Министерство науки и высшего образования Российской Федерации

Калужский филиал
федерального государственного бюджетного
образовательного учреждения высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(КФ МГТУ им. Н.Э. Баумана)

Факты в среде CLIPS

Методические указания по выполнению лабораторной работы

Калуга,2024

Содержание

Цель и задачи работы, требования к результатам её выполнения

Целью выполнения лабораторной работы является формирование практических навыков по работе с фактами в среде CLIPS.

Основными задачами выполнения лабораторной работы являются:

1. понять что такое факт применительно к экспертным системам, его структуру и виды;
2. изучить основные команды для работы с фактами в среде CLIPS;
3. научиться создавать, удалять, изменять, сохранять и загружать факты;
4. овладеть навыками очищения и повторной инициализации базы знаний.

Результатами работы являются:

- созданные в среде CLIPS факты;
- сохранённые в файлах содержимые базы знаний и скриптов;
- подготовленный отчёт.

Краткая характеристика объекта изучения, исследования

CLIPS представляет собой современный инструмент, предназначенный для создания экспертных систем (*expert system tool*). Первоначально аббревиатура CLIPS была названием языка — C *Language Integrated Production System* (язык C, интегрированный с продукционными системами), удобного для разработки баз знаний и макетов экспертных систем. CLIPS состоит из интерактивной среды - экспертной оболочки со своим способом представления знаний, гибкого и мощного языка и нескольких вспомогательных инструментов. Сейчас, благодаря доброй воле своих создателей, CLIPS является абсолютно свободно распространяемым программным продуктом. Всем желающим доступен как сам CLIPS последней версии, так и его исходные коды. Официальный сайт CLIPS располагается по адресу: http://www.clipsrules.net/. Этот сайт поможет вам получить как сам CLIPS, так и всевозможный материал для его изучения и освоения (документацию, примеры, советы специалистов, исходные коды и многое другое).

Система, претендующая называться экспертной, должна обладать знаниями. Эти знания, естественно, должны быть ориентированы на конкретную предметную область, и из этих знаний должно непосредственно вытекать решение проблемы. Именно поэтому знания в экспертных системах предполагают определённую организацию и интеграцию (отдельные факты, сведения должны каким-либо образом соотноситься друг с другом и образовывать между собой определённые связи), т.е. знания должны быть соответствующе представлены.

Применение CLIPS для построения систем, основанных на знаниях, может быть обусловлено следующими причинами:

- этот язык является свободно распространяемым программным продуктом;
- его исполнительная система обладает вполне приемлемой производительностью;
- язык имеет чётко сформулированный синтаксис;
- в него включено множество опробованных на практике конструкций из других инструментальных средств;
- язык допускает вызов внешних функций, написанных на других языках программирования; в свою очередь модули, написанные на CLIPS, могут быть вызваны программами, написанными на других языках;
- язык включает средства, позволяющие комбинировать порождающие правила и объектно-ориентированный подход.

CLIPS предлагает эвристические и процедурные подходы для представления знаний. Также средства CLIPS позволяют применять и объектно-ориентированный подход к организации знаний. Кроме того, язык предоставляет возможности комбинировать эти подходы.

В CLIPS используется оригинальный Lisp-подобный язык программирования, ориентированный на разработку ЭС. Кроме того, CLIPS поддерживает ещё две парадигмы программирования: объектно-ориентированную и процедурную.

Работа с CLIPS

Работа со средой CLIPS начинается с получения исходного кода, сборки исполняемого файла и сохранение его в системный каталог или каталог текущего пользователя:

```
# Сохраняем название папки в переменную dir
dir=clips_core_source_640

# Заходим в каталог /tmp/
cd /tmp/

# Скачиваем архив
wget -cO clips.tar.gz \
  "https://sourceforge.net/projects/clipsrules/files/CLIPS/6.40/$dir.tar.gz"
# -c - продолжить скачивание с того места, где закончилось предыдущее
# -O - переименовать скачиваемый файл

# Распаковываем архив
tar xf clips.tar.gz
# -x - режим распаковки
# -f - после опции следует название архива
# Примечание. У команды tar принято опускать '-' перед группой нераздельных
# команд, идущей сразу после "tar"

# Заходим в каталог /tmp/clips_core_source_640/core/
cd $dir/core/

# Запускаем процесс компиляции
make
# Чтобы ускорить компиляцию, можно задействовать N потоков процессора:
make -j N
# Или задействовать все потоки:
make -j $(nproc)

# ------------------------------------------------------------------------
# Если нужно установить CLIPS только для текущего пользователя:
# Создаём каталог
mkdir -p ~/.local/bin
# -p - создать все папки в указанном пути и не выводить ошибку если все
# или некоторые папки уже существуют

# Копируем исполняемый файл в созданный каталог
cp clips ~/.local/bin/
# ------------------------------------------------------------------------

# -----------------------------------------------------
# Если нужно установить CLIPS для всех пользователей:
sudo cp clips /usr/local/bin/
# -----------------------------------------------------
```

Если команда ниже ничего не выводит,

```
echo "$PATH" | grep ~/\.local/bin
```

то нужно добавить следующее в `~/.profile`:

```
# set PATH so it includes user's private bin if it exists
if [ -d "$HOME/.local/bin" ]; then
  PATH="$HOME/.local/bin:$PATH"
fi
```

Тогда при следующем входе в систему путь `~/.local/bin` будет добавлен в переменную `PATH` и все программы по данному пути будут доступны для исполенния (без указания пути). Если нужно это сделать без перезахода в систему, то можно зайти в login shell, которая прочитает содержимое файла `.profile`:

```
bash -l
```

Также можно в non-login shell (при обычном запуске терминала или команды `bash`/`sh`) написать «`source ~/.profile`» и получить тот же (временный) результат.

Команда `clips` запускает оболочку CLIPS:

```
         CLIPS (6.4 2/9/21)
CLIPS>
```

Командой выхода при работе с CLIPS является команда (**`exit`**) [5, с. 301, раздел 13.1.7]:

```
(exit [<integer-expression>])
```

Необязательный аргумент `<integer-expression>` позволяет указать код состояния выхода и передаётся функции выхода на C.

Экспертные системы, созданные с помощью CLIPS, могут быть запущены тремя основными способами:

- вводом соответствующих команд и конструкторов языка непосредственно в среду CLIPS;
- использованием интерактивного оконного интерфейса CLIPS (например, для версий *Windows* или *Macintosh*);
- с помощью программ-оболочек, реализующих свой интерфейс общения с пользователем и использующих механизмы представления знаний и логического вывода CLIPS.

Основные элементы языка CLIPS

Синтаксис языка CLIPS можно разбить на три основные группы элементов, предназначенных для написания программ:

- примитивные типы данных;
- функции, использующиеся для обработки данных;
- конструкторы, предназначенные для создания таких структур языка, как факты, правила, классы и т.д.

CLIPS поддерживает 8 *примитивных типов данных*: `INTEGER`, `FLOAT`, `SYMBOL`, `STRING`, `EXTERNAL-ADDRESS`, `FACT-ADDRESS`, `INSTANCE-ADDRESS`, `INSTANCE-NAME`.

Для хранения численной информации предназначаются типы `FLOAT` и `INTEGER`, для символьной — `SYMBOL` и `STRING`.

Число в CLIPS может состоять только из цифр 0–9, десятичной точки «.», знака «+» или «-» и экспоненциального символа «e» с соответствующим знаком, в случае представления числа в экспоненциальной форме.

Количество значащих цифр зависит от аппаратной реализации. В этой же связи могут возникать ошибки округления.

Как в любом языке программирования, особенную осторожность необходимо проявлять при сравнении чисел с плавающей точкой, а также при сравнении с ними целых чисел.

Примеры целых чисел:

```
237 15 +12 -32
```

Примеры чисел с плавающей точкой:

```
237e3 15.09 +12.0 -32.3e-7
```

Последовательность символов, которая не удовлетворяет числовым типам, обрабатывается как тип данных SYMBOL.

Тип данных SYMBOL в CLIPS — последовательность символов, состоящая из одного или нескольких любых печатных символов кода ASCII. Как только в последовательности символов встречается символ-разделитель, SYMBOL заканчивается. Следующие символы служат разделителями: любой непечатный ASCII-символ (включая пробел, символ табуляции, CR, LF), двойные кавычки «ï», «(» «)», «&», «|», «<», «~», «;». Символы-разделители не могут включаться в SYMBOL за исключением символа «<», который может быть первым символом в SYMBOL. Кроме того, SYMBOL не может начинаться с символа «?» или последовательности символов «$?», поскольку эти символы зарезервированы для переменных. Заметим, что CLIPS различает регистр символов. Ниже приведены примеры выражений символьного типа:

```
foo Hello B76-HI bad_value
127A 742-42-42 @+=-% Search
```

Тип данных STRING — это последовательность символов, состоящая из нуля и более печатных символов и заключенная в двойные кавычки. Если внутри строки встречаются двойные кавычки, то перед ними необходимо поместить символ «\». То же справедливо и для самого «\».

Несколько примеров:

```
"fooa and bI numbera\'quote"
```

Отметим, что строка "abcd" не то же самое, что abcd. Они содержат одинаковые наборы символов, но являются экземплярами разного типа.

Основные команды среды CLIPS

Очистка памяти осуществляется при помощи одной из следующих команд: (**clear**)или (**reset**). Обе команды не имеют возвращаемого значения.

При выполнении команды (**clear**)[5, с. 301, раздел 13.1.6] или (**reset**)[5, с. 301, раздел 13.1.8] текущий индекс фактов обнуляется.

Команда (**clear**)очищает текущий список фактов (а также все определённые конструкторы, которые уже были и ещё будут рассмотрены ниже). В отличие от (**reset**), команда (**clear**)не добавляет в список фактов факты, объявленные в (**deffacts**).

Функция (**clear**)полностью очищает систему, т.е. удаляет все правила, факты и прочие объекты базы знаний CLIPS, добавленные конструкторами, приводит систему в начальное состояние, необходимое для каждой новой программы.

Для загрузки данных (кроме фактов) из файла в рабочую память можно использовать команду (**load**) [5, с. 299, раздел 13.1.1] с указанием пути к файлу:

```
(load "file name")
(load file_name)
```

В первом примере аргумент имени файла является типом `STRING`, а во втором — `SYMBOL`. Далее по тексту для указания примера имени файла будем использовать тип `SYMBOL`.

Команда (**`load`**) возвращает символ `TRUE`, если файл был успешно загружен, иначе возвращается `FALSE`. [или лучше вставлять ссылку в самом конце описания команды?]

Для сохранения данных (кроме фактов) из рабочей памяти в файл можно использовать команду (**`save`**):

```
(save file_name)
```

Команда (**`save`**) возвращает символ `TRUE`, если файл был успешно сохранён, в противном случае он возвращает символ `FALSE` [5, с. 300, раздел 13.1.3].

Для загрузки/сохранения данных из файла/в файл можно использовать как относительный, так и абсолютный путь. Аргументом команд загрузки/сохранения является строка — имя файла (расширение файла не является обязательным).

Как можно заметить, наполнение списка фактов в CLIPS довольно кропотливое и длительное занятие. Если фактов достаточно много, этот процесс может растянуться на несколько часов или даже дней. Так как список фактов хранится в оперативной памяти компьютера, теоретически, из-за сбоя компьютера или, например, неожиданного отключения питания, список фактов можно безвозвратно потерять. Чтобы этого не произошло, а также для того чтобы сделать работу по наполнению базы знаний фактами более удобной, CLIPS предоставляет команды сохранения и загрузки списка фактов в файл — (**`save-facts`**) [5, с. 310–311, раздел 13.4.3] и (**`load-facts`**) [5, с. 311–312, раздел 13.4.5] соответственно.

Команда (**`save-facts`**) сохраняет факты из текущего списка фактов в текстовый файл. На каждый факт отводится одна строчка. Неупорядоченные факты сохраняются вместе с именами слотов. В функции существует возможность ограничить область видимости сохраняемых фактов. Для этого используется аргумент `<save-scope>` (границы видимости). Он может принимать значения **`local`** и **`visible`**. В случае если этот аргумент принимает значение **`visible`**, то сохраняются все факты, присутствующие в данный момент в системе. Если в качестве аргумента используется ключевое слово **`local`**, то сохраняются только факты из текущего модуля. По умолчанию аргумент `<save-scope>` принимает значение **`local`**. После аргумента `<save-scope>` может следовать список определённых в системе шаблонов (`<deftemplate-names>*`). В этом случае будут сохранены только те факты, которые связаны с указанными шаблонами. Команда (**`save-facts`**) возвращает количество сохранённых фактов и имеет следующий синтаксис:

```
(save-facts <file-name> [<save-scope> <deftemplate-names>*])
```

Замечание. В синтаксисе команды символ «*» означает любое количество параметра, указанного перед данным символом. Примерами такого параметра могут быть как отдельные аргументы, так и составные конструкции. Но отдельно стоящий символ «*» является значением типа `SYMBOL`. Символ «+» означает то же, что и символ «*», только минимальное количество параметра, стоящего перед «+» должно быть равно 1, а не 0 (как в случае с «*»).

Для загрузки сохранённых ранее файлов используется команда (**load-facts**). Она возвращает количество загруженных фактов или **-1**, если не удалось получить доступ к файлу.

Функция имеет следующий формат:

```
(load-facts <file-name>)
```

Среду CLIPS трудно назвать дружественной в плане редактирования текстов программ. Поэтому при наборе и редактировании исходных кодов мы рекомендуем пользоваться текстовым редактором. Исходные коды могут помещаться в любые файлы (не обязательно *.clp*), редактироваться там, а затем загружаться в CLIPS следующей командой:

```
clips -f file_name.clp
```

Зачастую (например, для подготовки отчёта) необходимо осуществить запись всех операций в файл. Для этого подойдут следующие две команды.

Для начала записи информации из основного экрана CLIPS в выбранный файл можно использовать команду (**dribble-on**):

```
(dribble-on file_name)
```

Команда (**dribble-on**) возвращает значение `TRUE`, если файл был успешно открыт, в противном случае возвращается `FALSE` [5, с. 304, раздел 13.2.1].

Для прекращения записи информации из основного экрана CLIPS в файлы следует использовать команду (**dribble-off**). Она возвращает значение `TRUE`, если файл был успешно закрыт, в противном случае возвращается `FALSE` [5, с. 305, раздел 13.2.2].

Задачи и порядок выполнения работы

Факты — одна из основных форм представления данных в CLIPS. Каждый факт представляет собой определённый набор данных, сохраняемый в текущем списке фактов — рабочей памяти системы. Список фактов представляет собой универсальное хранилище фактов и является частью базы знаний. Объём списка фактов ограничен только памятью вашего компьютера. Список фактов хранится в оперативной памяти компьютера, но CLIPS предоставляет возможность сохранять текущий список в файл и загружать список из ранее сохранённого файла.

В системе CLIPS фактом является список неделимых (или атомарных) значений примитивных типов данных. CLIPS поддерживает два типа фактов — упорядоченные факты (`ordered facts`) и неупорядоченные факты или *шаблоны* (`non-ordered facts` или `template facts`). Ссылаться на данные, содержащиеся в факте, можно либо используя строго заданную позицию значения в списке данных для упорядоченных фактов, либо указывая имя значения для шаблонов.

Факты можно добавлять, удалять, изменять и дублировать, вводя соответствующие команды с клавиатуры либо из программы.

Упорядоченные факты

Упорядоченные (ориентированные) факты состоят из поля, обязательно являющегося данным типа `SYMBOL`, и следующей за ним, возможно пустой, последовательности полей, разделённых пробелами. Ограничением факта служат круглые скобки.

Первое поле факта определяет так называемое *отношение*, или *связь* факта (*relation*). Термин «связь» означает, что данный факт принадлежит некоторому определённому конструктором или неявно объявленному шаблону. Приведём несколько примеров фактов:

```
(name tanya olya lena)
(book one two)
```

Количество полей в факте не ограничено. Поля в факте могут хранить данные любого примитивного типа CLIPS, за исключением первого поля, которое обязательно должно быть типа `SYMBOL`. Следующие слова зарезервированы и не могут быть использованы в качестве первого поля: **`test`**, **`and`**, **`or`**, **`not`**, **`declare`**, **`logical`**, **`object`**, **`exist`** и **`forall`**.

Добавление фактов в рабочую память осуществляется при помощи команды (**`assert`**) [5, с. 248, раздел 12.9.1], которая имеет следующий синтаксис:

```
(assert <fact>)
```

Функция **`assert`** — одна из наиболее часто применимых команд в системе CLIPS. Без использования этой команды нельзя написать даже самую простую экспертную систему и запустить её на выполнение в среде CLIPS.

Функция **`assert`** позволяет добавлять факты в список фактов текущей базы знаний. Каждым вызовом этой функции можно добавить произвольное число фактов. При использовании команды (**`assert`**) необходимо помнить, что первое поле факта обязательно должно быть значением типа `SYMBOL`. В случае удачного добавления фактов в базу знаний функция возвращает адрес последнего добавленного факта. Если во время добавления некоторого факта произошла ошибка, команда прекращает свою работу и возвращает значение `FALSE`.

Пример добавления факта в рабочую память:

```
(assert (name tanya olya lena))
```

Следует заметить, что при вводе двух одинаковых фактов, т.е. факты со всеми совпадающими полями, последний введённый будет проигнорирован, а команда вернёт первый введённый факт. Данную установку системы можно изменить с помощью команды (**`set-fact-duplication`**) (значение по умолчанию `FALSE`) [5, с. 312, раздел 13.4.7].

Слотам неупорядоченного факта, значения которых не заданы, будут присвоены значения по умолчанию.

Для просмотра фактов (всех типов), содержащихся в рабочей памяти, можно использовать команду (**`facts`**) [5, с. 309, раздел 13.4.1].

После добавления факта в базу знаний рано или поздно встанет вопрос о том, как его оттуда удалить. Для удаления фактов из текущего списка фактов в системе CLIPS

предусмотрена функция **retract**. Каждым вызовом этой функции можно удалить произвольное число фактов. Удаление некоторого факта может стать причиной удаления других фактов, которые логически связаны с удаляемым. Команда (**retract**) [5, с. 249, раздел 12.9.2] имеет следующий синтаксис:

```
(retract <retract-specifier>+ | *)

<retract-specifier> ::= <fact-specifier> | <integer-expression>
```

Приведём несколько примеров её использования:

```
(retract 5)
(retract ?fact-to-retract) ; Данная переменная хранит адрес факта
(retract (* ?index ?multiplier)) ; Результат функции умножения - индекс факта
```

Термин (аргумент) `<retract-specifier>` включает в себя: переменные, привязанные в левой части к адресам фактов, индекс желаемого факта (например, `3` для факта, помеченного как `f-3`), или выражение, вычисляющее индекс желаемого факта. Если в качестве аргумента используется символ «*», все факты будут удалены из рабочей памяти (фактов). Команда (**retract**) ничего не возвращает.

Необходимо заметить, что функция **retract** не оказывает никакого воздействия на индекс следующих добавленных фактов, т.е. этот индекс не обнуляется. Если после удаления всех введённых фактов добавить в систему какой-нибудь факт, то он получит индекс f-n, несмотря на то что список фактов в данный момент пуст.

Неупорядоченные факты

Так как упорядоченный факт для представления информации использует строго заданные позиции данных, то для доступа к ней пользователь должен знать не только то, какие данные сохранены в факте, но и какое поле содержит эти данные. Неупорядоченные (неориентированные) факты (или шаблоны) предоставляют пользователю возможность задавать абстрактную структуру факта путём назначения имени каждому полю. Для создания шаблонов, которые впоследствии будут применяться для доступа к полям факта по имени, используется конструктор (**deftemplate**). Конструктор (**deftemplate**) аналогичен определениям записей или структур в таких языках программирования, как Pascal или C [5, с. 35–38, раздел 3].

Конструктор (**deftemplate**) задаёт имя шаблона и определяет последовательность из нуля или более полей неупорядоченного факта, называемых также слотами. Слот состоит из имени, заданного значением типа `SYMBOL`, и следующего за ним, возможно пустого, списка полей. Как и факт, слот с обеих сторон ограничивается круглыми скобками. В отличие от упорядоченных фактов слот неупорядоченного факта может жёстко определять тип своих значений. Кроме того, слоту могут быть заданы значения по умолчанию.

Для описания шаблона используют команду (**deftemplate**). Общий синтаксис:

```
(deftemplate <deftemplate-name> ["comment"]
  (slot <slot-name> <template-attribute>*)*
  (multislot <slot-name> <template-attribute>*)*

  <template-attribute> ::= <default-attribute> | <constraint-attribute>
  <default-attribute> ::= (default ?DERIVE | ?NONE | <expression>*) |
                          (default-dynamic <expression>*)
)
```

При создании шаблона с помощью конструктора (**deftemplate**) каждому полю можно назначать определённые атрибуты, задающие значения по умолчанию или ограничения на значение слота. Рассмотрим эти атрибуты подробней.

Атрибут `default`/`default-dynamic` определяет значение, которое будет использовано в том случае, если при создании факта не задано конкретное значение слота. В CLIPS существует два способа определения значения по умолчанию, поэтому в конструкторе (**deftemplate**) предусмотрено два различных атрибута, задающих значения по умолчанию: `default` и `default-dynamic`.

Атрибут `default` определяет статическое значение по умолчанию. С его помощью задаётся выражение, которое вычисляется один раз при конструировании шаблона. Результат вычислений сохраняется вместе с шаблоном. Этот результат присваивается соответствующему слоту в момент объявления нового факта. Если в качестве значения по умолчанию используется ключевое слово **?DERIVE**, то это значение будет извлекаться из ограничений, заданных для данного слота. По умолчанию для всех слотов установлен атрибут (`default` **?DERIVE**).

Если вместо выражения для значения по умолчанию используется ключевое слово **?NONE**, то значение поля обязательно должно быть явно задано в момент выполнения операции добавления факта. Добавление факта без определения значений полей с атрибутом (`default` **?NONE**) вызовет ошибку.

Атрибут `default-dynamic` предназначен для установки динамического значения по умолчанию. Этот атрибут определяет выражение, которое вычисляется всякий раз при добавлении факта по данному шаблону. Результат вычислений присваивается соответствующему слоту.

Простой слот может иметь только одно значение по умолчанию. У составного слота может быть определено любое количество значений по умолчанию (количество значений по умолчанию должно соответствовать количеству данных, сохраняемых в составном слоте).

В качестве дополнительных параметров поля можно указывать:

1. тип поля (возможно использование сразу нескольких типов) (`type INTEGER FLOAT STRING SYMBOL NUMBER LEXEME EXTERNAL-ADDRESS FACT-ADDRESS INSTANCE-NAME INSTANCE-ADDRESS`);
2. диапазон значений (`range <start-value> <end-value>`);
3. список возможных значений (`allowed-types <values-list>`)

   В качестве типов могут выступать: `allowed-symbols`, `allowed-strings`, `allowed-numbers`, `allowed-integers`, `allowed-floats`, `allowed-values`;
4. список возможных значений (`cardinality <start-value> <end-value>`).

В конструкторе (**deftemplate**) поддерживается проверка статических и динамических ограничений.

Статическая проверка выполняется во время использования определения шаблона некоторой командой или конструктором. Например, для записи значений в слоты шаблона. Иначе говоря, статическая проверка выполняется до запуска программы. При несоответствии используемых значений с установленными ограничениями пользователю выводится соответствующее предупреждение об ошибке.

Ссылка на индекс факта в командах на изменение значения факта или его дублирование не связывает факт с соответствующим шаблоном явно. Это делает статическую проверку неоднозначной. Поэтому в командах, использующих индекс факта, статическая проверка не выполняется. Статическая проверка ограничений всегда включена.

Помимо статической, CLIPS также поддерживает динамическую проверку ограничений. Если режим динамической проверки ограничений включен, то все новые факты, созданные с использованием некоторого шаблона и имеющие определённые значения, проверяются в момент их добавления в список фактов.

Если нарушение заданных ограничений произойдёт в момент выполнения динамической проверки в процессе выполнения программы, то выполнение программы прекращается и пользователю будет выдано соответствующее сообщение.

По умолчанию в CLIPS отключён режим динамической проверки ограничений. Эту среду установки можно изменить с помощью команды (**set-dynamic-constraint-checking**) [5, с. 303, раздел 13.1.13].

Приведём пример шаблона:

```
(deftemplate book
  (slot name (type STRING))
  (slot author (type STRING))
  (slot year (type INTEGER) (default 2006) (range 1400 2006))
  (slot pages (type INTEGER))
  (multislot notes (cardinality 1 4))
)
```

Пример неориентированного факта (экземпляра шаблона `book`):

```
(book
  (name "Sun")
  (author "Tom")
  (year 1990)
  (pages 120)
)
```

Приведём пример ввода неориентированного факта (экземпляра шаблона `book`):

```
(assert (book (name "Life") (author "Bob") (year 1890) (pages 300)))
```

Для просмотра списка всех шаблонов в рабочей памяти можно использовать команду (**list-deftemplates**). Данная команда ничего не возвращает [5, с. 308, раздел 13.3.2].

Для просмотра конкретного шаблона можно использовать команду (**ppdeftemplate**) [5, с. 308, раздел 13.3.1], которая имеет следующий синтаксис:

```
(ppdeftemplate <deftemplate-name> [<logical-name>])
```

Если аргумент `<logical-name>` не указан или равен `t`, то вывод отправляются на логическое имя `stdout`, в противном случае они отправляются на указанное логическое имя. Если используется логическое имя `nil`, то текст используется в качестве возвращаемого значения этой команды (типа `STRING`), а не отправляется в место назначения вывода; в противном случае команда (**`ppdeftemplate`**) ничего не возвращает.

Для удаления из памяти конкретного шаблона (не имеющего зависимых фактов) можно использовать команду (**`undeftemplate`**) [5, с. 308–309, раздел 13.3.3], которая ничего не возвращает и имеет следующий синтаксис:

```
(undeftemplate <template-identifier>)
```

Для удаления всех шаблонов, находящихся в рабочей памяти (и не имеющих зависимых фактов), используется следующая команда:

```
(undeftemplate *)
```

Для полноты картины следует также упомянуть о неявно создаваемых шаблонах. При использовании факта или ссылки на упорядоченный факт (например, в правиле) CLIPS неявно создаёт соответствующий шаблон с одним составным слотом. Имя неявно созданного составного слота не отображается при просмотре фактов. Неявно созданным шаблоном можно манипулировать и сравнивать его с любым тождественным, определённым пользователем шаблоном, несмотря на то что он не имеет отображаемой формы [5, с. 38, раздел 3.4].

Конструктор (`deffacts`)

CLIPS предоставляет конструктор (**`deffacts`**), предназначенный для работы с фактами [5, с. 39, раздел 4]. Данный конструктор позволяет определять список фактов, которые будут автоматически добавляться всякий раз после выполнения команды (**`reset`**), очищающей текущий список фактов. Факты, добавленные с помощью конструктора (**`deffacts`**), могут использоваться и удаляться так же, как и любые другие факты, добавленные в базу знаний пользователем или программой с помощью команды (**`assert`**).

Общий синтаксис:

```
(deffacts <union-specifier> ["comment"]
  (<fact-specifier>)*
)
```

Приведём пример использования конструктора (**`deffacts`**):

```
(deffacts books
  (book (name "One") (author "Liz") (year 2001) (pages 200))
  (book (name "Two") (author "Liz") (year 2002) (pages 300))
  (book (name "Peace and war") (author "Tolstoy") (year 2002) (pages 300))
  (book (name "Three") (author "Liz") (year 2003))
)
```

Добавление конструктора (**`deffacts`**) с именем уже существующего конструктора приведёт к удалению предыдущего конструктора, даже если новый конструктор содержит ошибки. В среде CLIPS возможно наличие нескольких конструкций (**`deffacts`**) одновременно и

любое число фактов в них (как упорядоченных, так и неупорядоченных). Факты всех созданных пользователем конструкторов (`deffacts`) будут добавлены при инициализации системы.

В поля факта могут быть включены динамические выражения, значения которых будут вычисляться при добавлении этих фактов в текущую базу знаний CLIPS.

Проверить работу конструктора (`deffacts`) можно воспользовавшись командой (`watch`):

```
(watch <watch-item>)
```

Для этого укажите значение `facts` для аргумента `<watch-item>`:

```
(watch facts)
```

Команда (`watch`) ничего не возвращает. Полный синтаксис команды (`watch`) можно найти в документации CLIPS [5, с. 305–307, раздел 13.2.3].

После этого введите в (оболочку) CLIPS приведённый выше конструктор (`deffacts`). Затем введите команду (`reset`). В среде CLIPS должен появится следующий текст:

```
⟹ f-1     (book (name "One") (author "Liz") (year 2001) (pages 200) (notes nil))
⟹ f-2     (book (name "Two") (author "Liz") (year 2002) (pages 300) (notes nil))
⟹ f-3     (book (name "Peace and war") (author "Tolstoy") (year 2002) (pages 300) (notes nil))
⟹ f-4     (book (name "Three") (author "Liz") (year 2003) (pages 0) (notes nil))
```

Как видно из этого примера, команда (`watch`) позволяет выводить дополнительную информацию о процессе работы CLIPS. Для отключения вывода дополнительной информации можно воспользоваться командой (`unwatch`). Данная команда имеет идентичный команде (`watch`) синтаксис и возвращаеме значение [5, с. 307, раздел 13.2.4].

Используя функции `assert` и `retract`, можно выполнять большинство необходимых для функционирования правил действий. В том числе и изменения предшествующего факта.

Для изменения упорядоченных фактов доступен только этот способ. Для упрощения операции изменения неупорядоченных фактов CLIPS предоставляет функцию `modify`, которая позволяет изменять значения слотов таких фактов. (`modify`) просто упрощает процесс изменения факта, но её внутренняя реализация эквивалентна вызовам пар функций `retract` и `assert`. Вызов `modify` позволяет изменять только один факт. В случае удачного выполнения функция возвращает новый индекс модифицированного факта [5, с. 250–251, раздел 12.9.3].

Если в процессе выполнения произошла какая-либо ошибка, то пользователю выводится соответствующее предупреждение и функция возвращает значение `FALSE`.

Синтаксис команды (`modify`):

```
(modify <fact-specifier> (<slot-specifier> <new-value>)*)
```

Значение, возвращаемое этой функцией, является адресом только что изменённого факта. Если идентичная копия вновь изменённого факта уже существует в рабочей памяти фактов, то возвращается адрес факта существующей копии; в противном случае адрес и индекс факта изменённого факта сохраняются. Если добавление вновь изменённого факта вызывает ошибку, то возвращается `FALSE`.

Обратите внимание на движение фактов в базе знаний CLIPS при выполнении функции

modify — сначала удаляется старый факт, а затем добавляется новый факт, идентичный предыдущему, но с новым значением заданного слота.

Если в шаблоне заданного факта отсутствует слот, значение которого требуется изменить, CLIPS выведет соответствующее сообщение об ошибке. Если заданный факт отсутствует в списке фактов, пользователь также получит соответствующее предупреждение.

Помимо функции **modify**, в CLIPS существует ещё одна очень полезная функция, упрощающая работу с фактами, — функция **duplicate**. Эта функция создаёт новый неупорядоченный факт заданного шаблона и копирует в него определённую пользователем группу полей уже существующего факта того же шаблона. По выполняемым действиям функция **duplicate** аналогична **modify**, за исключением того, что она не удаляет старый факт из списка фактов. Одним вызовом функции **duplicate** можно создать одну копию некоторого заданного факта. Как и функция **modify**, **duplicate** в случае удачного выполнения возвращает индекс нового факта, а в случае неудачи — значение `FALSE` [5, с. 251–252, раздел 12.9.4].

Синтаксис команды (**duplicate**):

```
(duplicate <fact-identifier> (<slot-specifier> <new-value>)*)
```

Если добавляемый с помощью (**duplicate**) факт уже присутствует в списке фактов, будет выдана соответствующая информация об ошибке и возвращено значение `FALSE`. Факт при этом добавлен не будет. Это поведение можно изменить, разрешив существование одинаковых фактов в базе знаний.

Кроме функции **assert**, CLIPS предоставляет ещё одну функцию, полезную при добавлении фактов, — **assert-string**. Эта функция принимает в качестве единственного аргумента символьную строку, являющуюся текстовым представлением факта (в том виде, в котором вы набираете его, например, в функции **assert**), и добавляет его в список фактов. Функция **assert-string** может работать как с упорядоченными, так и с неупорядоченными фактами. Одним вызовом функции **assert-string** можно добавить только один факт [5, с. 252, раздел 12.9.5].

Синтаксис команды (**assert-string**):

```
(assert-string <string-expression>)
```

Строковое выражение (типа `STRING`) должно быть заключено в кавычки. Функция преобразует заданное строковое выражение в факт CLIPS, разделяя отдельные слова на поля, с учётом определённых в системе на текущий момент шаблонов. Если в строке необходимо записать внутреннее строковое выражение, представляющее, скажем, некоторое поле, то для включения в строковое выражение символа кавычек используется *обратная косая черта* (*backslash*).

Приведём пример использования команды (**assert-string**):

```
(assert-string "book-name \" Peace and war \"")
```

Для добавления содержащегося в поле символа обратной косой черты используйте её

дважды. Если обратная косая черта должна содержаться внутри подстроки, её необходимо использовать четыре раза.

Если добавление факта прошло удачно, функция возвращает индекс только что добавленного факта, в противном случае функция возвращает сообщение об ошибке и значение FALSE. Функция **assert-string** не позволяет добавлять факт в случае, если такой факт уже присутствует в базе знаний (если вы ещё не включили возможность присутствия одинаковых фактов).

Для просмотра списка определённых/созданных конструкторов (**deffacts**) существует команда (**list-deffacts**):

```
(list-deffacts [<module-name>])
```

Если аргумент <module-name> не указан, то выводятся имена всех (**deffacts**) в текущем модуле. Если аргумент <module-name> указан, то выводятся имена всех (**deffacts**) в указанном модуле. Если же аргумент <module-name> является символом «*», то выводятся имена всех (**deffacts**) во всех модулях. Эта команда не имеет возвращаемого значения [5, с. 313, раздел 13.5.2].

Для просмотра списка созданных конструкторов (**deffacts**) существует команда (**ppdeffacts**):

```
(ppdeffacts <deffacts-name> [<logical-name>])
```

Команда (**ppdeffacts**) отправляет исходный текст конструктора (**deffacts**) на логическое имя в качестве вывода. если аргумент <logical-name> равен t или не указан, то вывод отправляется на логическое имя stdout, в противном случае они отправляются на указанное логическое имя. Если используется логическое имя nil, то текст используется в качестве возвращаемого значения (типа STRING), а не отправляется в место назначения вывода; в противном случае эта команда не имеет возвращаемого значения [5, с. 313, раздел 13.5.1].

```
(ppfact <fact-specifier> [<logical-name> [<ignore-defaults-flag>]])
```

Команда (**ppfact**) отображает один факт, помещая каждый слот и его значение в отдельную строку. Дополнительно можно указать логическое имя, на которое отправляется вывод, и слоты, содержащие значения по умолчанию, могут быть исключены из вывода. Если аргумент <logical-name> равен t или не указан, то выходные данные отправляются на логическое имя stdout, в противном случае они отправляются на указанное логическое имя. Если используется логическое имя nil, то текст используется в качестве возвращаемого значения (типа STRING), а не отправляется в место назначения вывода; в противном случае эта команда не имеет возвращаемого значения.

Если аргумент <ignore-defaults-flag> не указан или равен FALSE, то выводятся все слоты факта, в противном случае слоты со статическими значениями по умолчанию отображаются только в том случае, если их текущее значение слота отличается от их начального значения по умолчанию [5, с. 309–310, раздел 13.4.2].

Команда (`undeffacts`) удаляет ранее опредёленные конструкторы (`deffacts`). Команда не имеет возвращаемого значения и имеет следующий синтаксис:

```
(undeffacts <deffacts-name>)
```

Все факты, перечисленные в удалённом конструкторе (`deffacts`), больше не будут добавляться при вызове команды (`reset`). если символ «*» используется для аргумента `<deffacts-name>`, то все конструкторы (`deffacts`) будут удалены (если не существует (`deffacts`) с именем `*`) [5, с. 313–314, раздел 13.5.3].

Задание на лабораторную работу

В среде CLIPS создать несколько упорядоченных фактов.

Определить неупорядоченные факты (шаблоны), описывающие предложенные объекты (сущности) согласно полученному варианту. Использовать не менее 6 полей с различными описателями.

Создать 8 – 10 ненаправленных фактов, объединённых в конструкцию (`deffacts`) и демонстрирующих работу каждого описательного поля.

Сохранить факты в файл.

Добавить созданные факты в конструктор (`deffacts`).

Продемонстрировать удаление, изменение фактов с последующей реинициализацией фактов.

Требования к реализации

Объекты (сущности) выбираются в соответствии с вариантом задания, который назначается преподавателем.

Все факты должны быть сохранены в файл посредством соответствующих команд CLIPS.

Варианты заданий

1. Компьютеры
2. Мониторы
3. Принтеры
4. Музыкальные инструменты
5. Геометрические фигуры в пространстве
6. Животные
7. Автомобили
8. Книги
9. Компьютерные игры
10. Музыкальные композиции
11. Функции одной переменной
12. Кошки
13. Языки программирования
14. Операционные системы
15. Студенты
16. Преподаватели
17. Предметы, изучаемые в университете
18. Небесные тела
19. Оружие
20. Цветы

21. Деревья
22. Птицы
23. Дома
24. Города
25. Страны
26. Мобильные телефоны
27. Фирмы
28. Фрукты
29. Лодки
30. Фотоаппараты
31. Стулья
32. Стереосистемы (музыкальные центры)
33. Одежда
34. Водоёмы
35. Часы
36. Носители информации
37. Пассажирские поезда
38. Сетевые карты
39. Веб-браузеры
40. Кондитерские изделия

Контрольные вопросы и задания

1. Перечислите примитивные типы данных поддерживаемые CLIPS и их назначение.
2. Чем тип `SYMBOL` отличается от типа `STRING`? Приведите пример.
3. Какие команды необходимо использовать для загрузки и сохранения данных?
4. С помощью какой команды можно очистить рабочую память?
5. Как сохранить весь текст, выводимый при запуске CLIPS программы?
6. Перечислите типы фактов в CLIPS.
7. Какой командой можно добавить факт в рабочую память?
8. Какой командой можно изменить факт в рабочей памяти?
9. Какой командой можно удалить факт из рабочей памяти?
10. Чем команда (**`duplicate`**) отличается от команды (**`modify`**)?
11. Приведите пример задания шаблона факта.
12. Для каких целей необходим (**`slot`**) и (**`multislot`**)?
13. Какие существуют описательные поля в неупорядоченных фактах и каково их назначение?
14. Как удалить шаблоны из памяти?
15. Чем действие команды (**`clear`**)отличается от команды (**`reset`**)?

Форма отчёта по лабораторной работе

На выполнение лабораторной работы отводится 2 занятия (4 академических часа: 3 часа на выполнение и сдачу лабораторной работы и 1 час на подготовку отчёта).

Номер варианта студенту выдаётся преподавателем.

Отчёт на защиту предоставляется в ЭОИС, формат файла pdf.

Структура отчёта (на отдельном листе(-ах)): титульный лист, формулировка задания (вариант), этапы выполнения работы (со скриншотами), результаты выполнения работы (скриншоты и содержимое файлов), выводы.

Основная литература

1. *Исаев С. В.* Интеллектуальные системы : учебное пособие / С. В. Исаев ; О. С. Исаева. — Красноярск : Сибирский федеральный университет, 2017. — 120 с. — Текст: электронный. — URL: https://www.iprbookshop.ru/84365.html.
2. *Пальмов С. В.* Интеллектуальные системы и технологии : учебное пособие / С. В. Пальмов. — Самара : Поволжский государственный университет телекоммуникаций и информатики, 2017. — 195 с. — Текст: электронный. — URL: https://www.iprbookshop.ru/75375.html.
3. *Пятаева А. В.* Интеллектуальные системы и технологии : учебное пособие / А. В. Пятаева ; К. В. Раевич. — Красноярск : Сибирский федеральный университет, 2018. — 144 с. — Текст: электронный. — URL: https://www.iprbookshop.ru/84358.

Электронные ресурсы:

4. *Трофимов В. Б.* Экспертные системы в АСУ ТП : учебник / В. Б. Трофимов ; И. О. Темкин. — М., Вологда : Инфра-Инженерия, 2020. — 284 с. — Текст: электронный. — URL: https://www.iprbookshop.ru/98489.html.
5. CLIPS Reference Manual. Volume I. Basic Programming Guide. Version 6.40. — 428 с. — URL: https://www.clipsrules.net/documentation/v640/bpg640.pdf.