Na CD: Lisp, Prolog, Clips, pakiety i biblioteki dla sieci neuronowych, algorytmów genetycznych, systemów eksperckich, sztucznego życia, konwersacji naturalnej

narzędzia • programy • sieci

Software 2.0

nr 2/2001 (74)
cena 26,75
(zawiera 7% VAT)

Sztuczna Inteligencja

W numerze:

Sieci neuronowe
Algorytmy genetyczne
Wnioskowanie rozmyte
Konwersacja naturalna
Systemy eksperckie
Prolog
Lisp
Security Updates
Olimpiada Informatyczna

INDEX 35497X ISSN 1508-1659

Systemy eksperckie

Marcin Włoszczyk

Na CD:

Pakiet CLIPS został zamieszczony na płytce CD dołączonej do pisma. Znajdziecie tam również proste przykłady.

Systemy eksperckie wykorzystują wiedzę i procedury wnioskowania do analizy problemów na tyle trudnych, że do ich rozwiązania niezbędna jest fachowa wiedza z danej dziedziny. Zaznaczyć należy fakt, że wskazana dziedzina wiedzy jest mocno ograniczona i w znacznej większości przypadków ogranicza się do wąskiego zakresu specjalizacji. System ekspercki nie może zatem rozwiązywać problemów z wielu dziedzin. Dlaczego? Ponieważ jest to, po pierwsze, niezgodne z założeniem — ekspertem jest się raczej w tylko jednej dziedzinie. Po drugie, byłoby to dość trudne w implementacji — system taki musiałby posiadać dość rozległą wiedzę, co wiąże się z rozbudowanymi strukturami danych i — co najważniejsze — rozbudowanymi regułami wnioskowania. Skuteczność systemu określa baza wiedzy oraz zbiór reguł produkcji (wnioskowania) — czyli sam mechanizm wnioskujący. Często bazy wiedzy tworzy nie jeden, a wielu ekspertów z danej dziedziny — chodzi o to, by „udostępnić" jak najwięcej informacji i zasad.

Cykl życia systemu eksperckiego wygląda następująco:

- planowanie — czyli założenia co taki system ma robić i gdzie będzie potrzebny
- pozyskiwanie wiedzy — definiowanie faktów i reguł
- kodowanie — implementacja jako program lub bazy reguł i faktów do systemu szkieletowego
- ocena systemu — czy założenia zostały spełnione, czy program jest optymalny, itp.

Systemy eksperckie najlepiej stosować w dziedzinach o słabo sformalizowanej wiedzy, gdzie procedury typowo algorytmiczne nie wchodzą w rachubę (np. zagadnienia typowo matematyczne lub oparte na matematyce). Można podać wiele przykładów wykorzystywania systemów eksperckich, np. w medycynie — system *MYCIN* służący do diagnozowania i terapii infekcji bakteryjnych krwi, *DENDRAL* — określenie struktury cząstek związków organicznych na podstawie wzorów chemicznych i informacji spektrograficznej. *PROSPECTOR* — geologia, ocena złóż surowców mineralnych i *XCON* — konfiguracja komputerów VAX.

Wyżej wymienione systemy są „całościowe", czyli poświęcone tylko jednej dziedzinie, do której zostały zaprojektowane. Istnieją również systemy „uniwersalne" czyli takie, do których programista sam może napisać bazę wiedzy i reguły wnioskowania. Systemy takie określane są jako szkieletowe i do nich należy m.in. CLIPS (*C Language Integrated Production System*).

Zanim przejdę do przedstawienia zasad tworzenia systemów eksperckich, postaram się zilustrować zasadę działania systemu eksperckiego.

Cykl wnioskowania można zawrzeć w pętli:

- **WHILE NOT** *rozstrzyganie konfliktu* -- wybierz regułę o największym priorytecie
- **akcja:** wykonuj kolejno akcje opisane po prawej stronie reguł
- **dopasowanie:** uaktualnij agendę uzupełniając ją o reguły których lewa strona jest spełniona
- **sprawdzenie warunku stopu:** JEŻELI warunek stopu jest spełniony to **STOP**

Struktura systemu eksperckiego przedstawiona została na Rysunku 1:

- mechanizm wnioskujący [agenda] — tu znajdują się reguły, których lewa strona (akcje) są spełnione
- baza wiedzy i pamięć robocza — z bazy wiedzy, która może być np. plikiem tekstowym, należy przenieść zapisaną wiedzę do pamięci roboczej, z której odczytywane są odpowiednie fragmenty informacji (wiedzy) jak i reguł
- mechanizm wyjaśniający — (niestety rzadko spotykany w szkieletowych systemach eksperckich) — jego działanie polega na wyjaśnieniu użytkownikowi jaką drogą system doszedł do takiego, a nie innego wniosku
- mechanizm pozyskiwania wiedzy
- interfejs użytkownika

O autorze:

Autor jest studentem Politechniki Poznańskiej, kierunek informatyka. Interesuje się sztuczną inteligencją (w tym systemami eksperckimi) zajmuje się głównie w kategoriach sterowania.
Kontakt z autorem e-mail: *marcwlo44@poczta.wp.pl*

Czym jest CLIPS?

CLIPS (*C Language Integrated Production System*) jest szkieletowym systemem eksperckim, czyli takim, gdzie programista sam definiuje bazy wiedzy i reguły wnioskowania, najczęściej w języku wewnętrznym danego systemu. Dla CLIPS jest to język nieco zbliżony do C.

Sterowanie wnioskowaniem w CLIPS oparte jest na wnioskowaniu wstecz (bazującym na algorytmie *Rete*), które polega na porównywaniu prawych stron reguł (RHS) z bazy wiedzy z faktami znajdującymi się w pamięci roboczej. Odpalenie reguły polega na wprowadzeniu do pamięci roboczej jej lewej strony (LHS) — brzmi to może trochę zawile, ale okaże się, że jest to dość proste do zrozumienia, a pisząc dowolny system, w zasadzie nie będziemy na to zwracać uwagi. Dla bardziej dociekliwych najlepszym wyjaśnieniem będzie przykład — definicja reguły:

```
(defrule <rule-name> [(<comment>)] [<decla-
            ration>] ; Zasady reguły
  <conditional-element>* ; Lewa strona
            reguł (LHS)
=>
  <action>*) ; Prawa strona reguł (RHS)
```

Najprościej rzecz ujmując: wcześniej wykonane reguły pozostawiają po sobie fakty, które są wynikiem wnioskowania. Mogą to być też po prostu zdefiniowane fakty lub grupy faktów — które są niczym innym jak prawą stroną faktów (RHS). Kolejnym etapem wnioskowania jest odpalenie następnej reguły, której lewa strona (LHS) jest spełniona, czyli reguły posiadającej w definicji deklaracje faktów, które to muszą być spełnione by odpaliła (w systemach eksperckich mówimy, że reguły odpalają, a nie uruchamiają się) — jest to jej warunek startu. Reguła, która startuje jako pierwsza może wcale nie mieć zdefiniowanej lewej strony — jej warunkiem startowym jest pusty fakt.

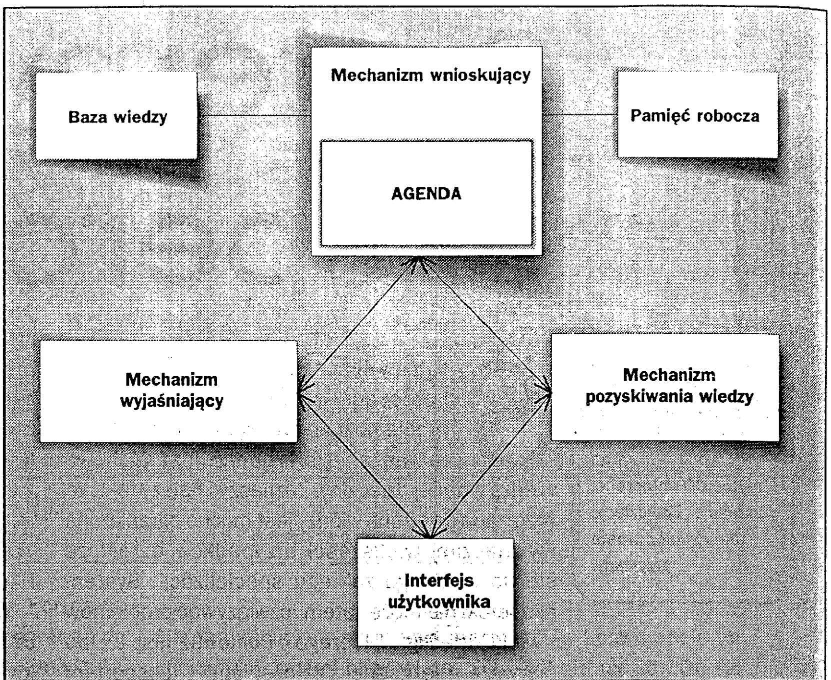
Zaczynamy pracę

Okno CLIPS jest typowym oknem edycyjnym, na początek interesować nas będzie menu: *File -> Load Constructs* oraz *Execution -> Reset / Run / Clear CLIPS / Step*. Pierwsze służyć będzie do ładowania „programików”, drugie do ich uruchamiania.

Kolejnymi pożytecznymi oknami CLIPSa będą okna podglądu faktów, agendy i instancji (menu: *Window*). Przydadzą się nam, by poznać zasadę działania i wnioskowania, a także do sprawdzania poprawności pracy pisanego programu (systemu).

Aby załadować plik programu należy najpierw z menu *Execution* wybrać opcję Clear CLIPS, następnie *File-> Load Constructs* i załadować plik **.clp* (można też załadować wersję binarną, ale nie będzie można jej edytować — poprawiać). Aby uruchomić program należy wejść do menu *Execution* wybrać opcję *Reset*, a następnie *Run*.

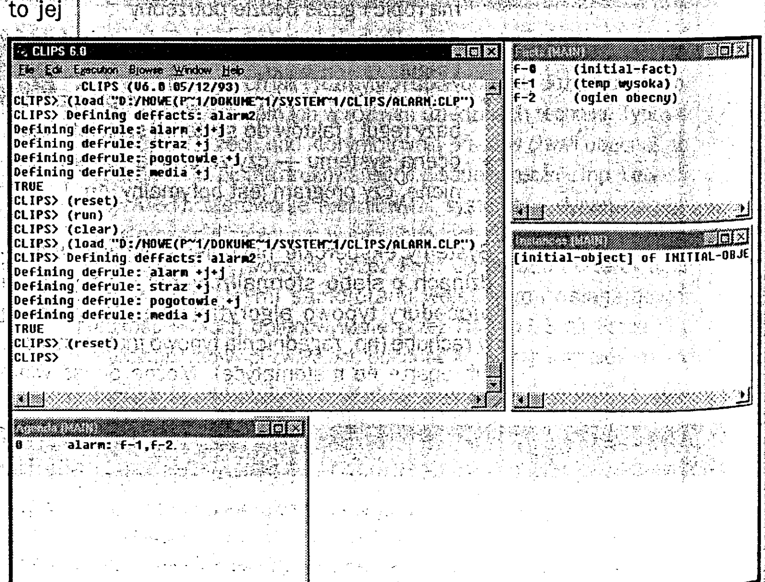
Operacje *Clear* CLIPS, *Reset* i *Run* można podać również w oknie, wpisując je koniecznie (!) w nawiasach okrągłych (podobnie jak na Rysunku 2). W CLIPSie nie należy używać polskich znaków dialektycznych — program nie jest na to „uodporniony” — może zacząć zachowywać się dziwnie.



Rysunek 1.
Struktura systemu eksperckiego.

Definiowanie zasad i faktów

Rozważania będziemy prowadzić na przykładzie drzewa genealogicznego, czyli sprawdzimy kto jest kim w rodzinie.



Rysunek 2.
Okna programu CLIPS podczas pracy.

Fakty:
Definiowanie faktów jest dość proste i wygląda następująco:

```
(deffacts <deffacts-name> [<comment>]
   <RHS-pattern>*)
```

a na przykładzie:

```
(deffacts mezczyzni
   (mezczyzna Piotr)
   (mezczyzna Adam)
   (mezczyzna Jan)
   (mezczyzna Marek)
   (mezczyzna Krzysztof)
   (mezczyzna Jacek))
```

Jak widać nazwą faktu jest tutaj `mezczyzni`, a tajemniczy `<RHS-pattern>` to po prostu prawa strona zasad, czyli pewne fakty, np. że Piotr jest mężczyzną. Zdefiniować można również fakty, które mają znacznie więcej atrybutów:

Tworzenie systemu eksperckiego najlepiej rozpocząć od narysowania grafu

```
(deffacts zywiciele
   (rodzic Maria Jan)
   (rodzic Piotr Jan)
   (rodzic Krzysztof Jacek)
   (rodzic Krzysztof Iwona))
```

Tu opisane są fakty, kto jest rodzicem — para: rodzic, potomek. Tak definiować możemy wszystko, co w danym systemie będzie potrzebne jako wiedza — w tym wypadku wystarczą nam: mężczyźni, kobiety i rodzice.

Zasady:
Definiowanie zasad wygląda podobnie do faktów, ale tu mamy większe pole do popisu. Zatem przyjrzyjmy się składni:

```
(defrule <rule-name> [<comment>] [<dec
          laration>] ; Zasady reguły
   <conditional-element>* ; Lewa strona
          reguł (LHS)
=>
   <action>*) ; Prawa strona reguł (RHS)
```

Na przykładzie zapis zasady wygląda nastepująco:

```
(defrule mamusia
   (kobieta ?x)
   (rodzic ?x ?y)
=> (assert (matka ?x ?y))
   (printout t ?x " jest matka " ?y crlf))
```

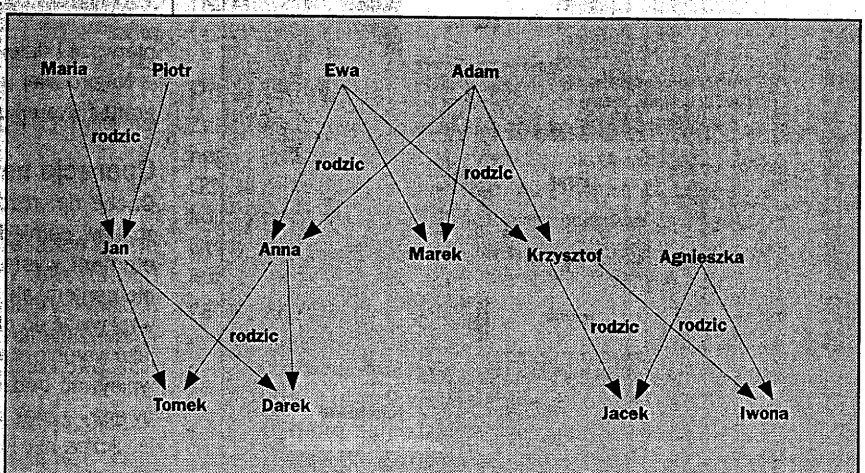
Mamy tu regułę określającą kto jest matką `?y`, zatem najpierw (co jest logiczne) należy wybrać kobiety i wstawić je pod jakąś zmienną `?x` (zmienne oznaczamy przez `?nazwazmiennej` i nie rozróżniamy ich typów), kolejnym krokiem jest sprawdzenie faktu czy dany `?x` (`kobieta`) jest rodzicem i tu wystarczy sprawdzić, czy `kobieta` jest pierwszą wartością w fakcie `zywiciele` (zdefiniowany wyżej) i jeśli tak, to musi być matką. `Assert` dodaje fakt do listy faktów — wnioskowanie — zatem widać jak CLIPS definiuje nowy fakt na podstawie istniejących. Ten etap można by pominąć i po prostu wypisać wyniki, ale fakt `matka` będzie potrzebny później. Z okna faktów można odczytać kto jest matką danego `?y`, ale dla elegancji należało by wypisać wyniki na okno terminala, czy robocze okno CLIPSa — do tego służy nam komenda `printout t` gdzie "t" oznacza wyjście ciągu na terminal. Podanie np. nazwy pliku powodować będzie zapis do niego. `Crlf` oznacza znak nowej linii.

Jak zacząć tworzyć system

Najlepiej zacząć od narysowania grafu. Trzymając się przykładu drzewa genealogicznego (choć systemem eksperckim bym tego nie nazwał...) należałoby zacząć od stworzenia czegoś na podobieństwo Rysunku 3.

Niezbyt to przyjazne, ale pomocne przy tworzeniu reguł. Aby dowiedzieć się kto jest, np. babcią należy nieco bardziej rozbudować lewą stronę reguł. Skorzystamy z powstałej wcześniej reguły `matka`.

```
(defrule babcia
   (kobieta ?x)
   (matka ?x ?y)
   (rodzic ?y ?z)
=> (assert (babka ?x ?z))
   (printout t ?x " jest babka " ?z crlf))
```



Rysunek 3.
Przykładowy graf dla naszego systemu.

Priorytety odpalania reguł zależą od kolejności zapisu w programie

I znów wszystko oczywiste: babcia musi być kobietą ?x, ?x musi być matką ?y i ?y musi mieć dziecko czyli być rodzicem ?z, oczywiście należy dodać kolejny fakt babka, który może być potrzebny do przejścia grafu w innym celu.

Operacje wewnątrz lewej strony reguł

```
(defrule siostra
   (rodzic ?x ?y)
   (kobieta ?y)
   (rodzic ?x ?z&~?y)
=> (assert (siostra ?y ?z))
   (printout t ?y " jest siostra " ?z crlf))
```

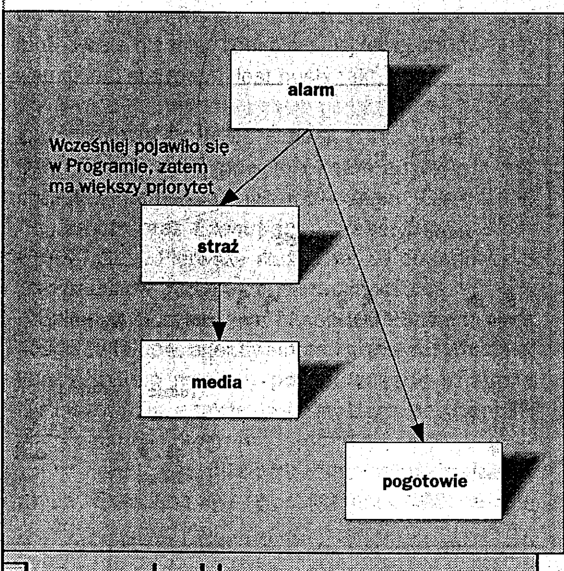
Na przykładzie reguły siostra można zauważyć coś, czego dotychczas nie było. Zapis rodzic ?x ?z&~?y oznacza, że zmienne z i y nie mogą być takie same — konstrukcja powinna być skądś znana — dopasowanie wzorca, np. ?x&?y oznacza, że obie zmienne muszą być takie same.

Zasada działania CLIPS — przykład

Teraz mały przykład zasady działania CLIPSa. Zgodnie z zasadą, że system działa w pętli WHILE pierwszym zadaniem jest: „wybierz regułę o największym priorytecie", zatem krótki przykład zawarty na Listingu 1.

Regułą startową będzie alarm. Przez assert zostanie dodany fakt wlaczam alarm, fakt ten powoduje, że uaktywnia się reguła straz, i powinna uaktywnić się reguła pogotowie, ale kontynuowane są akcje ze straz, zatem kolejną regułą będzie media, a po niej dopiero pogotowie. Ten krótki przykład ilustruje jak należy pisać — najważniejsze reguły jako pierwsze i dalej wg priorytetu — kolejności zapisania.

Najprościej przedstawić ten przykład można za pomocą grafu z Rysunku 4.



Rysunek 4.
Graf zdarzeń przykładowej akcji alarmowej z Listingu 1.

Operacje na ciągach

Chcąc np. rozdzielić ciąg, który składa się z dowolnych znaków, pomiędzy którymi występuje jakiś charakterystyczny element wystarczy przepisywać podciągi pomiędzy tym elementem, ale łatwiej będzie przepisywać go, albo po prostu wypisywać do okna terminala. Reguła obieranie zdefiniowane jako warunki startowe ma separator oraz zmienną ?x, do której wpisywać będziemy aktualną zawartość ciągu. Tu drobna uwaga — znak $ oznacza grupę elementów faktu ciag, jednak jej liczba nie jest określona — po prostu są to wszystkie elementy do wystąpienia zdefiniowanej wartości separatora (tu ?s), podobnie jest z drugiej strony — $?koniec oznacza wszystkie elementy od zdefiniowanego separatora lub

LISTING 1

Przykład działania zasad w CLIPSie.

```
(defrule MAIN::alarm   ; tak tez mozna zapisac
   (temp wysoka)
   (ogien obecny)
   =>
   (assert (wlaczam alarm)))

(defrule MAIN::straz
   (wlaczam alarm)
   =>
   (assert (wezwanie strazy)))

(defrule MAIN::pogotowie
   (wlaczam alarm)
   =>
   (assert (wysylam pogotowie)))

(defrule MAIN::media
   (wezwanie strazy)
   =>
   (assert (media TV)))
```

LISTING 2

Podstawowe operacje na ciągach.

```
(deffacts podciagi
(separator #)
(ciag a b # c d e # f # g h))

(defrule dodaj#
(separator ?ss)
?y <- (ciag $?calosc ?z &~?ss)
=>
(retract ?y)
(assert (ciag $?calosc ?z ?ss)))   ; nowy ciag z konczacym separatorem

(defrule obieranie
(separator ?s)
?x <- (ciag $?poczatek ?s $?koniec); fakt ciag jest przepisywany do ?x
   ; i usuniety
=>
(retract ?x)   ; usuwamy ?x
(assert (ciag $?koniec))   ; tworzymy nowy fakt ciag i wypelniamy
   ; go nowymi danymi - reszta ciagu
   ; poczatkowego
(printout t "wynik" $?poczatek crlf))
```

LISTING 3

Przykład implementacji funkcji silnia.

```
(deffacts dane
   (silnia 0 1)    ; definiujemy warunki poczatkowe
   (silnia 1 1)
   (silnia 6)      ; obliczamy te wartosc
   (licznik 1))
(defrule silowanie
   (silnia ?n ?n1)
   ?x <- (licznik ?n)
   (silnia ~?n)
=> (retract ?x)
   (assert (licznik (+ ?n 1)) (silnia (+ ?n 1) (* (+ ?n 1) ?n1))))
```

LISTING 4

Operacje na plikach.

```
(defrule otworz ; otwarcie plikow
=>
 (open "c:\\clips\\dane.txt" dane "r")     ; odczyt
 (open "c:\\clips\\wyniki.asc" result "w")    ; zapis
 (assert (faza czytanie)))

(defrule czytaj ; czytanie z plikow
 ?f<-(faza czytanie)
=>
 (retract ?f)
 (assert (cisnienie (read dane))))
```

dowolnej zmiennej rozdzielającej ciąg. Komenda `retract` służy do usuwania niepotrzebnych zmiennych. Reguła `dodaj#` służy do opcjonalnego dodania znaku separatora. Bardziej obrazowe są tu jednak komentarze Listing 2.

Innymi ważniejszymi operacjami na ciągach są: `str-compare <ciag1> <ciag2>` służy do logicznego porównania dwóch ciągów; `str-length <ciag>` zwraca długość ciągu; `sub-string <wartosc_poczatkowa> <wartosc_koncowa> <ciag>` — wyłuskuje część ciągu zawartą między `wartosc_poczatkowa` i `wartosc_koncowa`.

Operacje arytmetyczne

Wszystkie operacje arytmetyczne należy wykonywać w nawiasach. Należy zauważyć fakt, że wszystkie działania zapisywane są wg zapisu `znak_<zmienna1> <zmienna2>+`, gdzie `znak` może przyjmować jedną z: `+ - * / DIV` — są to operacje wieloargumentowe; `ABS FLOAT INTEGER MAX MIN` są jednoargumentowe, np. `abs <zmienna1>`.

Przykład funkcji silnia z użyciem wielu operacji matematycznych przedstawia Listing 3.

Operacje wejścia-wyjścia

Czytanie z klawiatury jest dość proste i w zasadzie opiera się na funkcji `assert`, gdzie pod tworzony fakt (tu pod nazwą `zmienna`) z wartością czytaną z klawiatury. Wartość ta może mieć dowolną długość a zakończenie wpisywania to po prostu *<ENTER>*, czy jak kto woli — znak nowej linii: `assert (zmienna (read))`

Zapisywanie do pliku jest nieco bardziej rozbudowane i wymaga przynajmniej trzech operacji: otwarcia z odpowiednim atrybutem (`w` — zapis, `r` — odczyt, `r+` — tryb rw), czyli: `open "nazwapliku.txt" wyjscie "w"`, gdzie `wyjście` to nazwa pliku widoczna w programie. Na końcu programu albo każdego liścia grafu, z którego nie ma już wyjścia należy zamknąć plik komendą `close wyjscie`. Wpisywanie do pliku realizowane jest podobnie jak komenda `printout` — `printout wyjscie "Zalecany papier do druku: " ?x crlf`.

Odczyt z pliku jest dość podobny do pobierania danych z klawiatury `assert (cisnienie (read dane))`, gdzie przez funkcję `assert` tworzymy nowy fakt o nazwie `cisnienie`. Tutaj `dane` jest nazwą zdefiniowanego pliku do odczytu. Odczyt z pliku jest sekwencyjny, linia po linii podobnie jak w przypadku wpisywania danych z klawiatury. Znakiem rozdzielającym jest znak nowej linii.

Przypadek końca pliku można sprawdzić w sposób następujący:

```
(defrule sprawdz
   (cisnienie ?c)
   (test (eq ?c EOF)) ; w tym miejscu testujemy koniec
            pliku
=>
   (assert (faza zamykanie)))
```

W powyższym przykładzie użyto funkcji `test`, służy ona do sprawdzania czy dana zmienna równa jest pewnej wartości. Należy posługiwać się standardowymi operacjami porównania: `eq` oraz `neq`.

Tworzenie pętli nieskończonej

Fakt `zeton`, tworzony jest na samym początku. Wchodzimy do reguły `petla-nieskonczona`, tam `zeton` jest usuwany (`retract`), a następnie tworzony od nowa — znów wywoływana jest reguła `petla-nieskonczona`...

```
(deffacts startuj
(zeton))

(defrule petla-nieskonczona
?i<- (zeton)
=>
(retract ?i)
(assert (zeton)))
```

Podsumowanie:

Zalety systemów eksperckich w zasadzie zależą od konkretnego zastosowania i czasu pracy z konkretnym systemem. Gdy zagadnienie jest trywialne, po pewnym czasie osoba korzystająca z systemu łatwo posiądzie wiedzę i system wtedy będzie raczej przeszkadzać niż pomagać (chyba, że jest to system-nauczyciel). Pamiętać należy o pewnych ograniczeniach systemów eksperckich:

- nie posiadają wiedzy przyczynowo-skutkowej — choć czasami, z pozoru, może na to wyglądać
- nie potrafią samodzielnie pozyskiwać wiedzy — nie należy tego mylić z wprowadzaniem danych potrzebnych do analizy.

■