панели **Схема**, позволяя быстро перемещаться между секциями исходного кода (рис. 3.25).

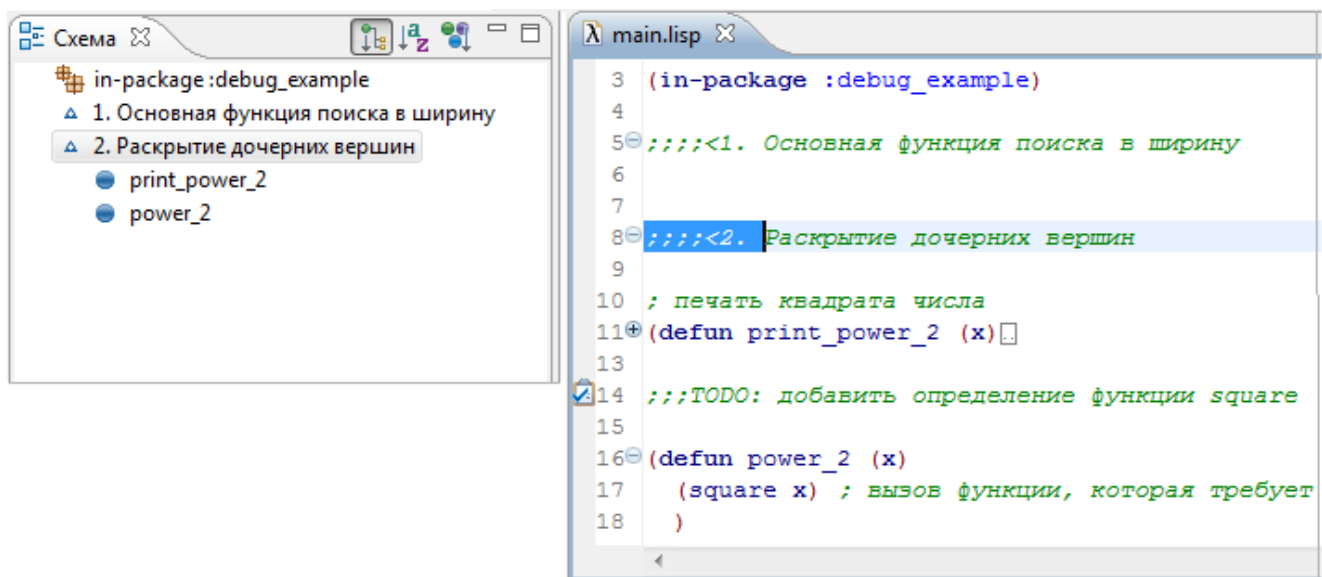


Рисунок 3.25 — Разделение исходного кода на секции

3.9. Добавление файлов в проект

Чтобы добавить файл в проект, необходимо щелкнуть правой кнопкой мыши на имени проекта в панели Lisp Navigator и выбрать **Создать > Lisp File**. Откроется окно диалога, в котором следует ввести имя файла.

Новый файл можно также скопировать в папку проекта и затем щелкнуть правой кнопкой мыши на имени проекта в панели Lisp Navigator и выбрать **Обновить**.

Затем следует добавить в начало нового файла выражение:

```
(in-package :имя_проекта)
```

Также требуется отредактировать файл проекта **имя_проекта.asd**, который используется при компиляции и загрузке проекта (**asdf** — a system definition facility). Для этого добавьте следующее выражение (`:file "имя_нового_файла" :depends-on ("defpackage")`) в раздел `:components`. Например, для проекта `helloworld` файл `helloworld.asd` следует отредактировать следующим образом:

```
(defpackage #:helloworld-asd
  (:use :cl :asdf))

(in-package :helloworld-asd)

(defsystem helloworld
  :name "helloworld"
  :version "0.1"
  :serial t
  :components ((:file "defpackage")
                (:file "main" :depends-on ("defpackage"))
                (:file "имя_нового_файла" :depends-on ("defpackage")))
  :depends-on ())
```

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Бондарев В.Н. Искусственный интеллект: учеб. пособие / В.Н. Бондарев, Ф.Г. Аде.— Севастополь : Изд-во СевНТУ, 2002. — 615с.
2. Введение в интегрированную среду разработки Eclipse CDT: методические указания к лабораторным работам по дисциплине “Основы программирования и алгоритмические языки” для студ. дневной и заочной формы обучения направления 09.03.02 — “Информационные системы и технологии”/ СевГУ; сост. В. Н. Бондарев, Т.И. Сметанина. — Севастополь: Изд-во СевГУ, 2015. — 40 с.
3. Казарин С.А. Среда разработки Java-приложений Eclipse (ПО для объектно-ориентированного программирования и разработки приложений на языке Java): учеб. пособие [Электронный ресурс] / С.А.Казарин, А.П.Клишин. — М.: Федеральное агентство по образованию, 2008. — 77 с. — Режим доступа: http://window.edu.ru/window_catalog/files/r58397/Eclipse_Java.pdf. — Последний доступ: 19.12.2011. — Название с экрана.
4. Kolos Sergey Lisp Programming with Cusp [Электронный ресурс] / Sergey Kolos. — Режим доступа: <http://www.sergeykolos.com/cusp/intro> . — Последний доступ: 19.12.2011. — Название с экрана.
5. Rahimberdiev Askar Проект Eclipse [Электронный ресурс] / Askar Rahimberdiev // RSDN Magazine .— №4.— 2004. — Режим доступа: <http://www.rsdn.ru/?article/devtools/eclipse.xml>. — Последний доступ: 19.12.2011. — Название с экрана.

ПРИЛОЖЕНИЕ А

(справочное)


Использование библиотек

Для установки библиотек, если они не были установлены ранее, необходимо выполнить следующее:

- загрузите архив [cusp0.9.390.zip](http://www.sergeykolos.com/cusp/archive/), который находится по адресу <http://www.sergeykolos.com/cusp/archive/>), и разархивируйте его во временную папку;
- скопируйте из папки **features** папку **jasko.tim.lisp.libs_1.1.1** в папку **c:\Eclipse\features**;
- скопируйте из папки **plugins** папку **jasko.tim.lisp.libs_1.1.1** в папку **c:\Eclipse\plugins**.

Аналогичным образом можно устанавливать и другие библиотеки, поставляемые отдельно. Например, можно загружать библиотеки с сайта <http://www.cliki.net>.

По умолчанию плагин Cusp хранит библиотеки в папке **eclipse/plugins/jasko.tim.lisp.libs_1.1.1/libs**. Если библиотека размещается в другой папке, то в окне параметров (**Окно > Параметры > Lisp**) на странице **Implementations** в поле **Path to libraries** необходимо указать путь к библиотеке. После чего следует перезапустить Eclipse.

Чтобы проверить работу одной из установленных библиотек, нажмите на панели **REPL** кнопку  — **Load Installed Packages**. Откроется окно, изображенное на рис. А.1. В этом окне можно просмотреть список доступных для загрузки библиотек и их назначение. Более подробное описание библиотек протестированных в Cusp приведено в [4].

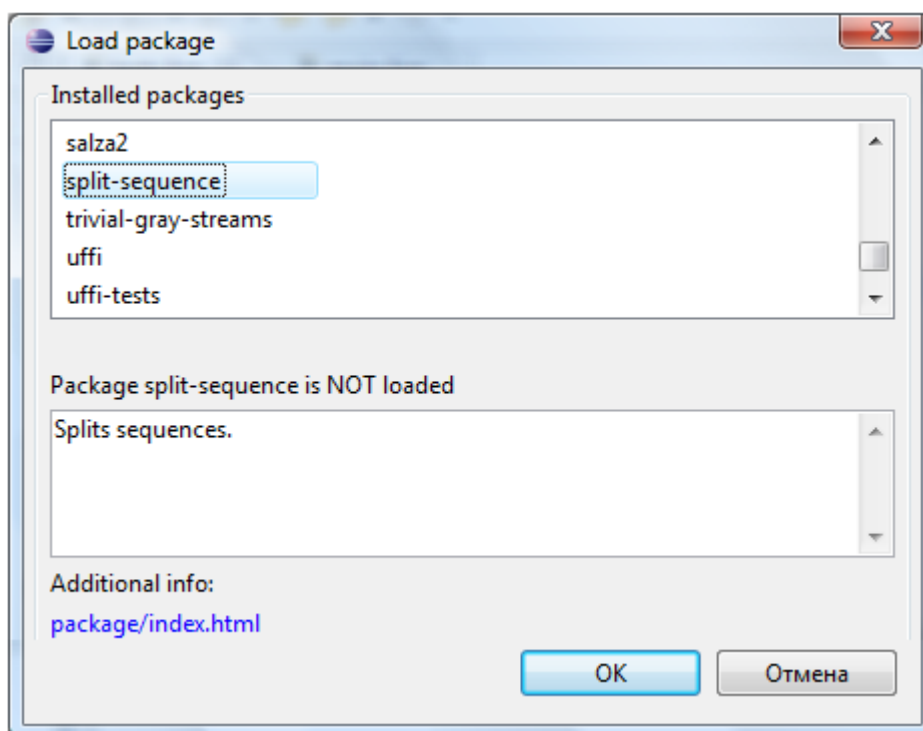


Рисунок А.1 — Окно загрузки пакетов

Выберите в списке **split-sequence** и подтвердите **ОК**. В результате будет загружен и скомпилирован пакет **split-sequence** для работы с последовательностями. Это можно увидеть, просмотрев список доступных пакетов (кнопка **Change Package**). Чтобы убедиться в работоспособности функций пакета, введите в нижней части окна REPL s-выражение: `(split-sequence::sort '(2 9 3 1 8 4 7 5 6) ' >)`. В результате получим упорядоченную последовательность: `(9 8 7 6 5 4 3 2 1)`.

После того как библиотека установлена, можно использовать функции, которые она экспортирует в своих программах. Для этого следует:

- 1) добавить имя библиотеки в раздел `:depends-on` файла проекта (`.asd`):
`:depends-on (:split-sequence)`
- 2) добавить имя библиотеки в раздел `:use` файла `defpackage.lisp`:
`(:use :cl :split-sequence)`


В дальнейшем, при повторном сеансе работы с Eclipse, не требуется загрузка библиотеки с помощью кнопки **Load Installed Packages**, так как она будет загружаться автоматически вместе с проектом.

ПРИЛОЖЕНИЕ Б

(справочное)

Тестирование

Среда **Cusp** содержит встроенные средства для тестирования, которые обеспечивают интерфейс для взаимодействия с пакетом **lisp-unit**. Этот пакет загружается по умолчанию (см. **Окно > Параметры > Lisp > Implementation** свойство **Use LispUnit integrated with Cusp**).

Для проверки интерфейса тестирования создайте проект **new-lisp1** с настройками, предлагаемыми по умолчанию. После компиляции и загрузки проекта нажмите на панели REPL кнопку  — **Run tests**. Откроется окно с загруженными пакетами, для которых определены файлы тестов (рис Б.1). Подтвердите выбор проекта, нажав ОК. В результате будет выполнен тест, результаты которого выводятся в панели **Tests** (рис Б.1).

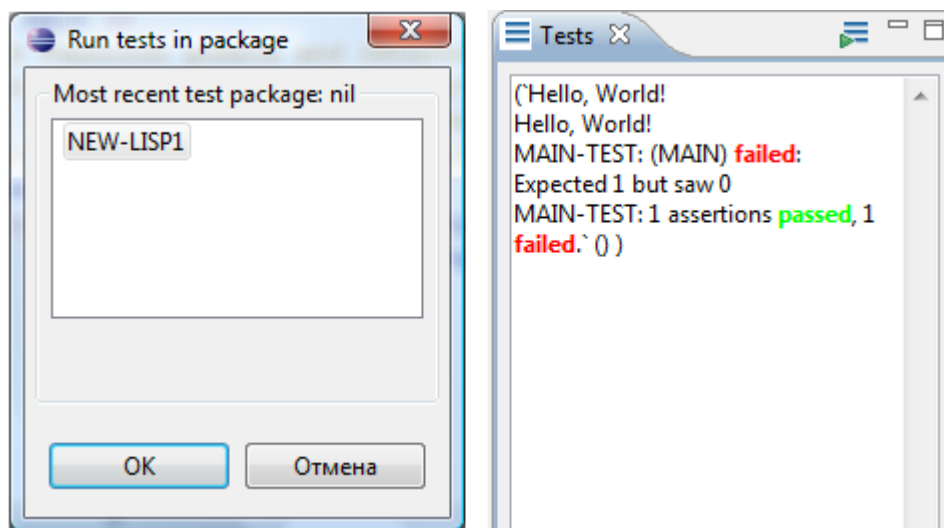


Рисунок Б.1 — Выполнение тестирования

ПРИЛОЖЕНИЕ В

(справочное)

Пример проекта

Рассмотрим пример проекта **depth**, реализующего поиск пути на графе состояний методом в глубину. Полное описание программы приведено в [1].

В.1. Файл main.lisp

```

;;; 2009-12-17 19:35:34
;;; Behold the power of Lisp! (Почувствуй силу Лиспа!)

;;; ПРИМЕР РЕШЕНИЯ ЗАДАЧИ ПОИСКА В ГЛУБИНУ НА ГРАФЕ СОСТОЯНИЙ

(in-package :depth)

;;; <1. Определение графа состояний
(setq graph '( (a b) (a e)           ; связи вершины "a"
               (b a) (b f) (b c)     ; связи вершины "b"
               (c b) (c f)           ; связи вершины "c"
               (d e) (d g)           ; связи вершины "d"
               (e a) (e d) (e f)     ; связи вершины "e"
               (f b) (f c) (f e) (f h) ; связи вершины "f"
               (g d) (g h)           ; связи вершины "g"
               (h g) (h f)))         ; связи вершины "h"

;;; <2. Функция поиска в глубину
(defun depth-first(start goal)
  ; поместить под список (start start) в open
  (setq open (list (list start start)))
  ; инициализировать список closed
  (setq closed nil)
  (when
    (loop
      (cond ((null open) (return nil)) ; цикл
            (t
             ; на графе нет пути
             ;; запомнить первый подсписок списка open
             (setq n (first open))
             ;; внести n в начало closed
             (setq closed (cons n closed))
             ;; проверить, является ли (first n) целевой вершиной
             (if (eq (first n) goal) (return t)) ; на графе есть путь
             ;; добавить в начало open дочерние вершины
             (setq open (append (list-children (first n)) (rest open)))
             ;; печать промежуточных результатов
             (terpri)
             (princ "closed=") (prin1 closed)
             (terpri)
             (princ "open=") (prin1 open)
             )
      )
    )
  ; вызов функции, восстанавливающей по списку closed искомый путь
  (back-way goal start)
)

;;; <3. Функция, возвращающая дочерние вершины
(defun list-children(n)
  (setq L nil)

```

```

(dolist (temp graph L) ;выполнить действия для каждого подсписка из graph
  ;; инвертировать подсписок temp
  (setq rtemp (rev temp))
  ;; проверяем вершину "n" и её дочернюю вершину
  (if
    (and
      (eq n (first temp)) ; выясняем, содержит ли temp вершину "n"
      ;; проверяем, входит ли очередная дочерняя вершина
      ;; в списки OPEN и CLOSED
      (not (assoc (first rtemp) open))
      (not (assoc (first rtemp) closed))
    )
    ;; добавляем дочернюю вершину в результирующий список L
    (setq L (cons rtemp L))
  )
)
)

;;;<4. Вспомогательные функции

;; функция, инвертирующая подсписок temp
(defun rev(temp)
  (setq a (first temp))
  (setq b (second temp))
  (list b a))

;;функции, восстанавливающая по списку closed искомый путь
(defun back-way (goal start)
  (setq g goal)
  (setq L nil)
  (dolist (temp closed L)
    (if (eq (first temp)g)
      (progl (setq L (cons (rev temp) L))
        (setq g (second temp))
      )
    )
  )
)
)

```

В.2. Файл defpackage.lisp

```

(in-package :common-lisp-user)

(defpackage :depth
  (:nicknames :depth)
  (:use :cl
    ;; Packages you want to import go here
  )
  (:export

    ;; Exported symbols go here

  ))

```

В.3. Файл depth.asd

```

;;; 2009-12-17 19:35:34
;;; Think of this as your project file.
;;; Keep it up to date, and you can reload your project easily
;;; by right-clicking on it and selecting "Load Project"

```

```
(defpackage #:depth-asd
  (:use :cl :asdf))

(in-package :depth-asd)

(defsystem depth
  :name "depth"
  :version "0.1"
  :serial t
  :components ((:file "defpackage")
               (:file "main" :depends-on ("defpackage")))

  ; As you add files to your project,
  ; make sure to add them here as well

  )
:depends-on ()))
```

B.4. Результаты

1. depth>
 (depth-first 'a 'h)

 closed=((A A))
 open=((E A) (B A))
 closed=((E A) (A A))
 open=((F E) (D E) (B A))
 closed=((F E) (E A) (A A))
 open=((H F) (C F) (D E) (B A))
((A A) (A E) (E F) (F H))
2. depth>
 (depth-first 'h 'a)

 closed=((H H))
 open=((F H) (G H))
 closed=((F H) (H H))
 open=((E F) (C F) (B F) (G H))
 closed=((E F) (F H) (H H))
 open=((D E) (A E) (C F) (B F) (G H))
 closed=((D E) (E F) (F H) (H H))
 open=((A E) (C F) (B F) (G H))
((H H) (H F) (F E) (E A))
3. depth>
 (depth-first 'h 'm)

 closed=((H H))
 open=((F H) (G H))
 closed=((F H) (H H))
 open=((E F) (C F) (B F) (G H))
 closed=((E F) (F H) (H H))
 open=((D E) (A E) (C F) (B F) (G H))
 closed=((D E) (E F) (F H) (H H))
 open=((A E) (C F) (B F) (G H))
 closed=((A E) (D E) (E F) (F H) (H H))
 open=((C F) (B F) (G H))
 closed=((C F) (A E) (D E) (E F) (F H) (H H))
 open=((B F) (G H))
 closed=((B F) (C F) (A E) (D E) (E F) (F H) (H H))
 open=((G H))
 closed=((G H) (B F) (C F) (A E) (D E) (E F) (F H) (H H))
 open=NIL
NIL

Приложение Г (справочное)

Запись и чтение s-выражений из файлов

```
(in-package :files)

;;;<1. Запись s-выражений в файл
(defun writer ( )
  (with-open-file (store "d:/test1.txt" :direction :output
                     :if-exists :append
                     :if-does-not-exist :create )
    (princ "Введите s-выражение")
    (terpri)
    (princ "Для завершения ввода напечатайте eof")
    (terpri)
    (do                                     ;цикл
      ((item (read) (read)))              ;чтение s-выражения в item
      ((eq item 'eof) 'end-of-session ) ;проверка условия item=eof
      (print item store )                 ;запись в файл
    )
  )
)

;;;<2. Чтение s-выражений из файла
(defun reader ( )
  (with-open-file (store "d:/test1.txt" :direction :input)
    (do                                     ;цикл
      ;чтение из файла "store" до обнаружения конца файла - nil
      ((item (read store nil 'finished) ;чтение первого s-выражения в item
              (read store nil 'finished))) ;чтение очередного s-выражения
      ((eq item 'finished) "Конец файла");проверка условия item=finished
      (print item)          ;вывод s-выражения
    )
  )
)
```

ПРИЛОЖЕНИЕ Д

(справочное)

Получение справочной информации из Интернет

Основная справочная информация может быть получена с помощью системы HyperSpec. Для поиска описания символа Лиспа в этой системе необходимо навести указатель мыши на символ программы и нажать клавиши **Alt+N**, либо выбрать **Лисп > HyperSpec**. В результате откроется веб-страница системы HyperSpec (рис. Д.1).

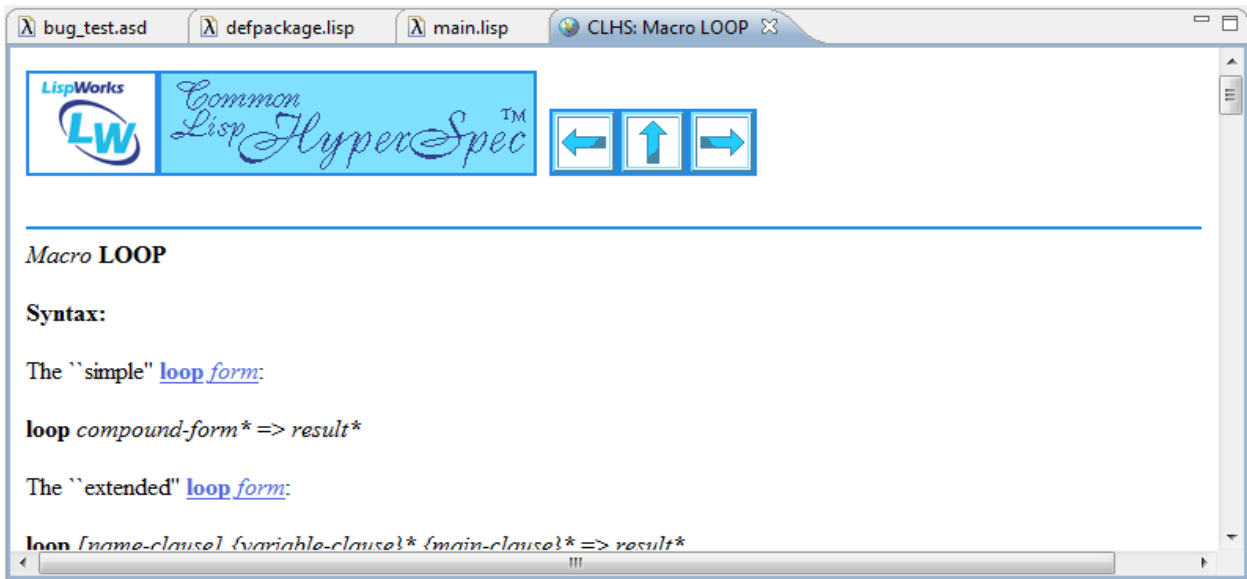


Рисунок Д.1 — Окно справочной системы HyperSpec

Иная возможность получения справочных данных основана на использовании справочной системы LispDoc. Для поиска сведений в этой системе необходимо навести указатель мыши на символ программы и нажать клавиши **Alt+L**, либо выбрать **Лисп > LispDoc**. В результате система LispDoc отобразит пример кода с использованием символа и список ссылок на доступную Лисп-документацию (рис. Д.2).

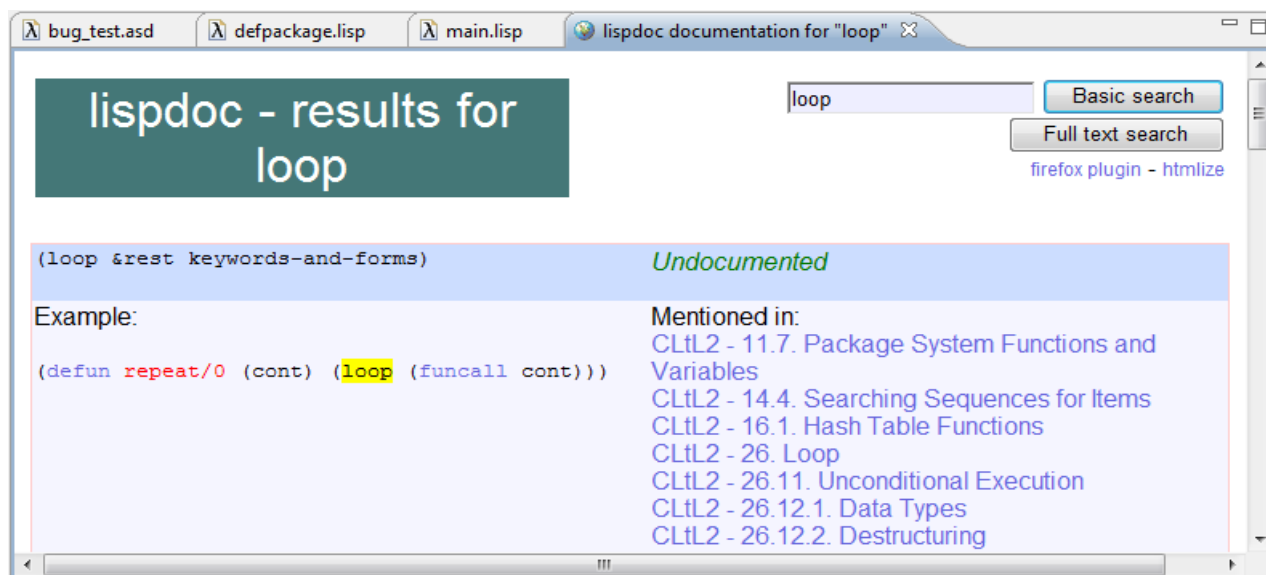


Рисунок Д.2 — Окно справочной системы LispDoc

Заказ № _____ от «_____» _____ 2015г. Тираж _____ экз.
Изд-во СевГУ