Представление и обработка продукций в CLIPS.

Представление правил в базе знаний. Типы условных элементов.

Правила являются основным способом представления знаний в CLIPS.

Для задания правил используется конструкция `defrule` со следующим синтаксисом:

```
(defrule <имя_правила> ["<комментарий>"]
 [<объявление>]
<условный элемент>*; Левая часть правила (антецедент)
=>
<действие>*) ;Правая часть правила (консеквент)
```

где имя_правила> – идентификатор символьного типа, уникальный для данной группы правил;

<комментарий> – необязательное поле комментария;

<объявление> – необязательный элемент, позволяющий задавать дополнительные свойства правила (например, значимость) с помощью оператора declare;

<условный элемент>* – произвольная последовательность условных элементов;

<действие>* – произвольная последовательность действий.

Пример задания правила:

```
(defrule R1
 (days 2)
 (works 100)
 =>
 (printout t crlf "Свободного времени нет" crlf)
 (assert (freetime "no"))).
```

В левой части правило содержит два условных элемента, сопоставляемых с упорядоченными фактами, а в правой – две команды.

В команде вывода сообщений `printout` параметр `t` задает стандартный режим вывода, а `crlf` – символ возврата и перевода курсора на новую строку.

Команда `assert` добавляет в факт-список новый упорядоченный факт.

Антецедент правила состоит из последовательности условных элементов (УЭ).

Если все УЭ правила удовлетворяются при текущем состоянии базы данных, то правило помещается в список готовых к выполнению правил – агенду.

В CLIPS используется шесть типов условных элементов:

- УЭ-образцы (Pattern Conditional Elements);
- УЭ-проверки (Test Conditional Elements);
- УЭ “ИЛИ” (Or Conditional Elements);
- УЭ “И” (And Conditional Elements);
- УЭ “НЕ” (Not Conditional Elements);
- УЭ “Существует” (Exists Conditional Elements);
- УЭ “Для всех” (Forall Conditional Elements);
- логические УЭ (Logical Conditional Elements).

УЭ-образец состоит из совокупности ограничений на поля, масок (wildcards) полей и переменных, используемых при сопоставлении УЭ с образцом – фактом или экземпляром объекта.

УЭ-образец удовлетворяется каждой сущностью, которая удовлетворяет его ограничениям.

Ограничения на поля используются для проверки одного поля или слота факта либо экземпляра объекта.

Ограничение на поле может состоять из единственного литерала или из нескольких связанных ограничений.

В УЭ-образцах используются следующие конструкции:

- литеральные ограничения (Literal Constraints);
- одно и многоместные маски (Single- and Multifield Wildcards);
- одно и многоместные переменные (Single- and Multifield Variables);
- ограничения со связками (Connective Constraints);
- предикатные ограничения (Predicate Constraints);
- ограничения возвращаемым значением (Return Value Constraints).

Литеральное ограничение не содержит переменных и масок, а задает точное значение (константу целого, вещественного, символьного или строкового типа, либо имя экземпляра), которое должно сопоставляться с полем образца.

Все литеральные ограничения должны совпадать с соответствующими полями сопоставляемой сущности.

Упорядоченный УЭ-образец содержит только литералы и имеет следующий синтаксис:

`(<constant-1> ... <constant-n>)`. Например, `(data 1 one “two”)`.

Пример УЭ-образца для неупорядоченных фактов:

```
(person (name Bob) (age 20)).
```

Одно- и многоместные маски позволяют игнорировать некоторые поля в процессе сопоставления.

Одноместная маска обозначается символом “?” и сопоставляется с любым значением, занимающим точно одно поле в соответствующем месте сопоставляемой сущности.

Многоместная маска, обозначается парой символов “$?” и сопоставляется с любыми значениями, занимающими произвольное число полей в сопоставляемой сущности.

Маски могут использоваться в одном образце в любых комбинациях.

Не допускается лишь использование многоместной маски в одноместном слоте (содержащем единственное поле) неупорядоченных фактов или объектов.

Например, УЭ `(data ? blue red $?)` будет сопоставляться со следующими упорядоченными фактами:

```
(data 1 blue red),
(data 5 blue red 6.9 "avto"),
```

но не будет сопоставлен со следующими фактами:

```
(data 1.0 blue "red"),
(data 1 blue).
```

Одно- и многоместные переменные используются для запоминания значений полей, с целью их дальнейшего использования в других условных элементах антецедента или в операторах консеквента правила.

Одноместные переменные начинаются с символа “?”, за которым следует символьное значение, начинающееся с буквы.

Например: ?x, ?var, ?age.

Многоместные переменные начинаются с префикса “$?”, за которым также следует символьное значение, начинающееся с буквы.

Например: $?y, $?zum.

В именах переменных не допускается использование кавычек.

При первом появлении переменная работает так же, как в маске, т.е. связывается с любым значением в данном поле(ях).

Последующие появления переменной требуют, чтобы поле сопоставлялось со связанным значением переменной.

Имена переменных являются локальными в пределах каждого правила.

Пусть имеется три факта:

```
(data 2 blue green),
(data 1 blue),
(data 1 blue red)
```

и правило:

```
(defrule find-data-1
(data ?x ?y ?z)
=>
(printout t ?x " : " ?y " : " ?z crlf))
```

УЭ данного правила будет сопоставляться с первым и третьим фактом, поэтому в результате срабатывания правила будет выведено:

```
1 : blue : red
2 : blue : green
```

Ограничения со связками используются для связывания индивидуальных ограничений и переменных друг с другом с помощью связок & (“и”), | (“или”) и ~ (“не”), используемых в традиционном смысле.

Старшинство операций обычное, за исключением случая, когда первым ограничением является переменная, за которой следует связка &.

В этом случае первая переменная трактуется как отдельное ограничение, которое также должно удовлетворяться.

Например, ограничение ?x&red|blue трактуется как ?x&(red|blue), а не как (?x&red)|blue.

Пример правила с УЭ, содержащим ограничения со связками:

```
(defrule r1
 (data (value ?x&~red&~green))
 =>
 (printout t "slot value = " ?x crlf)).
```

Например, для факта `(data (value blue))` это правило выведет сообщение:

```
 slot value = blue.
```

Предикатное ограничение позволяет ограничить значение поля, основываясь на истинности булевого выражения.

Для этого используется предикатная функция, которая вызывается в процессе сопоставления с образцом и возвращает в случае неудачи символьное значение FALSE.

Если возвращается значение FALSE, то ограничение не удовлетворяется, в противном случае оно удовлетворяется.

Предикатное ограничение задается с помощью символа “:”, за которым следует вызов предикатной функции.

Данное ограничение может использоваться в комбинации с ограничением со связками, а также связанной переменной.

В последнем случае переменная сначала связывается некоторым значением, а затем к ней применяется предикатное ограничение.

В таком варианте предикатные ограничения часто применяются для проверки типов данных.

При этом в качестве предикатных функций используются встроенные функции CLIPS, в частности:

`(numberp <выражение>)` – функция возвращает значение TRUE, если `<выражение>` имеет числовой тип (`integer` или `float`), в противном случае возвращается символ FALSE;

`(floatp <выражение>)` – функция возвращает значение TRUE, если `<выражение>` имеет тип `float`, иначе возвращается символ FALSE;

(integerp <выражение>) – функция возвращает значение TRUE, если <выражение> имеет тип integer, иначе – символ FALSE;

(symbolp <выражение>) – функция возвращает значение TRUE, если <выражение> имеет тип symbol, иначе – символ FALSE;

(stringp <выражение>) – функция возвращает значение TRUE, если <выражение> имеет тип string, иначе – символ FALSE;

Пусть заданы факты: `((data 1) (data 2) (data red))`. Тогда для определения значений числового типа можно использовать следующий УЭ

```
(data ?x&:(numberp ?x)),
```

который сопоставится с первыми двумя фактами.

Тот же результат может быть получен использованием УЭ `(data ?x&~:(symbolp ?x))`.

Ограничение возвращаемым значением использует в качестве ограничения значение, возвращаемое внешней функцией.

Эта функция вызывается непосредственно из УЭ-образца с использованием следующего синтаксиса:

```
=<вызов-функции>
```

Возвращаемое функцией значение одного из базовых типов подставляется непосредственно в УЭ-образец на позицию, из которой была вызвана функция, и используется далее как литеральное ограничение.

Например, следующее правило, содержащее УЭ-образец с ограничением возвращаемым значением**:**

```
(defrule twice
 (data (x  ?x) (y  =(* 2 ?x)))
 =>...)
```

будет сопоставляться со всеми неупорядоченными фактами, у которых значение в слоте `y` равно удвоенному значению слота `x`.

Условный элемент-проверка имеет следующий синтаксис:

```
(test <function-call>).
```

УЭ-проверка удовлетворяется, если функция, вызываемая из него, возвращает значение, отличное от FALSE.

Как и в предикатном ограничении, можно сравнивать уже связанную некоторым значением переменную, используя любые функции (алгебраическое и логическое сравнение, вызов внешних функций).

В УЭ-проверку могут быть встроены внешние функции любого вида.

В следующем правиле проверяется, что модуль разности двух чисел не меньше трех:

```
(defrule example-1
 (data ?x)
 (value ?y)
 (test (>= (abs (- ?y ?x)) 3))
 =>...)
```

Условный элемент “ИЛИ” задается следующей конструкцией:

```
(or <УЭ-1> ... < УЭ-N>)
```

и удовлетворяется, если удовлетворяется хотя бы один УЭ внутри этой конструкции.

Наличие такого УЭ позволяет сократить число правил, т.к. тоже самое можно было бы записать множеством правил с одинаковой правой частью.

При этом правило будет активизироваться несколько раз, по числу удовлетворяемых комбинаций.

Например, правило

```
(defrule r1
 (man stud)
 (or (spec computeer) (age 20))
 =>...)
```

эквивалентно двум следующим:

```
(defrule r2
 (man stud)
 (spec computeer)
 =>...)
(defrule r3
 (man stud)
 (age 20)
 =>...)
```

Условный элемент “И” задается следующей конструкцией:

```
(and <УЭ-1> ... < УЭ-N>)
```

и удовлетворяется, если удовлетворяются все УЭ внутри этой конструкции.

В CLIPS все УЭ в антецедентах правил неявно объединены по “И”, однако использование УЭ “И” для явного задания конъюнктивной связи позволяет комбинировать УЭ “И” и УЭ “ИЛИ” в любых сочетаниях.

Пример такой комбинации приведен в следующем правиле:

```
(defrule r1
 (sys-mode search)
 (or (and (distance high) (resol little))
    (and (distance low) (resol big)))
 =>...)
```

Условный элемент “НЕ” задается следующей конструкцией:

```
(not <УЭ>)
```

и удовлетворяется, если содержащийся внутри него УЭ не удовлетворяется.

Предварительно связанные переменные могут использоваться внутри УЭ “НЕ” как свободные.

Однако, переменные, которые связываются внутри УЭ-“НЕ”, могут использоваться только в этом образце.

Следующее правило ищет факты, у которых второе поле – `red`, а третье и четвертое поля не совпадают:

```
(defrule not-double
 (not (data red ?x ?x))
 =>...)
```

Условный элемент “Существует” имеет следующий синтаксис:

```
(exists <УЭ-1> ... <УЭ-N>)
```

и используется для определения, удовлетворяется ли группа УЭ, специфицированных внутри условного элемента “Существует”, хотя бы одним набором образцов-сущностей в базе данных.

Например, правило:

```
(defrule example
 (exists (a ?x) (b ?x))
 =>...)
```

будет активизировано, если в базе данных имеется хотя бы одна пара фактов, содержащих в первых полях значения `a` и `b`, а вторые поля которых совпадают.

Условный элемент “Для всех” имеет следующий синтаксис:

```
(forall <УЭ-1> ... <УЭ-N>)
```

и используется для определения, удовлетворяется ли группа УЭ, специфицированных внутри условного элемента “Для всех”, для каждого появления УЭ-1.

Например, следующее правило активизируется, если каждый студент научился чтению, письму и арифметике:

```
(defrule all-students-passed
 (forall  (student ?name)
      (reading ?name)
      (writing ?name)
      (arithmetic ?name))
 =>
 (printout t "All students passed." crlf))
```

Логические условные элементы обеспечивают возможность *поддержания истинности* различных сущностей (фактов и экземпляров), создаваемых правилами, использующими логические УЭ.

Сущность-образец, создаваемая оператором правой части правила, может быть сделана логически зависимой от сущностей-образцов, сопоставляемых с *логическим* УЭ в антецеденте правил.

Сущности-образцы, сопоставляемые с логическими УЭ в антецеденте правил обеспечивают *логическую поддержку* фактам и экземплярам, создаваемым в консеквенте правила.

Сущность-образец может логически поддерживаться несколькими группами сущностей-образцов из одного или различных правил.

Если любая поддерживающая сущность удаляется из группы поддерживающих сущностей и не существует никаких других поддерживающих групп, то поддерживаемая сущность удаляется из рабочей памяти.

Сущность-образец имеет безусловную поддержку, если она создается без логической поддержки, т.е. с помощью конструкций `deffacts`, `definstaces`, с помощью высокоуровневых команд или правил без логической поддержки образцов.

Безусловная поддержка сущности удаляет всю логическую поддержку (без удаления самой сущности), при этом дальнейшая логическая поддержка безусловно поддерживаемой сущности игнорируется.

Удаление правила, генерировавшего для сущности логическую поддержку, удаляет логическую поддержку, генерируемую этим правилом, но не влечет удаления сущности, даже если для нее не осталось логической поддержки.

Логический УЭ имеет следующий синтаксис:

```
(logical <УЭ>+)
```

Логический УЭ группирует образцы точно так же, как УЭ “И” и может использоваться в сочетании с УЭ “И”, УЭ “ИЛИ” и УЭ “НЕ”.

Однако логические УЭ можно применять только в первых образцах правила.

Например, следующее правило допустимо:

```
(defrule ok
 (logical (a))
 (logical (b))
 (c)
=>
 (assert (d)))
```

Вместе с тем, следующее правило является недопустимым:

```
(defrule not-ok-1
 (logical (a))
 (b)
 (logical (c))
 =>
 (assert (d)))
```