Дефиниране на манипулатори на съобщения

Обектите се манипулират чрез изпращане към тях на съобщения посредством функцията **`send`**. Резултатът от дадено съобщение може да бъде или подходяща върната стойност, или полезен страничен ефект. С помощта на конструкцията **`defmessage-handler`** се определя поведението на обектите от даден клас по отношение (в отговор) на дадено съобщение. Реализацията на едно съобщение се осъществява чрез т. нар. ***манипулатори на съобщение*** (***message handlers***).

Типове манипулатори на съобщения

primary: извършва основната работа за съобщението;

before: извършва подготвителни (предварителни) действия преди да бъде изпълнен ***primary*** манипулаторът;

after: извършва допълнителни (заключителни) действия след изпълнението на ***primary*** манипулатора;

around: дефинира подходяща среда за изпълнението на останалите манипулатори.

Ред на изпълнение на различните типове манипулатори: ***around***, ***before***, ***primary***, ***after***.

Синтаксис на дефиницията на манипулатор на съобщение

```
(defmessage-handler
   <class-name> <message-name>
   [<handler-type>] [<comment>]
   (<parameter>* [<wildcard-parameter>])
   <action>*)
<handler-type> ::=
  around | before | primary | after
<parameter> ::= <single-field-variable>
<wildcard-parameter> ::=
  <multifield-variable>
```

Всеки манипулатор на съобщение се определя еднозначно от класа, името и типа си. Манипулаторите на съобщения никога не се извикват директно. Когато потребителят изпрати съобщение към даден обект, CLIPS подрежда приложимите манипулатори на съобщения, прикрепени към класа/класовете на обекта, и след това ги изпълнява.

Пример

```
> (clear)
> (defclass A (is-a USER) (role concrete))
> (defmessage-handler A delete before ()
    (printout t "Deleting an instance of
       class A ... " crlf))
> (defmessage-handler USER delete after ()
    (printout t "System completed
       deletion of an instance." crlf))
> (watch instances)
> (make-instance a of A)
```

```
==> instance [a] of A
[a]
> (send [a] delete)
Deleting an instance of class A ...
<== instance [a] of A
System completed deletion of an instance.
TRUE
> (unwatch instances)
>
```

Параметри на манипулаторите на съобщения

Явни параметри

- **Задължителни:** в синтактичната дефиниция са означени като **`<parameter>*`**;
- **Незадължителен:** в синтактичната дефиниция е означен като **`[<wildcard-parameter>]`**. Служи за задаване на възможност за използване на променлив брой фактически параметри.

Active instance параметър

Използва се за означаване на екземпляра, към който е изпратено съобщението (екземпляра, върху който се изпълнява съобщението).

Всички манипулатори на съобщения имат по един неявен параметър, наречен **`?self`**, който се свързва с активния за съобщението екземпляр. Името на този параметър е резервирано и не може да се среща в явен вид сред имената на параметрите в дефиницията на манипулатора. Променливата **`?self`** не може да се свързва явно в тялото на манипулатора.

Пример

```
> (clear)
> (defclass A (is-a USER) (role concrete))
> (make-instance a of A)
[a]
> (defmessage-handler A print-args
    (?a ?b $?c)
    (printout t (instance-name ?self)
       " " ?a " " ?b " and " (length$ ?c)
          " extras: " ?c crlf))
```

```
> (send [a] print-args 1 2)
[a] 1 2 and 0 extras: ()
> (send [a] print-args a b c d)
[a] a b and 2 extras: (c d)
>
```

Действия на манипулаторите на съобщения

Тялото на един манипулатор на съобщение съдържа поредица от действия, които се изпълняват последователно, когато се извика този манипулатор. Стойността, която връща манипулаторът, съвпада с оценката на последното действие от тялото.

Действията на манипулатора могат директно да манипулират слотовете на активния екземпляр. Има няколко функции, които оперират неявно с активния екземпляр (например **`dynamic-get`** и **`dynamic-put`**) и могат да бъдат извиквани само в тялото на манипулатор на съобщение.

За достъп до слотовете на активния екземпляр може да се използва означението **`?self:<slot-name>`**.

Функцията **`bind`** може да се използва за задаване на стойност(и) на слот на активния екземпляр:

```
(bind ?self:<slot-name> <value>*)
```

Пример 1

```
> (clear)
> (defclass A (is-a USER) (role concrete)
    (slot s1 (default 1))
    (slot s2 (default 2)))
> (defmessage-handler A print-all-slots ()
    (printout t ?self:s1 " " ?self:s2
              crlf))
> (make-instance a of A)
[a]
> (send [a] print-all-slots)
1 2
>
```

Пример 2

```
> (defmessage-handler A set-s1 (?value)
     (bind ?self:s1 ?value))
> (send [a] set-s1 34)
34
>
```

Разгледаните средства за достъп до слот, както и стандартните манипулатори за достъп **`get-<slot-name>`** и **`put-<slot-name>`**, ***се свързват статично със съответния слот на класа*** по време на дефинирането на манипулатора на съобщението (или класа). Затова трябва да се внимава, когато средствата за директен достъп до слот от типа на посочените могат да бъдат изпълнявани като резултат от съобщение, изпратено към екземпляр на подклас на класа, за който е дефиниран (към който е прикрепен) манипулаторът.

Ако подкласът предефинира слота, то средствата за директен достъп до този слот, включени в манипулаторите на съобщения, прикрепени към суперкласа, не могат да бъдат използвани.

Тези манипулатори на съобщения осъществяват достъп до съответния слот само на суперкласа, а не и на подкласа.

Пример

```
> (clear)
> (defclass A (is-a USER)
     (slot s (create-accessor read)))
> (defclass B (is-a A) (role concrete)
     (slot s))
> (make-instance b of B)
[b]
> (send [b] get-s)
 Static reference to slot s of class A
 does not apply to [b] of B
FALSE
>
```

С цел директен достъп до нови варианти на слотове на подкласовете в рамките на манипулатори на съобщения, прикрепени към суперкласовете, могат да се използват функциите **dynamic-get** и **dynamic-put**:

(dynamic-get <slot-name-expression>)

(dynamic-put <slot-name-expression> <expression>*)

С помощта на тези функции се извлича/записва стойност от/в дадения слот на активния екземпляр. Те се изпълняват само в телата на манипулатори на съобщения. Аргументите им се оценяват и се осъществява реален достъп до слота едва при изпълнението на функцията.

Затова е възможно да се осъществява достъп до различни слотове при различните изпълнения на манипулатора.

За целта слотът на подкласа трябва да включва в дефиницията си фасета **`(visibility public)`**.

Пример

```
> (clear)
> (defclass A (is-a USER) (slot s))
> (defmessage-handler A get-s ()
    (dynamic-get s))
> (defclass B (is-a A) (role concrete)
    (slot s (visibility public)))
> (make-instance b of B)
[b]
> (send [b] get-s)
nil
```

По-често използвани вградени манипулатори на съобщения

- Инициализиране на екземпляр

```
(defmessage-handler USER init primary ())
```

Потребителят не може да изпраща това съобщение директно; то се изпраща от системните функции **`make-instance`** и **`initialize-instance`**.

- Изтриване на екземпляр

```
(defmessage-handler USER delete
    primary ())
```

- Показване (извеждане) на екземпляр

```
(defmessage-handler USER print primary ())
```

Извеждат се сведения за името на екземпляра, класа му, имената и стойностите на слотовете му.

- Директно модифициране на екземпляр

```
(defmessage-handler USER
      direct-modify primary
       (?slot-override-expressions))
```

Модифицира слотовете на даден екземпляр директно, вместо изпращане на **`put-`** съобщения към тях.

Изпращане на съобщения (Message Dispatch)

Когато към даден обект се изпрати определено съобщение с помощта на функцията **`send`**, CLIPS разглежда списъка от класовете – предшественици на класа на активния екземпляр с цел да определи пълното множество от манипулатори на съобщения, които са приложими към съобщението. CLIPS използва ***ролите*** (around, before, primary или after) и ***специфичността*** на тези манипулатори на съобщения с цел да установи реда им и след това ги изпълнява. При това, манипулатор на дадено съобщение, който е присъединен към подклас на класа, към който е присъединен друг манипулатор на това съобщение, се смята за по-специфичен.

Приложимост на манипулаторите на съобщения

Даден манипулатор е приложим към дадено съобщение, ако името му съвпада с това на съобщението и е прикрепен към (дефиниран за) клас, който се съдържа в списъка на класовете – предшественици на екземпляра, към който е изпратено съобщението.

Ред на изпълнение на манипулаторите на съобщения

Множеството от всички приложими манипулатори на даденото съобщение се разделя на 4 групи в зависимост от ролята им и след това всяка от тези 4 групи се сортира според специфичността на класовете на елементите. Наредбата сред around, before и primary манипулаторите е в посока от по-голяма специфичност към по-голяма общност; after манипулаторите се подреждат от най-общия към най-специфичния.

Редът на изпълнение на приложимите манипулатори е следният:

- Започва изпълнението на най-специфичния от around манипулаторите (ако тялото му съдържа обръщение към функцията **`call-next-handler`**, тази стъпка се повтаря);
- Един след друг (в посока от най-специфичния към най-общия) се изпълняват before манипулаторите;
- Изпълнява се най-специфичният от primary манипулаторите (възможно е тялото му да съдържа обръщение към функцията **`call-next-handler`** и ако е така, изпълнява(т) се и следващият (следващите) по специфичност primary манипулатор(и));

- Един след друг (в посока от най-общия към най-специфичния) се изпълняват after манипулаторите;
- Завършва изпълнението на активираните around манипулатори (в посока от най-общия към най-специфичния).

Забележка. Ако един манипулатор на съобщение се извиква за изпълнение от друг манипулатор на същото съобщение, тогава се казва, че първият манипулатор е ***скрит*** (***shadowed***) от втория. За извикване на скрити манипулатори на съобщения (shadowed message handlers) се използват функциите **`call-next-handler`** и **`override-next-handler`**.

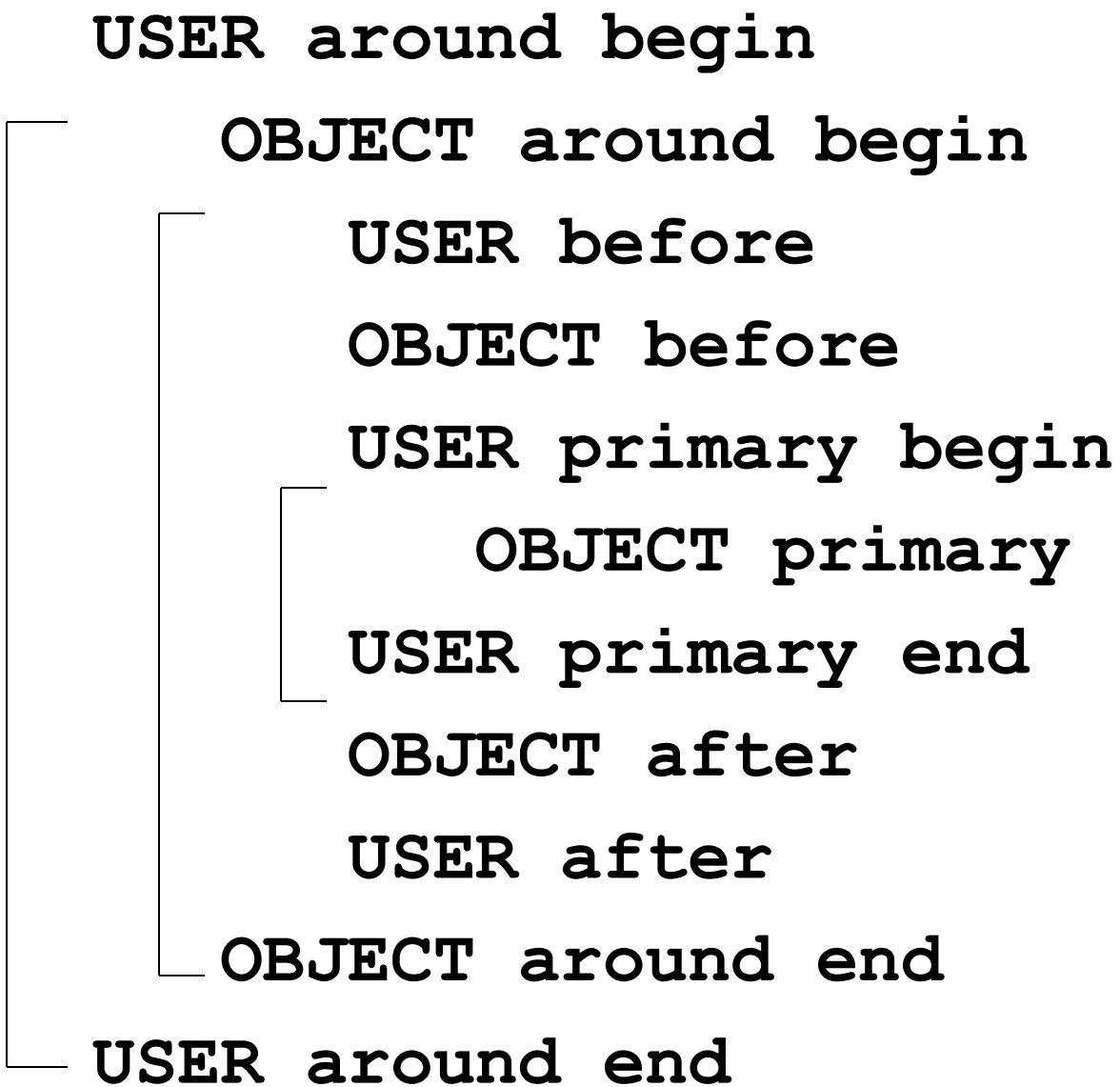
Пример

```
(defmessage-handler USER my-message
  around () (call-next-handler))
(defmessage-handler USER my-message
  before ())
(defmessage-handler USER my-message ()
  (call-next-handler))
(defmessage-handler USER my-message
  after ())
(defmessage-handler OBJECT my-message
  around () (call-next-handler))
```

```
(defmessage-handler OBJECT my-message
  before ())
(defmessage-handler OBJECT my-message ())
(defmessage-handler OBJECT my-message
  after ())
```

Забележка. Манипулаторите от тип around трябва задължително да включват в телата си явни обръщения към други манипулатори.

Схема на изпълнението



Манипулиране на екземпляри на класове

Изпращане на съобщения

```
(send <object-expression>
      <message-name-expression>
      <expression>*)
```

Създаване на екземпляри

```
(make-instance
  [<instance-name-expression>]
  of <class-name-expression>
  <slot-override>*)

<slot-override> ::=
  (<slot-name-expression> <expression>*)
```

Функцията **make-instance** работи по следния начин:

- Ако екземпляр с посоченото име вече съществува, този екземпляр получава съобщение **delete**, което има следния вид: **(send <instance-name> delete)**
- Създава се нов неинициализиран екземпляр на определения клас, който получава избраното име
- Всички директни присвоявания на слотове (slot-overrides) се оценяват и се изпълняват съответни **put** съобщения от вида

```
(send <instance-name>
   put-<slot-name> <expression>*)
```

- Новият екземпляр получава системното съобщение **`init`**. Обикновено манипулаторът, прикрепен към класа USER, реализира това съобщение. Този манипулатор извиква функцията **`init-slots`**, която използва стойностите, валидни по подразбиране, за всички слотове, за които няма директни присвоявания. Тези стойности по подразбиране се записват директно, без използване на съобщения

Конструкция definstances

Синтаксис:

```
(definstances <definstances-name>
  [<comment>]
  <instance-definition>*)
```

Пример

```
> (clear)
> (defclass A (is-a USER) (role concrete)
    (slot x (create-accessor write)
        (default 1)))
> (definstances A-OBJECTS
    (a1 of A)
    (a2 of A (x 65)))
> (reset)
>
```

СЪПОСТАВЯНЕ С ОБЕКТНИ ОБРАЗЦИ

Особености на съпоставянето с обектни образци

Екземпляри на потребителски класове могат да бъдат съпоставяни с образци, включени в левите страни на правилата. Даден обектен образец може да се съпостави успешно само с обекти, които са екземпляри на клас, дефиниран преди образеца. Всеки клас, който може да има екземпляри, съпоставими с някакъв образец, не може да бъде унищожен или променен, докато не бъде изтрит този образец.

Класовете, свързани с образците в лявата страна на правилото, не могат да бъдат променяни, докато не приключи изпълнението на дясната му страна, дори ако това правило бъде изтрито в резултат от изпълнението на съответно действие в дясната страна.

Когато бъде създаден или изтрит определен екземпляр, се прави опит за актуализиране на всички негови съпоставяния със съответни обектни образци. Когато обаче се извърши промяна на слот, тази промяна може да окаже влияние само върху съпоставянето на тези образци, които явно цитират същия слот.

Промените в нереактивните слотове или екземпляри на нереактивни класове нямат ефект върху съпоставянето на левите страни на правилата. По време на изпълнението на функциите **`make-instance`**, **`initialize-instance`**, **`modify-instance`**, **`message-modify-instance`**, **`duplicate-instance`**, **`message-duplicate-instance`** и **`object-pattern-match-delay`** съпоставянето на съответните обектни образци се отлага.

Отложено съпоставяне при манипулиране с екземпляри

При манипулирането с екземпляри (т.е. при създаването, модифицирането или изтриването на екземпляри) е възможно съпоставянето с обектните образци в левите страни на правилата да бъде отложено, докато приключат съответните манипулации. Отлагането на съпоставянето се осъществява с помощта на функцията **`object-pattern-match-delay`**. То продължава до завършването на действията, включени в обръщението към функцията.

Синтаксис:

```
(object-pattern-match-delay <action>*)
```

Пример

```
> (clear)
> (defclass A (is-a USER) (role concrete)
    (pattern-match reactive))
> (defrule match-A
     (object (is-a A))
  =>)
> (make-instance a of A)
[a]
```

```
> (agenda)
0   match-A: [a]
For a total of 1 activation.
> (make-instance b of A)
[b]
> (agenda)
0   match-A: [b]
0   match-A: [a]
For a total of 2 activations.
```

```
> (object-pattern-match-delay
     (make-instance c of A)
     (printout t "After c ... " crlf)
     (agenda)
     (make-instance d of A)
     (printout t "After d ... " crlf)
     (agenda))
After c ...
0   match-A: [b]
0   match-A: [a]
For a total of 2 activations.
```

```
After d ...
0   match-A: [b]
0   match-A: [a]
For a total of 2 activations.
> (agenda)
0   match-A: [d]
0   match-A: [c]
0   match-A: [b]
0   match-A: [a]
For a total of 4 activations.
>
```

Модифициране на екземпляри

За модифициране на екземпляри са предвидени четири функции. Тези функции позволяват актуализирането на слотовете на даден екземпляр да бъде извършвано в блок, без за целта да е необходима поредица от съответни **`put-`** съобщения.

Всяка от тези функции връща стойност TRUE, когато изпълнението на тялото й завърши с успех, и FALSE – в противния случай.

- Директно модифициране на екземпляр с отложено съпоставяне

Функцията **`modify-instance`** използва съобщението **`direct-modify`**, за да промени стойностите на някои от слотовете на екземпляра. Съпоставянето с обектните образци се отлага, докато завършат всички модификации на слотове в съответния екземпляр.

Синтаксис:

```
(modify-instance <instance>
  <slot-override>*)
```

- Директно модифициране на екземпляр с незабавно съпоставяне

Функцията **`active-modify-instance`** използва съобщението **`direct-modify`**, за да промени стойностите на някои от слотовете на екземпляра. Съпоставянията с обектните образци се извършват по време на модифицирането на слотовете на екземпляра.

Синтаксис:

```
(active-modify-instance <instance>
  <slot-override>*)
```