Materiały do ćwiczeń z przedmiotu „Sztuczna Inteligencja”

Artur Michalski

Kierunek Informatyka

Studia Inżynierskie

Szkieletowy system ekspercki CLIPS

Część 1

• Wprowadzenie

• Typy wartości danych

• Fakty

• Reguły

• Operacje we/wy

• Śledzenie działania programu

Wprowadzenie

Głównym celem zajęć laboratoryjnych z przedmiotu „Sztuczna Inteligencja" jest zaprojektowanie oraz zaimplementowanie prostego systemu eksperckiego. Z tego względu konieczne jest zapoznanie się z narzędziem przeznaczonym do realizacji tego zadania. Należy ono do szerokiej klasy aplikacji zwanych systemami z bazą wiedzy i jest to szkieletowy system ekspercki CLIPS (clipsrules.sf.net). Obrany cel zajęć laboratoryjnych powoduje, że mają one dwojaki charakter. Z jednej strony konieczne jest przeprowadzenie ćwiczeń, w ramach których możliwe będzie zaznajomienie się zarówno z regułową formą reprezentacji wiedzy w systemie CLIPS, jak i mechanizmami związanymi z procesem wnioskowania z użyciem tej wiedzy. Z drugiej zaś strony umiejętności nabyte w ten sposób należy następnie wykorzystać do zaimplementowania systemu eksperckiego w wybranej przez twórcę dziedzinie.

Poniżej zaprezentowano materiał, będący opisem wszystkich tych elementów systemu CLIPS, które są niezbędne do wykonania projektu własnego, prostego systemu eksperckiego. Większość omawianych przykładów polega na rozwiązywaniu prostego zadania programistycznego, z wykorzystaniem formy reprezentacji i sposobu przetwarzania, charakterystycznego dla systemów reguł produkcji.

Druga części opracowania zawiera opis wymagań jakie musi spełnić realizowany projekt systemu eksperckiego.

Typy wartości danych

Symbole (inaczej: słowa)

Ciąg znaków, rozpoczynający się od widocznego znaku kodu ASCII,
po którym mogą pojawić się inne znaki. Reprezentuje pojedynczą wartość.

Symbol nie może rozpoczynać się od znaku:

```
< | & $ ? + - ( ) ;
```

oraz nie może zawierać:

```
< | & ( ) ;
```

Przykłady poprawnie zdefiniowanych słów:

```
alarm
lista
pozar_budynku
imie-i-nazwisko
!?#$*
```

Łańcuchy znakowe (ang. *string*):

Dowolny ciąg widocznych znaków.
Ogranicznik początku i końca w postaci cudzysłowu.
Długości zero lub więcej znaków.

Przykłady poprawnie zdefiniowanych łańcuchów znakowych:

```
"Długość listy wynosi: "
"Zamknięcie zaworu nr 17"
"\"cudzysłowy\" oraz \\ ukośniki\\"
"<;—)"
```

Liczby:

Wartości numeryczne zmiennoprzecinkowe składają się ze znaku, liczby i wykładnika.
Znak i wykładnik są opcjonalne.

Przykłady poprawnie zdefiniowanych liczb:

```
1    1.5       +3   .99       —100      3.1415e66
```

Fakty

Podstawowa forma reprezentacji danych, na których prowadzone jest wnioskowanie w systemie CLIPS. Każdy fakt składa się z jednego lub więcej **pól**, ujętych między dwa okrągłe nawiasy: otwierający i zamykający.

Przykłady poprawnie zdefiniowanych faktów:

```
(jedno-pole)
(dwa pola)
(max-predkosc 200 km/h)
(osoba "Jan Kowalski")
(cena 99 zlotych 99 groszy)
```

Zapis składni

W opisie składni obiektów języka system CLIPS prezentowanej w tym opracowaniu zawsze podane jest słowo kluczowe polecenia albo symboliczna nazwa pierwszego pola, po której pojawiają się (bądź nie) kolejne elementy danej konstrukcji języka; zapis składni może wskazywać zarówno dokładną liczbę składowych, jak i ogólnie pewną ich liczbę; w dalszej części dokumentacji konsekwentnie będzie wykorzystywana taka właśnie notacja.

Przykładowo dla faktów:

```
(obiad pomidorowa zraz szarlotka)
(obiad spaghetti vitello-arosto tiramisu)
(obiad rosol pieczen)
```

możemy podać dokładną składnię:

```
(obiad <pierwsze danie> <danie główne> <deser>)
```

lub też bardziej ogólnie składnię:

```
(obiad <<dania>>)
```

Czasami wystąpienie danego elementu w konstrukcji jest opcjonalne:

```
(obiad <pierwsze danie> <danie główne> [<deser>])
```

Praca z systemem CLIPS

Środowisko systemu CLIPS opiera się na interpreterze poleceń, który służy do wykonywania różnych komend systemowych.

Uruchomienie systemu:

```
C:\> clips
     CLIPS(V6.24 06/06)
CLIPS>
```

W tym momencie interpreter jest gotowy do przyjęcia polecenia od użytkownika.

Zakończenie pracy z systemem:

```
CLIPS> (exit)
C:\>
```

Każda polecenie systemu CLIPS musi być umieszczone między **lewym i prawym okrągłym nawiasem**; ich brak powoduje błędy składniowe; nie ma w systemie komend, które pozbawione są nawiasów.

Operacje na faktach

Dodawanie faktu do pamięci:

Wszystkie fakty zdefiniowane w systemie CLIPS są przechowywane w globalnej pamięci określanej mianem pamięci roboczej. Nowe fakty mogą być dodane do pamięci za pomocą polecenia `assert`.

Składnia

```
(assert <<fakt>>)
```

Przykład

```
CLIPS> (assert (alarm pozar))
<Fact-0>
CLIPS>
```

Wszystkie fakty w pamięci roboczej są numerowane (indeksowane), poczynając od wartości zero.

Wyświetlanie faktów znajdujących się pamięci:

Zawartość pamięci roboczej może być wyświetlona za pomocą komendy `(facts)`.

Składnia

```
(facts [<od> [<do> [<max>]]])
```

gdzie `<od>`, `<do>` oraz `<max>` to liczby całkowite dodatnie, odpowiadające indeksom faktów, które mają być wyświetlone.

Przykład

```
CLIPS> (assert (alarm powodz))
CLIPS> (facts)
f-0 (alarm pozar)
f-1 (alarm powodz)
CLIPS>
```

Usuwanie faktu z pamięci:

Fakty można bezpowrotnie usuwać z pamięci roboczej za pomocą polecenia `(retract)`. Argumentami polecenie nie są fakty, lecz ich indeksy.

Składnia

```
(retract <<indeks-faktu>>)
```

Przykład

```
CLIPS> (assert (alarm pozar)(alarm powodz))
CLIPS> (facts)
f—0 (alarm pozar)
f—1 (alarm powodz)
CLIPS> (retract 0)
CLIPS> (facts)
f—1 (alarm powodz)
CLIPS>
```

Reguły produkcji

Definiowane reguł:

Reguły produkcji to podstawowa forma reprezentacji wiedzy w systemie CLIPS. Definiujemy je za pomocą polecenia `(defrule)`.

Składnia

```
(defrule <nazwa-reguły> [<opcjonalny-komentarz>]
[<<warunki>>]  ; lewa strona reguły
=>
[<<akcje>>])   ; prawa strona reguły
```

Przykład

```
CLIPS> (defrule pozar "Uwaga!!!"
(alarm pozar)
=>
(assert (wezwanie straz)))
CLIPS>
```

Uwagi:

Reguła może nie posiadać żadnych warunków i/lub akcji.
Jeżeli reguła nie posiada warunków, to jest uaktywniania specjalnym predefiniowanym faktem systemowym o treści `(initial-fact)`.

Agenda:

Dla każdej reguły zdefiniowanej w systemie CLIPS podejmowana jest próba spełnienia jej warunków poprzez znalezienie w pamięci roboczej faktów, które można dopasować do lewej strony reguły. Warunki są zatem traktowane jak wzorce faktów poszukiwanych w pamięci roboczej. Jeśli wszystkie warunki są spełnione (dla każdego wzorca znajdzie się odpowiedni fakt w pamięci roboczej), to reguła jest uaktywniana i umieszczana na **agendzie**. Agenda jest wewnętrzną strukturą mechanizmu wnioskowania w systemie CLIPS i reprezentuje listę aktywnych reguł. Z zawartością agendy można się zapoznać, wydając polecenie `(agenda)`.

Składnia

```
(agenda)
```

Przykład

```
CLIPS> (agenda)
0    pozar     f—0
CLIPS>
```

Uwagi:

Aktywacja reguł nie wymaga wykonywania żadnych dodatkowych poleceń ze strony użytkownika. Realizowana jest całkowicie automatycznie przez system po tym jak zaobserwowane zostanie spełnienie warunków reguły w chwili pojawienia się pewnych nowych faktów i/lub nowych reguł.

Wykonanie akcji reguły:

Do rozpoczęcia procesu przetwarzania (wnioskowania) w systemie CLIPS wykorzystywana jest komenda `(run)`. Pozwala ona systemowi przejść do wykonania akcji reguły.

Składnia

```
(run [<limit>])
```

gdzie `<limit>` oznacza liczbę kolejnych uaktywnianych reguł, których akcję zamierzamy wykonać.

Przykład

```
CLIPS> (run)
CLIPS> (agenda)
CLIPS>
```

Uwagi:

Po wykonaniu akcji reguły (tzw. odpaleniu reguły) jest ona automatycznie usuwana z agendy i nie będzie ponownie aktywowana przez te same fakty. Jedynie usunięcie tych faktów i dodanie do pamięci roboczej po raz drugi może spowodować ponowną aktywację reguły.

Inne operacje na regułach:

Wyświetlenie listy nazw wszystkich aktualnie zdefiniowanych reguł jest możliwe dzięki poleceniu `(rules)`.

Składnia

```
(rules) lub (list-defrules)
```

Komenda `(ppdefrule)` umożliwia zapoznanie się z treścią pojedynczej reguły.

Składnia

```
(ppdefrule <nazwa-reguły>)
```

Regułę można również bezpowrotnie usunąć z systemu CLIPS za pomocą polecenia `(undefrule)`.

Składnia

```
(undefrule <nazwa-reguły>)
```

Operacje wejścia/wyjścia

Polecenie zapisu na standardowe urządzenie wyjściowe:

Oprócz operacji dodawania i usuwania faktu z pamięci roboczej akcją prawej strony reguły może być również polecenie (`printout`), wykorzystywane do wyświetlania napisów na konsoli (terminalu).

Składnia

```
(printout <nazwa-logiczna> <<argumenty>>)
```

gdzie `<nazwa-logiczna>` oznacza urządzenie wyjściowe (np. **`t`** to standardowe urządzenie wyjściowe jakim jest terminal), zaś `<<argumenty>>` to wszystkie elementy, których wartości chcemy wyświetlić.

Przykład

```
CLIPS> (defrule pozar
(alarm pozar)
=>
(printout t "Wezwij straż pożarną!" crlf))
CLIPS> (assert (alarm pozar))
CLIPS> (run)
Wezwij straz pozarna!
CLIPS>
```

Uwagi:

Symbol `crlf` jest predefiniowanym symbolem systemowym, który oznacza przejście na początek nowego wiersza i może być wykorzystywany tylko jako argument polecenia `printout`.

Grupy faktów

Definiowanie grupy faktów:

Wykorzystywanie polecenia `assert` do wprowadzania dużych zbiorów faktów jest bardzo niewygodne, dlatego też wszystkie fakty, które są niezbędne do rozpoczęcia procesu wnioskowania definiuje się za pomocą komendy `(deffacts)`.

Składnia

```
(deffacts <nazwa-grupy> [<opcjonalny-komentarz>]
<<fakty>>)
```

Umieszczenie tak zdefiniowanych faktów w pamięci roboczej jest możliwie dopiero po wydaniu komendy `(reset)`. Dzięki temu zawsze na dowolnym etapie procesu wnioskowania możemy ustawić system w stan początkowy, określony za pomocą faktów zapisanych w grupach.

Składnia

```
(reset)
```

Przykład

```
CLIPS> (deffacts status-alarmu
(alarm pozar)
(pozar-grupy A))
CLIPS> (reset)
CLIPS> (facts)
f-0 (initial-fact)
f-1 (alarm pozar)
f-2 (pozar-grupy A)
CLIPS>
```

Uwagi:

Wykonanie polecenia `(reset)` powoduje również automatyczne skasowanie wszystkich faktów, które znajdowały się dotychczas w pamięci roboczej, wyzerowanie licznika indeksów faktów oraz dodanie systemowego faktu `(initial-fact)` z indeksem 0.

Inne operacje na grupach faktów:

Wyświetlenie listy nazw wszystkich aktualnie zdefiniowanych w systemie grup faktów jest możliwe dzięki poleceniu `(list-deffacts)`.

Składnia

```
(list—deffacts)
```

Komenda `(ppdeffacts)` umożliwia zapoznanie się z zawartością pojedynczej grupy.

Składnia

```
(ppdeffacts <nazwa-grupy>)
```

Grupę faktów można również bezpowrotnie usunąć z systemu CLIPS za pomocą polecenia `(undeffacts)`.

Składnia

```
(undeffacts <nazwa-grupy>)
```

Obsługa zbiorów definicji

Wszystkie obiekty, których zostały zdefiniowane w systemie CLIPS mogą być przechowywane w plikach zapisanych na zewnętrznych nośnikach danych. Pliki te są plikami tekstowymi i rozpoznawane są dzięki rozszerzeniu `*.clp`.

Operacje odczytu i zapisu definicji:

Definicje reguł i grup faktów można załadować do systemu CLIPS ze zbioru na dysku za pomocą polecenia `(load)`.

Składnia

```
(load <ścieżka-do-pliku>)
```

Można również zapisać takie definicje do pliku i służy do tego komenda `(save)`.

Składnia

```
(save <ścieżka-do-pliku>)
```

Argumentem poleceń jest ścieżka, wskazująca położenie pliku docelowego w formacie zgodnym z językiem ANSI C.

Operacja zerowania systemu:

System CLIPS umożliwia skasowanie wszystkich obiektów zapisanych w pamięci systemu. Przywracany jest wtedy taki stan systemu jaki mamy tuż po jego uruchomieniu – system pozbawiony jakichkolwiek danych i jakiejkolwiek wiedzy.

Składnia

```
(clear)
```

Uwagi:

Okienkowy interfejs środowiska systemu CLIPS ułatwia realizację tych poleceń – możliwy jest zapis i odczyt definicji bez znajomości składni komend jedynie za pomocą myszki i rozwijanego menu.

Debugowanie przebiegu wnioskowania

Śledzenie zmian:

Dzięki poleceniu `(watch)` możliwe jest w systemie CLIPS śledzenie faktów, reguł, ich aktywacji oraz wykonań.

Składnia

```
(watch {facts, rules, activations, all})
```

gdzie argument `all` oznacza śledzenie wszystkich obiektów programu, biorących udział w procesie wnioskowania.

Jeżeli śledzenie dotyczy tylko faktów (opcja `facts`), generowane są komunikaty zarówno wtedy gdy fakt jest dodawany do pamięci roboczej, jak i wtedy gdy jest usuwany.

Jeśli śledzone są uaktywnienia (opcja `activations`), system CLIPS generuje odpowiednie komunikaty za każdym razem kiedy reguła (dokładniej: jej aktywacja) jest dodawana do agendy lub z niej usuwana.

Jeżeli śledzenie ma dotyczyć tylko reguł (opcja `rules`), to system informuje nas zawsze kiedy dochodzi do odpalenia reguły.

Do zakończenia procesu śledzenia służy polecenie `(unwatch)`.

Składnia

```
(unwatch {facts, rules, activations, all})
```

Śledzenie dopasowań:

System CLIPS oferuje możliwość sprawdzenia częściowych dopasowań faktów do warunków reguł. Służy do tego polecenie `(matches)`.

Składnia

```
(matches <nazwa-reguły>)
```

Przykład

```
CLIPS> (defrule odciac-zasilanie
(alarm pozar)
(pozar-grupy C)
=>
(printout t "Odetnij zasilanie!" crlf))
CLIPS> (watch facts)
CLIPS> (assert (alarm pozar))
==> f-0 (alarm pozar)
CLIPS> (matches odciac-zasilanie)
Matches for Pattern 1
f-0
Matches for Pattern 2
None
Activations
None
CLIPS>
```

Pułapki:

Pułapki w systemie CLIPS odnoszą się do momentu odpalania reguły, a nie wykonania pojedynczej akcji reguły.

Założenie pułapki za pomocą polecenia `(set-break)` oznacza, że system zatrzyma proces wnioskowania tuż przed odpaleniem reguły, na której ustawiono pułapkę. Musi jednak wcześniej dojść do uaktywnienia reguły. Jeżeli reguła nie była aktywna, to pułapka nie zadziała, gdyż odpalane są wyłącznie te reguły, które wcześniej były uaktywnione.

Składnia

```
(set—break <nazwa-reguły>)
```

Można wyświetlić listę reguł, na których założono pułapki, korzystając z komendy `(show-breaks)`.

Składnia

```
(show—breaks)
```

Usunięcie pułapki realizujemy za pomocą polecenia `(remove-break)`.

Składnia

```
(remove-break [<nazwa-reguły>])
```

Jeżeli podany jest argument `<nazwa-reguły>`, to pułapka usuwana jest wyłączenie ze wskazanej reguły. W przypadku przeciwnym usuwane są wszystkie pułapki.

Przykład

```
CLIPS> (defrule 1-sza
=>
(assert (odpal druga)))
CLIPS> (defrule 2-ga
(odpal druga)
=>
(assert (odpal trzecia)))
```

```
CLIPS> (defrule 3-cia
(odpal trzecia)
=> )
CLIPS> (watch all)
CLIPS> (reset)
==> f-0 (initial-fact)
==> Activation      0 1-sza: f-0
CLIPS> (run)
FIRE 1    1-sza: f-0
==> f-1 (odpal druga)
==> Activation      0 2-ga: f-1
FIRE 2    2-ga: f-1
==> f-2 (odpal trzecia)
==> Activation      0 3-cia: f-2
FIRE 3    3-cia: f-2
3 rules fired
```

A teraz podobnie ale z pułapkami:

```
CLIPS> (set-break 2-ga)
CLIPS> (set-break 3-cia)
CLIPS> (watch all)
CLIPS> (reset)
==> f-0 (initial-fact)
==> Activation      0 1-sza: f-0
CLIPS> (run)
FIRE 1 1-sza: f-0
==> f-1 (odpal druga)
==> Activation      0 2-ga: f-1
Breaking on rule 2-ga
1 rules fired
CLIPS> (run)
FIRE 1 2-ga: f-1
==> f-2 (odpal trzecia)
==> Activation      0 3-cia: f-2
Breaking on rule 3-cia
1 rules fired
CLIPS> (run)
FIRE 1 3-cia: f-2
1 rules fired
CLIPS>
```

Szkieletowy system ekspercki CLIPS

Część 2

• Zmienne

• Ograniczniki wartości pól

• Wyrażenia numeryczne

• Ćwiczenia

• Priorytety reguł

• Operacje logiczne

• Wyrażenia boolowskie

• Operacje we/wy na plikach

• Ćwiczenia końcowe

Zmienne proste

Identyfikator zmiennej:

W systemie CLIPS zmienna służy – podobnie jak w innych językach programowania – do przechowywani wartości. Nie ma natomiast mechanizmu deklarowania zmiennych. Możliwe jest jednak rozpoznanie identyfikatora zmiennej dzięki określonej składni – identyfikator zaczyna się znakiem zapytania, po którym następuje ciąg znaków zgodny z definicją symbolu.

Przykłady poprawnych identyfikatorów:

```
?licznik
?X
?suma
?koszt_netto
```

Zanim użyjemy zmiennej w akcji reguły (po prawej stronie reguły) musi ona przyjąć pewną wartość w trakcie dopasowania faktu do warunku reguły - jest to tzw. proces wiązania zmiennej.

Przykład

```
CLIPS> (defrule dziadek
(jest-dziadkiem ?imie)
=>
(printout t ?imie " zostal dziadkiem." crlf))
CLIPS> (assert(jest-dziadkiem Jan))
CLIPS> (run)
Jan zostal dziadkiem.
CLIPS>
```

Operator wiązania indeksu faktu:

Usunięcie faktu w systemie CLIPS możliwe jest tylko po podaniu jego indeksu w poleceniu `retract`. Aby kasowanie faktów było wygodne i możliwe do wykonania przez regułę w trakcie procesu wnioskowania, konieczne jest użycie systemowego operatora wiązania zmiennej z indeksem przypisanym faktowi. W tym celu w warunkach reguły (i tylko tam) stosujemy operator “<– “, poprzedzający wzorzec faktu, który zamierzamy odnaleźć w pamięci i usunąć.

Przykład

```
(deffacts dziadkowie
     (jest-dziadkiem Jan)
     (jest-dziadkiem Jacek)
     (jest-dziadkiem Tomasz))

(defrule modifkuj-dane
     ?indeks <– (jest-dziadkiem ?imie)
     =>
     (retract ?indeks)
     (assert   (ma-wnuka ?imie)
               (jest-mezczyzna ?imie)))
```

Przykład

```
(deffacts dane
     (zeton))

(defrule petla-prosta
     ?stary <– (zeton)
     =>
     (printout t ”Zapetlenie ...” crlf)
     (retract ?stary)
     (assert (zeton)))
```

Wielokrotne wystąpienie zmiennej:

Zmienne, które pojawiają się wielokrotnie w części warunkowej reguły (po lewej stronie) mają pewną ważną cechę. Otóż, wiązanie zmiennej następuje podczas pierwszego jej wystąpienia, a wszystkie inne jej wystąpienia muszą przyjąć tę samą wartość.

Przykład

```
(deffacts dane
     (ojciec Tomek Jan)
     (ojciec Robert Jan)
     (ojciec Jan Jacek)
     (ojciec Bartek Adam)
     (ojciec Zuza Adam)
     (ojciec Adam Marek))

(defrule szukaj-dziadka
     (ojciec ?wnuk ?ojciec)
     (ojciec ?ojciec ?dziadek)
     =>
     (printout t ”Dziadkiem ” ?wnuk ” jest ” ?dziadek crlf))
```

Po wykonaniu polecenia `(reset)` wszystkie zdefiniowane fakty zostaną dopasowane do pierwszego warunku reguły `szukaj-dziadka`. Zmienna `?ojciec` zostanie związana z wartościami Jan, Jacek, Adam i Marek. Fakty posiadające taką właśnie wartość ostatniego pola, spełniły pierwszy warunek i są to wszystkie spośród nich. Warunek drugi reguły może spełnić jednak już tylko część faktów, gdyż każda z tych wartości (Jan, Jacek, Adam ,Marek) musi się powtórzyć jako drugie pole faktu dopasowywanego do drugiego wzorca. Przykładowo, kiedy pierwszy warunek spełnia fakt `(ojciec Robert Jan)`, drugi warunek może spełnić tylko fakt `(ojciec Jan Jacek)`.

Zmienna anonimowa (prosta):

W pewnych przypadkach spełnienie warunku reguły nie zależy od konkretnej wartości pola. Zależy nam jedynie na samym wystąpieniu danych w fakcie, zaś ich wartość nie ma dla nas znaczenia. Korzystamy wtedy z tzw. zmiennej anonimowej, która jest reprezentowana tylko przez znak ”?” bez nazwy i oznacza, że wartości przechowywane w polu nie są nam potrzebne do przetwarzania, choć muszą w nich wystąpić w trakcie dopasowywania.

Przykład

```
(deffacts persony
     (osoba Jan piwne czarne)
     (osoba Anna piwne brazowe)
     (osoba Tomasz niebieskie blond)
     (osoba Jacek zielone brazowe)
     (osoba Beata niebieskie zielone))

(defrule brazowe-wlosy
     (osoba ?imie   ?  brazowe) ; kolor oczu bez znaczenia
     =>
     (printout t ?imie ” ma brazowe wlosy.” crlf))
```

Zmienne wielopolowe (wielowartościowe)

Identyfikator zmiennej wielopolowej:

Zmienna wielopolowa reprezentuje we wzorcu dowolną liczbę pól faktu dopasowywanego do warunku reguły. Dowolna liczba pól oznacza tutaj sytuację, w której w fakcie spełniającym warunek występuje zero lub więcej wartości. Identyfikator zmiennej wielopolowej zaczyna się ciągiem dwóch znaków "$?", bezpośrednio po którym następuje ciąg znaków zgodny z definicją symbolu.

Przykład

```
(deffacts listy
     (lista a)
     (lista b d)
     (lista a b c)
     (lista a c c e)
     (lista c d b))

(defrule wyswietlaj-listy
     (lista $?elementy)
     =>
     (printout t "Zawartosc listy: " $?elementy crlf))

(defrule odszukaj-c
     (lista $?przed c $?po) ; c jest gdzies w srodku
     =>
     (printout t "Przed 'c' mamy: " $?przed crlf)
     (printout t "Po 'c' mamy: " $?po crlf))
```

Note:

Dopasowanie do zerowej liczby pól oznacza, że w fakcie może nie być żadnych danych poza rzecz jasna pierwszym polem symbolicznym, które jest obowiązkowe i którego wzorzec zmiennej wielopolowej nie obejmuje.

Zmienna anonimowa wielopolowa:

Wystąpienie zmiennej wielopolowej anonimowej – podobnie jak określonej zmiennej wielopolowej – oznacza dopasowanie do dowolnej liczby pól z tą różnicą, że wartości przechowywane w fakcie nie są nigdzie zapamiętywane. Oznaczeniem zmiennej anonimowej wielopolowej jest ciąg dwóch znaków ”$?”.

Przykład

```
(deffacts listy
     (lista a)
     (lista b d)
     (lista a b c)
     (lista a c c e)
     (lista c d b))

(defrule szukaj-list
     (lista $?)
     =>
     (printout t ”Znalazlem liste!” crlf))
```

Uwagi:

Choć anonimowe zmienne wielopolowe mają duże znaczenie w wielu przypadkach definiowania wzorców, należy unikać ich nadużywania, gdyż mogą powodować znaczne zwiększenie wymagań zasobowych oraz spadek wydajności systemu.

Ograniczniki wartości pól

Ograniczniki wartości pola są dodatkowym mechanizmem wzbogacającym sposoby definiowania wzorców w warunkach reguły. Pozwalają na nakładanie ograniczeń na wartość pola, dzięki czemu można dokładniej wskazać jakie wartości może ono przyjąć w trakcie dopasowania. Ograniczenia te mają charakter relacji logicznych i wyróżniamy trzy ich typy: negacja wartości pola, dysjunkcja wartości pól oraz ich koniunkcja. Mogą być stosowane tylko we wzorcach po lewej stronie reguły.

Negacja wartości pola:

Negacja wartości symbolizowana jest przez znak tyldy ”~”. Wpływa ona na obiekt, który zapisano we wzorcu bezpośrednio po niej, uniemożliwiając dopasowanie do niego. Inaczej można stwierdzić, że służy do definiowania dopełnienia wartości w warunkach, kiedy dopasowanie powinno zachodzić, gdy wartość pola w fakcie jest różna od tej, którą zanegowaliśmy we wzorcu.

Przykład

```
(deffacts persony
     (osoba Jan piwne czarne)
     (osoba Anna piwne brazowe)
     (osoba Tomasz niebieskie blond)
     (osoba Jacek zielone brazowe)
     (osoba Beata niebieskie zielone))

(defrule wlosy-inne-niz-brazowe
     (osoba ?imie   ?  ~brazowe)
     =>
     (printout t ?imie ” nie ma brazowych wlosow.” crlf))
```

Dysjunkcja wartości pól**:**

Dysjunkcja wartości jest symbolizowana przez znak pionowej kreski ”|”. Ogranicznik ten pozwala zapisać we wzorcu wszystkie alternatywne wartości jakie może przyjąć pole w trakcie dopasowania.

Przykład

```
(defrule wlosy-brazowe-lub-czarne
     (osoba ?imie  ?  brazowe|czarne)
     =>
     (printout t ?imie ” ma ciemne wlosy.” crlf))
```

Koniunkcja wartości pól**:**

Dysjunkcja wartości jest symbolizowana przez znak ”&” (ang. *ampersand*). Wykorzystywanie tego ogranicznika ma sens tylko w połączeniu razem z innymi ogranicznikami , gdyż w przeciwnym przypadku albo prowadzi do błędów, albo jest nadmiarowy i nie wnosi w gruncie żadnych nowych ograniczeń na wartość.

Przykład

```
(defrule brazowe-lub-czarne
     (osoba ?imie ? ?kolor&brazowe|czarne)
     =>
     (printout t ?imie  ” ma wlosy ” ?color ”.” crlf))
```

Łączenie ograniczników:

Ograniczniki wartości pól mogą być wykorzystywane razem w jednym wzorcu i tym samym tworzyć bardziej złożone ograniczenia nakładane na wartość pola w trakcie dopasowania. Konstruowanie takich złożonych ograniczeń może odbywać się zarówno dla stałych, jak i zmiennych występujących we wzorcach, dzięki czemu mamy do dyspozycji bardzo silny mechanizm definiowania warunków dopasowania reguł.

Przykład

```
(defrule skomplikowany
     (osoba ?im1 ?o1&niebieskie|zielone ?w&~czarne)
     (osoba ?im2&~?im1 ?o2&~?o1 ?w2&red|?w1)
=>
(printout t ?im1 ” ma oczy ” ?o1 ” i włosy ” ?w1 ”.” crlf)
(printout t ?im2 ” ma oczy ” ?o2 ” i włosy ” ?w2 ”.” crlf)
)
```

Uwagi:

Zmienne występujące w złożonych ogranicznikach są wiązane tylko wtedy, gdy zajmują w nim pierwszą pozycję. W przeciwnym przypadku muszą być już związane, gdyż inaczej doprowadzi to do błędu. Łączenie ogranicznikiem koniunkcji tylko stałych zawsze prowadzi do błędu, ponieważ nie ma to logicznego uzasadnienia (nie można wymagać aby jedno pole przyjmowało równocześnie kilka wartości). Łączenie ogranicznikiem dysjunkcji negacji różnych stałych nie ma żadnego znaczenia praktycznego, gdyż spełnienie tego ograniczenia jest zawsze możliwe.

Wyrażenia numeryczne

Funkcje i wyrażenie arytmetyczne:

- wyrażenia zapisywane są w notacji prefiksowej
- mamy wszystkie podstawowe operatory i funkcje: `+`, `-`, `*`, `/`, `**`, `sqrt`, `mod`, `div`
- nie ma żadnego wbudowanego mechanizmu priorytetów operatorów
- wyrażenia mogą być komendami interpretera poleceń, o ile operacja wykonywana jest tylko na stałych
- ewaluacja wyrażeń jest możliwa wewnątrz polecenia `assert`, o ile wyrażenie jest elementem tworzonego faktu

Przykład

```
CLIPS> (+ 2 2)
4
CLIPS> (+ 2 (* 3 4))
14
CLIPS> (* (+ 2 3) 4)
20
CLIPS> (** 2 7)
128
CLIPS> (assert (rezultat (+ 2 3 4)))
CLIPS> (facts)
f-0 (rezultat 9)
```

Uwagi:

Przetwarzanie numeryczne nie jest domeną systemu CLIPS. Choć system ten dysponuje pełną biblioteką wszystkich funkcji arytmetycznych, został stworzony z myślą o realizowaniu przede wszystkim obliczeń symbolicznych. Nie ma zatem praktycznego sensu implementowanie w nim algorytmów numerycznych, jeśli istotną ich cechą ma być wydajność.

Przykład

Sumowanie pól prostokątów.

```
(deffacts dane
     (prostokat 10 6)
     (prostokat 7 5)
     (prostokat 6 8)
     (prostokat 2 5)
     (suma 0))

(defrule pierwsza-proba ”Uwaga! Petla!”
     (prostokat ?wys ?szer)
     ?s <— (suma ?total)
     =>
     (retract ?s)
     (assert (suma =(+ ?total (* ?wys ?szer)))))
```

Poprawnie:

```
(defrule pola—prostokatow
     (prostokat ?wys ?szer)
     =>
     (assert (pole (* ?wys ?szer))))

(defrule suma-pol
     ?s <— (suma ?total)
     ?a <- (pole ?area)
     =>
     (retract ?s ?a)
     (printout t ”Dodajemy ” ?area ” do ” ?total crlf)
     (printout t ”Nowa suma: ” (+ ?total ?area) crlf)
     (assert (suma (+ ?total ?area))))
```

Funkcja jawnego wiązania:

Zmienne w systemie CLIPS zazwyczaj wiązane są w warunkach reguły w trakcie dopasowywania wzorców. Czasami jednak zachodzi potrzeba wprowadzenia zmiennej i tym samym jej wiązania (bo nie ma mechanizmu jedynie deklarowania zmiennych) w akcji reguły. Zmienna taka najczęściej pełni rolę pomocniczą i służy do przechowywania rezultatów pośrednich obliczeń.

Składnia

```
(bind <zmienna> <wartość>)
```

Przykład

```
(defrule pola—prostokatow
     (prostokat ?wys ?szer)
     =>
     (assert (pole (* ?wys ?szer))))

(defrule suma-pol
     ?s <— (suma ?total)
     ?a <- (pole ?area)
     =>
     (retract ?s ?a)
     (printout t ”Dodajemy ” ?area ” do ” ?total crlf)
     (bind ?nowe-total (+ ?total ?area))
     (printout t ”Nowa suma: ” ?nowe-total crlf)
     (assert (suma ?nowe-total)))
```

Uwagi:

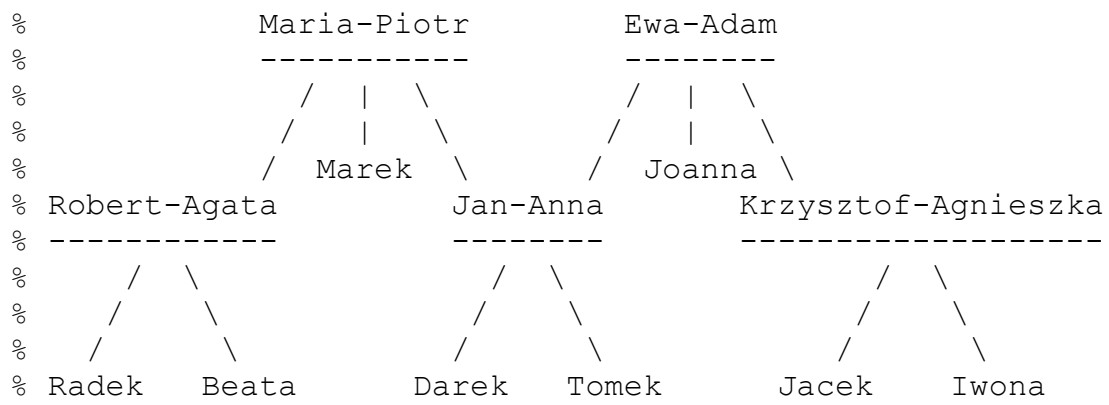
Akcją po prawej stronie reguły może być praktycznie każda komenda, której można użyć w trybie interpretacji poleceń w środowisku CLIPS. Jedynym wyjątkiem są tutaj tzw. konstruktory systemu CLIPS, których słowa kluczowe zaczynają się od przedrostka ”def”. Należą do nich między innymi takie definicje jak `defrule` czy `deffacts` oraz kilka innych.

Ćwiczenie:

Dla pewnego przykładowego drzewa genealogicznego należy zdefiniować kilka najpopularniejszych relacji pokrewieństwa w postaci reguł. Zakładamy, że dane zapisane w drzewie są reprezentowane przez trzy rodzaje faktów: `(kobieta <imię>)`, `(mezczyzna <imię>)` oraz `(rodzic <imię_rodzica> <imię_dziecka>)`. Opisy deklaratywne definiowanych relacji w postaci zdań języka naturalnego zostały zamieszczone poniżej. Każda ze zdefiniowanych reguł powinna generować fakt binarny (dwa pola, tak jak w fakcie `rodzic`), a jej warunki mogą odwoływać się do innych – wcześniej zdefiniowanych – faktów.

Relacja `matka`: „Dla każdego X i Y, jeżeli X jest kobietą oraz X jest rodzicem Y, to X jest matką Y”
Relacja `ojciec`: „Dla każdego X i Y, jeżeli X jest mężczyzną oraz X jest rodzicem Y, to X jest ojcem Y”
Relacja `siostra`: „Dla każdego X i Y, jeżeli X jest kobietą oraz X ma tego samego rodzica Z co Y, to X jest siostrą Y”
Relacja `brat`: „Dla każdego X i Y, jeżeli X jest mężczyzną oraz X ma tego samego rodzica Z co Y, to X jest bratem Y”
Relacja `babcia`: „Dla wszystkich X i Y, jeżeli X jest rodzicem Z, Z jest rodzicem Y i X jest kobietą, to X jest babcią Y”
Relacja `dziadek`: : „Dla wszystkich X i Y, jeżeli X jest rodzicem Z, Z jest rodzicem Y i X jest mężczyzną, to X jest dziadkiem Y”
Relacja `wuj`: „Dla wszystkich X i Y, jeżeli X jest bratem Z i Z jest matką Y, to X jest wujem Y”
Relacja `stryj`: „Dla wszystkich X i Y, jeżeli X jest bratem Z i Z jest ojcem Y, to X jest wujem Y”

Przykładowe drzewo genealogiczne:



Ćwiczenie:

Mając zapisaną informację o kolorze oczu, kolorze włosów oraz narodowości osób w postaci faktów o formacie:

```
(person <imie> <oczy> <wlosy> <narodowosc>)
```

należy zdefiniować następujące reguły dla:

- osób o niebieskich lub zielonych oczach z brązowymi włosami, pochodzących z Francji;
- osób o oczach innych niż niebieskie i włosach różnych od czarnych, których kolor oczu i włosów nie mogą być takie same;
- pary osób, z których pierwsza ma piwne lub niebieskie oczy, włosy różne od blond i będzie pochodzić z Niemiec, zaś druga ma zielone oczy, ten sam kolor włosów co osoba pierwsza; kolor oczu drugiej osoby może też być czarny, jeśli oczy osoby pierwszej też są czarne.

Operacje logiczne

Operator dysjunkcji warunków:

W systemie CLIPS domniemanym spójnikiem warunków reguł jest operator koniunkcji wzorców. Wszystkie dotychczasowe przykłady wykorzystywały niejawnie ten operator. Do dyspozycji mamy jednak wszystkie podstawowe operatory logiczne, które oznaczamy odpowiednio za pomocą słów kluczowych: `and`, `or` oraz `not`. Choć w systemie regułowym – jakim jest przecież CLIPS – nie ma konieczności używania operatora dysjunkcji, może on jednak skrócić zapis zbioru reguł, które mają alternatywne warunki dla tych samych akcji.

Przykład

```
(defrule odciac-zasilanie-1
     (alarm powodz)
     =>
     (printout t ”Odetnij zasilanie!” crlf))

(defrule odciac-zasilanie—2
     (pozar-grupy A)
     =>
     (printout t ”Odetnij zasilanie!” crlf))
```

Wykorzytując operator dysjunkcji może dwie powyższe reguły zapisać jako jedną regułę:

```
(defrule odeciac-zasilanie
     (or  (alarm powodz)
          (pozar-grupy A))
     =>
     (printout t ” Odetnij zasilanie!” crlf))
```

Operator dysjunkcji warunków (ciąg dalszy):

Uwagi:

System CLIPS interpretuje regułę z operatorem dysjunkcji dokładnie tak samo jak odpowiedni zbiór alternatywnych reguł bez operatora `or`. Może to powodować wielokrotne odpalenie tej samej reguły (z operatorem `or`), co można kontrolować poprzez wprowadzenie dodatkowego warunku po lewej stronie reguły.

```
(defrule odciac-zasilanie
     ?p<– (stan-zasilania wlaczone) ; dodatkowy warunek
     (or  ?r <– (alarm powodz)
          ?r <– (pozar-grupy A))
     =>
     (retract ?p ?r)
     (assert (stan-zasilania wylaczone))
     (printout t ”Odetnij zasilanie!” crlf))
```

Operator koniunkcji warunków:

W przeciwieństwie do operatora `or`, operator `and` wymaga, aby dla wszystkich wzorców zapisanych w części warunkowej reguły znaleziono w pamięci roboczej fakty, które zostaną do nich dopasowane. Skoro operator koniunkcji warunków jest domyślnym operatorem, to faktyczna potrzeba jego wykorzystania ma miejsce tylko wtedy, gdy chcemy go połączyć z innymi operatorami, definiując bardziej złożoną postać części warunkowej reguły.

Przykład

```
(defrule odciac-zasilanie
     ?p <- (stan-zasilania wlaczone)
     (or  (alarm powodz)
          (and (alarm pozar)
               (pozar-grupy A))
          (system-spryskiwaczy aktywny))
     =>
     (retract ?p)
     (assert (stan-zasilania wylaczone))
     (printout t "Odetnij zasilanie!" crlf))
```

Operator negacji warunków:

Operator logiczny `not`, traktowany jest jako zapis sytuacji, w której chcemy wykryć brak w pamięci roboczej faktu, który można byłoby dopasować do zanegowanego wzorca. Tak interpretacja negacji logicznej wynika z złożenia o „zamkniętości świata”, zgodnie z którym, jeśli nie znajdziemy informacji, potwierdzających pewien fakt, to przyjmujemy, iż jego negacja jest spełniona.

Przykład

```
(deffacts dane
     (para 3 4)
     (para 4 5)
     (trojka 4 4 6))

(defrule demonstracja-not-1
     (para ?x ?y)
     (para ?y ?z)
     (not (trojka ?x ?y ?z))
     =>
     (printout t ”Brak trojki: ” ?x ” ” ?y ” ” ?z crlf))

(defrule demonstracja—not—2
     (not (trojka ?x ?y ?z))
     (para ?x ?y)
     (para ?y ?z)
     =>
     (printout t ”Brak jakiejkolwiek trojki!” crlf))
```

Uwagi:

Zmienne, które występują po raz pierwszy w negacji wzorca nie mogą być wykorzystywane w innych warunkach ani w akcjach reguły. Natomiast zmienne związane we wcześniejszych warunkach (bez negacji) mogą być swobodnie używane, również w dalszych warunkach z negacjami czy w akcjach reguły.

Wyrażenia boolowskie

W systemie CLIPS nie ma jawnego typu boolowskiego, ale istnieje możliwość zapisywanie wyrażeń boolowskich z użyciem dokładnie tym samych słów kluczowych, które stosujemy definiując spójniki logiczne warunków reguł tj. `and`, `or` oraz `not`. Należy jednak pamiętać, że kontekst w jakim pojawiają się te operatory jest inny.

Funkcja predykatowa test:

Podstawowym procesem realizowanym w trakcie wnioskowania w systemie CLIPS jest dopasowywanie wzorców. Właśnie dlatego pojedyncze warunki reguły oznaczają poszukiwanie danych, które go spełniają, a nie ewaluację wyrażeń boolowskich, które znamy z innych języków przetwarzania symbolicznego. Wyrażenie boolowskie może być jednak elementem części warunkowej reguły, o ile użyjemy predefiniowanej funkcji systemowej `test`, która służy do ewaluacji tych wyrażeń. Jeśli wyrażenie boolowskie, będące argumentem tej funkcji, jest prawdziwe, to warunek reguły, który zawiera wywołanie tej funkcji jest spełniony.

Składnia

```
(test <wyrażenie-boolowskie>)
```

Przykład

```
(deffacts dane
     (liczba 1)
     (liczba 5)
     (liczba 7)
     (zakres 1 6))
(defrule liczba-poza-zakresem
     (liczba ?nr)
     (zakres ?lewy ?prawy)
     (test (or (not (integerp ?nr))
               (< ?nr ?lewy)
               (> ?nr ?prawy)))
     =>
     (printout t "Liczba " ?nr " nie miesci sie w
      zakresie [" ?lewy "," ?prawy "]." crlf))
```

Uwagi:
Jeżeli wyrażenie boolowskie jest złożone, to używamy tylko jednego wywołania funkcji `test`. Obejmuje ono - jako swój argument - całe złożone wyrażenie.

Predykaty boolowskie:

Poniżej zamieszczono wszystkie operatory i funkcje boolowskie, występujące w systemie CLIPS z podziałem na różne kategorie.

Logiczne funkcje boolowskie:

```
(and <<predykat-boolowski>>)
(or <<predykat-boolowski>>)
(not <predykat-boolowski>)
```

Operatory porównania:

```
(eq  <wyrażenie> <wyrażenie>)
(neq <wyrażenie> <wyrażenie>)
(=  <wyr-numeryczne> <wyr-numeryczne>)
(!= <wyr-numeryczne> <wyr-numeryczne>)
(>  <wyr-numeryczne> <wyr-numeryczne>)
(>= <wyr-numeryczne> <wyr-numeryczne>)
(<  <wyr-numeryczne> <wyr-numeryczne>)
(<= <wyr-numeryczne> <wyr-numeryczne>)
```

Predykaty typu wartości:

```
(numberp <wyrażenie>)
(stringp <wyrażenie>)
(wordp <wyrażenie>)
(integerp <wyrażenie>)
(evenp <wyrażenie>)
(oddp <wyrażenie>)
```

Operacje we/wy na plikach

W systemie CLIPS może operować nie tylko na danych przechowywanych w pamięci roboczej, ale równie przetwarzać informacje zapisane plikach tekstowych. Tradycyjnie zbiory takie nie oferują dostępu swobodnego, lecz wymagają liniowego przetwarzania.

Funkcja odczytu:

Operacją, która umożliwia odczyt z pliku lub ze standardowego urządzenia wejściowego jest funkcja `read`. W systemie CLIPS ma ona charakter wyrażenia i pozbawiona jest zarówno wskazania typu odczytywanej wartości, jak i miejsca jego zapamiętywania. W związku z tym przechowanie tego co zostało odczytane wymaga podobnych zabiegów jak zapamiętywanie wyników ewaluacji wyrażeń arytmetycznych – tworzenia faktów.

Składnia

```
(read [<nazwa-logiczna>])
```

Jeżeli pominiemy argument `<nazwa-logiczna>`, to odczyt następuje ze standardowego urządzenia wejściowego tj. terminala.

Przykład

```
(defrule wczytaj-liczbe
     (not (liczba ?))
     =>
     (printout t ”Podaj liczbe: ”)
     (assert (liczba (read))))
(defrule sprawdz-typ
     ?i <- (liczba ?nr)
     (test (not (numberp ?nr)))
     =>
     (retract ?i)
     (printout t crlf ”To nie jest liczba!” crlf))
(defrule sprawdz
     ?i <- (liczba ?nr)
     (test (numberp ?nr))
     (test (> ?nr 100))
     =>
     (retract ?i)
     (printout t crlf ”Za duza!” crlf))
```

Operacja otwarcia pliku:

Podobnie jak ma to miejsce w innych językach programowania, w systemie CLIPS zanim wykonamy na pliku operację odczytu lub zapisu, musimy go najpierw otworzyć tak, aby uzyskać do niego dostęp.

Składnia

```
(open <ścieżka> <nazwa-logiczna> [<tryb>])
```

gdzie `<tryb>` może przyjąć jedną w wartości reprezentowaną przez łańcuch:
“r” – dostęp w trybie wyłącznie odczytu (domyślny),
“w” – dostęp w trybie wyłącznie zapisu,
“r+” – dostęp w trybie odczytu i zapisu,
“a” – dostęp w trybie dopisywania,
zaś parametr `<nazwa-logiczna>` jest globalną stałą symboliczną, która zostanie skojarzona z plikiem na dysku wskazanym przez parametr `<ścieżka>`.

Operacja zamknięcia pliku:

W chwili kiedy dalszy dostęp do pliku nie jest już potrzebny powinniśmy go zamknąć, używając do operacji `close`.

Składnia

```
(close [<nazwa-logiczna>])
```

Możliwe jest zamknięcie jednoczesne wszystkich plików, jeżeli zamiast nazwy logicznej podamy argument symboliczny `all`.

Przykład

```
(defrule otworz-pliki
     =>
     (close all) ; aby uniknac bledu otwierania otwartych
     (open ”dane.dat” we ”r”)
     (open ”wyniki.txt” wy ”w”)
     (assert (faza odczyt)))
```

```
(defrule wczytaj-dane
     ?f <- (faza odczyt)
     =>
     (retract ?f)
     (assert (cisnienie (read we))))

(defrule koniec-pliku
     (cisnienie ?wartosc)
     (test (eq ?wartosc EOF))
     =>
     (assert (faza zamykanie)))

(defrule przetwarzanie-1
     (cisnienie ?c)
     (test (numberp ?c))
     (test (> ?c 100))
     =>
     (printout wy "Cisnienie: " ?c " ponad 100!" crlf)
     (assert (faza odczyt)))

(defrule przetwarzanie-2
     (cisnienie ?c)
     (test (numberp ?c))
     (test (<= ?c 100))
     =>
     (printout wy "Cisnienie: " ?c " ok!" crlf)
     (assert (faza odczyt)))

(defrule zamknij-pliki
     ?f <— (faza zamykanie)
     =>
     (retract ?f)
     (close we)
     (close wy))
```

Uwagi:

Predefiniowania stała symboliczna `EOF` służy do wykrywania końca pliku. Liniowe przetwarzanie plików tekstowych wymusza odpowiedni porządek w odpalaniu reguł. Uzyskujemy to za pomocą dwóch faktów sterujących `(faza odczyt)` oraz `(faza zamykanie)`. Pierwszy fakt zapewnia kontynuację odczytu danych z pliku, drugi zaś pozwala zakończyć przetwarzanie zamknięciem pików.

Projekt systemu eksperckiego w systemie CLIPS

Część 3

• Cel

• Założenia

• Wymagania opcjonalne

• Uwagi

Cel

Celem tej części laboratoriów jest zaprojektowanie i implementacja regułowego systemu eksperckiego z wykorzystaniem narzędzia przeznaczonego do tworzenia tego rodzaju aplikacji - systemu szkieletowego CLIPS. Projektowany system ma być niezależną aplikacją, działającą poza środowiskiem interpretera systemu CLIPS, wyposażoną we własny okienkowy interfejs graficzny w Javie. W tym celu należy wykorzystać bibliotekę CLIPSJNI w wersji 0.2, którą można znaleźć tutaj. Tematyka systemu nie jest zupełnie dowolna - co roku aktualizowana jest lista dziedzin dopuszczonych do projektowania (lub ewentualnie zakazanych).

Założenia

Realizowany projekt musi spełniać wymaganie stawiane tego typu projektom. Wszystkie kluczowe zostały omówione poniżej.

- Modularny charakter projektu z zachowaniem separacji między sterowaniem a wiedzą i danymi przedmiotowymi

  Kryterium projektowe bardzo istotnej dla systemów z bazą wiedzy. Oznacza między innymi, że dane przedmiotowe nie mogą znaleźć się w warstwie interfejsu aplikacji i obok – oczywistego w tym projekcie – założenia, iż baza wiedzy jest oddzielona także od sterowania, gwarantuje wyraźny podział systemu eksperckiego na trzy główne komponenty: interfejs, mechanizm wnioskowania, bazę danych i wiedzy. Oprócz tego wśród elementów systemu można wyróżnić dodatkowe pliki zasobowe takie jak obrazy graficzne, zbiory multimedialne czy dane niezbędne do uruchamiania aplikacji w różnych wersjach językowych.

- Rozmiar bazy wiedzy

  Minimalny rozmiar bazy wiedzy powinien oscylować w granicach 100 reguł produkcji. Brane są pod uwagę tylko reguły, uczestniczące w procesie wnioskowania i dialogu z użytkownikiem. Te ostatnie tylko, jeżeli są kontekstowe. Inne reguły służące np. do wykrywania i korekty błędów nie są uwzględniane.

- Ograniczone wykorzystywanie faktów sterujących

  Stosowanie faktów sterujących jest uzasadnione tylko w pewnych sytuacjach. Przypadki nadużywania faktów sterujących (w istocie pozaprzedmiotowych) to przede wszystkim systemy, w których część „faktów” tworzona jest sztucznie jedynie w celu narzucenia bardzo ścisłej kontroli procesowi wnioskowania. Fakty takie w rzeczywistości nie zawierają żadnych danych z dziedziny, w której pracuje system. Niekiedy stosowanie takich faktów jest dopuszczalne, ale tylko tam gdzie twórca systemu zamierza wyróżnić pewne fazy/etapy w procesie wnioskowania (np. zbieranie kilku koniecznych na początku faktów o użytkowniku), a nie tam gdzie chce poddać wnioskowanie nadmiernej kontroli, ograniczając je tym sposobem najczęściej do wyłącznie jednej ścieżki wnioskowania.

- Obecność kontekstu w pytaniach/dialogu oraz związanych z nimi regułach

  Przebieg dialogu z użytkownikiem powinien odzwierciedlać naturę procesu wnioskowania, który opiera się na poszukiwaniu rozwiązania, a nie jego ewaluacji/wyliczaniu drogą znanego z góry algorytmu. W dialogu powinny być zatem zawsze uwzględniane dotychczas zgromadzone informacje (choć niekoniecznie wszystkie), gdyż właśnie w ten sposób w systemie regułowym realizujemy mechanizm rozwiązywania problemu, polegający na dochodzeniu do wniosków końcowych różnymi ścieżkami a nie sekwencyjnym wykonywaniu kroków pewnego ustalonego algorytmu.

- Spójność i konsekwencja w reprezentacji danych przedmiotowych

  Fakty zawierające dane przedmiotowe nie mogą w trakcie przetwarzania, przechowywania i komunikacji na styku Java-CLIPS zmieniać swej postaci czy też formy reprezentacji w sposób uniemożliwiających ich jednoznaczną identyfikację i właściwą interpretację, taką jak wynika z treści pytań stawianych użytkownikowi i wybieranych przez niego odpowiedzi. Założenie to wymaga zatem od twórcy systemu czytelnych nazw obiektów, pojawiających się w kodzie

bazy wiedzy (zwłaszcza wartości opcji wyboru prezentowanych użytkownikowi) oraz zachowywania ich typu (oryginalne w danej dziedzinie wartości symboliczne pozostają w bazie wiedzy symbolicznymi, numeryczne – numerycznymi itd.) Jeżeli wykorzystywane są dodatkowe pliki z zasobami, zawierające pełną treść pytań i odpowiedzi prezentowanych użytkownikowi, to odpowiadające im wartości w bazie wiedzy mogą mieć skróconą formę. Powinna być ona jednak symboliczna i czytelna. Z takich plików zasobowych korzystać muszą wtedy zarówno warstwa interfejsu realizowana w Javie, jak i warstwa wnioskowania realizowana w CLIPSie.

- Rozwiązania końcowe to nie statyczne dane

  Konkluzje, które otrzymujemy w momencie zakończenie procesu wnioskowania nie mogą być reprezentowane w bazie wiedzy w postaci faktów. Powinny mieć postać reguł produkcji lub też powinny być dopiero konstruowane w trakcie wnioskowania (jak np. w systemach planowania tras/działań). Do wniosków należy bowiem dojść na drodze poszukiwania rozwiązania w trakcie kontekstowego dialogu z użytkownikiem, a nie poprzez selekcję i projekcję pewnych grup faktów dobieranych na podstawie zbioru wartości atrybutów podanych przez użytkownika. Taki charakter rozwiązania oznaczałby, iż korzystamy z mechanizmów właściwych systemom baz danych, a to nie może być przedmiotem oceny aplikacji SI jaką jest system ekspercki.

- Właściwa i naturalna forma wyboru odpowiedzi przez użytkownika na poziomie interfejsu

  Forma odpowiedzi udzielanych przez użytkownika w trakcie dialogu powinna wynikać z natury przedmiotu pytania a nie z ograniczeń narzuconych sztucznie przez założenia przyjęte (często wtedy błędnie) przy projektowaniu aplikacji. W interfejsie powinny być zatem obecne zarówno opcje jednokrotnego, jak i wielokrotnego wyboru, o ile wymaga tego charakter pytania (a właściwie charakter udzielanych na nie odpowiedzi). Należy unikać rozwijanych list, gdyż ich zastosowanie jest uzasadnione tylko w przypadku bardzo dużej liczby (rzędu kilkudziesięciu) możliwych wartości.

Wymagania opcjonalne

- Możliwość ignorowania konkretnych wartości odpowiedzi przez użytkownika

  Dużym ułatwieniem – zwłaszcza w bardziej specjalistycznych systemach eksperckich – jest dopuszczenie sytuacji w której użytkownik nie dokonuje wyboru konkretnej wartości odpowiedzi lecz korzysta z opcji „nieistotne/nie wiem/dowolny”. Pozwala to zwiększyc szanse na znalezienie końcowego rozwiązania przy niewielkim dodatkowym nakładzie prac podczas projektowania systemu – wystarczy tylko w warunkach akcji dodać alternatywę wartości konkretnej z wartością ignorującą. Rozwiązania tego należy jednak używać z umiarem, tak aby nie doprowadzać do stany, w którym wszystkie rozwiązania końcowe zdefiniowane w bazie wiedzy są na końcu wnioskowania poprawne.

- Wyprowadzanie na standardowe urządzenie wyjściowe (konsola) informacji o stanie systemu

  Bardzo pomocnym rozwiązaniem, które ułatwia śledzenie pracy systemu jest wyprowadzanie na konsolę danych, opisujących aktualny stan procesu wnioskowania. Interfejs okienkowy służy raczej do realizacji pewnej formy dialogu między użytkownikiem a systemem. Konsola natomiast może być drugim kanałem, którym docierają informację do użytkownika, mające bardziej techniczny charakter. W najprostszym przypadku mogą to być tylko fakty aktualnie znajdujące się w pamięci roboczej, ale można także sygnalizować dokładniej wszelkie zmiany, które mają miejsce w systemie w trakcie przetwarzania tj. dodanie faktu, usunięcie faktu, aktywacje reguł na agendzie, odpalenia reguł itp.

Uwagi

Gotowy projekt dostarczamy w postaci źródłowej wraz z oddzielnymi skryptami kompilacji (plik o nazwie kompiluj.cmd) i uruchomienia (plik o nazwie

uruchom.cmd) stworzonymi przy założeniu, iż wszystkie niezbędne biblioteki (CLIPSJNI.jar CLIPSJNI.dll) oraz kluczowe pliki (baza wiedzy czy kod źródłowy aplikacji w Javie) znajdują się w bieżącym katalogu. Należy dołączyć także pliki z zasobami, jeśli takie są wykorzystywane przez aplikację (np. zbiory z obrazkami graficznymi). Wykluczone jest dostarczanie całych folderów projektów, pochodzących od środowisk programistycznych Javy (IDEs) takich jak Netbeans, Eclipse czy JBuilder. Postać wysłanego projektu musi być taka, aby możliwe było jego skompilowanie i uruchomienie na platformie, posiadającej jedynie odpowiednią wersję (najlepiej najnowszą) Runtime'u Javy (JRE) plus ewentualnie dodatkowe biblioteki (inne niż CLIPSJNI), nie wchodzące w skład bibliotek standardowych JRE (o konieczności posiadania których należy rzecz jasna poinformować).