2015

TJD

JAZYK CLIPS

KOŠÍK, MICHAL

Obsah

Úvod

Konvenčné programovacie jazyky, ako je napr. C alebo FORTRAN, sú navrhnuté a optimalizované na procedurálnu manipuláciu s dátami (ako sú napr. čísla alebo polia). Naproti tomu ľudia veľmi často zvyknú pristupovať k riešeniu zložitých problémov abstraktne, čo sa ale do konvenčných programovacích jazykov implementuje zložitejšie a neintuitívne.

Jedným z výstupov výskumu v oblasti umelej inteligencie bol aj vývoj techník umožňujúcich prácu s informáciami na vyššom stupni abstrakcie. Tieto techniky boli pretransformované do programovacích jazykov alebo nástrojov, ktoré sú schopné pracovať s informáciami vo forme podobnej tej, s akou pracuje ľudská logika a tým pádom aj umožňujú jednoduchší prepis týchto informácií a znalostí do programu. Tieto nástroje, ktoré emulujú ľudské rozhodovanie a skúsenosti v určitej dobre definovanej oblasti, v ktorej sa nachádza problém, ktorý sa snažíme vyriešiť, sa nazývajú expertné alebo produkčné systémy.

Tieto systémy sú často vyvíjané pomocou programovania založenom na pravidlách. V tomto programovacom paradigme pravidlá reprezentujú heuristiku, ktorá špecifikuje určitú sadu akcií, ktoré budú vykonané v určitej situácii. Každé pravidlo pozostáva z podmienkovej (*if)* časti a časti vykonávajúcej nejaké akcie (*then part)*. Podmienková časť obsahuje množinu vzorov špecifikujúcich fakty, ktoré spôsobia, že pravidlo je použiteľné. Proces naviazania faktov na vzory sa nazýva „*pattern matching*“. Expertný systém poskytuje mechanizmus, nazývaný „*inference engine*“, ktorý automaticky naviaže fakty na vzory a rozhoduje sa, ktoré pravidlá sú použiteľné. *Then* časť pravidla sa vykoná, keď je pravidlo použiteľné a keď *inference engine* vydá príkaz na jeho vykonanie. *Inference engine* funguje nasledovne:

1. Vyberie pravidlo, ktoré je použiteľné.
2. Vykonaj akcie tohto pravidla – toto môže ovplyvniť zoznam pravidiel, ktoré sú vykonateľné tým, že akcie pridajú alebo odoberú fakty.
3. Pokiaľ existuje nejaké použiteľné pravidlo, vráť sa naspäť na bod 1.

CLIPS (skratka z „*C Language Integrated Production System*“) je softwarový nástroj, ktorý ponúka kompletné prostredie pre konštrukciu expertných systémov založených na pravidlách a/alebo objektoch. Jeho hlavnými vlastnosťami sú:

- *Reprezentácia znalostí* – CLIPS ponúka ucelený nástroj na prácu s rôznymi typmi znalostí, ktorý podporuje 3 rozdielne programovacie paradigmy: procedurálne, objektovo-orientované a založené na pravidlách. Procedurálne schopnosti CLIPSu sú podobné tým, ktoré obsahujú jazyky ako C, Java, Ada a LISP. Objektovo-orientované programovanie umožňuje modelovať komplexné systémy ako modulárne komponenty, ktoré môžu byť v budúcnosti opätovne použiteľné. A programovanie založené na pravidlách umožňuje reprezentovať znalosti ako sadu akcií, ktoré budú v danej situácii vykonané.
- *Prenositeľnosť* – CLIPS je napísaný v jazyku C a bol nainštalovaný na veľkom počte rôznych operačných systémov bez toho, aby sa kvôli nim musel nejak upravovať. Tieto systémy boli napr. Windows XP, Windows 7/8, MacOS X, Unix. CLIPS môže byť portovaný na každý operačný systém, ktorý má kompilátor spĺňajúci štandard ANSI C alebo C++. V prípade problémov (alebo nutnosti zmeniť či rozšíriť jeho funkcionalitu) je možné upraviť priamo zdrojové kódy CLIPSu, ktoré sú voľne dostupné.

- *Integrovateľnosť a rozšíriteľnosť* – CLIPS môže byť vložený do procedurálneho kódu, volaný ako procedúra a môže byť integrovaný do jazykov ako sú C/C++, Java, FORTRAN a Ada. Okrem toho dokáže byť CLIPS jednoducho rozšírený o novú funkcionalitu pomocou niekoľkých definovaných rozhraní.
- *Interaktívny vývoj* – štandardná verzia CLIPSu ponúka interaktívne textové vývojové prostredie, ktoré obsahuje debugger, nápovedu a integrovaný editor. Pre Windows, Mac OS a unixové distribúcie s X Window existuje grafická nadstavba, ktorá ešte viac zjednodušuje prácu.
- *Validácia a verifikácia* – CLIPS ponúka niekoľko nástrojov, ktoré umožňujú previesť validáciu a verifikáciu expertného systému, podporujú modulárny dizajn a rozdelenie databázy znalostí, statickú a dynamickú kontrolu hodnôt argumentov funkcií a jednotlivých slotov, ako aj sémantickú analýzu pravidiel.
- *Dokumentácia* – CLIPS je výborne zdokumentovaný. Medzi 2 hlavné dokumenty patria *Reference Manual* a *User's Guide*.
- *Nízke náklady* – CLIPS je udržiavaný ako public domain software.

História

Pôvod CLIPSu siaha až do roku 1984 do americkej agentúry pre letectvo a vesmír (NASA). V tom období sekcia NASA nazývaná Artificial Intelligence Section vyvinula viac než 10 prototypov nástrojov na tvorbu expertných systémov. No aj napriek veľkému potenciálu, ktorý veľká časť týchto nástrojov mala, sa iba niektoré z nich dočkali reálneho využitia. Neschopnosť dodať nástroj na tvorbu expertných systémov, ktorý by spĺňal požiadavky, ktoré naň kládla NASA, sa pričíta použitiu jazyka LISP ako hlavného programovacieho jazyka, na ktorom bola väčšina týchto nástrojov postavená. Existovali 3 hlavné problémy, s ktorými sa pri použití LISPu muselo počítať: nízka dostupnosť LISPu na rôznych typoch počítačov, vysoká cena LISPovských nástrojov a podporovaného hardwaru a v neposlednom rade náročná integrácia LISPu do ostatných jazykov.

V rámci Artificial Intelligence Section vedeli, že použitie „štandardného" jazyka, ako napr. jazyka C, by eliminovalo väčšinu problémov. V tej dobe ale neexistoval expertný systém, ktorý by nemal aspoň jeden z vyššie spomenutých problémov. Z toho dôvodu sa v Artificial Intelligence Section (AIS) rozhodli, že vytvoria vlastný nástroj na tvorbu expertných systémov.

Prvý prototyp CLIPSu bol vytvorený za menej ako 2 mesiace a bol predstavený na jar 1985. Veľká pozornosť bola venovaná tomu, aby bol tento nástroj kompatibilný s vyvíjanými expertnými systémami v rámci AIS. Práve preto má CLIPS veľmi podobnú syntax, ako je časť syntaxe nástroja pre vývoj expertných systémov ART, ktorý bol vyvinutý spoločnosťou Inference. Napriek tomu bol CLIPS vyvíjaný bez asistencie zo strany Inference a bez prístupu ku zdrojovým kódom ARTu.

Pôvodným zámerom pri vývoji CLIPSu bolo získať znalosti o tom, ako sa vytvára nástroj na tvorbu expertných systémov a zároveň položiť základy pre tvorbu nástroja, ktorý by v budúcnosti nahradil používané komerčné nástroje, ako bol napr. ART. CLIPS verzie 1.0 ukázal, že tieto zámery je možné splniť. Po ďalšom vývoji sa ukázalo, že CLIPS bude low-cost nástroj na tvorbu expertných systémov vhodný pre tréningové účely. Počas ďalšieho roku vývoja sa autori zamerali na zlepšenie prenositeľnosti, výkonu, funkcionality a tvorbu dokumentácie. CLIPS verzie 3.0 bol vydaný v lete 1986 a stal sa dostupným aj záujemcom mimo agentúru NASA.

Ďalší vývoj transformoval CLIPS z tréningového nástroja na nástroj vhodný pre vývoj a reálne nasadenie expertných systémov. Verzie 4.0 (vydaná v lete 1987) a 4.1 (vydaná na jeseň 1987) priniesli značný nárast výkonu, integráciu do ďalších jazykov a niekoľko ďalších vylepšení. Verzia 4.2 (vydaná v lete 1988) priniesla kompletné prepísanie CLIPSu za účelom zlepšenia modularity jeho kódu. Súčasťou tejto verzie bol tiež kompletný popis softwarovej architektúry CLIPSu a nástroje na verifikáciu a validáciu napísaných pravidiel. Verzia 4.3 (vydaná v lete 1989) priniesla ďalšie drobné vylepšenia.

Pôvodne CLIPS používal iba tzv. programovanie založené na pravidlách, kde bol použitý forward chaining založený na algoritme Rete. Verzia 5.0, ktorá bola vydaná na jar 1991, priniesla 2 nové programovacie paradigmy – procedurálne a objektovo-orientované. Objektovo-orientovaný programovací jazyk, ktorý CLIPS ponúka, sa nazýva *CLIPS Object-Oriented Language* (*COOL*). Verzia 5.1 (vydaná na jar 1991) priniesla iba niekoľko vylepšení a opráv týkajúcich sa grafických rozhraní pre Windows a MacOS. Verzia 6.1 (vydaná v roku 1998) zasa odstránila podporu pre staré non-ANSI C kompilátory a pridala podporu pre C++ kompilátory a takisto aj možnosť profilovania času stráveného v jednotlivých konštruktoch, ako aj možnosť písať užívateľom definované funkcie. Konečne verzia 6.2, ktorá bola vydaná na jar roku 2002, pridala podporu pre viaceré prostredia, do ktorých môže byť program vložený a zároveň priniesla niekoľko vylepšení vývojového prostredia.

Momentálne je CLIP udržiavaný nezávisle na NASA ako public domain software.

Základné informácie

Predchádzajúca kapitola mala za úlohu zbežne predstaviť CLIPS a jeho fungovanie. Poďme sa teraz naňho pozrieť podrobnejšie.

Ako už bolo spomenuté, CLIPS je nástroj na vývoj expertných systémov. Bol špeciálne navrhnutý na uľahčenie vývoju softwaru určeného na modelovanie ľudských znalostí alebo expertízy.

V rámci CLIPSu existujú 3 spôsoby, ako je možné reprezentovať znalosti:

1. *Pravidlá* (rules) – sú primárne určené pre heuristické znalosti založené na skúsenostiach.
2. *Funkcie* (deffunctions a generic functions) – sú určené hlavne pre procedurálne znalosti.
3. *Objektovo-orientované programovanie* – je taktiež primárne určené pre procedurálne znalosti. Šesť všeobecne akceptovaných vlastností objektovo-orientovaného programovania (objekty, spracovanie správ, abstrakcia, zapuzdrenie, dedičnosť a polymorfizmus) sú v CLIPSe podporované. Pravidlá sa môžu pokúsiť naviazať svoje vzory na fakty a aj na objekty.

Dôležitý pojem je tzv. *shell* – je to časť CLIPSu, ktorá prevádza inferenciu (inak nazývanú aj *reasoning*). CLIPS shell poskytuje tri základné prvky expertného systému:

1. Zoznam faktov a zoznam inštancií – globálna pamäť pre dáta.
2. Databázu znalostí – obsahuje všetky pravidlá.
3. Inference engine – prevádza vykonávanie pravidiel.

V CLIPSe je možné vyvinúť ako expertný systém využívajúci iba pravidlá alebo iba objekty, tak aj ich kombináciu. Vo všeobecnosti program napísaný v CLIPSe pozostáva z pravidiel, faktov a objektov. Inference engine rozhoduje, ktoré pravidlá budú použité a kedy. Expertný systém založený na pravidlách a napísaný v CLIPSe je program založený na datách, kde fakty (a príp. objekty) sú práve tými dátami, ktoré spôsobujú použitie pravidiel pomocou inference engine.

Toto je jedna z oblastí, v ktorej sa CLIPS líši od procedurálnych jazykov, kde vykonanie nejakej procedúry môže nastať aj bez dát. Napr. výraz PRINT 2+2 napísaný v jazyku BASIC môže byť okamžite vykonaný. Je to totiž kompletný výraz, ktorý na svoje vykonanie nepotrebuje žiadne dáta. Naproti tomu, CLIPS potrebuje dáta na to, aby mohol spustiť vykonávanie pravidiel.

Takto CLIPSom vytvorený expertný systém môže byť použitý buď samostatne (ako aplikácia), alebo môže byť volaný z procedurálneho jazyka, ako je napr. C. V tom prípade systém prevedie svoju funkciu a vráti kontrolu nazad do kódu, odkiaľ bol zavolaný. Je ale možný aj opačný prístup – procedurálny kód môže byť definovaný ako externá funkcia a zavolaná priamo z CLIPSu. Vtedy sa po skončení externej funkcie vracia kontrola naspäť do CLIPSu.

Fakty

Ako pri iných programovacích jazykoch, tak aj CLIPS rozpoznáva isté kľúčové slová. Napr. **(assert)** slúži na pridanie nových dát do zoznamu faktov. Napr.

```
(assert (duck))
```

vezme **duck** ako argument a pridá ho do zoznamu faktov. Pokiaľ sa tento príkaz vloží do CLIPS vývojového prostredia, systém zareaguje odpoveďou `<Fact-X>`, ktorá znamená, že CLIPS úspešne uložil fakt a dal mu identifikátor X, kde X je celé číslo. CLIPS automaticky pomenováva novo-vložené fakty pomocou sekvenčne narastajúceho celého čísla. Toto číslo sa zároveň použije aj pre tzv. *fact identifier* (*identifikátor faktu*), čo je jednoznačný identifikátor každého faktu v zozname.

Príkazy v CLIPSe sú ohraničené zátvorkami. Táto syntax je prevzatá z LISPu (ART, z ktorého CLIPS pôvodne vychádzal, mal tiež syntax založenú na jazyku LISP). Napriek tomu, že CLIPS je napísaný v jazyku C, štýl LISPu jeho vývoj ovplyvnil.

Štandardne pri spustení CLIPSU (alebo pri jeho resetovaní) sa do zoznamu faktov vloží tzv. **initial-fact** s identifikátorom **f-0**. Zvyčajne slúži pre pohodlné počiatočné aktivovanie pravidiel. Obsah databáze faktov môžeme vypísať pomocou príkazu **(facts)**. V našom prípade sa zobrazí nasledujúci výpis:

```
CLIPS>(facts)
f-0    (initial-fact)
f-1    (duck)
```

CLIPS štandardne neakceptuje duplikáty faktov. Nie je teda možné vložiť fakt s rovnakým názvom. Toto obmedzenie je ale možné obísť pomocou príkazu **(set-fact-duplication)**.

Fakt môže byť zo zoznamu faktov odstránený pomocou príkazu **(retract fact-index)**, kde index je číslo v rámci identifikátora faktu. Pri odstránení sa nemenia *fact-identifiers* ostatných faktov, takže je možné, že číslovanie nebude konzistentné. Pri pokuse o odstránenie neexistujúceho faktu vypíše CLIPS chybovú správu.

Na rýchle odstránenie všetkých faktov je možné použiť príkaz **(clear)**. Tento príkaz obnoví CLIPS do pôvodného stavu, ktorý je pri jeho štarte. To znamená, že zároveň aj odstráni všetky pravidlá.

Každý fakt pozostáva z jedného alebo viacerých *polí* (*fields*). Pole je zástupný symbol (placeholder), ktorý môže byť buď pomenovaný alebo nepomenovaný a ktorý môže mať asociovanú nejakú hodnotu. Pomenované zástupné symboly smú byť použité iba s tzv. *deftemplates*, ktoré sú popísané ďalej.

Fakt `(duck)` má iba jeden (nepomenovaný) zástupný symbol – jedná sa teda o fakt s jediným poľom (*single-field fact*). Poradie nepomenovaných polí je dôležité. Napr. pokiaľ by existoval fakt `(Brian, Donald)`, ktorý by bol interpretovaný nejakým pravidlom tak, že Brian je otcom Donalda, tak potom fakt `(Donald, Brian)` by znamenal, že Donald je otcom Briana. Naproti tomu pri pomenovaných poliach nie je ich poradie nijak dôležité.
Na to, aby sme mohli pokračovať, je potrebné si najprv definovať niekoľko pojmov. *Zoznam* (*list*) je skupina položiek, z ktorých každá môže mať nejakú hodnotu. *Usporiadaný* (*ordered*) *zoznam* je taký, v ktorom je pozícia dôležitá. *Multipole* (*multifield*) je sekvencia polí, z ktorých každé môže mať hodnotu. Napr. už zmienený fakt `(Donald, Brian)` je faktom s multipoľom. Pokiaľ pole nemá žiadnu hodnotu, môžeme použiť špeciálny symbol `nil`. Napr. `(Donald nil)` znamená že Donald nie je nikoho otcom.

Existuje hneď niekoľko typov polí: *float*, *integer*, *symbol*, *string*, *external-address*, *fact-address*, *instance-name* a *instance-address*. Typ každého poľa je určený podľa typu hodnoty, ktorá je v ňom uložená. V nepomenovanom poli je typ určený implicitne podľa typu hodnoty. V *deftemplates* je možné explicitne deklarovať, s akým typom má dané pole pracovať.

Symbol

Symbol je typ, ktorý začína zobraziteľným ASCII znakom a nasleduje $0 - n$ ďalšími zobraziteľnými znakmi. Jednotlivé polia sú oddelené buď medzerami alebo zátvorkami. Fakty nie je možné do seba zanárať.

CLIPS je taktiež *case-sensitive*. Okrem toho majú niektoré znaky špeciálny význam:

`" ( ) & | < ~ ; ? $`

Niektoré znaky slúžia ako oddeľovače a ukončujú symboly:

- Ľubovoľný nezobraziteľný znak, vrátane medzery, enteru a tabulátoru
- Úvodzovky
- Otváracie a zatváracie zátvorky
- & a |
- Menšie ako <; tento ale môže byť prvým znakom symbolu
- Tilda ~
- Bodkočiarka – naznačuje začiatok komentára, ktorý je až do konca riadku
- ? a $? – nemôžu byť na začiatku symbolu (jednalo by sa totiž o premennú), ale môžu byť v jeho vnútri

Príklady symbolov: `duck`, `duck1`, `duck_soup`, `duck-soup`, `duck1-1_soup-soup`, `d!?#%^`

String

Druhým typom poľa je *string*. Musí začínať a končiť úvodzovkami. Tieto úvodzovky sú súčasťou poľa. Medzi nimi sa môže objaviť $0 - n$ znakov. Napríklad: `"duck"`, `"duck1"`, `"duck/soup"`, `"duck soup"`, `"duck soup is good!!!"` Pokiaľ chceme do poľa pridať ďalšie úvodzovky, musíme pred ne dať spätné lomítko, rovnako ako v jazyku C.

Čísla

Ďalšími dvoma typmi sú numerické polia. Všetky čísla v rámci CLIPSu sú brané ako *long int* alebo *double* v jazyku C. Čísla bez desatinnej čiarky sú automaticky brané ako *int*, pokiaľ sa zmestia do jeho rozsahu. Inak sú brané ako *double*. Zápis čísel v plávajúcej desatinnej čiarke je rovnaký, ako v jazyku C. Od CLIPS verzie 6.0 nie je možné vložiť do zoznamu faktov iba samotné číslo ako fakt. Musí vždy začať symbolom, napr. `(number 1)`.

Fakt môže byť *usporiadaný* (*ordered*) alebo *neusporiadaný* (*unordered*). Doteraz ukázané fakty boli vždy usporiadané, pretože poradie polí hralo pri ich interpretácii úlohu. Usporiadané fakty používajú pozíciu polí na definíciu dát.

Je zvykom, že prvé pole pri fakte popisuje vzťah nasledujúcich polí.

Pravidlá

Na to, aby mohol expertný systém vykonávať svoju prácu, potrebuje okrem faktov aj pravidlá. Pravidlo je podobné príkazu *if then* v procedurálnych jazykoch. Takéto pravidlo môže byť popísané takto:

***IF** isté podmienky sú splnené*

***THEN** vykonaj tieto akcie*

Nové pravidlo sa vkladá pomocou príkazu **(defrule)**. Po kľúčovom slove nasleduje meno pravidla a za ním voliteľný komentár v úvodzovkách. Ďalej nasleduje jeden alebo niekoľko vzorov. CLIPS sa snaží naviazať vzor pravidla na nejakú vzorovú entitu, ktorá môže byť buď fakt alebo užívateľom definovaná trieda. Vzorový príklad faktu je napr. tento:

```
(defrule rule_name "optional_comment"
     (vzor_1) ; Ľavá strana pravidla (LHS)
     (vzor_2)
     ...
     (vzor_N)
     =>
     (akcia_1) ; Pravá strana pravidla (RHS)
     (akcia_2)
     ...
     (akcia_M)

)
```

Konkrétny príklad je napríklad tento:

```
(defrule duck "Here comes the quack" ; Hlavička pravidla

(duck)                               ; Vzor

 =>                                  ; THEN šípka

(assert (sound-is quack)))           ; Akcia
```

Každé pravidlo musí mať jedinečné meno – nové pravidlo s rovnakým menom prepíše existujúce. Akcia je v skutočnosti funkcia, ktorá zvyčajne nemá žiadnu návratovú hodnotu, ale vykoná užitočnú akciu, ako je napr. vloženie alebo vymazanie faktu. Funkcia v CLIPSe je časť spustiteľného kódu, ktorý má vlastné jedinečné meno a ktorý vráti nejakú hodnotu alebo má užitočný vedľajší účinok.

CLIPS sa snaží naviazať vzory v pravidle s faktami v zozname faktov. Pokiaľ sa mu podarí naviazať všetky vzory, pravidlo je *aktivované* a umiestnené do *agendy*. *Agenda* je kolekcia aktivovaných pravidiel. Agenda môže byť aj prázdna. Keď je v agende viacero aktivovaných pravidiel, CLIPS automaticky (podľa *priority* a aktuálnej *stratégie*) rozhodne, ktoré z nich *vypáli* (*fire*). Keď je agenda prázdna, program automaticky zastaví svoju činnosť.

Ako zoznam vzorov, tak aj zoznam akcii môže byť prázdny.

Práca s pravidlami a beh programu

CLIPS vždy vykoná akcie pravidla v agende, ktoré má najvyššiu *prioritu*. Toto pravidlo je potom z agendy odstránené a ďalšie pravidlo s najvyššou prioritou je vypálené. Aktuálna agenda sa dá zobraziť pomocou príkazu **(agenda)** alebo v grafickom rozhraní CLIPSu ako samostatne pod-okno.

```
CLIPS> (agenda)
0 duck: f-1
For a total of 1 activation.
```

Prvé číslo 0 udáva prioritu pravidla, `f-1` je zasa identifikátor faktu, ktorý je možné naviazať na vzor pravidla. Pokiaľ nie je priorita pravidla určená explicitne, CLIPS mu priradí 0, čo reprezentuje strednú prioritu. Možný rozsah priorít je $\langle -10000; 10000 \rangle$. Prioritu je možno nastaviť pomocou kľúčových slov **(declare salience)**.

Na spustenie programu stačí zadať príkaz **(run)**. Pravidlo `duck` sa vypáli a do zoznamu faktov sa pridá nový fakt. Pokiaľ sa ale príkaz **(run)** zadá znova, žiadne pravidlo sa už nevypáli. Dôvod je jednoduchý. Pravidlo je aktivované pokiaľ sú vzory naviazané na:

1. Úplne novú entitu, ktorá predtým neexistovala.
2. Entitu, ktorá už síce existovala, ale bola buď vymazaná alebo znovu vložená.

Inference engine radí aktivované pravidlá podľa ich priority. Toto radenie sa nazýva aj *riešenie konfliktov*, pretože eliminuje konflikty pri rozhodovaní, ktoré pravidlo má vypáliť. Keďže nami vložený fakt `(duck)` sa už použil (čo viedlo k vloženiu nového faktu `(quack)`), druhý krát už v rámci nášho pravidla použitý nebude.

Spôsob, akým sa vyberá pravidlo, ktoré bude následne vypálené (resp. akým sa novo-aktivované pravidlo vkladá do agendy), sa nazýva *stratégia* (*strategy*). CLIPS ponúka dohromady 7 rôznych stratégií:

1. *Depth* – jedná sa o štandardnú stratégiu, podľa ktorej sú novo-aktivované pravidlá vložené v rámci agendy nad už aktivované pravidlá s rovnakou prioritou.
2. *Breadth* – novo-aktivované pravidlá sú vložené pod už aktivované pravidlá s rovnakou prioritou.
3. *Simplicity* – medzi pravidlami s rovnakou prioritou sú novo-aktivované pravidlá vložené nad už existujúce pravidlá, ktoré majú rovnakú alebo vyššiu *špecifickosť* (*specificity*). Špecifickosť pravidla je určená počtom porovnaní ktoré musia byť vykonané na jeho ľavej strane (LHS). Každé porovnanie s konštantou alebo v minulosti naviazanou premennou pridá jeden bod

k špecifickosti. Rovnako aj každé zavolanie funkcie v rámci LHS. Boolovské funkcie and, or a not nepridávajú nič k špecifickosti, ale ich argumenty už áno. Vnorené volania funkcií nepridávajú k špecifickosti.

4. *Complexity* - medzi pravidlami s rovnakou prioritou sú novo-aktivované pravidlá vložené nad už existujúce pravidlá, ktoré majú rovnakú alebo nižšiu špecifickosť.
5. *LEX* – medzi pravidlami s rovnakou prioritou sú novo-aktivované pravidlá vložené do agendy pomocou použitia OPS5 stratégie. Tá priradí každému faktu (a inštancii) časovú pečiatku, ktorá udáva relatívny čas, kedy bol fakt naposledy použitý (relatívny vzhľadom k ostatným faktom a inštanciám). Potom je aktivované pravidlo, ktoré pracuje s objektami alebo faktami s nedávnejším časovou pečiatkou umiestnené v agende nad aktivované pravidlá s dávnejšími časovými pečiatkami. Pokiaľ sú všetky časové pečiatky 2 pravidiel rovnaké, pravidlo s viacerými časovými pečiatkami je umiestnené nad pravidlo s menej pečiatkami. Pokiaľ sú rovnaké aj tie, uprednostní sa pravidlo s väčšou špecifickosťou.
6. *MEA* – na určenie poradia pravidiel sa používa iba prvá časová pečiatka. Pokiaľ sa podľa nej nedá určiť poradie v agende, určí sa podľa stratégie LEX.
7. *Random* – každému aktívnemu pravidlu je pridelené náhodne vygenerované číslo, ktoré sa použije na určenie poradia medzi aktívnymi pravidlami s rovnakou prioritou. Toto číslo ostane zachované aj pri zmene stratégie, takže poradie pravidiel ostane zachované aj pri opätovnej zmene stratégie naspäť na random (pre všetky aktívne pravidlá, ktoré ostali v agende z doby pred prvou výmenou stratégie).

Premenné

Podobne ako ostatné jazyky, aj CLIPS dokáže pracovať s premennými a ukladať do nich hodnoty. Oproti faktu, ktorý je statický, je obsah premennej dynamický – hodnoty k nej priradené sa môžu meniť. Pokiaľ by sme chceli zmeniť fakt, museli by sme ho najprv odstrániť a potom znova vložiť so zmenenými hodnotami.

Meno premennej, tzv. *identifikátor* (*variable identifier*), vždy začína otáznikom, na ktorým nasleduje symbol, ktorý je menom premennej, napr. `?x`.

Pred tým, ako môže byť premenná použitá, jej musí byť priradená nejaká hodnota. Inak CLIPS zahlási chybu. Premenné sa môžu vyskytovať na oboch stranách pravidiel v ľubovoľnom počte a s ľubovoľným množstvom opakovaní. Nasleduje príklad pravidla s premennou:

```
(assert (duck-sound quack))
(defrule make-quack
(duck-sound ?sound)
=>
(assert (sound-is ?sound)))
```

Fakt `(duck-sound quack)` sa naviaže na premennú `?sound` a toto pravidlo sa vykoná. Po jeho vykonaní pribudne v zozname faktov nový fakt `(sound-is quack)`.

Pomocou premenných je možné aj fakty odstraňovať. Na to, aby CLIPS dokázal v rámci pravidla fakt odstrániť, potrebuje jeho adresu (*fact-address*) naviazať na premennú v rámci ľavej strany pravidla. Adresa faktu sa špecifikuje pomocou ľavej šípky:

```
CLIPS> (assert (bachelor Dopey))
CLIPS> (defrule get-married
          ?duck <- (bachelor ?name)
          =>
          (printout t ?name " is now married " ?duck crlf)

          (retract ?duck))
```

Tento kód po spustení vypíše `Dopey is now married` **`<Fact-1>`**, kde `<Fact-1>` je adresa (identifikátor) faktu `(bachelor Dopey)`, ktorá sa naviazala na premennú `?duck`. Zároveň tento kód aj ukazuje, že premenné môžu byť použité ako na získanie hodnoty faktu, tak aj jeho adresy.

Pokiaľ chceme CLIPSU naznačiť, že niektoré pole faktu nás nezaujíma, môžeme to spraviť pomocou tzv. *single-field wildcard* – **?**:

```
CLIPS> (defrule dating-ducks
(bachelor Dopey ?)
=>
(printout t "Date Dopey" crlf))
CLIPS>
(deffacts duck
     (bachelor Dicky)
     (bachelor Dopey)
     (bachelor Dopey Mallard)
     (bachelor Dinky Dopey)
     (bachelor Dopey Dinky Mallard))
```

Tento kód spôsobí to, že na ľavú stranu pravidla naviaže fakt `(bachelor Dopey Mallard)`, ktorý ako jediný spĺňa podmienku. Je možné použiť aj niekoľko wildcards za sebou: napr. `(bachelor Dopey ? ?)`. V tomto prípade by sa na ľavú stranu pravidla naviazal fakt `(bachelor Dopey Dinky Mallard)`.

Namiesto zložitej správy jednotlivých polí faktov je jednoduchšie použiť tzv. *multifield wildcard* – **$?**, ktorá reprezentuje $0 - n$ polí. Na rozdiel od nej sa *sigle-field wildcard* musí naviazať na práve jedno pole.

WIldcards majú dôležitú úlohu, pretože môžu byť prichytené k symbolickému poľu a vytvoriť tak premennú, ako je napr. `?x`, `$?x`, `?name`, `$?name`. Premenná môže byť buď *single-field* alebo *multifield*, v závislosti od toho je na LHS použité `?` alebo `$?`. Na pravej strane je vždy použité iba `?x`, kde `x` je meno ľubovoľnej premennej.

Deftemplate

Deftemplate je skratka od „*define template*“ a jej hlavné využitie je pri písaní pravidiel s dobre definovanou štruktúrou vzorov.

Deftemplate by sa dala prirovnať štruktúre napr. z jazyka C – podobne ako štruktúra, deftemplate definuje skupinu polí, ktoré majú nejaký vzťah.

Deftemplate je zoznam pomenovaných polí, ktoré sa nazývajú *sloty* (*slots*). Deftemplate umožňuje pristupovať ku konkrétnemu poľu podľa jeho mena, na rozdiel od prístupu na základe poradia (ako to bolo ukazované v príkladoch uvedených doteraz).

Slot je buď **single-slot** alebo **multislot**. Single-slot (zjednodušene nazývaný iba **slot**) obsahuje presne jedno pole, zatiaľ čo **multislot** môže obsahovať $0 - n$ polí. V rámci **deftemplate** môže byť použité ľubovoľné množstvo slotov a multislotov.

Vo všeobecnosti má **deftemplate** s N slotmi nasledujúcu štruktúru:

```
(deftemplate <name>
     (slot-1)
     (slot-2)
     ...
     (slot-N))
```

Konkrétny príklad môže potom vyzerať napríklad takto:

```
(deftemplate person ;meno deftemplate relácie
     "basic person information" ;voliteľný komentár v úvodzovkách
     (slot person_name      ;meno poľa
          (type STRING)     ;typ poľa (voliteľný)
          (default ?DERIVE));počiatočná hodnota poľa (voliteľná)
     (slot assets           ;meno poľa
          (type SYMBOL)     ;typ poľa
          (default rich))   ;počiatočná hodnota poľa
     (slot age              ;meno poľa
          (type NUMBER)     ;NUMBER môže byť INTEGER alebo FLOAT
          (default 80)))    ;počiatočná hodnota poľa s vekom
```

Táto **deftemplate** má 3 single-sloty, konkrétne *person_name*, *assets* a *age*. Počiatočné hodnoty sú vložené pri zavolaní príkazu (reset), pokiaľ nie sú explicitne uvedené žiadne iné hodnoty:

```
CLIPS> (assert (person)) ; vložme nový fakt
<Fact-0>                 ; odpoveď systému
CLIPS> (facts)           ; vypíšme zoznam faktov
f-0 (prospect (name "") (assets rich) (age 80))
For a total of 1 fact.
```

`?DERIVE` je kľúčové slovo, ktoré vyberie vhodný typ pre slot, napr. „“ v prípade slotu typu `STRING`. Pokiaľ chceme *explicitne* nastaviť hodnoty jednotlivých polí, môžeme to urobiť (bez ohľadu na ich poradie):

```
(assert (prospect (age 99) (name "Dopey")))
<Fact-1>
CLIPS> (facts)
f-1 (prospect (name "Dopey") (assets rich) (age 99))
```

Pri **deftemplate** je dôležité si uvedomiť, že `NUMBER` nie je primitívny typ poľa, ale že sa jedná o zložený typ, ktorý môže byť `INTEGER` alebo `FLOAT`. Používa sa v prípadoch, keď užívateľa nezaujíma, aký typ

čísla sa ukladá. Pokiaľ by sme chceli dosiahnuť rovnakú funkcionalitu slotu `age`, mohli by sme použiť tento zápis: `(slot age (type INTEGER FLOAT) (default 80))`. Tento príklad zároveň ukazuje možnosť špecifikácie viacerých dátových typov, ktoré môže slot prijímať.
Pokiaľ je to potrebné, môžeme špecifikovať konkrétne hodnoty alebo rozsah, ktoré má slot povolené prijímať, pomocou zápisu, ako je napr. `allowed-integers 1 4 8`. Ak by sme chceli prijímať rozsah, môžeme použiť napr. funkciu `(range 1 10)`. Nie je ale možné pre jeden slot vymenovať povolené hodnoty a zároveň aj špecifikovať rozsah hodnôt.

Deftemplate vzor môže byť použitý ako hocijaký "obyčajný" vzor so všetkým, čo k tomu prináleží.

Ako už bolo spomenuté vyššie, **deftemplate** umožňuje použiť viacero hodnôt v multislote. Ako príklad si skúsme predstaviť, že chceme brať slot `person_name` v relácii `person` ako niekoľko polí, teda aby osoba mohla mat aj viacero mien (resp. aby bolo jednoducho oddeliteľné meno a priezvisko). Toto sme schopní docieliť drobnou úpravou:

```
(multislot person_name
     (type SYMBOL)
     (default ?DERIVE))
```

Na vytlačenie informácií o osobe môžeme použiť nasledujúce pravidlo:

```
(defrule happy_relationship
     (person (person_name $?p_name) (assets ?net_worth) (age
?months))
     =>
     (printout t "Person: " ?p_name crlf ; Pozor! Nie $?name
                            ?net_worth crlf
                            ?months " months old" crlf))
```

Po vložení ukážkovej osoby a zavolaní funkcie **(run)**, dostaneme nasledujúci výstup:

```
CLIPS> (deffacts duck-bachelor
(prospect (name Dopey Wonderful) (assets rich) (age 99)))
CLIPS> (reset)
CLIPS> (run)
Prospect: (Dopey Wonderful)
Rich
99 months old
```

Zátvorky vo výpise okolo Dopeyho mena vkladá CLIPS automaticky aby upozornil, že sa jedná o hodnotu z multislotu. Keďže meno nie je v tejto verzii STRING, CLIPS ho berie za 2 nezávislé polia – *Dopey* a *Wonderful*.

Obmedzovače a počítanie s číslami

Pravidlá vo forme, v akej boli doteraz prezentované, umožňovali iba jednoduché testovanie splniteľnosti podmienok a následne vykonanie určitých akcií. CLIPS ale umožňuje obmedziť množinu hodnôt, ktoré môže vzor v pravidle na svojej ľavej strane mať.

Základným typom obmedzovača poľa je tzv. *connective constraint*, kt. má 3 podtypy:

1. *Tildu* (~) ktorá nepovoľuje hodnotu poľa; uvedený je príklad expertného systému, ktorý radí, čo má človek urobiť na prechode pre chodcov so semaforom:

```
(defrule walk
     (light ~green)
     =>
     (printout t "Don't walk" crlf))
```

   Pokiaľ na semafore nesvieti zelená, systém vytlačí správu, že nemáme ísť cez cestu.
2. *Bar constraint* (|), ktorá povolí naviazať hociktorú zo skupiny hodnôt:

```
(defrule cautious
     (light yellow | blinking-yellow)
     =>
     (printout t "Be cautious" crlf))
```

   Pokiaľ na semafore svieti *alebo* bliká oranžová, musíme byť opatrní.
3. *Ampérsand* (*&*), ktorý hovorí, že pole musí obsahovať všetky hodnoty spojené *&*:

```
(defrule not-yellow-red
     (light ?color&~red&~yellow)
     =>
     (printout t "Go, since light is " ?color crlf))
```

   Systém radí ísť, pokiaľ nie je na semafore červená a ani oranžová (takže napr. pre blikajúcu oranžovú toto pravidlo nevypáli).

Ďalším typom je tzv. *predicate constraint* obmedzovač, ktorý, pokiaľ nie je splnený, spôsobí nenaviazanie faktu. Tento obmedzovač môže napr. testovať, či je zadaná hodnota číslo, adresa faktu, nepárne číslo a tak ďalej.

Okrem práce so symbolickými faktami dokáže CLIPS takisto prevádzať numerické výpočty, hoci to nie je jeho primárny účel.

CLIPS ponúka základné matematické funkcie `+`, `-`, `*`, `/`, `div`, `max`, `min`, `abs`, `float` a `integer`. Numerické výrazy sú reprezentované v rovnakom štýle, ako v rámci LISPu, teda v *prefixovej forme zápisu*. Matematické funkcie môžu byť použité na oboch stranách pravidiel. Na ľavej strane pravidla môže byť funkcia použitá vtedy, keď použijeme operátor priradenia (=) k tomu, aby sme prikázali CLIPSu výraz vyhodnotiť namiesto toho, aby ho použil pre naviazanie vzoru.
V matematickom výraze nemusia byť operátori nutne binárne – tá istá sekvencia aritmetických výpočtov je použitá aj pre viac, ako 2 argumenty, napr.:

```
(defrule addition
     (numbers ?x ?y ?z)
     =>
     (assert (answer-plus (+ ?x ?y ?z)))) ; ?x + ?y + ?z
```

Toto pravidlo dokáže sčítať 3 čísla, napr. vo fakte `(numbers 1 2 3)` a vytvoriť odpoveď `(answer-plus 6)`. Pri práci s číslami sa CLIPS snaží zachovať vo výsledku typ operátorov – teda napr. súčet 2 `FLOAT` čísel dá výsledok typu `FLOAT`, keď sčítame 2 `INTEGER` čísla, dostaneme

INTEGER a pri mixe typov FLOAT. CLIPS ale umožňuje aj explicitnú konverziu medzi typmi pomocou funkcií **(integer x)** a **(float x)**, kde x je výraz, ktorého výsledok chceme previesť na daný dátový typ. Prioritu operátorov je možné špecifikovať zátvorkami.

Okrem základných matematických operácií (vrátane porovnávaní) CLIPS ponúka aj rozšírené, ktoré zahŕňajú napr. trigonometrické, hyperbolické a pod.

Funkcie a procedurálne programovanie

Podobne ako iné jazyky, aj CLIPS dovoľuje užívateľovi nadefinovať si vlastné funkcie, a to pomocou príkazu **deffunction**. Takto definované funkcie sú globálne a môžu byť volané rovnako, ako doteraz ukázané funkcie. Deffunction môže byť použitá ako argument v inej funkcii. Príkaz **(printout)** môže byť použitý všade v rámci funkcie a nie iba na konci, pretože výpis je v ňom implementovaný ako vedľajší efekt.

Syntax **deffunction** vyzerá takto:

```
(deffunction <function-name> [optional comment]
     (?arg1 ?arg2 ...?argM [$?argN]) ;zoznam argumentov
     (<action1>
     <action2>
     ...
     <action(K-1)>
     <actionK>)
```

Mená argumentov môžu byť rovnaké, ako mená premenných v pravidle bez toho, aby spôsobili konflikt. Posledný argument môže byt multipole. Napriek tomu, že každá z akcií môže vrátiť nejakú hodnotu, tieto hodnoty nie sú vrátené užívateľovi, ale sú zablokované. Deffunction vráti iba hodnotu poslednej akcie (actionK). Táto akcia môže byť buď funkciou, premennou alebo konštantou.

CLIPS ponúka veľké množstvo preddefinovaných funkcií, ktoré umožňujú meniť fakty, pristupovať k jednotlivým poliam, pracovať s číslami a pod. Plný zoznam týchto funkcií sa nachádza v manuáli.

Štruktúry pre procedurálne programovanie

CLIPS ponúka niektoré štruktúry z procedurálneho programovania ako sú:

- *Cyklus* **while**, vrátane možnosti ho ukončiť pomocou príkazu break
- *Vetvenie* **if**-**then**-**else**
- *Funkciu* **return**, ktorá okamžite ukončí aktuálne spracovávanú **deffunction**, obecnú funkciu, metódu alebo message handler
- *Funkcie* **open** *a* **close** na prácu so súbormi. Súbor, s ktorým CLIPS potom pracuje, je známy pod svojim globálnym menom, pod ktorým k nemu dokážu pristúpiť všetky pravidlá
- *Funkciu* **eval**, ktorá vyhodnotí ľubovoľný reťazec alebo symbol okrem konštrukcií s typu „def*“, ako je napr. **defrule**, **deffacts**, ...

Z pravej strany pravidla môže byť zavolaná ľubovoľná funkcia.

Objektovo-orientované programovanie v CLIPSe

Ako už bolo zmienené, CLIPS podporuje tri paradigmy: pravidlá, objekty a procedúry. Pravidlá už boli prebrané. Procedúry sú podporované pomocou generických funkcií, **deffunction** a užívateľom definovaných externých funkcií. Z objektovo-orientovaného paradigmu podporuje CLIPS všetkých 6 hlavných vlastností – triedy, abstrakciu, dedičnosť, zapúzdrenie, polymorfizmus a dynamickú väzbu.

CLIPS používa multiparadigmový prístup, ktorý umožňuje *kombinovať* všetky tri modely. Na prácu s objektami slúži *CLIPS Object-Oriented Language* (*COOL*). Základné pojmy z objektovo-orientovaného programovania sa používajú aj v ramci CLIPSu a COOLu. Okrem toho, atribúty objektu sú tu reprezentované *slotmi* a správanie sa objektu je vyjadrené jeho *handlermi správ*, ktoré prijímajú správy posielané tomu-ktorému objektu.

Základná práca s triedami a objektami

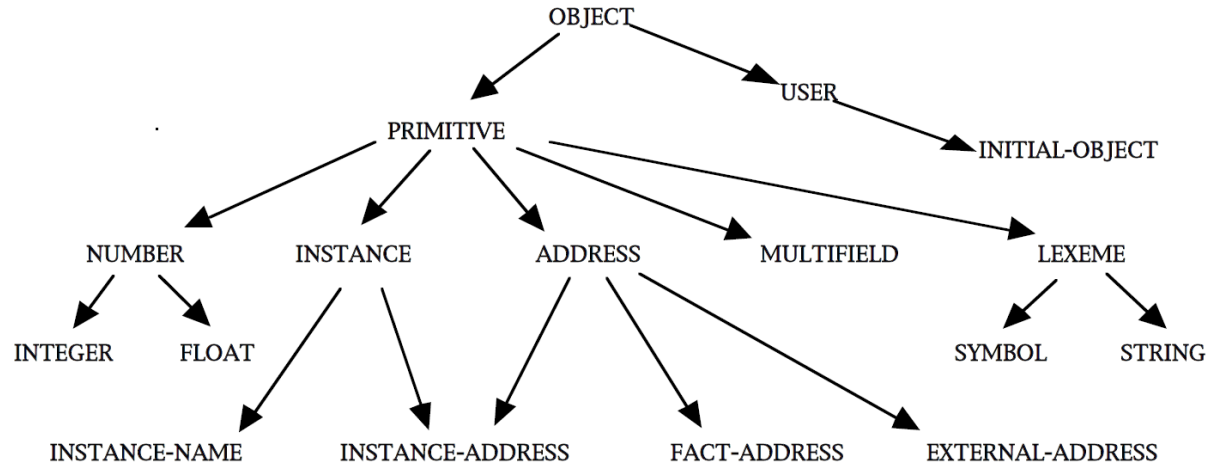
Všetky triedy môžu byť iba v „*je*“ (*is-a*) relácii, nie v „*má*“ (*has-a*) relácii. Existuje *základná trieda* (nazývaná `OBJECT`), od ktorej sú odvodené všetky ostatné triedy. Táto trieda nemá rodičovskú triedu. Každá zdedená trieda dedí aj všetky sloty a handlery správ rodičovskej triedy (pokiaľ nie je dedenie nejakého slotu a jemu prináležiacich handlerov správ explicitne blokované).

Definícia jednoduchej triedy bez slotov prebieha v CLIPSe pomocou príkazu:

```
(defclass <class> (is-a <direct-superclasses>))
```

CLIPS je *case-sensitive* a je zaužívanou konvenciou, že názvy tried sú písané kapitálkami. Takto sú pomenované aj všetky vstavané triedy. `<direct-superclasses>` je zoznam priamych rodičovských tried – CLIPS podporuje *viacnásobnú dedičnosť*. Rodičovská trieda musí byť vytvorená pred triedami, ktoré z nej dedia; v opačnom prípade zahlási CLIPS chybu. Užitočný príkaz je **(superclass c1 c2)**, ktorý vracia TRUE, pokiaľ c1 je priamym rodičom c2, inak vracia FALSE. Príkaz (subclass c1 c2) funguje na rovnakom princípe, ale testuje, či c1 je priamym potomkom c2. **(list-defclasses)** zasa dokáže vypísať všetky definované triedy. Nezobrazuje však ich hierarchiu. Na to slúži príkaz **(browse-classes)**, ktorý má voliteľný parameter udávajúci meno triedy, od ktorej sa má začať výpis.

Základné triedy, ktoré sú okamžite dostupné, ako aj ich hierarchiu, môžeme vidieť na Obrázku 1.



Obrázok 1: Hierarchie systémových tried v jazyku CLIPS

Preddefinované dátové typy CLIPSu sú taktiež definované ako triedy, takže s nimi môžeme v rámci COOLu pracovať. Triedy sú očíslované podľa svojej úrovne. Úroveň 0 má základná trieda `OBJECT`. Každý priamy potomok triedy ma úroveň o 1 stupeň vyššiu.

CLIPS rozlišuje medzi abstraktnou a konkrétnou triedou. Toto rozlíšenie prebieha pomocou *deskriptora* **(role concrete)** a **(role abstract)** v rámci definície triedy. Abstraktná trieda slúži iba na dedenie, nie je z nej možné vytvoriť priamu inštanciu (je ale možné vytvárať nepriame inštancie, čo sú inštancie potomkov, ktorí sú konkrétnymi triedami). Konkrétne triedy inštancie vytvárať môžu. Normálne triedy, ktoré dedia od abstraktných tried, sú takisto abstraktnými triedami. Výnimkou sú triedy, ktoré dedia od systémových (preddefinovaných) tried – tie sú konkrétne, pokiaľ nie je explicitne požiadané, aby boli tiež abstraktné.

Silno sa odporúča, aby všetky užívateľom definované triedy boli odvodené od triedy `USER`. Vtedy CLIPS automaticky poskytne handlery pre vypísanie triedy, jej inicializáciu, aj zmazanie.

Existujú jazyky (napr. Smalltalk), v ktorých je všetko, vrátane tried, objektom. CLIPS ale medzi nich nepatrí, pretože triedy objektami nie sú. Syntax pre meno konkrétnej inštancie je symbol uzavretý v `[]:[<name>]`
Tieto zátvorky nie sú v skutočnosti súčasťou mena inštancie, ktoré je vlastne symbol, ale chránia meno inštancie pred nejednoznačnosťou.

Existujú 2 funkcie na konverziu medzi symbolom a menom inštancie: **(symbol-to-instance-name)** a **(instance-name-to-symbol)**.

Keďže triedy nie sú objekty, nie je možné po vzore čisto objektovo-orientovaného prístupu poslať triede správu, aby vytvorila svoju inštanciu. Namiesto toho musíme zavolať funkciu **(make-instance)**. Jej základná syntax je:

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

Pokiaľ sa nedefinuje meno inštancie (zátvorky pri mene sú pre triedy zdedené od USER voliteľné), CLIPS ho vygeneruje automaticky pomocou **(gensym*)** funkcie a toto meno vráti:

```
CLIPS> (make-instance [Dorky] of DUCK)
      [Dorky]
CLIPS> (make-instance of DUCK)
      [gen1]
```

Podobne ako pri príkazoch **(rules)** a **(facts)**, CLIPS pozná funkciu **(instances)**, ktorá vypíše všetky vytvorené inštancie tried.

Ohľadom tried je dôležité uviesť ešte 2 pravidlá:

1. V rámci jedného modulu nemôžu mať 2 inštancie rovnaké meno.
2. Trieda, ktorá má vytvorenú nejakú inštanciu, nemôže byť nijak zmenená – napr. nemôžme premenovať jej sloty ani meniť jej handlery.

Pri príkaze **(reset)** sú všetky inštancie tried zmazané (rovnako, ako sa deje aj s faktami) a inštancia **[initial-instance]** je vytvorená, rovnako ako fakt **(initial-fact)**. Taktiež rovnako, ako **(deffacts)** definuje fakty, **(definstances)** automaticky definuje inštancie, ktoré sa vytvoria, keď je vykonaný príkaz **(reset)**:

```
CLIPS> (definstances DORKY_OBJECTS "The Dorky Cousins"
(Dorky of DUCK)
(Dorky_Duck of DUCK))
CLIPS> (instances)
[initial-object] of INITIAL-OBJECT
For a total of 1 instance.
CLIPS> (reset)
CLIPS> (instances)
[initial-object] of INITIAL-OBJECT
[Dorky] of DUCK
[Dorky_Duck] of DUCK
For a total of 3 instances.
CLIPS>
```

Pokiaľ chceme permanentne zmazať inštanciu, môžeme použiť funkciu **(unmake-instance)**. Druhým spôsobom je poslanie **delete** správy. Táto správa automaticky zmaže každú inštanciu zdedenú od `USER` triedy, ktorá ju príjme. Pokiaľ inštancia nie je zdedená od `USER` triedy, musíme najprv vytvoriť správny handler správy delete.

Posielanie správ objektom predstavuje jediný správny spôsob, ako môžu medzi sebou objekty komunikovať. Na to slúži funkcia **(send)**. Podobne, ako **deftemplate** dáva štruktúru vzoru pravidla, dávajú sloty štruktúru objektu. Ako pre **deftemplate**, tak aj pre objekty je **slot** pomenované miesto, v ktorom môžu byť uložené dáta. Na rozdiel od **deftemplate** ale objekty získavajú svoje sloty z tried a triedy používajú dedičnosť. *Neviazaný slot* je slot, ktorý nemá priradenú žiadnu hodnotu. Všetky sloty *musia* byť viazané.

Najjednoduchšie bude ukázať vytvorenie slotu a poslanie správy na príklade. Vytvoríme si triedu `DUCK` s 2 slotmi, ktoré na začiatku nebudú obsahovať žiadne dáta (takže ich hodnota bude *nil*):

```
CLIPS> (clear)
CLIPS> (defclass DUCK (is-a USER) (role concrete)
     (slot sound (create-accessor read-write))
     (slot age (create-accessor read-write)))
CLIPS> (definstances DORK_OBJECTS
     (Dorky_Duck of DUCK))
CLIPS> (reset)
CLIPS> (send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound nil)
(age nil)
CLIPS> (send [Dorky_Duck] put-sound quack)
quack
CLIPS> (send [Dorky_Duck] print)
[Dorky_Duck] of DUCK
(sound quack)
(age nil)
```

Sloty sú vytlačené v poradí, v akom sú definované v triede. Pokiaľ by trieda `DUCK` dedila od inej triedy obsahujúcej sloty, tieto zdedené sloty by boli vytlačené ako prvé.

Aspekty slotov

Tak ako sloty popisujú inštancie, *aspekty* (*facets*, ďalej v texte nazývané anglickým názvom) popisujú sloty. Jeden facet je použitý aj v príklade – **create-accessor**, ktorý vytvorí prístupové handlery (**put-** a **get-**) pre triedu. Všetky sloty majú štandardne vytvorené prístupové handlery automaticky, takže tento facet je v príklade iba na ilustráciu a nie je tam nevyhnutne potrebný. Pomocou **put-** handleru je ale potom poslať slotu **sound** správu, ktorá mu priradí hodnotu `quack`. Pokiaľ by **put-** alebo **get-** neuspeli, CLIPS vypíše chybovú hlášku:

```
(send [Dorky_Duck] put-color white) ; Slot color neexistuje
[MSGFUN1] No applicable primary message-handlers found for
putcolor.
FALSE
```

Existuje veľké množstvo facetov – ich úplný výčet sa nachádza v dokumentácii CLIPSu. Tu si popíšeme iba niekoľko najzaujímavejších:

- **default** – nastaví počiatočnú hodnotu slotu pri vytvorení alebo inicializácii inštancie. Za kľúčovým slovo default môže byť ľubovoľný platný CLIPSovský výraz (vrátane funkcií), ktorý neobsahuje premenné. Default môže byť **default-static**, ktoré spôsobí vyhodnotenie výrazu pri prvom zavolaní a ďalej sa už nevyhodnocuje (teda všetky inštancie triedy majú tú istú počiatočnú hodnotu) alebo **default-dynamic**, kedy sa výraz vyhodnocuje pri každej inicializácii inštancie, takže každá inštancia môže mať inú počiatočnú hodnotu. Príklad facetu nastavujúceho počiatočnú hodnotu je napr. `(slot ID (default-dynamic (gensym*)))`, kde **(gensym*)** je funkcia vracajúca pri každom zavolaní celé číslo o 1 väčšie, ako predchádzajúce.
- **storage** – definuje, kde bude uložená hodnota slotu – buď v inštancii (cez **storage local** facet) alebo v triede (cez **storage shared** facet). Uloženie hodnoty v rámci inštancie je prednastavené správanie.
- **access type** – spôsob, akým je možné pristupovať k slotom. Môže byť trojakého typu: čítanie aj zápis (**read-write**), iba čítanie (**read-only**) alebo prístup iba počas inicializácie (**initialize-only**).
- **propagation** – určuje, či je potomok môže dediť rodičov slot.
- **visibility** – určuje, či bude slot viditeľný potomkovi (**public**) alebo nie (**private**).

Handlery

Handlery podporujú zapuzdrenie objektov. Jediný správny spôsob, akým môže objekt reagovať na správu, je mať pre ňu vytvorený špeciálny *handler*, ktorý ju dokáže prijať a vykonať príslušné akcie.

Všeobecný formát handleru správ vyzerá takto:

```
(defmessage-handler <class-name> <message-name> [handler-type]
     [comment]
     (<parameters>* [wildcard-parameter])
     <action>*)
```

Hranaté zátvorky predstavujú voliteľný parameter. Hoci handler môže obsahovať niekoľko akcií, vracia iba hodnotu *poslednej z nich*. Poďme sa pozrieť na konkrétny príklad handleru. Predefinujeme si handler pre správu **print** pre nami vytvorenú triedu `USER`:

```
CLIPS> (defclass DUCKLING (is-a USER)
          (slot sound (default quack))
          (slot age (visibility public)))
CLIPS> (definstances DUCKY_OBJECTS
          (Dorky_Duck of DUCK (age 2)))
CLIPS> (defmessage-handler USER print before ()
          (printout t "*** Starting to print ***" crlf))
CLIPS> (send [Dorky_Duck] print)
*** Starting to print ***
[Dorky_Duck] of DUCK
(age 2)
(sound quack)
```

Takto nadefinovaný handler môžu využiť všetky triedy, ktoré majú triedu `USER` ako predka, vrátane triedy `DUCK`. Prázdne zátvorky pri definícii handleru znamenajú, že handler neberie žiadne parametre. Typ handleru je **before**, čo znamená, že sa vykoná pred spracovaním print správy, ako môžeme vidieť z výpisu. Opačným typom je typ **after**, ktorý sa vykoná po spracovaní **print** správy. Dôležité je poznamenať, že trieda `USER` už má definovaný (pomocou dedičnosti) tzv. *primárny* (**primary**) handler pre správu **print**, ktorý sa použije aj pri inštancii triedy `DUCK`. Posledným možným typom handleru je **around** handler, ktorého hlavnou úlohou je pripraviť prostredie objektu pre ďalšie handlery.

V závislosti od typu handleru, CLIPS vie, kedy ho má vykonať – **around** handler sa začne vykonávať pred akýmkoľvek iným handlerom. Potom nasleduje prevedenie **before** handleru, **primary** handlerov a nakoniec sa vykonajú **after** handlery. Pokiaľ bol špecifikovaný **around** handler, po skončení behu všetkých ostatných handlerov sa ukončí aj on.

Trieda `USER` ponúka nasledujúce preddefinované primárne handlery správ:

- **init** – inicializuje inštanciu
- **delete** – vymaže inštanciu
- **print** – vytlačí informácie o objekte
- **direct-modify** – priamo modifikuje sloty
- **message-modify** – modifikuje sloty pomocou put- správ
- **direct-duplicate** – duplikuje inštanciu bez put- správy
- **message-duplicate** – duplikuje inštanciu za použitia správ

Tieto primárne handlery sú preddefinované a *nemôžu byť modifikované* (pokiaľ nechceme meniť zdrojové kódy CLIPSu). Užívateľ si však môže definovať handlery typu **before**, **after** a **around**.

Pokiaľ potrebujeme v rámci handleru pristúpiť ku konkrétnej inštancii a jej slotu, CLIPS to umožňuje cez špeciálnu premmenú **?self**, ktorá nemôže byť explicitne uvedená ako argument handleru a ani ju nie je možné naviazať na inú hodnotu.

?self je užitočný, pretože nám umožňuje sprístupniť hodnotu slotu na čítanie. Na získanie hodnoty použijeme zápis `?self:<slot-name>`. Pomocou **?self** ale nie je možné zapísať do slotu hodnotu. Na zápis (rovnako aj na čítanie) môžu byť z handleru inštancie použité funkcie (**dynamic-get)** a (**dynamic-put)**, teda nie je teda nutné posielať správy (aj keď to je tiež možné: `(send ?self dynamic-get-<slot>)`).
Rozdiel medzi týmito funkciami a použitím **?self** je v tom, že `?self:<slot>` môže byť použité iba v triedach a potomkoch, ktorí zdedili slot `<slot>`, pretože `?self:<slot>` je vyhodnotený staticky cez dedičnosť. Teda pokiaľ potomok predefinuje slot, handler rodičovskej triedy, ktorý bude na jeho obsluhu zavolaný, skončí s chybovou hláškou. Naproti tou **(dynamic-get)** a **(dynamic-set)** prevádzajú dynamickú kontrolu slotov. Na to ale potrebujú aby **visibility** facet slotu bol **public**. Nasleduje príklad toho, ako je možné pomocou **(dynamic-put)** zmeniť hodnotu stolu inštancie:

```
(defmessage-handler DUCK lie-about-age (?change)
(bind ?new-age (- ?self:age ?change))
(dynamic-put age ?new-age)
(printout t "*** Starting to print ***" crlf
     "I am only " ?new-age crlf
     "*** Finished printing ***" crlf))
```

Tento handler používa premennú `?new_age` na to, aby ju odčítal od aktuálneho veku. Čo keby sme ale chceli zaručiť, aby vek nebolo možné meniť? Môžeme vytvoriť handler typu **around**, ktorý na to dohliadne (tento prístup slúži skôr na ilustráciu **around** handleru, než ako príklad reálneho riešenia daného problému):

```
CLIPS> (defmessage-handler DUCK lie-about-age around (?change)
          (bind ?old-age ?self:age)
          (if (next-handlerp) then
               (call-next-handler))
          (bind ?new-age ?self:age)
          (if (<> ?old-age ?new-age) then
               (printout t "Dorky_Duck is lying!" crlf
                    "Dorky_Duck is lying!" crlf
                    "He's really " ?old-age crlf)))
CLIPS> (make-instance [Dorky_Duck] of DUCK (age 3))
[Dorky_Duck]
CLIPS> (send [Dorky_Duck] lie-about-age 1)
*** Starting to print ***
I am only 2
*** Finished printing ***
Dorky_Duck is lying!
Dorky_Duck is lying!
He's really 3
CLIPS>
```

Handler typu **around** funguje takto:

1. Začni pred ostatnými handlermi.

2. Zavolaj ďalší handler buď pomocou funkcie **(call-next-handler)**, ktorému sú predané tie isté argumenty, ako **around** handleru, alebo pomocou funkcie **(override-next-handler)**, ktorej je možné predať iné parametre.
3. Pokračuj vo svojej činnosti, keď skončí posledný handler.

Kľúčové slovo **(call-next-handler)** slúži na zavolanie ďalšieho handleru. Môže byť použité aj viackrát. Na otestovanie, či nejaký „ďalší handler", existuje funkcia **(next-handlerp)**, ktorá vráti `FALSE`, pokiaľ už ďalší handler neexistuje. Hodnotu môže vracať iba handler typu **around** alebo **primary**. Ostatné môžu mať iba vedľajšie účinky.

Spojenie pravidiel a objektov

Jednou nových vlastností CLIPSu verzie 6.0 je jeho schopnosť naviazania vzorov na objekty. Nasledujúci príklad ukazuje, ako môže byť hodnota slotu `sound` naviazaná na vzor v pravidle:

```
CLIPS> (defclass DUCK (is-a USER)
          (multislot sound (default quack quack)))
CLIPS> (make-instance [Dorky_Duck] of DUCK)
[Dorky_Duck]
CLIPS> (make-instance [Dinky_Duck] of DUCK)
[Dinky_Duck]
CLIPS> (defrule find-sound
          ?duck <- (object (is-a DUCK)(sound $?find))
          =>
          (printout t "Duck " (instance-name ?duck) " says "
          ?find crlf))
CLIPS> (run)
Duck [Dinky_Duck] says (quack quack)
Duck [Dorky_Duck] says (quack quack)
```

V podmienke pravidla môžeme vidieť, že na premennú `?duck` sa pokúšame naviazať inštanciu triedy `DUCK` alebo niektorého z jej potomkov (pokiaľ takí existujú) a z tejto inštancie sa snažíme získať (multi)pole `sound`. Za názvom slotu môže byť obmedzovač obsahujúci `?`, `#?`, `&` a `|`. Okrem toho môžeme v podmienke pravidla špecifikovať aj mená inštancií pomocou obmedzovača **name** – toto je kľúčové slovo a nemôže byť použité ako názov slotu. Príklad použitia obmedzovača **name**:

```
CLIPS> (defrule find-sound
?duck <- (object (is-a DUCK)(sound $?find)(name [Dorky_Duck]))
=>
(printout t "Duck " (instance-name ?duck) " says " ?find crlf))
CLIPS> (run)
Duck [Dorky_Duck] says (quack quack)
CLIPS>
```

Vidíme, že pravidlo bolo vykonané iba pre *jedinú* inštanciu.

CLIPS (resp. COOL) umožňuje pokročilú správu a výber alebo filtrovanie objektov na základe dotazov, čo ešte viac umožňuje spojiť objektovo-orientovaný model s modelom založeným na pravidlách.

Porovnanie CLIPSu s inými nástrojmi

V tejto kapitole sa nachádza porovnanie CLIPSu ako s predstaviteľom procedurálnych jazykov – jazykom C – tak aj s jazykom Prolog, ktorý zastupuje opačný prístup k dosiahnutiu rozhodnutia – *backward chaining*.

CLIPS vs. jazyk C

Hoci je samotný CLIPS implementovaný v jazyku C, poskytuje oproti nemu niekoľko užitočných schopností:

- Implementovaný inference engine, ktorý sa dokáže na základe vstupných dát a pravidiel dostať (často aj ich komplexným aplikovaním) k záveru. Poskytuje teda istú formu vstavanej umelej inteligencie schopnej riešiť aj netriviálne problémy.
- Jednoduché rozšírenie funkcionality aplikácie a zvýšenie jej flexibility iba pomocou pridávania, odoberania a zmeny pravidiel. Systém ani nie je nutné rekompilovať, čo sa ocení hlavne pri rozsiahlejších systémoch, kde expertný systém tvorí iba časť celého systému. Okrem toho sa zložité zanorené podmienky z jazyka C dajú modelovať niekoľkými jednoduchými pravidlami.

Oproti jazyku C však ale prináša aj niekoľko nevýhod:

- Výpočtová náročnosť – pretože CLIPS aplikuje pravidlá tak dlho, pokiaľ už nie je možné aplikovať žiadne ďalšie, môže pri získavaní výsledku prehľadať veľkú časť stavového priestoru problému, čo môže trvať veľmi dlho.
- Potrebný čas na zžitie sa s CLIPSom – keďže veľká časť vývoja prebieha v procedurálnych a objektovo-orientovaných jazykoch, môže jazyk byť založený na pravidlách zo začiatku ťažký na pochopenie.

Vo všeobecnosti je vhodné CLIPS (resp. programovanie založené na pravidlách) použiť všade tam, kde sa vstupné dáta dajú reprezentovať ako fakty a narábanie s nimi, ako pravidlá. Užitočnosť nasadenia CLIPSu je najväčšia tam, kde v jednom okamihu je zvyčajne možné použiť viacero pravidiel (teda viacero aktivovaných pravidiel sa nachádza v agende) – vtedy práve vyniknú vlastnosti CLIPSu a podobných nástrojov, ktoré dokážu vybrať „vhodné“ pravidlo. Pokiaľ by sme vždy mali na výber iba z jedného pravidla, použitie CLIPSu by stratilo zmysel a jednoduchšie by pravdepodobne bolo použiť iný nástroj či programovací jazyk (ako napr. C).

CLIPS vs. Prolog

CLIPS aj Prolog sú oba nástroje, ktoré pracujú s pravidlami a za použitia inference enginov a matematickej logiky robia rozhodnutia. Oba dokážu pracovať ako v dávkovom (systém dostane na začiatku svojho behu všetky dôležité dáta a z nich vyvodí nejaký záver), tak aj v interaktívnom režime (systém aktívne komunikuje s užívateľom).

V čom sa ale CLIPS a Prolog líšia, je spôsob, akým sa dostanú k vyvodeniu záveru. CLIPS používa tzv. *forward chaining*, zatiaľ čo Prolog používa *backward chaining*. Nasleduje popis oboch typov vyvodzovania a ich následné porovnanie.

Forward chaining

Inference engine, ktorý používa forward chaining, postupne (podľa nejakej heuristiky – v prípade CLIPSu to je stratégia, ktorá zoraďuje pravidlá v agende) prehľadáva pravidlá až kým nenájde také, ktorého

podmienková (*if*) časť je splniteľná. Keď takéto pravidlo nájde, odvodí (prevedie) časť s akciami (*then*), ktorá zvyčajne spôsoby zmenu informácii (faktov), s ktorými engine pracuje. Povedané inými slovami, *forward chaining* mechanizmus začína s nejakými faktami, na ktoré postupne aplikuje pravidlá a snaží sa nájsť všetky možné závery, ktoré sa z faktov dajú vyvodiť. Aj preto sa tento prístup nazýva „*data driven approach*".

Činnosť systému implementujúceho *forward chaining* sa dá popísať ako opakovanie 2 krokov:

1. Nájdi tie pravidlá, ktoré sa na základe aktuálneho stavu zoznamu faktov dajú vypáliť.
2. Vyber z nich (na základe aktuálnej stratégie) jedno a to vypáľ.

Tieto kroky sa opakujú tak dlho, až kým už neexistuje žiadne pravidlo, ktoré môže vypáliť.

Backward chaining

Inference engine implementujúci *backward chaining* používa iný prístup – prehľadáva pravidlá, až kým nenájde nejaké, ktoré má vo svojej *then* klauzule uvedený cieľ, ktorý chce systém dosiahnuť. Pokiaľ ale o podmienkovej (*if*) klauzule tohto pravidla nedokáže povedať, či je splniteľná alebo nie, tak sa pridá do zoznamu cieľov – pokiaľ má totiž cieľ platiť, musí poskytnúť dáta, ktoré tento cieľ podporujú. Inými slovami – *backward chaining* začína s požadovaným záverom a spätne sa snaží nájsť fakty, ktoré tento záver podporujú. Preto sa tento prístup nazýva aj „*goal-driven approach*".

Činnosť systému implementujúceho *backward chaining* sa dá popísať takto:

1. Nájdi medzi pravidlami také ktoré dokážu potvrdiť požadovaný záver.
2. Tieto pravidlá postupne spracuj:
    a. Pre každé pravidlo postupne vyhodnocuj jednotlivé podmienky v podmienkovej (*if*) časti:
        i. Ak splniteľnosť podmienky momentálne nedokážeš potvrdiť, pridaj potvrdenie jej pravdivosti do zoznamu cieľov a rekurzívne zavolaj vyhodnocovanie.
        ii. Pokiaľ vieš, že podmienka nie je splniteľná, alebo si nedokázal jej splniteľnosť určiť, pokračuj v cykle krokom 2.
    b. Pokiaľ všetky podmienky v podmienkovej časti pravidla boli splnené, pridaj do zoznamu faktov tie fakty, ktoré boli špecifikované v „*then*" časti pravidla, vymaž cieľ zo zoznamu cieľov a vráť sa z tohto vyhodnocovania.

Backward chaining systém skončí s úspechom, pokiaľ je zoznam cieľov prázdny. Pokiaľ mu v kroku 2 dôjdu pravidlá, skončí s neúspechom.

Porovnanie forward a backward chainingu

Použitie *forward* (*FC*) alebo *backward chainingu* (*BC*) závisí od úlohy, ktorú ma daný systém riešiť – zatiaľ čo systémy s *BC* sú vhodné pre diagnostiku a klasifikáciu, *FC* systémy štandardne dosahujú lepšie výsledky pri plánovaní alebo monitorovaní. To súvisí s ich postupom pri práci - zatiaľ čo *BC* sa snaží obmedziť preskúmavanie stavového priestoru iba na cesty, ktoré sú nevyhnutné pre splnenie cieľa, *FC* takéto obmedzenie štandardne nemá. Navyše, pri *FC* musíme písať extra pravidlá pre správu podcieľov; *BC* to zvláda automaticky.

Takisto pri *BC* je prehľadávanie stavového priestoru riadené hlavným cieľom (a podcieľmi), takže sú aplikované iba pravidlá, ktoré sú nutné na dosiahnutie cieľa. Zatiaľ čo pri *FC* nie je beh systému smerovaný ku konkrétnemu cieľu, teda nie je zrejmé, kedy skončí a s akým výsledkom.

Vo všeobecnosti, systém implementujúcu *forward chaining* je vhodné použiť tam, kde množstvo spôsobov, ktorými je možné dosiahnuť konkrétny záver, je veľké, ale počet záverov, ktoré môžeme dosiahnuť pomocou faktov, je relatívne malý. Pokiaľ naopak fakty vedú k veľkému množstvu záverov, ale existuje relatívne malé množstvo spôsobov, ako ich dosiahnuť, je lepšou voľbou systém implementujúci *backward chaining*.

Záver

CLIPS je rozhodne zaujímavým nástrojom, ktorý v sebe implementuje hneď tri rôzne programovacie paradigmy – programovanie založené na pravidlách, procedurálne a objektovo orientované programovanie, ktoré zároveň dokáže efektívne navzájom prepojiť. Jeho veľkou výhodou je jeho integrovateľnosť do veľkého množstva programovacích jazykov, voľne dostupné zdrojové kódy, ako aj možnosť volať externé funkcie. Od svojho vytvorenia v osemdesiatych rokoch urazil veľký kus cesty a okrem agentúry NASA bol použitý aj na veľkom množstve rôznych projektov (napr. v rámci nemeckého Deutsches Zentrum für Luft- und Raumfahrt). Momentálne je dostupná beta verzia 6.30, ktorá ukazuje, že tento nástroj je neustále vyvíjaný.

Použitá literatúra

[1] CLIPS User's Guide (http://clipsrules.sourceforge.net/OnlineDocs.html)

[2] CLIPS Programming Guide (http://clipsrules.sourceforge.net/OnlineDocs.html)