CLIPS áttekintés

Produkciós rendszerek fejlesztése

CLIPS történet

- **CLIPS** = **C** **L**anguage **I**ntegrated **P**roduction **S**ystem
- Fejlesztették: NASA's Johnson Space Center (80-as évek közepén)
- C nyelvet alkalmazták a megvalósításra, LISP szintaktikát követtek

  (C nyelv választása: hatékonyabb kód, LISP fordítóktól való függetlenség, más nyelveken írt modulokkal integrálás)
- Alap változat: produkciós szabály interpreter.
- Objektum orientált kiterjesztés: **COOL** = **C**LIPS **O**bject-**O**riented **L**anguage

Mi is ez valójában?

- Klasszikus szabályalapú szakértoi rendszer
- CLIPS – szabályleíró nyelv
- CLIPS – kiegészíto komponensek
- Elorekövetkeztetés (CLIPS) *vs.* visszakövetkeztetés (pld. MYCIN)

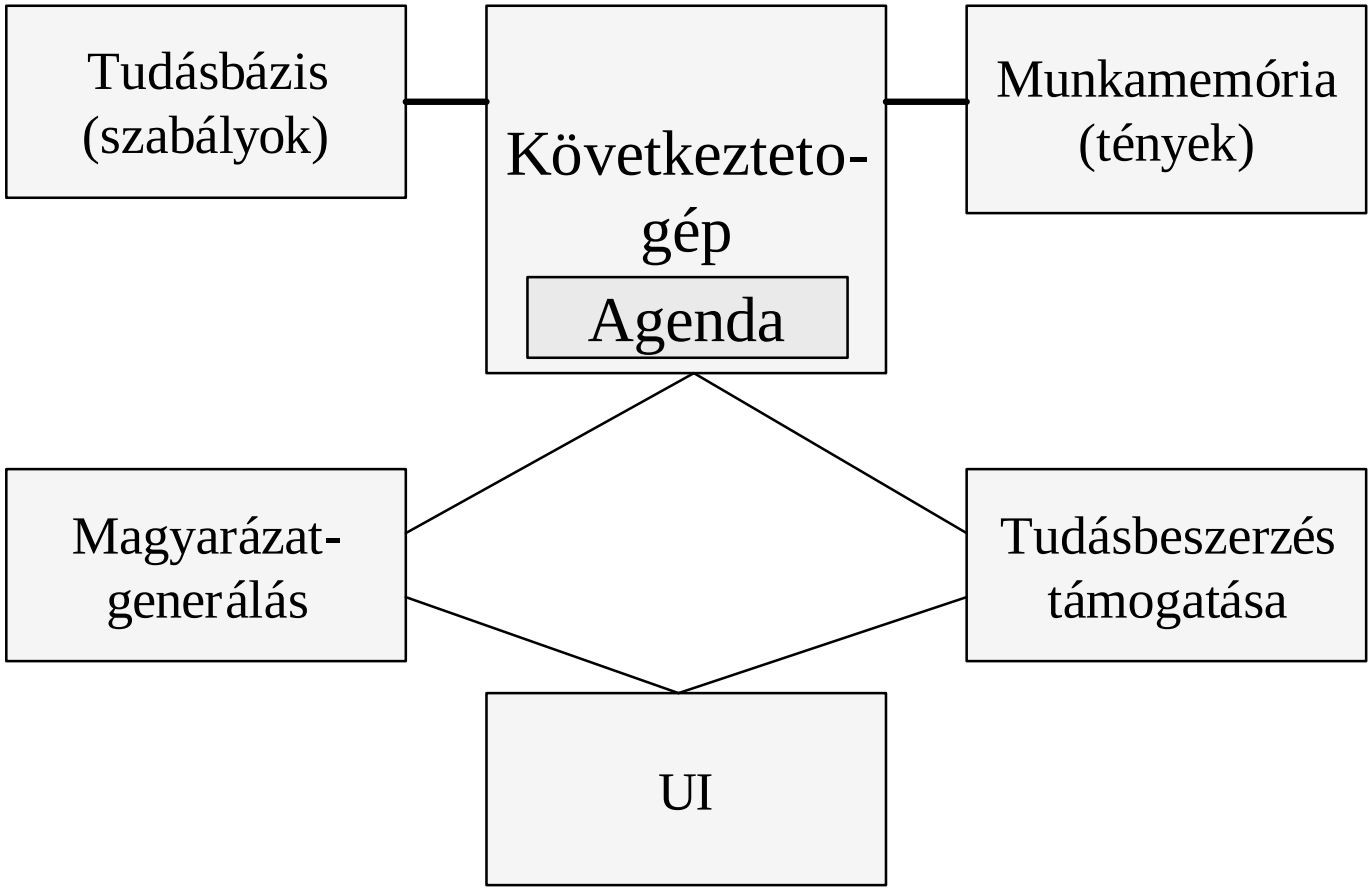
Miért érdekes a CLIPS?

- A CLIPS elonyei:
  - Magas-szintu interpreter
  - Produkciós szabály interpreter
  - Objektum orientált programozási nyelv
  - LISP-szeru procedurális nyelv
  - Számos különbözo platformon fut: UNIX, Linux, Windows, MacOS
  - Egy public-domain jól dokumentált szoftver

CLIPS, mint szabályleíró nyelv

- Az eredeti CLIPS két modult tartalmazott:
  - produkciós szabályleíró nyelv
  - procedurális nyelv
- A szabályleíró nyelv fontosabb komponensei
  - ténybázis
  - szabálybázis
  - következtetogép

Szabályalapú szakértoi rendszer komponensei



Szabályalapú szakértoi rendszer komponensei

- Ténybázis (ténylista): a probléma kezdeti vagy aktuális állapotát reprezentálja. *Adat, ami alapján következtethetünk.*
- Szabálybázis (tudásbázis): szabályok halmaza, amely leírja, hogy hogyan juthatunk el a problémától a megoldásig.
  (A CLIPS csak elorefele következtetést támogat.)

Szabályalapú szakértoi rendszer komponensei

- Következtetogép: vezérli a végrehajtást. Illeszti a tényeket a szabályokhoz, hogy meghatározza mely szabályok alkalmazhatóak.
- Felismer-végrehajt ciklusban dolgozik:
  - **Illesztés (match):** tények szabályok feltétel részéhez illesztése
  - **Választás (choose):** mely szabályok alkalmazhatóak
  - **Végrehajtás (execute):** a gyoztes szabályban leírt akciók végrehajtása

Elorekövetkeztetés *vs.* Hátrakövetkeztetés

- **A CLIPS egy elorekövetkezteto rendszer.**
  Illeszti a szabály feltétel részét (LHS) a munkamemória tartalmához, és végrehajtja az akció részét (RHS) a kiválasztott szabálynak.
  Kiinduló tények -> konklúziók.
- Hátrakövetkeztetésnél (híres példa MYCIN) a cél elore ismert és a cél felderítéséhez ennek elofeltételeit vizsgáljuk. Ha egy elofeltételnek további elofeltételei vannak, akkor ezt részcélnak tekintjük és ugyanúgy felderítjük, mint a célt, mindaddig, amíg nincsenek további részcélok.
- Sok esetben használható mindketto,
  vagy a két módszer kombinációja

A CLIPS elérhetosége

- http://www.ghg.net/clips/CLIPS.html

A Clips indítása

1. UNIX alatt az interpreter indítása

```
clips
```

interpreter bejelentkezése

```
CLIPS>
```

Kilépés CLIPS: `(exit)`.

2. Változatok:
   xclips (grafikus felület unix alatt)
   winclips (grafikus felület windows alatt)
   CLIPS ActiveX komponens (integráláshoz)

Alapveto CLIPS parancsok

- `(exit)` kilépés
- `(clear)` a munkakörnyezet megtisztítása a tényektol, szabályoktól és minden aktív definíciótól
- `(reset)` visszaállítja a ténybázist a kiindulási állapotra (kitöröl minden létezo tényt, létrehoz egy `initial-fact` tényt létrehozza a programban definiált tényeket `(deffacts)`
  A reset kiadárára minden futtatás elott szükség van!
- `(run)` végrehajtja a betöltött programot a CLIPS interpreterben a definiált szabályok és tények felhasználásával.

Alapveto CLIPS parancsok

- `(load “filename.clp”)`
  betölti a fájlban található CLIPS programot.
  Elvégzi a program szintaktikai elemzését.
- `(facts)` listázza a pillanatnyilag érvényes szabályokat
- `(rules)` listázza a szabálybázis szabályait.

Néhány fontos Clips utasítás

- Tény hozzáadása a ténybázishoz:

```
(assert (first-fact asserted))
```

- Szabály definiálása:

```
(defrule first_rule
   (first-fact asserted)
=>
   (assert (second-fact asserted)))
```

“Hello World” CLIPS-ben

```
(defrule start
  (initial-fact)
=>
  (printout t “Hello, world!” crlf))
```

Futtatás

- A *hello-world.clp fájl* szerkesztése
- CLIPS környezet indítása (`clips` vagy `xclips`)
- Program betöltése
  `(load hello-world.clp)`
- Betöltés után a CLIPS visszajelez
  (sikeres betöltés- TRUE):

```
(load hello-world.clp)
defining defrule start +j
TRUE
```

Futtatás (2)

- Alaphelyzetbe állítás(`reset`)
- Indítás: (`run`)

  ^E - reset

  ^R - run

- Kilépés: **^Q** vagy (`exit`)

Mezok

- 7 féle adattípust ismer a CLIPS (tokenek).
  - float: [+\-] <digit>* [.<digit>*] [e|E[+|-]<digit>*]
  - integer: [+|-] <digits> *
  - symbol: [<] <char>*
  - string: “<char>* “ (e.g. “John”, “848-3000”)
  - external address
  - instance name
  - instance address

Érvényes kifejezések

- Példák
  - `fire`
  - `emergency-fire`
  - `activate_sprinkler_system`
  - `shut-down-electrical-junction-387`
  - `!?#$^*`

(A CLIPS kisbetu-nagybetu érzékeny!)

Tények

- A tények egyszeru kifejezések, amelyek tartalmaznak egy relációnevet, és opcionálisan további rekeszeket. Példa:
  `(person (name “John”))`
- Gyakran használunk keret struktúrákat:
  `(deftemplate)` konstrukció

(deftemplate)

```
(deftemplate <relation_name> [<comment>]
  <slot-definition>*
)
```

<slot-definition> is:

- (slot <slot-name>)
- (field <slot-name>)
- (multislot <slot-name>)

Érvényes tények

- Példák:

```
(single-field)
(two fields)
(speed 38 mph)
(cost 78 dollars 23 cents)
(name “John Doe”)
```

(deftemplate) példa

```
(deftemplate person “an example template”
        (multislot name)
        (slot age)
        (slot eye-color)
        (slot hair-color))
CLIPS> defining deftemplate: person
TRUE
```

Tények (2)

- A Deftemplate lehet implicit = sorrend megadása:
  `(numbers 1 2 3)`
- Tények hozzáadása: `(assert <fact>)`
  Tény törlése: `(retract <fact-index>*)`
- Tények listázása: `(facts)` visszaadja a tényeket és ezek azonosítóit, pl. `f-0`

Példák tényekre

```
CLIPS> (deftemplate course “electives”
        (slot number))
CLIPS> (assert (course (number comp674))
               (course (number comp672)))
<Fact-1>
CLIPS> (facts)
f-0 (course (number comp674))
f-1 (course (number comp672))
For a total of 2 facts
CLIPS> (retract 1)
CLIPS> (facts)
f-0 (course (number comp674))
For a total of 1 fact
```

Tények módosítása

- Módosítás:

(modify <fact-index> <slot-modifier>***)**

<slot-modifier> is (<slot-name> <slot-value>)

Példa:

```
CLIPS>  (modify 0 (number comp675))
<fact-2>
CLIPS> (facts)
f-2 (course (number comp675))
for a total of 1 fact
```

Tények duplikálása

- Másolat készítése:

Példa (folytatva)

```
CLIPS> (duplicate 2 (number comp775))
<fact-3>
CLIPS> (facts)
f-2 (course (number comp675))
f-3 (course (number comp775))
For a total of 2 facts
```

- A `(duplicate)` megváltoztatja a tényt az eredeti törlése nélkül

Tények csoportjának hozzáadása

```
(deffacts <deffacts-name> [<comment>]
   <facts>*)
(reset)
```

Tények törlése

- Tényeket lehet törölni, visszavonni

```
(retract <fact-index>)
(retract 2)
```

- Több tény törlése

```
(retract 1 2)
```

(deftemplate) példa

- Több deklaráció egyszerre:

```
(deffacts student-Ids
  (student (name Tarzan))
  (student (name Jane) (age 19)))
```

- Eredmény:

```
(student (name Tarzan) (age 18))
(student (name Jane) (age 19))
```

Szabályok (1)

- LHS => RHS
- Szintaktika:

```
(defrule <rule-name> [<comment>]
    [<declaration>] ; salience
     <patterns>*     ;LHS, premises, patterns,
                     ;conditions, antecedent
   =>
     <actions>*)   ;RHS, actions, consequent
```

Szabályok (2)

- Példa:

```
(defrule class-A-fire-emergency
    (emergency fire)
=>
    (printout t “FIRE!!!” crlf))
```

- Egy szabálynak több premisszája lehet:

```
(defrule class-B-fire-emergency
    (emergency fire)
    (fire-class B)
=>
    (printout t “Use carbon dioxide extinguisher”
crlf))
```

Agenda

- Ha a LHS mintái illeszkednek az érvényes tényekhez, akkor a szabály aktiválja a CLIPS és felrakja az agendára.
- Szabályo sorrendje:
  salience (*prioritás*).
- Ha az agenda üres a program leáll.
- **Refraction**: minden szabályt egyszer hajt végre a rendszer egy adott tényhalmazra => `(refresh)`

Salience

- Alaphelyzetben az agenda egy stack..
- A legutoljára aktivált szabályt hajtja végra.
- Salience, prioritás változtat a sorrenden.
- Alapértelmezett érték: 0.

Konfliktus feloldási stratégia

- Frissesség
  - Utoljára felkerült szabályok preferálása.
  - CLIPS time-tag-ek
- Specifikusság
  - Az a szabály, amely a legjobban illeszkedik az adott helyzetre preferált.
  - Hasznos, ha általános és kivételkezelo szabályokkal dolgozunk
- Cilkusmentesség
  - Ugyanazokra az adatokra csak egyszer hajtja végre a szabályokat.
  - Megakadályozza a ciklusokat
  - Használt szabályok újratöltése (`refresh`))

Konfliktus feloldási stratégia

- Elodleges a prioritás.
- 7 stratégia állítható:
  - The depth strategy
  - The breadth strategy
  - The simplicity strategy
  - The complexity strategy
  - The LEX strategy
  - The MEA strategy
  - It is possible also to set strategy to random
- szintaktika: (set-strategy <strategy>)

Változók

- Változó neve `?` Karakterrel kezdodik:
- Példa:
  `(course (number ?cmp))`
- Változókat használunk
  - Mintaillesztés
  - I/O
  - Tények mutatói

Példák változókra

- ```
  (defrule grandfather
      (is-a-grandfather ?name)
  =>
      (assert (is-a-man ?name))
  )
  ```
- ```
  (defrule grandfather
      (is-a-grandfather ?name)
  =>
      (assert (is-a-father ?name))
      (assert (is-a-man ?name))
      (printout t ?name " is a grandfather" crlf)
  )
  ```

Példák tények törlésére

```
(defrule change-grandfather-fact
  ?old-fact <- (is-a-grandfather ?name)
=>
  (retract ?old-fact)
  (assert (has-a-grandchild ?name)
  (is-a-man ?name))
)
```

Példák tények törlésére (2)

- Több tény törlése:

```
(retract ?fact1 ?fact2 ?fact3)
```

- Minden tény törlése:

```
(retract *)
```

Standard I/O

- Nyomtatás STDOUT: `(printout t …)`
- újsor: `crlf`
- Olvasás STDIN :`(read)`

Standard I/O Példák (1)

- Billentyuzet példa:

```
(defrule to-start “Rule to start & enter a name”
  (phase choose-name)
=>
  (printout t “Enter your name” crlf)
  (assert (your-name =(read)))) ; ’=’ is optional
```

- Változók használata:

```
(defrule to-start
=>
  (printout t “Enter something: ”)
  (bind ?something (read))
  (printout t “You have entered ” ?something crlf))
```

Standard I/O Példák (2)

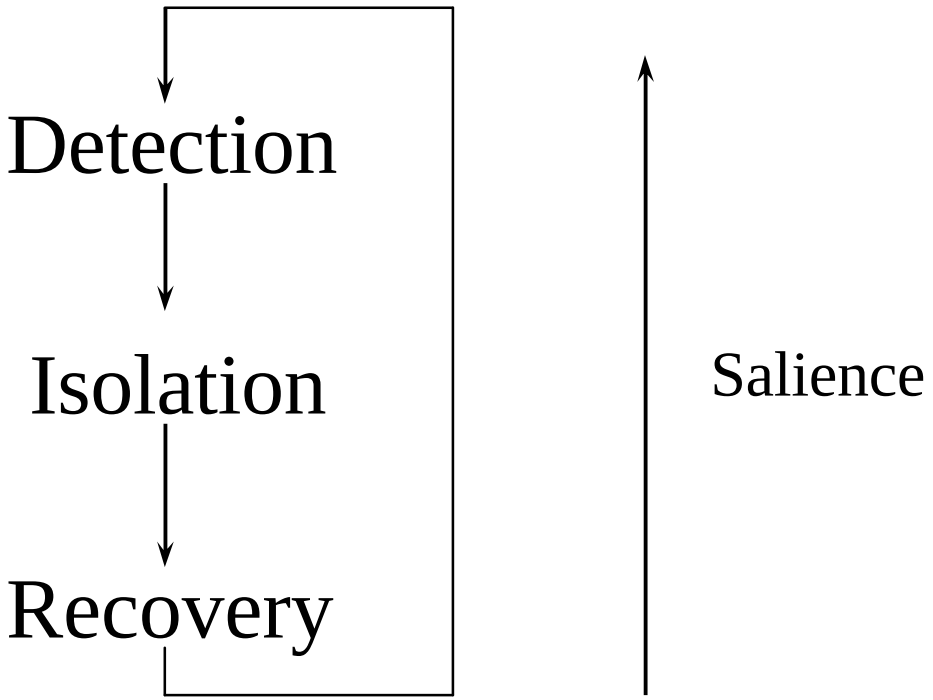
- Összetettebb példa:

```
(defrule continue-check
     ?phase <- (phase check-continue)
=>
     (retract ?phase)
     (printout t “Do you want to continue?” crlf)
     (bind ?answer (read))
     (if    (or (eq ?answer yes) (eq ?answer y))
      then  (assert (phase continue))
      else  (halt))
)
```

File I/O

- File I/O
  - `(load <filename>)`
  - `(save <filename>)`

Prioritás példa



Vezérlési struktúra példa (1)

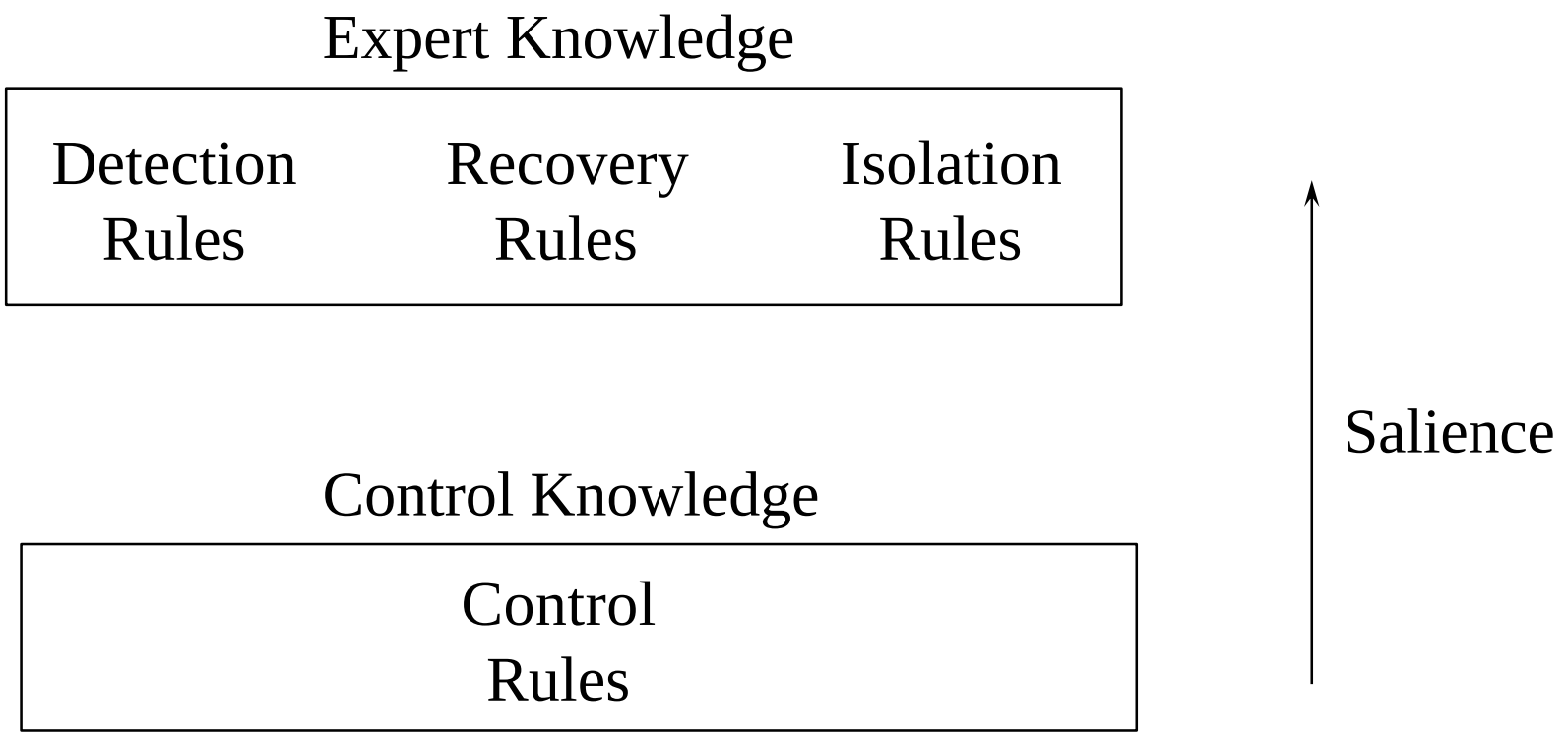
- ```
  (defrule detection-to-isolation
      ?phase <- (phase detection)
      (declare (salience –10))
  =>
      (retract ?phase)
      (assert (phase isolation)))
  ```
- ```
  (defrule isolation-to-recovery
      ?phase <- (phase isolation)
      (declare (salience –10))
  =>
      (retract ?phase)
      (assert (phase recovery)))
  ```

Vezérlési struktúra példa(2)

- ```
  (defrule recovery-to-detection
       ?phase <- (phase recovery)
       (declare (salience –10))
  =>
       (retract ?phase)
       (assert (phase detection)))
  ```

- ```
  (defrule find-fault-location-and-recovery
       (phase recovery)
       (recovery-solution switch-device ?replacement on)
  =>
       (printout t “Switch device” ?replacement “on”
  crlf))
  ```

A vezérlési és probléma tudás elválasztása



Vezérlési struktúra példa(3)

- Az elozo szabályok általánosabb formában

```
(deffacts control-information
   (phase detection)
   (phase-after detection isolation)
   (phase-after isolation recovery)
   (phase-after recovery detection))

(defrule change-phase
   (declare (salience -10))
   ?phase <- (phase ?current-phase)
   (phase-after ?current-phase ?next-phase)
=>
   (retract ?phase)
   (assert (phase ?next-phase)))
```

Vezérlési struktúra példa(4)

- Fázisok szekvenciájaként (ciklikusan)

```
(deffacts control-information
   (phase detection)
   (phase-sequence isolation recovery detection))

(defrule change-phase
   (declare (salience -10))
   ?current-phase <- (?phase ?current-phase)
   (phase-sequence ?next-phase $?other-phases)
=>
   (retract ?current-phase)
   (assert (phase ?next-phase))
   (assert (phase-sequence $?other-phases ?next-phase)))
```