

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)

Правила в среде CLIPS
Методические указания по выполнению лабораторной работы

Калуга, 2024

Цель и задачи работы, требования к результатам её выполнения

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

Основными задачами выполнения лабораторной работы являются:

1. изучить основные команды для работы с правилами в среде CLIPS;
2. понимать синтаксис формирования правил;
3. научиться создавать и удалять правила, построенные на различных логических элементах;
4. научиться задавать нужную для конкретной задачи стратегию разрешения конфликтов и просматривать план решения задачи.

Результатами работы являются:

- созданные в среде CLIPS правила;
- подготовленный отчёт.

Краткая характеристика объекта изучения, исследования

CLIPS поддерживает эвристическую и процедурную парадигму представления знаний. Для представления знаний в процедурной парадигме CLIPS предоставляет такие механизмы, как правила, о которых и пойдет речь в данной лабораторной работе. Правила в CLIPS служат для представления эвристик, или так называемых «эмпирических правил», которые определяют набор действий, выполняемых при возникновении некоторой ситуации. Разработчик экспертной системы определяет набор правил, которые вместе работают над решением некоторой задачи. Правила состоят из предпосылок и следствия.

Предпосылки правила представляют собой набор условий (или условных элементов), которые должны удовлетвориться, для того чтобы правило выполнилось. Предпосылки правил удовлетворяются в зависимости от наличия или отсутствия некоторых заданных фактов в списке фактов или некоторых созданных объектов, являющихся экземплярами классов, определенных пользователем.

Следствие правила представляется набором некоторых действий, которые необходимо выполнить, в случае если правило применимо к текущей ситуации. Таким образом, действия, заданные вследствие правила, выполняются по команде механизма логического вывода, если все предпосылки правила удовлетворены. В случае если в данный момент применимо более одного правила, механизм логического вывода использует так называемую стратегию разрешения конфликтов (conflict resolution strategy), которая определяет, какое именно правило будет выполнено. После этого CLIPS выполняет действия, описанные вследствие выбранного правила (которые могут оказать влияние на список применимых правил), и приступает к выбору следующего правила. Этот процесс продолжается до тех пор, пока список применимых правил не опустеет.

Основу экспертной системы составляет база знаний о предметной области, которая накапливается в процессе построения и эксплуатации экспертной системы. Знания в системе представлены, как правило, на некотором специальном языке и хранятся отдельно от собственно программного кода, который и формирует выводы и соображения. Этот компонент программы принято называть базой знаний.

Правила также хранятся в базе знаний.

Создание правил

Для добавления новых правил в базу знаний CLIPS предоставляет специальный конструктор (**defrule**) [5, с. 40–41, раздел 5.1]. В общем виде синтаксис данного конструктора можно представить следующим образом:

```
(defrule <rule-name> [<comment>]
  [<declaration>]           ; Rule Properties
  <conditional-element>*    ; Left-Hand Side (LHS)
  =>
  <action>*                 ; Right-Hand Side (RHS)
)
```

Имя правила должно быть значением типа **SYMBOL**. В качестве имени правила

нельзя использовать зарезервированные слова CLIPS. Повторное определение существующего правила приводит к удалению правила с тем же именем, даже если новое определение содержит ошибки.

Комментарии являются необязательными и, как правило, описывают назначения правила. Комментарии необходимо заключать в кавычки. Эти комментарии сохраняются и в дальнейшем могут быть доступны при просмотривании определения правила.

Определение правила может содержать объявление свойств правила, которое следует непосредственно после имени правила и комментариев. Более подробно свойства правила будут рассмотрены ниже.

В справочной системе и документации по CLIPS для обозначения предпосылок правила чаще всего используется термин «LHS of rule», а для обозначения следствия — термин «RHS of rule», поэтому в дальнейшем мы будем использовать аналогичную терминологию — левая и правая часть правила.

Левая часть правила задается набором условных элементов, который обычно состоит из условий, примененных к некоторым образцам. Заданный набор образцов используется системой для сопоставления с имеющимися фактами объектами. Все условия в левой части правила объединяются с помощью неявного логического оператора `and`. Правая часть правила содержит список действий, выполняемых при активизации правила механизмом логического вывода. Для разделения правой и левой части правил используется символ « \Rightarrow ». Правило не имеет ограничений на количество условных элементов или действий. Единственным ограничением является свободная память вашего компьютера. Действия правила выполняются последовательно, но тогда и только тогда, когда все условные элементы в левой части этого правила удовлетворены.

Если в правой части правила не определено ни одно действие, правило может быть активировано и выполнено, но при этом ничего не произойдет.

Для запуска CLIPS-программ используется команда (**run**):

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

`<integer-expression>` является необязательным аргументом команды. В простейшем случае в качестве этого аргумента можно использовать целую константу. Если данный аргумент задан и он положителен, то CLIPS запустит на выполнение заданное число правил из плана решения задачи. Если данное число больше числа правил в плане решения задачи, то будут запущены все правила. В случае если аргумент не задан или является отрицательным, план решения задачи также будет выполнен полностью [5, с. 319–320, раздел 13.7.2].

План решения задачи (`agenda`) можно просмотреть с помощью команды (**agenda**) [5, с. 319, раздел 13.7.1]:

```
(agenda)
```

Для демонстрации работы с правилами наберите следующую программу:

```
(defrule rule1
  (name tanya)
  =>
  (println "Hello Tanya")
)
(assert (name tanya))
(facts)
(agenda)
(run)
(exit)
```

Правило rule1 проверяет наличие в рабочей памяти факта (name tanya) и в случае удачной проверки на экране будет выведено следующее:

```
Hello Tanya
```

В случае необходимости выполнения определенного правила, команду run применяют с числовым параметром — номером правила.

Свойства правил

Свойства правил позволяют задавать характеристики правил до описания левой части правила. Для задания свойства правила используется ключевое слово declare. Одно правило может иметь только одно определение свойства, заданное с помощью declare.

Основное свойство правил — это свойство salience. Свойство правила salience позволяет пользователю назначать приоритет для своих правил. Объявляемый приоритет должен быть выражением, имеющим целочисленное значение из диапазона от -10000 до +10000. Выражение, представляющее приоритет правила, может использовать глобальные переменные и функции. В случае если приоритет правила явно не задан, ему присваивается значение по умолчанию — 0.

Значение приоритета может быть вычислено в одном из трех случаев: при добавлении нового правила, при активации правила и на каждом шаге основного цикла выполнения правил. Два последних варианта называются динамическим приоритетом (dynamic salience). По умолчанию значение приоритета вычисляется только во время добавления правила. Для изменения этой установки можно использовать команду (set-salience-evaluation) [5, с. 321–222, раздел 13.7.9].

Команды и функции для работы с правилами

После создания правил с помощью конструктора (defrule) возможно выполнение ряда операций с уже существующим правилом. CLIPS поддерживает множество различных команд, оперирующих с правилами. В данном разделе мы рассмотрим наиболее часто используемые команды: (ppdefrule), (list-defrules) и (undefrule).

С помощью команды (ppdefrule) можно просмотреть определение правила в том виде, в котором оно было создано с помощью конструктора (defrule) [5, с. 314, раздел 13.6.1]. Команда имеет следующий синтаксис:

```
(ppdefrule имя_правила)
```

Для того чтобы получить полный список правил, присутствующих в CLIPS в данный момент, используется команда (**list-defrules**):

```
(list-defrules имя_модуля)
```

Полный синтаксис этой команды содержит необязательный аргумент <имя-модуля>. Если данный аргумент не задан, то будет выведен список правил, определенных в текущем модуле. В случае явного задания модуля будет выведен список правил, принадлежащих конкретному модулю. Данный аргумент может принимать значение «*». В этом случае на экран будет выведен список всех правил из всех модулей [5, с. 314, раздел 13.6.2].

Для удаления правила используется команда (**undefrule**):

```
(undefrule <defrule-name>)  
(undefrule rule_name)  
(undefrule имя_правила)
```

В качестве параметра команда (**undefrule**) принимает имя правила, которое нужно удалить. Если в качестве имени правила был задан символ «*», то будут удалены все правила [5, с. 314–315, раздел 13.6.3].

CLIPS не содержит специальных команд для изменения существующих правил. Чтобы изменить существующее правило, пользователю необходимо заново определить данное правило с помощью конструктора (**defrule**). При этом существующее определение правила будет автоматически удалено из системы, даже если новый конструктор содержал ошибки, вследствие чего новое правило добавлено не было.

CLIPS предоставляет возможность просматривать списки наборов данных (фактов или объектов), способных активировать заданное правило. Команда (**matches**) выводит информацию обо всех возможных наборах данных, способных активировать это правило:

```
(matches имя_правила)
```

Как вы уже успели убедиться, создавать правила конструктором (**defrule**) каждый раз, по мере необходимости используя для этого среду CLIPS, достаточно неудобно. Для облегчения участи пользователя CLIPS позволяет загружать конструкторы правил (как, впрочем, и все остальные конструкторы) из текстового файла. Для этого используются команды (**load**) и (**save**) (см. методические указания к лабораторной работе №1).

Текстовый формат — не единственный способ хранения конструкторов CLIPS. Команды (**bsave**) и (**bload**) позволяют сохранять и загружать конструкторы в двоичном виде. Двоичные файлы загружаются гораздо быстрее, чем текстовые, но занимают больше места (так как, кроме конструкторов, они хранят полную информацию о текущем состоянии среды). Еще одним неудобством пользования двоичных файлов является то, что создавать их можно только непосредственно в среде CLIPS.

Стратегии разрешения конфликтов

План решения задачи — это список всех правил, имеющих удовлетворенные условия

при некотором, текущем состоянии списка фактов и объектов, которые еще не были выполнены.

CLIPS поддерживает семь различных стратегий разрешения конфликтов: стратегия глубины (depth strategy), стратегия ширины (breadth strategy), стратегия упрощения (simplicity strategy), стратегия усложнения (complexity strategy), LEX (LEX strategy), MEA (MEA strategy) и случайная стратегия (random strategy). По умолчанию в CLIPS установлена стратегия глубины. Текущая стратегия может быть установлена командой (**set-strategy**) (которая переупорядочит текущий план решения задачи, базируясь на новой стратегии) [5, с. 320–321, раздел 13.7.5].

Стратегия глубины

Только что активированное правило помещается выше всех правил с таким же приоритетом. Например, допустим, что факт-А активировал правило-1 и правило-2 и факт-Б активировал правило-3 и правило-4, тогда, если факт-А добавлен перед фактом-Б, в плане решения задачи правило-3 и правило-4 будут располагаться выше, чем правило-1 и правило-2. Однако позиция правила-1 относительно правила-2 и позиция правила-3 относительно правила-4 будут произвольными [5, с. 43, раздел 5.3.1].

Стратегия ширины

Только что активированное правило помещается ниже всех правил с таким же приоритетом. Например, допустим, что факт-А активировал правило-1 и правило-2 и факт-Б активировал правило-3 и правило-4, тогда, если факт-А добавлен перед фактом-Б, в плане решения задачи правило-1 и правило-2 будут располагаться выше, чем правило-3 и правило-4. Однако позиция правила-1 относительно правила-2 и позиция правила-3 относительно правила-4 будут произвольными [5, с. 43, раздел 5.3.2].

Стратегия упрощения

Между всеми правилами с одинаковым приоритетом только что активированные правила размещаются выше всех активированных правил с равной или большей определенностью (specificity). Определенность правила вычисляется по числу сопоставлений, которые нужно сделать в левой части правила. Каждое сопоставление с константой или заранее связанной с фактом переменной добавляет к определенности единицу. Каждый вызов функции в левой части правила, являющийся частью условного элемента test, также добавляет к определенности единицу. Логические функции and, or и not не увеличивают определенность правила, но их аргументы могут сделать это. Вызовы функций, сделанные внутри функций, не увеличивают определенность правила [5, с. 43, раздел 5.3.3].

Стратегия усложнения

Между всеми правилами с одинаковым приоритетом, только что активированные правила размещаются выше всех активированных правил с равной или меньшей определенностью [5, с. 43, раздел 5.3.4].

Стратегия LEX

Для определения места активированного правила в плане решения задачи используется «новизна» образца, который активировал правило. CLIPS маркирует каждый факт или объект временным тегом для отображения относительной новизны каждого факта или объекта в системе. Образцы, ассоциированные с каждой активацией правила, сортируются по убыванию тегов для определения местоположения правила. Активация правила, выполненная более новыми образцами, располагается перед активацией, осуществленной более поздними образцами. Для определения порядка размещения двух активаций правил поодиночке сравниваются отсортированные временные теги для этих двух активаций, начиная с наибольшего временного тега. Сравнение продолжается до тех пор, пока не останется одна активация с наибольшим временным тегом.

Если активация некоторого правила выполнена большим числом образцов, чем активация другого правила, и все сравниваемые временные теги одинаковы, то активация с большим числом временных тегов помещается перед активацией с меньшим. Если две активации имеют одинаковое количество временных тегов и их значения равны, то правило с большей определенностью помещается перед правилом с меньшей. Условный элемент `not` в CLIPS имеет псевдовременной тег, который также используется в данной стратегии разрешения конфликтов. Временной тег условного элемента `not` всегда меньше, чем временной тег образца [5, с. 44, раздел 5.3.5].

Стратегия МЕА

Среди правил с одинаковым приоритетом только что активированные правила размещаются с использованием стратегии МЕА. Основное отличие стратегии МЕА от LEX в том, что в стратегии МЕА не производится сортировка образцов, активировавших правило. Сравниваются только временные теги первых образцов двух активаций. Активация с большим тегом помещается в план решения задачи перед активацией с меньшим. Если обе активации имеют одинаковые временные теги, ассоциированные с первым образцом, то для определения размещения активации в плане решения задачи используется стратегия LEX. Так же как и в стратегии LEX, условный элемент `not` имеет псевдовременной тег [5, с. 45, раздел 5.3.6].

Случайная стратегия

Каждой активации назначается случайное число, которое используется для определения местоположения среди активаций с одинаковым приоритетом. Это случайное число сохраняется при смене стратегий, таким образом тот же порядок воспроизводится при следующем выборе случайной стратегии [5, с. 45, раздел 5.3.7].

Задачи и порядок выполнения работы

Синтаксис правил

Левая часть правил содержит список условных элементов, которые должны удовлетворяться, для того чтобы правило было помещено в план решения задачи. Существует

семь типов условных элементов, используемых в левой части правил: образцы, `test`, `and`, `not`, `exists`, `forall` и `logical`.

Образцы — наиболее часто используемый условный элемент. Он содержит ограничения, которые служат для определения, удовлетворяет ли какой-нибудь элемент данных (факт или объект) образцу. Условие `test` используется для оценки выражения как части процесса сопоставления образов. Условие `and` применяется для определения группы условий, каждое из которой должно быть удовлетворено. Условие `or` — для определения одного условия из некоторой группы, которое должно быть удовлетворено. Условие `not` — для определения условия, которое не должно быть удовлетворено. Условие `exists` — для проверки наличия по крайней мере одного совпадения факта (или объекта) с некоторым заданным образцом. И наконец, условие `logical` позволяет выполнить добавление фактов и создание объектов в правой части правила, связанных с фактами и объектами, совпавшими с заданным образцом в левой части правила (поддержка достоверности фактов в базе знаний).

Переменные

В CLIPS имеется два различных групповых символа, которые используются для сопоставления полей в образцах. CLIPS интерпретирует эти групповые символы как место для подстановки некоторых частей данных, удовлетворяющих образцам. Групповой символ для простого поля записывается с помощью знака «?», который соответствует одному любому значению, сохраненному в заданном поле. Групповой символ составного поля записывается с помощью знака «\$?» и соответствует, возможно, пустой последовательности полей, сохраненной в составном поле. Групповые символы для простых и составных полей могут комбинироваться в любой последовательности. Нельзя использовать групповой символ составного поля для простых полей. По умолчанию не заданный в образце простой слот шаблона или объекта сопоставляется с неявно заданным групповым символом для простого поля. Аналогично не заданный в образце составной слот сопоставляется с неявно заданным групповым символом для составного поля [5, с. 52–54, раздел 5.4.1.3].

Групповые символы заменяют любые поля образца и могут принимать какие угодно значения этих полей. Значение поля может быть связано с переменными для последующего сопоставления, отображения и других действий. Это выполняется с помощью применения имени переменной следующим непосредственно после группового символа.

Имя переменной должно быть значением типа `SYMBOL` и обязательно начинаться с буквы. В имени переменной не разрешается использовать кавычки, т.е. строка не может использоваться как имя переменной или её часть.

Правила сопоставления образцов при использовании переменных в ограничениях образца аналогичны правилам, использующимся для групповых символов. В момент первого появления имени переменной она ведет себя так же, как и соответствующий групповой символ. В этот момент CLIPS вызывает значения поля с заданной переменной. Эта связь будет действовать только в рамках правила, в котором она возникла. Каждое правило имеет свой собственный список имен переменных со значениями, связанными с ними, эти переменные локальны для правил.

Связанные переменные могут быть использованы во внешних функциях. Символ «\$» имеет особое значение в левой части правил — этот оператор отображает, что некоторая, возможно пустая, последовательность полей требует сопоставления. В правой части правила символ «\$» ставится перед переменной для обозначения того, что перед использованием переменной в качестве аргумента функции необходимо раскрыть последовательность полей, содержащихся в переменной. Таким образом, при использовании переменных в качестве параметров функций (как в левой, так и правой части правил) перед именем переменной, содержащей значение составного поля, не должен стоять символ «\$» (за исключением случаев, когда требуется раскрыть последовательность полей). При использовании переменной, содержащей значение составного поля, в других случаях перед её именем должен стоять символ «\$». Нельзя применять переменную составного поля при операциях с простым полем образца шаблона или объекта.

Условные элементы

Как уже говорилось ранее, левая часть правил содержит список условных элементов, которые должны удовлетворяться, для того чтобы правило было помещено в план решения задачи. Рассмотрим эти условные элементы более подробно.

Условный элемент test

Условный элемент test предоставляет возможность наложения дополнительных ограничений на слоты фактов или объектов. Элемент test удовлетворяется, если вызванная в нём функция возвращает значение не-FALSE. Как и в случае предикатных ограничений образца, в условном элементе test можно использовать переменные, уже связанные со своими значениями. Внутри элемента test могут быть выполнены различные логические операции, например сравнения переменных [5, с. 67–68, раздел 5.4.2].

Выражение test вычисляется каждый раз при удовлетворении других условных элементов. Это означает, что условный элемент test будет вычислен больше одного раза, если обрабатываемое выражение может быть удовлетворено более чем одной группой данных. Использование условного элемента test может стать причиной автоматического добавления правилу некоторых условных выражений. Кроме того, CLIPS может автоматически переупорядочивать условные элементы test.

Приведём пример использования логического элемента test:

```

(defrule DifBiggerThan3_l "or equal"
  (data ?x)
  (data ?y)
  (test (≥ (abs (- ?x ?y)) 3))
  ⇒
  (println "abs(" ?x " - " ?y ") ≥ 3")
  (println "abs(" ?x " - " ?y ") = " (abs (- ?x ?y)) " ≥ 3")
)

(def facts data_for_test
  (data 1)
  (data 4)
  (data 6)
  (data 11)
)

```

Приведённое выше правило находит пару фактов data, причём разница между значениями первых полей этих фактов должна быть больше или равной 3.

Условный элемент or

Условный элемент or позволяет активировать правило любым из нескольких заданных условных элементов. Если какой-нибудь из условных элементов, объединенных с помощью or, удовлетворен, то и все выражение or считается удовлетворенным. В этом случае, если все остальные условные элементы, входящие в левую часть правила (но не входящие в or), также удовлетворены, правило будет активировано [5, с. 68–69, раздел 5.4.3].

Условный элемент or может объединять любое количество элементов.

Приведём пример использования логического элемента or:

```

(defrule System-Fault
  (error-status unknown)
  (or
    (temp ?x & : (integerp ?x) & : (> ?x 250))
    (resistor broken)
    (pump_status off)
  )
  ⇒
  (println "The system has a fault.")
)

```

Данное правило сообщит о поломке системы, если в списке фактов будет присутствовать факт (error-status unknown) и один из фактов (temp ?x) (причём если значение ?x целого типа и больше 20), (resistor broken) или (pump_status off).

Условный элемент and

Все условные элементы в левой части правил CLIPS объединены неявным условным элементом and. Это означает, что все условные элементы, заданные в левой части, должны удовлетвориться, для того чтобы правило было активировано. С помощью явного применения условного элемента and можно смешивать различные условия and и or и группировать элементы так, как этого требует логика правил. Условие and удовлетворяется, только если условия внутри явного and удовлетворены. В случае если остальные условия в левой части правила также истинны, правило будет активировано. Элемент and может объединять любое число условных элементов [5, с. 69–70, раздел 5.4.4].

Приведём пример использования логического элемента and:

```
(defrule System-Flow
  (error-status confirmed)
  (or
    (and (temp high) (valve closed))
    (and (temp low) (valve open))
  )
  =>
  (println "The system is having a flow problem.")
)
```

Данное правило сообщит о поломке системы, если в списке фактов будет присутствовать факт (error-status unknown) и одна из следующих пар фактов (temp high) (valve closed) или (temp low) (valve open).

Условный элемент not

Иногда важнее отсутствие информации, а не её присутствие, т.е. возникают ситуации, когда необходимо запустить правило, если образец или другой условный элемент не удовлетворяется (например, факт не существует). Условный элемент not предоставляет эту возможность. Элемент not удовлетворяется, только если условный элемент, который он содержит, не удовлетворяется [5, с. 70–72, раздел 5.4.5].

Условный элемент not может отрицать только одно выражение. Несколько условных элементов нужно отрицать с помощью нескольких элементов not. Тщательно следите за комбинациями not с or или and; результат не всегда очевиден.

Приведём пример использования логического элемента not:

```
(defrule double-pattern
  (color red)
  (not (color red ?x ?x))
  =>
  (println "No patterns with red green green!")
)
```

Данное правило проверяет наличие фактов с одинаковыми вторым и третьим цветами и выводит сообщение об их отсутствии, если условие выполняется.

Условный элемент exists

Условный элемент exists позволяет определить, существует ли хотя бы один набор данных (фактов или объектов), которые удовлетворяют условным элементам, заданным внутри элемента exists [5, с. 73–75, раздел 5.4.6].

CLIPS автоматически заменяет exists двумя последовательными условными элементами not.

Приведём пример использования логического элемента exists:

```

(deftemplate hero
  (multislot name)
  (slot status (default unoccupied))
)
(deffacts goal-and-heroes ; newline
  (goal save-the-day)
  (hero (name Death Defying Man))
  (hero (name Stupendous Man))
  (hero (name Incredible Man))
)
(defrule save-the-day
  (goal save-the-day)
  (exists (hero (status unoccupied)))
  =>
  (println "The day is saved.")
)

```

Данная программа определяет шаблон, имеющий составное поле с именем и простое поле, содержащее статус «не занят» по умолчанию. Конструктор deffacts определяет трех ничем не занятых героев и текущую цель — спасение мира. Правило проверяет, есть ли в данный момент эта цель, и в случае положительного ответа проверяет, если ли какой-нибудь еще не занятый герой. Если все условные элементы правила удовлетворены, оно сообщает, что мир спасен. Обратите внимание: несмотря на то что у нас все три героя не заняты, правило будет активировано только один раз.

Условный элемент forall

Условный элемент forall позволяет определить, что некоторое заданное условие выполняется для всех заданных условных элементов [5, с. 75–77, раздел 5.4.7].

CLIPS автоматически заменяет forall комбинацией условных элементов not и and.

Приведём пример использования логического элемента forall:

```

(defrule all-students-passed
  (forall
    (student ?name)
    (reading ?name)
    (writing ?name)
    (arithmetic ?name)
  )
  =>
  (println "All students passed.")
)

```

Правило all-students-passed определяет, прошли ли все студенты чтение, письмо и арифметику, используя условие forall.

Заметьте, что данное правило удовлетворяется, пока нет ни одного студента. При добавлении факта (student Bob) правило перестает удовлетворяться, так как нет фактов, подтверждающих, что Bob прошел все необходимые предметы. Правило не начнет удовлетворяться и после добавления фактов (reading Bob) и (writing Bob). А вот после добавления факта (arithmetic Bob) правило будет активировано и сможет вывести на экран соответствующую запись. Если добавить факт (student John), правило опять перестает удовлетворяться, так как один из студентов (John) не прошел все необходимые предметы (точнее, нет данных, что он их прошел). Используя условный элемент exists, вы без труда сможете изменить это правило так, чтобы оно не выполнялось в случае отсутствия студентов.

Условный элемент `logical`

Условный элемент `logical` предоставляет механизм поддержки достоверности для созданных правилом данных (фактов или объектов), удовлетворяющих образцам. Данные, созданные в правой части правила, могут иметь логическую зависимость от данных, удовлетворивших образцы в левой части правила. Такая зависимость называется логической поддержкой. Данные могут зависеть от группы данных или нескольких групп данных, удовлетворивших одно или несколько правил. Если удаляются данные, которые поддерживают некоторые другие данные, то зависимые данные также автоматически удаляются [5, с. 77–83, раздел 5.4.8].

Если некоторые данные созданы без логической поддержки (например, с помощью конструкторов (`deffacts`), (`definstances`) или команды (`assert`), введенной пользователем или вызванной в правой части правила), то считается, что они имеют безусловную поддержку. Безусловная поддержка удаляет все присутствующие в данный момент условные поддержки этих данных (но не удаляет сами данные). Дальнейшая логическая поддержка для данных с безусловной поддержкой игнорируется. Удаление правила, которое вызвало логическую поддержку для данных, удаляет логическую поддержку, сгенерированную этим правилом (но не удаляет данные, если у них еще есть логическая поддержка, сгенерированная другим правилом).

Приведём пример использования логического элемента `forall`:

```
(defrule rule7
  (logical (book (name Peace_and_war_1) (author "Tolstoy")))
  =>
  (assert (book (name Peace_and_war_2) (author "Tolstoy")))
)
```

Несколько примеров работы с правилами

Приведём несколько примеров работы с правилами.

Пример 1.

```
(defrule rule1
  (book (name ?x) (author ?y))
  =>
  (println ?y " write " ?x)
)
```

Данное правило находит все факты шаблона `book`, которые имеют слоты `name` и `author` с любыми значениями, и выводит эти значения на экран.

Пример 2.

```
(defrule rule2
  (book (name ?x) (author ?y))
  (book (name ?z&~?x) (author ?y))
  =>
  (println ?x " and " ?z " was written by " ?y)
)
```

Данное правило находит в списке фактов те пары фактов шаблона `book`, которые имеют

разные слоты `name` (о чем говорит запись `?z & ?x`, которая означает, что переменная `?z` не равна переменной `?x`) и одинаковые слоты `author`, и выводит значения этих переменных на экран. В данном примере было использовано ограничение вида «&» (логическое и). Ограничение «&» удовлетворяется, если два соседних ограничения удовлетворяются.

CLIPS предоставляет три связывающих ограничения, предназначенных для объединения отдельных ограничений и переменных в единое целое: «&» (логическое и), «|» (логическое или) и «~» (логическое не). Ограничение «|» удовлетворяется, если любое из двух соседних ограничений удовлетворяется. Ограничение «~» удовлетворяется, если следующее за ним ограничение не удовлетворяется. Связывающие ограничения могут комбинироваться почти произвольным образом и в любом количестве. Ограничение «~» имеет наивысший приоритет, далее следуют «&» и «|». В случае одинакового приоритета ограничение вычисляется слева направо. Существует одно исключение из правил приоритета, которое применяется при связывании переменных. Если первое ограничение — это переменная и за ней следует «&», то переменная является отдельным ограничением.

Ограничение «&» обычно служит только для объединения с другими ограничениями или связывания переменных. Заметьте, что связывающие ограничения могут использовать связанные переменные и в то же время сами производить связывание переменной со значением некоторого поля. Если имя переменной встретилось в первый раз, то для ограничения будут использоваться остальные члены условного элемента, а переменная будет связана с соответствующим значением поля. Если переменная уже была связана, то её значение работает как дополнительное ограничение для данного поля.

Некоторые действия в правой части правил, такие как `retract` и `unmake-instance`, оперируют с фактами или объектами, участвующими в левой части. Для того чтобы определить, какой факт или объект будет изменяться, необходимо присвоить переменной адрес конкретного факта или объекта. Присваивание адресов происходит в левой части правила и полученное значение называется адресом образца (`pattern-address`).

Стрелка влево «<-» — необходимая часть синтаксиса. Переменная, связанная с адресом факта или объекта, может сравниваться с другой переменной или использоваться внешней функцией. Переменная, связанная с адресом факта или объекта, может быть также использована для последующего ограничения полей в образце условного выражения. Однако нельзя связывать переменную в условном выражении `not`.

Пример 3.

```
(defrule rule3
  ?x <- (book (pages ?z))
  (test (<= ?z 200))
  =>
  (retract ?x)
)
```

Данное правило присваивает переменной `?x` адрес факта шаблона `book` с количеством страниц `?z`. С помощью команды (`test`) выполняется проверка, является ли количество страниц меньшим или равным `200`. При выполнении данного условия факт удаляется из рабочей памяти при помощи команды (`retract`) [5, с. 249, раздел 12.9.2].

Пример 4.

```
(defrule rule4 ⇒  
  (assert (book (name Assembler) (author "Zubkov") (year 2003)))  
)
```

Данное правило не имеет левой части. Оно добавляет в рабочую память факт (book (name Assembler) (author "Zubkov") (year 2003)).

Пример 5. Рассмотрим пример решения известной задачи построения башни из блоков. Программа переключается между двумя задачами: выбором очередного блока и установкой блока в башню.

В разделе шаблонов блоки представлены объектами, обладающими такими свойствами, как цвет, размер и положение. Если положение блока не определено, предполагается, что он находится в куче блоков (heap), еще не уложенных в башню. Шаблон on предоставляет в наше распоряжение средство, позволяющее описать размещение блоков одного (upper) на другом (lower). Информацию о текущей фазе решения проблемы (поиск или установка) несёт шаблон goal.

```
example5.clp  
; СТРАТЕГИЯ РАЗРЕШЕНИЯ КОНФЛИКТОВ  
(set-strategy mea)  
; Шаблоны  
; Объект block характеризуется цветом,  
; размером и положением.  
(deftemplate block  
  (slot color (type SYMBOL))  
  (slot size (type INTEGER))  
  (slot place (type SYMBOL) (default heap))  
)  
; Вектор on указывает, что блок «upper»  
; находится на блоке «lower».  
(deftemplate on  
  (slot upper (type SYMBOL))  
  (slot lower (type SYMBOL))  
  (slot place (type SYMBOL))  
)  
; Текущая цель (goal) может быть либо «найти» (find),  
; либо «уложить» (build).  
(deftemplate goal  
  (slot task (type SYMBOL))  
)  
; ИНИЦИАЛИЗАЦИЯ  
; Имеются три блока разных цветов и размеров.  
; Предполагается, что они находятся в куче.  
(defacts the-facts  
  (block (color red) (size 10))  
  (block (color yellow) (size 20))  
  (block (color blue) (size 30))  
)  
; ПРАВИЛА  
; Задать первую цель: найти первый блок.  
(defrule begin  
  ⇒  
  (assert (goal (task find)))  
)  
; Взять самый большой блок в куче (heap).  
(defrule pick-up  
  ?my-goal <- (goal (task find))  
  ?my-block <- (block (size ?S1) (place heap))  
  (not (block (size ?S2 & :(> ?S2 ?S1)) (place heap)))  
  ⇒
```



```

    (modify ?my-block (place hand))
    (modify ?my-goal (task build))
  )
; Установить первый блок в основание башни (tower).
; Этот блок не имеет под собой никакого другого.
(defrule place-first
  ?my-goal <- (goal (task build))
  ?my-block <- (block (place hand))
  (not (block (place tower)))
  =>
  (modify ?my-block (place tower))
  (modify ?my-goal (task find))
)
; Установить последующие блоки на тот,
; что лежит в основании башни.
(defrule put-down
  ?my-goal <- (goal (task build))
  ?my-block <- (block (color ?C0) (place hand))
  (block (color ?C1) (place tower))
  (not (on (lower ?C1)))
  =>
  (modify ?my-block (place tower))
  (assert (on (upper ?C0) (lower ?C1)))
  (modify ?my-goal (task find))
)
; Если в куче больше нет блоков, прекратить процесс,
(defrule stop
  ?my-goal <- (goal (task find))
  (not (block (place heap)))
  =>
  (retract ?my-goal)
)

(reset)
(facts)
(run)
(facts)
(exit)

```

Задание на лабораторную работу

В среде CLIPS создать шаблон согласно варианту задания (и возможно ещё несколько из той же предметной области) и несколько неупорядоченных фактов.

Разработать 20 правил к сформулированным шаблонам, используя команды и все логические операторы из теоретической части.

Задать стратегию разрешения конфликтов.

Продемонстрировать план решения и его результат.

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 функциональные составные части правил.
2. Приведите синтаксис создания правила.
3. Каков синтаксис функции для удаления правил?
4. Как запустить правила на выполнение?
5. Как можно просмотреть план решения?
6. Что такое приоритет правила и как его задать?
7. Как просмотреть наборы данных, способных активировать правило?
8. Зачем нужны различные стратегии разрешения конфликтов?
9. Перечислите и поясните доступные стратегии разрешения конфликтов
10. Поясните понятие определённости правила.
11. Перечислите типы условных элементов, используемых в левой части правил.
12. Поясните назначение переменных в CLIPS.
13. Что такое логическое связывание фактов и зачем оно нужно?
14. Как оперировать в правиле данными, участвующими в левой части правила?

Форма отчёта по лабораторной работе

На выполнение лабораторной работы отводится 3 занятия (6 академических часов: 5 часов на выполнение и сдачу лабораторной работы и 1 час на подготовку отчёта).

Номер варианта студенту выдаётся преподавателем.

Отчёт на защиту предоставляется в печатном виде.

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