



Artificial Intelligence Laboratory

ZINTEGROWANY PAKIET SZTUCZNEJ INTELIGENCJI SPHINX® 4.0

KRZYSZTOF MICHALIK

PC-SHELL 4.0

dla Windows 9x/NT/2000

SZKIELETOWY SYSTEM EKSPERTOWY

CZĘŚĆ 2

PODRĘCZNIK INŻYNIERA WIEDZY

KATOWICE 2003



Artificial Intelligence Laboratory

ul. Kossutha 7, 40-844 KATOWICE

tel./fax: tel.: (0-32) 254-41-01 w. 374

tel. kom. 0 502-99-27-28

e-mail: aitech@aitech.com.pl

WWW: <http://www.aitech.com.pl>

Copyright ©1990-2003 AITECH & Krzysztof Michalik

AITECH, Sphinx, CAKE oraz **Neuronix**

są prawnie zastrzeżonymi znakami towarowymi firmy

AITECH, ARTIFICIAL INTELLIGENCE LABORATORY

SPIS TREŚCI

Rozdział 1. SPHINX - JĘZYK REPREZENTACJI WIEDZY SYSTEMU PC-SHELL ...	1-1
INFORMACJE OGÓLNE	1-3
STRUKTURA JĘZYKA OPISU BAZY WIEDZY	1-4
Rozdział 2. OPIS ŹRÓDEŁ WIEDZY	2-1
DEKLARACJE ŹRÓDEŁ WIEDZY	2-3
ŹRÓDŁA TYPU <i>KB</i>	2-3
ŹRÓDŁA TYPU <i>METAPHOR</i> I <i>WHAT_IS</i>	2-4
ŹRÓDŁA TYPU <i>NEURAL_NET</i>	2-4
Rozdział 3. OPIS FASET	3-1
OGÓLNA STRUKTURA OPISU FASET	3-3
SZCZEGÓŁOWY OPIS FASET	3-3
Faseta <i>ASK</i>	3-3
Faseta <i>SINGLE</i>	3-4
Faseta <i>QUERY</i>	3-4
Faseta <i>UNIT</i>	3-5
Faseta <i>VAL</i>	3-5
Faseta <i>PARAM</i>	3-7
Faseta <i>PICTURE</i>	3-8
Faseta <i>SOUND</i>	3-8
Faseta <i>VIDEO</i>	3-9
Znaki sterujące atrybutami tekstu	3-9
Przykład użycia opisu faset	3-11
Rozdział 4. OPIS FAKTÓW I REGUŁ	4-1
BLOK OPISU FAKTÓW	4-3
BLOK OPISU REGUŁ	4-3
Ogólna składnia reguł	4-3
Wyrażenia relacyjne	4-4
Instrukcja przypisania	4-4
Zmienne parametryczne w regułach	4-5
Funkcje matematyczne	4-5
Mechanizm inteligentnego uzgadniania	4-5
Kilka uwag o odwzorowaniu wiedzy w trójkę OAW	4-6
Przykłady poprawnie zbudowanych reguł	4-7
Rozdział 5. PROGRAMOWANIE W SYSTEMIE PC-SHELL	5-1
WPROWADZENIE	5-3
TYPY ZMIENNYCH	5-3
Typy proste	5-3
Typ <i>variant</i>	5-4
Tablice	5-4
Rekordy	5-5
ZMIENNE PARAMETRYCZNE	5-5
FUNKCJE	5-5
Deklarowanie funkcji	5-6
Definiowanie funkcji	5-6
Konwersja argumentów	5-7
Tablice jako argumenty	5-7
Przekazywanie rezultatu przez funkcję	5-7
INSTRUKCJE PROGRAMOWANIA	5-8
Instrukcje proste i złożone	5-8
Instrukcja przypisania	5-8
Instrukcje języka programowania PC-Shell	5-9
WYKORZYSTANIE OKIEN DIALOGOWYCH	5-10
Funkcje dialogowe	5-12

Rozdział 6. APLIKACJE O ARCHITEKTURZE TABLICOWEJ	6-1
OGÓLNE ZASADY BUDOWY APLIKACJI TABLICOWYCH	6-3
Rozdział 7. APLIKACJE HYBRYDOWE	7-1
WPROWADZENIE DO APLIKACJI HYBRYDOWYCH	7-3
Rozdział 8. AUTOMATYCZNA PARAMETRIZACJA BAZ WIEDZY	8-1
WPROWADZENIE	8-3
DEKLAROWANIE ZMIENNYCH PARAMETRYCZNYCH	8-3
ZMIENNE PARAMETRYCZNE W BLOKU REGUŁ	8-4
ZMIENNE PARAMETRYCZNE W BLOKU STEROWANIA	8-4
KATEGORIE PARAMETRÓW	8-4
ZMIANA WARTOŚCI ZMIENNYCH PARAMETRYCZNYCH	8-5
STRUKTURA OPISU KATEGORII W PLIKU INICJALIZACYJNYM	8-7
WIZUALIZACJA ZMIENNYCH PARAMETRYCZNYCH	8-8
ZALETY PARAMETRIZACJI BAZ WIEDZY	8-8
Rozdział 9. ARKUSZE KALKULACYJNE	9-1
ARKUSZE KALKULACYJNE I ICH STOSOWANIE	9-3
NAJWAŻNIEJSZE INFORMACJE O ARKUSZU KALKULACYJNYM	9-3
ARKUSZE KALKULACYJNE – POJĘCIA PODSTAWOWE	9-3
Stałe	9-4
Formuły obliczeniowe	9-4
Operatory	9-4
Tryby adresowania	9-5
Nazwy	9-5
Funkcje	9-5
PROGRAMOWA OBSŁUGA ARKUSZY W JĘZYKU SPHINX	9-5
Rozpoczęcie pracy ze skoroszytem	9-5
Wczytanie danych do arkusza	9-6
Wyświetlenie arkusza na ekranie	9-6
Zamykanie widoku arkusza	9-6
Zapamiętanie skoroszytu	9-6
Zamknięcie skoroszytu	9-7
Odwołania do komórek i zakresów	9-7
PODRĘCZNE MENU ARKUSZA	9-7
Rozdział 10. WYKRESY	10-21
NAJWAŻNIEJSZE INFORMACJE O WYKRESACH	10-23
MECHANIZMY PROGRAMOWEJ OBSŁUGI WYKRESÓW	10-23
Tworzenie wykresu	10-23
Wstawianie danych	10-24
Formatowanie wykresu	10-24
Wizualizacja wykresu	10-25
Zapamiętywanie wykresu	10-25
Wczytanie definicji wykresu	10-25
Zamykanie wykresu	10-25
ŁĄCZENIE WYKRESU Z ARKUSZEM KALKULACYJNYM	10-25
Rozdział 11. DOSTĘP DO BAZ DANYCH	11-1
WPROWADZENIE	11-3
POBIERANIE DANYCH Z BAZY DANYCH	11-3
ZAPIS DANYCH DO BAZY	11-5
STEROWANIE TRANSAKcjAMI	11-5
Rozdział 12. DYNAMICZNA WYMIANA DANYCH (DDE)	12-1
WPROWADZENIE	12-3
PC-SHELL JAKO KLIENT DDE	12-4
PC-SHELL JAKO SERWER DDE	12-6
Wstawianie danych (poke)	12-6
Wykonywanie poleceń (execute)	12-7
Pobieranie danych (request)	12-7

PRZYKŁAD WYKORZYSTANIA MECHANIZMU DDE.....	12-10
Nawiązanie konwersacji z edytorem	12-10
Tworzenie pliku raportu	12-10
Wydruk raportu	12-11
Przykład tworzenia raportu.....	12-11
Rozdział 13. AUTOMATYZACJA OLE.....	13-1
SERWER AUTOMATYZACJI OLE	13-3
PRZYKŁAD WYWOŁANIA PC-SHELLA JAKO SERWERA OLE	13-6
OBSŁUGA AUTOMATYZACJI W JĘZYKU SPHINX	13-6
PRZYKŁAD OBSŁUGI AUTOMATYZACJI OLE	13-7
Rozdział 14. INTEGRACJA SYSTEMU PC-SHELL Z INNYMI APLIKACJAMI.....	14-1
FUNKCJE BIBLIOTECZNE SYSTEMU PC-SHELL	14-3
PRZYKŁADY WYKORZYSTANIA FUNKCJI BIBLIOTECZNYCH	14-7

SPHINX - JĘZYK REPREZENTACJI WIEDZY SYSTEMU PC-SHELL

INFORMACJE OGÓLNE

Język opisu bazy wiedzy systemu PC-Shell służy do formalnego opisu wiedzy eksperckiej z określonej dziedziny. Językami tego typu posługują się przede wszystkim inżynierowie wiedzy. Ich podstawowym zadaniem jest pozyskanie wiedzy od specjalisty, stosując różne techniki wypracowane przez teorię i praktykę dziedziny systemów ekspertowych i zakodowanie jej za pomocą jakiegoś języka formalnego. Wydaje się, że prezentowany język opisu bazy wiedzy jest dostatecznie przejrzysty, by mógł być wykorzystywany również przez niespecjalistów.

System PC-Shell jest systemem regułowym, stąd całość wiedzy o charakterze heurystycznym jest kodowana za pomocą reguł i faktów. Podstawową strukturą reprezentacji wiedzy jest tu trójka: obiekt-atrybut-wartość. Identyfikatory obiektów i atrybutów są symbolami rozpoczynającymi się od małej litery, po której może nastąpić dowolny ciąg znaków alfanumerycznych (liter i cyfr) oraz znaków podkreślenia „_”. Wartości atrybutów mogą być liczbami typu rzeczywistego, łańcuchami znakowymi lub reprezentowane przez zmienne. Liczby mogą być poprzedzone znakiem i zawierać kropkę dziesiętną (zakres wartości: od $3,4 \cdot 10^{-38}$ do $3,4 \cdot 10^{38}$). Łańcuchy znakowe są dowolnymi ciągami znaków zawartymi pomiędzy znakami cudzysłowu. W systemie wykorzystywane są również symbole, będące tu ciągami znaków, rozpoczynającymi się od małej litery, po której może nastąpić dowolny ciąg znaków złożony z liter i cyfr oraz znaku podkreślenia „_”. Nazwy zmiennych zbudowane są podobnie jak symbole, z tą różnicą, że muszą rozpoczynać się od dużej litery. Trójka obiekt-atrybut-wartość, w pełnej postaci, ma następującą składnię:

```

    atrybut ( obiekt )   operator_relacji   liczba      lub
    atrybut ( obiekt ) = łańcuch_znakowy   lub
    atrybut ( obiekt ) = zmienna

```

Identyfikator atrybutu (dla uproszczenia atrybut) jest jedynym obowiązkowym elementem trójki. Zarówno obiekt jak i wartość mogą być opuszczone. Wartość jest poprzedzona jednym z następujących operatorów relacji: „=”, „<”, „>”, „<=”, „>=”. W przypadku pojawienia się zmiennej jedynym dozwolonym operatorem relacji jest znak „=”.

Specjalnego wyjaśnienia wymaga rola operatorów zawierających znak „=”.

System PC-Shell wykorzystuje trzy postacie operatorów kwalifikowanych do tej kategorii:

1. „=”,
2. „==”,
3. „:=”.

Ad. 1.

Operator „=” określa relację przyporządkowania określonej wartości do atrybutu: *atrybut=wartość*. Jest jednocześnie separatorem oddzielającym identyfikator atrybutu od jego wartości. Operator „=” wykorzystywany jest głównie w bloku reguł i faktów.

Przykład:

```

temperatura = 36.6
sytuacja_finansowa = "bardzo dobra"

```

Ad. 2.

Dwuznak „==” oznacza test relacji równości, sprawdzając, czy wartości po lewej i prawej stronie tego operatora są takie same. Typowym kontekstem użycia są instrukcje *if*, *while* i *for* w bloku sterowania (*control*) bazy wiedzy.

Przykład:

```
if ( X == 10 )
  begin
    ciąg_instrukcji
  end;
while ( Y < 10 )
  begin
    ciąg_instrukcji
  end;
```

Ad. 3.

Dwuznak „:=” jest operatorem przypisania wartości do zmiennych. Typowym kontekstem użycia jest instrukcja przypisania w bloku *control*.

Przykład:

```
// deklaracja zmiennej typu znakowego
char Zmienna2;
// deklaracje zmiennych o wartościach rzeczywistych
float Zmienna1, Zmienna3;
Zmienna1 := 123.45;
Zmienna2 := "łańcuch znaków";
Zmienna1 := Zmienna3;
```

KOMENTARZE

W opisie bazy wiedzy można umieszczać komentarze. Tekst komentarza musi rozpoczynać się od dwuznaku „/*” i zawierać w całości w jednym wierszu (por. wcześniejszy przykład). Nie wolno używać komentarzy wewnątrz instrukcji języka programowania.

STRUKTURA JĘZYKA OPISU BAZY WIEDZY

Opis bazy wiedzy w systemie PC-Shell jest podzielony na pięć bloków: blok opisu źródeł, faset, reguł, faktów oraz sterowania. Obowiązkowe jest wystąpienie przynajmniej jednego bloku – faktów lub reguł. Ogólną strukturę opisu bazy wiedzy przedstawiono na Rys. 1. *nazwa_bazy_wiedzy* jest dowolnym symbolem.

W następnych rozdziałach wymienione bloki zostaną omówione szczegółowo.

```
knowledge base nazwa_bazy_wiedzy
  sources
    opis_źródeł
  end;
  facets
    opis_faset
  end;
  rules
    opis_reguł
  end;
  facts
    opis_faktów
  end;
  control
    program
  end;
end;
```

RYS. 1. OGÓLNA STRUKTURA JĘZYKA OPISU BAZY WIEDZY SYSTEMU PC-SHELL

Jak już wspomniano, system PC-Shell ma możliwość korzystania z wielu źródeł wiedzy. Ogólną strukturę opisu źródła wiedzy przedstawiono na Rys. 2.

```
knowledge source nazwa_źródła_wiedzy
  facets
    opis_faset
  end;
  rules
    opis_reguł
  end;
  facts
    opis_faktów
  end;
end;
```

RYS. 2. OGÓLNA STRUKTURA OPISU ŹRÓDŁA WIEDZY

Jeśli system korzysta z architektury tablicowej (tzn. źródeł wiedzy w postaci eksperckich baz wiedzy), to musi wystąpić moduł główny bazy wiedzy (*knowledge base*). W takim przypadku moduł główny nie może zawierać opisu reguł i faktów. Całość wiedzy eksperckiej musi być umieszczona w źródłach. Moduł główny pełni wtedy rolę sterującą (blok *control*) i kontrolną.

Z wymienionych bloków, musi wystąpić przynajmniej jeden: opis reguł, faktów lub *control*. W przypadku, gdy występuje blok reguł lub faktów, obowiązkowo musi wystąpić blok faset, dla deklaracji atrybutów użytych w regułach lub faktach. Przyjęto konwencję, że nawiasy klamrowe „{” i „}”, zapisane drukiem niewytłuszczonym, oznaczają konieczność wyboru jednego z ujętych pomiędzy nimi wariantów. W niektórych przypadkach dla oznaczenia wyboru będzie stosowany operator „|” również zapisany drukiem niewytłuszczonym. Nawiasy kwadratowe zapisane drukiem niewytłuszczonym oznaczają opcjonalność danego elementu składni.

OPIS ŹRÓDEŁ WIEDZY

DEKLARACJE ŹRÓDEŁ WIEDZY

System PC-Shell jest systemem hybrydowym o architekturze tablicowej. Oznacza to, że do rozwiązywania problemów może wykorzystywać wiele heterogenicznych źródeł wiedzy. W obecnej wersji systemu mogą to być: eksperckie bazy wiedzy, aplikacje oparte o sieci neuronowe oraz bazy danych z wyjaśnieniami tekstowymi.

```
sources
  opis_źródeł_wiedzy
end;
```

RYS. 3. OGÓLNA STRUKTURA BLOKU OPISU ŹRÓDEŁ

opis_źródeł_wiedzy musi składać się co najmniej z jednego opisu, składającego się z nazwy źródła oraz specyfikacji właściwości źródła (patrz Rys. 4). Nazwa źródła jest dowolną nazwą ustaloną przez inżyniera wiedzy, natomiast specyfikacje składają się z wyrażeń zawierających słowa kluczowe systemu. Po nazwie źródła musi wystąpić znak „:” (dwukropek).

```
nazwa_źródła:
  właściwość_1 ... właściwość_n;
```

RYS. 4. OGÓLNA STRUKTURA OPISU ŹRÓDŁA WIEDZY

W obecnej wersji dostępne są następujące typy właściwości źródła: *type* oraz *file*. Pełny format wymienionych właściwości przedstawiono poniżej.

```
type { kb | metaphor | what_is | neural_net }
file łańcuch_znaków;
```

RYS. 5. SKŁADNIA OPISU WŁAŚCIWOŚCI ŹRÓDŁA

Wyrażenie *type* służy do zadeklarowania typu źródła:

kb – eksperckie bazy wiedzy,

neural_net – sieci neuronowe,

metaphor – bazy danych zawierające wyjaśnienia typu *metafory*,

what_is – bazy danych zawierające wyjaśnienia typu „co to jest?”.

Wyrażenie *file* określa plik, w którym przechowywane jest źródło wiedzy.

łańcuch_znaków występujący po słowie *file* określa nazwę pliku i, jeśli jest to konieczne, ścieżkę dostępu. Należy dodać, że znaki „\” w ścieżce muszą być podwajane.

ŹRÓDŁA TYPU KB

Typowym źródłem w architekturze tablicowej jest baza wiedzy. W obecnej wersji PC-Shell umożliwia wykorzystanie w jednej aplikacji dowolną ilość różnych baz wiedzy, ujętych w formie źródeł wiedzy. Formalną strukturę opisu źródła wiedzy przedstawiono w rozdziale 1.

Nie jest dozwolone użycie w źródle opisów źródeł oraz bloku sterowania (programu). Moduł główny bazy wiedzy (oznaczony wyrażeniem *knowledge base*) nie może zawierać reguł i faktów. Natomiast wszystkie atrybuty używane w źródłach wiedzy mogą być zadeklarowane w bloku faset tego modułu, lecz ich definicja musi być identyczna z opisem atrybutów podanym w źródle wiedzy.

Odwołanie do źródeł może nastąpić w programie (blok *control*) za pomocą instrukcji *getSource* i *freeSource* (zob. Tom III Instrukcje języka Sphinx). Możliwe jest także ładowanie i uruchamianie źródeł jako samodzielnych baz wiedzy, w celu ich samodzielnego przetestowania. Dozwolone jest także wykorzystywanie tego samego źródła wiedzy przez wiele różnych aplikacji, które wykorzystują go do rozwiązania tego samego podproblemu.

Jeśli wszystkie źródła wiedzy typu *kb* znajdują się w tym samym katalogu i jest to bieżący katalog baz wiedzy, zadeklarowany w menu *Opcje|Katalogi* (zob. rozdział 3. „Podręcznika użytkownika”), to wyrażenie *file* nie musi zawierać ścieżek do poszczególnych źródeł. W przeciwnym wypadku, tzn. gdy źródła znajdują się w innych katalogach niż bieżący katalog baz wiedzy, to nazwę pliku przechowującego źródło należy poprzedzić ścieżką do katalogu, w którym się dane źródło znajduje.

Przykład:

```
// Dotyczy teoretycznej aplikacji dla decyzji
// kredytowych. Zakłada się, że ostateczna decyzja
// podejmowana jest po ustaleniu takich ocen jak:
// sytuacja finansowa klienta, zabezpieczenie
// kredytu, rentowność.

sources
  decyzja_kredytowa:
    type kb
    file "c:\\aitech\\bw\\decyzja.zw";
  sytuacja_finansowa:
    type kb
    file "c:\\aitech\\bw\\sytfina.zw";
  zabezpieczenie:
    type kb
    file "c:\\aitech\\bw\\gwaranc.zw";
  rentownosc:
    type kb
    file "c:\\aitech\\bw\\rentwnsc.zw";
end;
```

ŹRÓDŁA TYPU METAPHOR I WHAT_IS

Deklaracje *metaphor* i *what_is* określają źródła w formie baz wyjaśnień tekstowych, utworzonych za pomocą programu dbMaker. Źródła tego typu wykorzystywane są podczas wyjaśnień typu „co to jest?” oraz objaśnień konkluzji. Deklaracja *metaphor* definiuje bazę *metafor*, również utworzoną za pomocą programu CAKE. *Metafory* mogą być wykorzystywane podczas wyjaśnień typu „jak?”. Więcej informacji na temat tych źródeł i programu CAKE można znaleźć w instrukcji użytkowania programu CAKE.

W obecnej wersji systemu PC-Shell dozwolone jest zadeklarowanie co najwyżej jednego źródła typu *metaphor* i co najwyżej jednego źródła typu *what_is*.

Przykład:

```
sources
  metafora:
    type metaphor
    file "c:\\aitech\\bazy\\kredyt.dbm";
  coto:
    type what_is
    file "c:\\aitech\\bazy\\kredyt.dbw";
end;
```

ŹRÓDŁA TYPU NEURAL_NET

Podobnie jak w przypadku innych deklaracji źródeł wiedzy, deklaracje sieci neuronowych składają się ze zbioru oddzielnych deklaracji sieci neuronowych. Od wersji 2.3 dozwolona jest nieograniczona ilość źródeł typu sieć neuronowa.

Właściwość *file* odnosi się do pliku zawierającego definicję sieci neuronowej, utworzonego za pomocą systemu Neuronix (zob. „Neuronix – Podręcznik użytkownika”). Pliki tego typu mają standardowo

rozszerzenie *npr*. Pliki definiujące sieć zawierają informacje niezbędne do tego, by PC-Shell mógł wygenerować odpowiedni symulator sieci neuronowej.

Wygenerowanie odpowiedniej sieci neuronowej następuje dynamicznie w trakcie pracy systemu, na podstawie informacji zawartych w pliku wskazanym przez wyrażenie *file*.

Przykład:

```
sources  
  prognoza_finansowa:  
    type neural_net  
    file "c:\\aitech\\sieci\\prognoza.npr";  
end;
```


OPIS FASET

OGÓLNA STRUKTURA OPISU FASET

Fasetami określa się tu zbiór deklaracji odnoszących się do wybranych atrybutów. Blok faset zawiera wykaz wszystkich atrybutów używanych w bazie wiedzy, w tym również zawartych w źródłach wiedzy, wraz z przypisanymi do nich fasetami. Nie wszystkie atrybuty muszą być opisane fasetami, lecz wszystkie muszą być zadeklarowane w bloku faset.

Ogólną strukturę bloku opisu faset przedstawiono na Rys. 6.

```
facets
  opis_faset
end;
```

RYS. 6. OGÓLNA STRUKTURA BLOKU FASET

Opis faset składa się z deklaracji globalnych *ask*, *single* oraz zbioru atrybutów i związanych z nimi faset (Rys. 7). Deklaracja *ask* określa, czy system może zadawać pytania o prawdziwość warunków reguł. Jeśli wybrana zostaje opcja "yes", to system będzie zadawał pytania o prawdziwość warunków, które nie mogą być potwierdzone w oparciu o wiedzę zawartą w bazie wiedzy w postaci faktów i reguł. Opcja *no* oznacza, że system nie może zadawać pytań użytkownikowi w celu potwierdzenia prawdziwości warunków. Może jedynie wykorzystywać wiedzę zawartą w bazie wiedzy.

Od tych ogólnych reguł można określić wyjątki, stosując deklarację *ask* w deklaracji faset. System przyjmuje domyślnie: *ask yes* oraz *single no*.

```
[ ask { yes | no }; ]
[ single { yes | no }; ]
atrybut_1 [ deklaracje_faset_1 ];
...
atrybut_n [ deklaracje_faset_n ];
```

RYS. 7. STRUKTURA OPISU FASET

SZCZEGÓŁOWY OPIS FASET

W systemie PC-Shell dostępne są następujące rodzaje faset: *ask*, *single*, *unit*, *val* (ang. *values*) oraz *query*. Spośród wymienionych, musi wystąpić przynajmniej jedna; kolejność deklaracji jest dowolna.

FASETA ASK

```
ask { yes | no }
```

Określa czy system może stawiać pytania dotyczące danego atrybutu. Deklaracja umożliwia tworzenie wyjątków od globalnej deklaracji *ask*, która dotyczy wszystkich atrybutów w bazie wiedzy. Należy podkreślić, że system PC-Shell zadaje pytania jedynie w sytuacji, gdy nie potrafi potwierdzić warunku reguły lub hipotezy wykorzystując fakty i reguły zawarte w bazie wiedzy.

Przykład:

```
facets
  ask no;
  atrybut_1:
    ask yes
    query "Podaj temperaturę ciała:";
end;
```

FASETA SINGLE

```
single { yes | no }
```

Faseta *single* umożliwia zadeklarowanie, że w bazie wiedzy może wystąpić tylko jeden fakt zawierający atrybut, do którego odnosi się ta faseta. Ma to najczęściej miejsce w odniesieniu do atrybutów, których wartości wzajemnie się wykluczają. Dla przykładu, jeśli w bazie wiedzy występuje pewien fakt, np.: *temperatura_ciala=36.6* i atrybut *temperatura_ciala* opisano fasetą *single yes*, to system przyjmie, że w bazie wiedzy nie ma innego faktu stwierdzającego, że *temperatura_ciala=37.5*. Jednakże, gdyby taki fakt się pojawił jako drugi w kolejności, to system będzie go ignorował.

W praktyce zastosowanie fasety *single* umożliwia zredukowanie liczby pytań stawianych przez system o wartość danego atrybutu, przez co jego działanie staje się bardziej „inteligentne”. Sposób wykorzystania tej fasety pokazano w przykładowej bazie wiedzy *grzyby.bw*. Jeśli użytkownik udzieli systemowi odpowiedzi, że hymenofor jest blaszkowaty, to system nie będzie już pytał, czy np. hymenofor jest kolczasty. W tym wypadku fasetę zastosowano w odniesieniu do atrybutu, którego możliwe wartości wzajemnie się wykluczają.

Faseta *single* może pojawić się – w postaci deklaracji globalnej, przed opisami faset poszczególnych atrybutów (podobnie jak *ask*) i oznacza wtedy, że działaniem fasety *single* objęte są praktycznie wszystkie atrybuty używane w danej bazie wiedzy. Faseta *single* może być również użyta w odniesieniu do wybranych atrybutów. W ten sposób można definiować wyjątki, podobnie jak w przypadku fasety *ask*. Domyślnie przyjmuje się wartość *single no*.

Przykład:

```
facets
  ask yes;
  single yes;
  atrybut_1:
    single no;
  atrybut_2:
    single yes;
end;
```

FASETA QUERY

```
query pytanie
query { lista_pytań }
```

Jak już wspomniano system PC-Shell generuje automatycznie pytania o prawdziwość określonych warunków aktywnych reguł lub o wartości atrybutów występujących w tych warunkach. Treść zapytań jest formułowana automatycznie, w sposób jednolity dla całej bazy wiedzy. Faseta *query* umożliwia zdefiniowanie przez użytkownika własnej treści tych zapytań, w odniesieniu do wybranych atrybutów. W przypadku użycia tej fasety, system zada pytanie o treści zgodnej z określoną w fasecie, zamiast pytań automatycznie generowanych przez system. Pytanie jest łańcuchem znakowym.

W wersji 2.1 systemu wprowadzono możliwość definiowania oddzielnych pytań do każdej z wartości atrybutu. Umożliwia to uwarunkowanie pytań w zależności od wartości atrybutu w badanym warunku uzgadnianej reguły. Służy do tego druga postać definicji fasety *query*, gdzie w nawiasach klamrowych podaje się listę pytań odpowiednio do każdej z wartości zdefiniowanej w fasecie *val*. Oczywiście wymagane jest wtedy zdefiniowanie tejże fasety *val* w postaci *val oneof*, dodatkowo jest wymagane, aby ilość wartości danego atrybutu i zapytań na liście była zgodna. W przypadku zastosowania tej postaci zapytań należy się także liczyć z tym, że w przypadku zadania przez system tego typu zapytania lista możliwych odpowiedzi zostaje automatycznie ograniczona do podstawowych: tak, nie lub nie wiem.

W pytaniach mogą się znaleźć specjalne znaki sterujące niektórymi właściwościami tekstu, takimi jak kolor, krój pisma, czcionka. Omówienie znaków sterujących znajduje się w ostatnim punkcie tego rozdziału pt. Znaki

sterujące atrybutami tekstu. Dodatkowo z zapytaniem może być związany rysunek (mapa bitowa), dźwięk lub sekwencja wideo.

Przykład:

```
facets
  temperatura_ciała:
    query "Podaj temperaturę ciała:";
  występuje_ból_mięśni:
    query "Czy pacjent odczuwa bóle mięśni?";
  kolor:
    val oneof
    { "czerwony",
      "żółty",
      "zielone" }
    query
    { "Czy światło ma kolor czerwony?",
      "Czy światło ma kolor żółty?",
      "Czy światło ma kolor zielony?" };
end;
```

FASETA UNIT

```
unit jednostka_miary
```

Faseta *unit* umożliwia zadeklarowanie jednostki miary, w której wyrażane są wartości danego atrybutu. *jednostka_miary* jest w tym wypadku dowolnym tekstem, a ściślej łańcuchem znaków. Podczas wyświetlania informacji zawierającej dany atrybut (np. zapytania systemu, przeglądanie bazy wiedzy), dodatkowo – po wartości – będzie pojawiał się tekst zadeklarowany jako *jednostka_miary*.

Przykład:

```
facets
  wzrost_pacjenta:
    query "Podaj wzrost pacjenta w centymetrach:"
    unit "cm";
  waga_pacjenta:
    query "Podaj wagę pacjenta w kilogramach:"
    unit "kg";
end;
```

FASETA VAL

Określa zbiór dopuszczalnych wartości danego atrybutu. Wartości mogą być liczbami rzeczywistymi (typ *float*) lub łańcuchami znakowymi. W obecnej wersji systemu do określenia dozwolonych lub niedozwolonych wartości służą następujące deklaracje związane z faseta *val*: *oneof*, *someof*, *range*, *except*. W przypadku, gdy wprowadzona do systemu wartość (np. w formie odpowiedzi użytkownika, lub dynamicznie dodawanego faktu za pomocą instrukcji *freadFacts*) wykracza poza dopuszczalny zakres system automatycznie sygnalizuje błąd i nie pozwala na wprowadzenie takiego faktu do bazy wiedzy.

Ubocznym efektem działania faset *oneof* i *someof* może być automatycznie generowane okno zawierające zadeklarowane w fascie wartości.

VAL ONEOF

```
val oneof { wartość_1 , ... , wartość_n }
```

Faseta *val oneof* deklaruje dozwolony zbiór wartości atrybutu, z którym ta fasetta jest związana. Wartości *wartość_1*, ..., *wartość_n* mogą być wyłącznie liczbami lub łańcuchami znakowymi. Nie mogą w liście wartości wystąpić typy mieszane, tzn. liczby i łańcuchy znakowe.

Użycie tej fasety zakłada domyślnie wartość *yes* dla fasety *single*. Dlatego próba wprowadzenia kolejno dowolnych dwóch wartości z listy do bazy wiedzy zakończy się niepowodzeniem.

Zadeklarowany w tej fasecie zbiór wartości pojawia się w formie podpowiedzi, mających postać automatycznie generowanego okienka zawierającego zadeklarowaną listę wartości.

Użytkownik wybiera wartość z okna zgodnie z zasadami obowiązującymi w systemie Windows. Wybrana wartość zostaje automatycznie przypisana zmiennej (o ile występuje) lub uzgodniona z wartością stałą w aktywnym warunku reguły. W rezultacie fakt zawierający wybraną wartość jest wprowadzany do bazy wiedzy.

Przykład:

```
facets
// zapis poprawny
stan_wody:
  query "Określ stan wody w rzece:"
  val oneof { "niski", "średni", "wysoki" };
liczba_pomiarów:
  query "Podaj liczbę pomiarów"
  val oneof { 1, 2, 3, 4, 5 };
// zapis niepoprawny
temperatura_wody:
  query "Określ temperaturę wody"
  val oneof { "niska", 15, "średnia", 16 };
end;
```

VAL SOMEOF

```
val someof { wartość_1 , ... , wartość_n }
```

Podobnie jak fasetta *oneof*, deklaracja *someof* określa zbiór dopuszczalnych wartości danego atrybutu. Różnica polega na tym, że fasetta *someof* umożliwia wybranie kilku wartości z listy i umieszczenie ich w formie faktów w bazie wiedzy. Dlatego fasetta ta zakłada użycie fasety *single no* dla danego atrybutu lub, o ile taka deklaracja nie wystąpiła, niejawnie ustala taką wartość tej fasety. Jawne użycie fasety *single yes* łącznie z fasetą *someof* traktowane jest jako błąd.

Zastosowanie tej fasety może być odpowiednie dla tych atrybutów, które mają wartości niewykluczające się wzajemnie. Podobnie jak fasetta *oneof*, fasetta *someof* generuje podczas dialogu z użytkownikiem okno zawierające listę dopuszczalnych wartości. W tym przypadku jednak, użytkownik może wybrać jednocześnie kilka spośród nich.

VAL RANGE

```
val range przedział
val range { przedział_1 , ... , przedział_n }
```

gdzie:

przedział jest przedziałem otwartym, oznaczonym znakami () lub przedziałem domkniętym, oznaczonym znakami < >.

Faseta *val range* umożliwia deklarowanie wartości atrybutu w formie zbioru dopuszczalnych przedziałów. Wartości w tym przypadku mogą być wyłącznie liczbami. Nie jest dozwolone jednoczesne użycie którejś z

wymienionych faset: *oneof*, *someof*, *except*. Dla oznaczenia wartości minimalnej oraz maksymalnej w danej implementacji można używać odpowiednio symboli *min* oraz *max*.

Przykład:

```
facets
  temperatura_ciała:
    val range < 36, 42 >;
  parametr_X:
    val range { < -5, -1 >, < 1, 5 > };
  parametr_Y:
    val range { < min, 0 ), ( 0, max > };
end;
```

VAL EXCEPT

```
val except przedział
val except { przedział_1 , ... , przedział_n }
```

gdzie:

przedział jest przedziałem otwartym, oznaczonym znakami ()
lub przedziałem domkniętym, oznaczonym znakami <>.

Faseta ta określa zbiór dopuszczalnych wartości związanego z nią atrybutu, przez wyszczególnienie wartości niedozwolonych (w pewnym sensie odwrotnie do fasety *range*). Nie jest dozwolone jednocześnie użycie którejs z wymienionych faset: *oneof*, *someof*, *range*. Podobnie jak w przypadku fasety *range*, dla oznaczenia wartości minimalnej oraz maksymalnej w danej implementacji, można używać odpowiednio symboli *min* oraz *max*.

Przykład:

```
facets
  temperatura_ciała:
    val except { < min, 36), ( 42, max > };
  parametr_X:
    val except { < min, -5 ), ( -1, 1 ), ( max, 5 > };
  parametr_Y:
    val except { 0 };
end;
```

FASETA PARAM

```
param { zmienna_1 = wartość_1, ... , zmienna_n = wartość_n }
```

Faseta *param* umożliwia zadeklarowanie tzw. zmiennych parametrycznych oraz przypisanie im wartości domyślnych. Zmienne takie mogą pojawiać się w bazie wiedzy np. w charakterze wartości progowych lub przedziałów wartości, z którymi porównywane są rzeczywiste wartości danego atrybutu. Dzięki temu mechanizmowi znacznie ułatwiona jest procedura parametryzacji baz wiedzy, która ma miejsce w niektórych zastosowaniach. Zmiana wartości zmiennych parametrycznych może nastąpić zarówno z poziomu programu zawartego w bloku sterowania jak i w sposób interakcyjny za pomocą okna dialogowego, do którego dostęp możliwy jest z opcji *Narzędzia|Parametryzacja*. Mechanizm parametryzacji baz wiedzy został szerzej omówiony w rozdziale 8 niniejszego podręcznika pt. *Automatyczna parametryzacja baz wiedzy*.

Wartości domyślne *wartość_1*, ..., *wartość_n* nie mogą być sprzeczne z deklaracjami typu: *oneof*, *someof*, *range*, *except*.

Przykład:

```
facets
// przykłady deklaracji poprawnych
atrybut_1:
    val range < 1, 3 >
    param { PARMIN = 2.1, PARMAX = 2.8 };
atrybut_2:
    val oneof { "alfa", "beta", "gamma" }
    param { PAR1 = "alfa", PAR2 = "beta" };
// przykłady deklaracji niepoprawnych
atrybut_3:
    val range < 1, 3 >
    param { PARATR3 = -2, PARATR3 = 4 };
atrybut_4:
    val oneof { "alfa", "beta", "gamma" }
    param { PARATR4 = "omega", PARATR4 = "epsilon" };
end;
```

FASETA PICTURE

```
picture nazwa_pliku
picture { plik_1, ... , plik_n }
```

Faseta *picture* umożliwia związywanie atrybutów z rysunkami, np. w formie map bitowych. W obecnej wersji rysunek jest automatycznie pokazywany, gdy pojawia się zapytanie dotyczące atrybutu, z którym rysunek jest związany oraz w przypadku pojawienia się rozwiązania do zapytania o atrybut ze związanym rysunkiem. System umożliwia przypisanie rysunku bezpośrednio do atrybutu (pierwsza postać definicji) lub przypisuje każdej z wartości odrębny rysunek. Druga postać wymaga oczywiście wystąpienia w deklaracji atrybutu fasety *val* w postaci *oneof*. Ilość wartości determinuje ilość elementów listy nazw plików z rysunkami.

Obecnie dopuszczane są jedynie rysunki w formacie *bmp*.

Przykład:

```
facets
grzyb:
    val oneof { "pieczarka", "muchomor", "maślak" }
    picture { "piecz.bmp", "muchomor.bmp", "maslak.bmp" };
hymenofor:
    val oneof { "rurkowaty", "brak" }
    picture "hymenof.bmp";
// Pojawienie się zapytania o warunek grzyb = "maślak"
// spowoduje pojawienie się w oknie konsultacji obrazka
// "maslak.bmp". W przypadku, gdy w oknie konsultacji
// wyświetlana jest lista wartości, podświetlenie jednej
// z nich powoduje pojawienie się rysunku związanego
// z daną wartością. Natomiast pojawienie się zapytania
// o hymenofor z dowolną wartością spowoduje pojawienie się
// zawsze tego samego obrazka ("hymenof.bmp").
```

FASETA SOUND

```
sound nazwa_pliku
sound { plik_1, ... , plik_n }
```

Faseta *sound* umożliwia związywanie atrybutów z dźwiękami (mową, muzyką itp.). *nazwa_pliku* określa plik, w którym przechowywany jest zapis efektów dźwiękowych. Jej składnia jest identyczna jak fasety *picture*. Zadeklarowanie fasety *sound* powoduje pojawienie się podczas zapytania lub w oknie rozwiązania przycisku,

umożliwiającego odtworzenie zawartości danego pliku dźwiękowego. Jedynym akceptowanym w obecnej wersji systemu formatem plików dźwiękowych jest format *wav*.

FASETA VIDEO

```
video nazwa_pliku
video { plik_1, ... , plik_n }
```

Faseta *video* umożliwia związanie atrybutów z animacjami, np. w formie plików *avi* lub *mpg*. System umożliwia przypisanie animacji bezpośrednio do atrybutu (pierwsza postać definicji) lub do każdej z wartości atrybutu. Druga postać wymaga oczywiście wystąpienia w deklaracji atrybutu fasety *val oneof*. Ilość wartości determinuje ilość elementów listy nazw plików zawierających animacje.

Typy animacji uzależnione są od zainstalowanych w systemie sterowników, odpowiedzialnych za odtwarzanie tego typu plików. Standardowo w systemach Windows 9x oraz NT dostępne są sterowniki do odtwarzania plików typu *avi*.

ZNAKI STERUJĄCE ATRYBUTAMI TEKSTU

System PC-Shell umożliwia określenie sposobu formatowania niektórych tekstów wyświetlanych podczas konsultacji. W szczególności dotyczy to tekstów związanych z fasetą *query*, *unit* oraz znajdujących się na liście wartości faset *oneof* lub *someof*. Należy podkreślić, że znaki te są ignorowane w trakcie procesu wnioskowania. W związku z tym, w wartościach warunków reguł nie muszą występować identyczne łańcuchy, tzn. zawierające kody sterujące, łańcuchy znaków (takie jak w fasetach), by mogło dojść do uzgodnienia. Przeciwnie, kody sterujące nie powinny się pojawiać w tekstach reguł.

Inżynier wiedzy może kodować następujące atrybuty tekstu:

- rodzaj czcionki;
- krój pisma;
- kolor;
- format specjalny.

Znaki kodujące właściwości tekstu poprzedzane są otwierającym nawiasem kwadratowym, po którym następuje znak kodujący określony atrybut tekstu. Aby znak „[” był rzeczywiście wyświetlony, a nie interpretowany jako znak sterujący musi zostać podwojony, np. łańcuch „[[a+b]” będzie wyświetlony jako: „,[a+b]”.

RODZAJ CZCIONKI

Do kodowania rodzaju czcionek służą następujące znaki:

- A** Arial
- C** Courier
- N** Courier New
- F** Fixedsys
- M** Modern
- E** MS Sans Serif
- e** MS Serif
- R** Roman
- P** Script
- L** Small Fonts
- Y** Symbol
- Z** System
- T** Terminal
- t** Times New Roman
- W** Wingdings

Rodzaj czcionek jest dobierany automatycznie w sposób umożliwiający prawidłowe wyświetlanie na ekranie polskich znaków diakrytycznych (CE-Central Europe).

Przykład:

```
query "[CPodaj wielkość obrotów firmy]"
```

(wyświetlany tekst pytania w oknie dialogowym zostanie zapisany czcionką *Courier*)

Rodzaj czcionki może być zmieniany wielokrotnie w ramach pojedynczego łańcucha znaków, dozwolony jest więc następujący zapis:

```
query "[CPodaj [Awielkość obrotów [Cfirmy]"
```

Pytanie będzie zobrazowane następująco:

```
Podaj wielkość obrotów firmy
```

KRÓJ PISMA

Do kodowania kroju pisma służą następujące znaki:

- B** pogrubienie (*bold*) włączone
- b** pogrubienie wyłączony
- I** kursywa (*italic*) włączona
- i** kursywa wyłączona
- S** przekreślenie (*striketru*) włączone
- s** przekreślenie wyłączony
- U** podkreślenie (*underline*) włączone
- u** podkreślenie wyłączony

Dozwolona jest wielokrotna zmiana kroju pisma w obrębie pojedynczego tekstu.

Przykład:

```
query "[AWybierz [Brodzaj zabezpieczenia[b kredytu]"
```

Pytanie będzie zobrazowane następująco:

```
Wybierz rodzaj zabezpieczenia kredytu
```

KOLOR

Kolor tła jest przyjmowany domyślnie jako kolor systemowy. Do ustalania kolorów znaków tekstu służą następujące kody liczbowe (0-9):

- | | |
|--------------------------|-------------------------|
| 0 czarny | 5 jasnoniebieski |
| 1 czerwony | 6 żółty |
| 2 ciemnoniebieski | 7 szary |
| 3 jasnozielony | 8 ciemnozielony |
| 4 purpurowy | 9 biały |

Przykład:

```
query "[A[2Wybierz [B[1rodzaj zabezpieczenia[b[2 kredytu]"
```

(format tekstu będzie zbliżony do poprzedniego przykładu, z tym, że słowa „rodzaj zabezpieczenia” zostaną wyświetlone w kolorze niebieskim)

FORMAT SPECJALNY

- D** indeks górny (*superscript*) włączony
- d** indeks górny wyłączony
- G** indeks dolny (*subscript*) włączony
- g** indeks dolny wyłączony
- f** ułamki, do spacji

(w treści ułamków nie należy używać innych kodów sterujących)

q przywrócenie standardowych ustawień dotyczących tekstu

Przykład:

```
query "Podaj wartość v[G2[g [[ [fm/s[q ]]"
```

(format powyższego zapytania: „Podaj wartość v_2 [m/s]”)

PRZYKŁAD UŻYCIA OPISU FASET

```
facets
  single yes;
  decyzja_kredytowa;           // atrybuty bez faset
  gwarancje_kredytowe;
  profil_klienta;             // atrybuty z fasetami
    val oneof { "ok", "zły" };
  sytuacja_finansowa:
    val oneof { "dobra", "zła" };
  rodzaj_zabezpieczenia:
    query "PODAJ RODZAJ ZABEZPIECZENIA"
    val oneof { "hipoteka", "papiery wartościowe" };
  płynność_finansowa:
    query "OKREŚL PŁYNNOŚĆ FINANSOWĄ"
    val oneof { "dobra", "zachowana", "zła" };
  poziom_zadłużenia:
    query "PODAJ POZIOM ZADŁUŻENIA"
    val oneof { "dopuszczalny", "niedopuszczalny" };
  rentowność_sprzedaży:
    query "JAKA JEST RENTOWNOŚĆ SPRZEDAŻY?"
    val oneof { "niska", "wysoka", "bardzo wysoka" };
  dotychczasowe_kredyty:
    query "PODAJ POZIOM SPŁATY DOTYCHCZASOWYCH KREDYTÓW"
    val oneof { "spłacone", "nie spłacone" };
end;
```


OPIS FAKTÓW I REGUŁ

BLOK OPISU FAKTÓW

Blok opisu faktów umożliwia zapisanie wiedzy o charakterze faktograficznym, będącej zbiorem faktów na jakiś temat. Najczęściej będą to informacje względnie stałe, o charakterze parametrów dla bazy wiedzy. W praktyce ten blok nie musi wystąpić.

Opis faktów składa się ze zbioru faktów poprzedzonych słowem *facts* oraz zakończonych słowem *end* (Rys. 8).

```
facts
  opis_faktów
end;
```

RYS. 8. STRUKTURA BLOKU OPISU FAKTÓW

Podstawowym elementem faktów jest trójka *obiekt-atrybut-wartość* („trójka OAW”), która może być poprzedzona znakiem negacji. W systemie PC-Shell negacja jest oznaczona słowem *not*. W odróżnieniu od ogólnej składni trójki OAW (przedstawionej w rozdziale 1) fakty nie mogą zawierać zmiennych, a jedynym operatorem pomiędzy atrybutem i wartością jest znak „=”.

Dopuszczalna składnia faktu:

```
trójka_OAW;
not trójka_OAW;
```

Poniżej zamieszczono przykładowy, poprawnie zbudowany, blok faktów.

Przykład:

```
facts
  miesięczny_obrót_mld ( firma_X ) = 14.5;
  ocena_zabezpieczeń_kredytowych ( firma_Y ) = "bardzo dobra";
  not rodzaj_zabezpieczeń = "papiery wartościowe";
  rodzaj_zabezpieczeń = "krajowe depozyty gotówkowe";
  rodzaj_zabezpieczeń = "hipoteka";
  not płynność_finansowa_zachowana;
  ocena_zdolności_kredytowej ( klient_1 ) = "dobra";
end;
```

Podczas ładowania bazy wiedzy fakty umieszczane są na początku (przed regułami). W procesie wnioskowania fakty pobierane są do uzgodnienia w kolejności zgodnej z ich pozycją w bazie wiedzy (dotyczy to również reguł).

BLOK OPISU REGUŁ

OGÓLNA SKŁADNIA REGUŁ

Blok reguł pełni główną rolę z punktu widzenia reprezentacji wiedzy eksperckiej. Formalizm reguł jest dziedzinowo-niezależny i umożliwia kodowanie wiedzy praktycznie z każdej dziedziny.

Opis reguł składa się ze zbioru reguł poprzedzonych słowem *rules* oraz zakończonych słowem *end*.

```
rules
  opis_reguł
end;
```

RYS. 9. STRUKTURA BLOKU OPISU REGUŁ

Standardowo składnia reguł składa się z konkluzji oraz części warunkowej. Konkluzja oraz część warunkowa oddzielone są słowem kluczowym *if*. Część warunkowa musi zawierać przynajmniej jeden warunek. Reguły

```
[ nr_reguły: ]   konkluzja if
    warunek_1 & warunek_2 & ... & warunek_n;
[ nr_reguły: ]   konkluzja if
    warunek 1 | warunek 2 & warunek 3;
```

```
trójka_OAW
not trójka_OAW
wyrażenia_relacyjne
instrukcja_przypisania
```

liczba lub wyrażeniem dwuargumentowym. W przypadku wyrażenia dwuargumentowego oba argumenty są rozdzielone dwuargumentowymi operatorami: „-”, „+”, „*”, „/”. Argumenty mogą być liczbami, zmiennymi o przypisanych wcześniej wartościach typu liczba lub funkcjami matematycznymi. Ilustrację poprawnych instrukcji przypisania zawarto w poniższym przykładzie.

Przykłady poprawnie zbudowanych instrukcji przypisania

```
X := 5.1
X1 := X2
X3 := 2 + 5
X4 := X5 - X6
X6 := 2 + sin( X7 )
X8 := sin( X9 ) + cos( X10 )
```

ZMIENNE PARAMETRYCZNE W REGUŁACH

W części warunkowej reguł mogą być używane tzw. zmienne parametryczne. Identyfikatory zmiennych parametrycznych budowane są w taki sam sposób jak zwykłych zmiennych, z tą różnicą, że identyfikator jest poprzedzony znakiem „#” (np. #ZM_PARAM). Istota zmiennych tego typu oraz zasady korzystania z nich omówiono w rozdziale 8 pt. „Automatyczna parametryzacja baz wiedzy”. Z tego względu pomijamy tu szczegóły tej metody. Należy jedynie podkreślić, że zmienne parametryczne zawsze mają ustaloną wartość, która może być zmieniona w ściśle określony sposób. Mechanizm nawrotów nie wpływa na wartość zmiennych parametrycznych. Można w uproszczeniu założyć, że z punktu widzenia wnioskowania zmienne parametryczne zachowują się tak jak stałe.

FUNKCJE MATEMATYCZNE

Funkcje matematyczne składają się z identyfikatora funkcji i argumentu funkcji ujętego w nawiasy okrągłe. Argumentem funkcji może być liczba lub zmienna o przypisanej wcześniej wartości typu liczba.

W obecnej wersji systemu dostępne są następujące funkcje matematyczne:

sin	oblicza wartość równą sinusowi argumentu (argument jest wyrażony w radianach);
cos	oblicza wartość równą cosinusowi argumentu (argument jest wyrażony w radianach);
tan	oblicza wartość równą tangensowi argumentu (argument jest wyrażony w radianach);
exp	obliczanie wartości potęgi o podstawie e ;
abs	wyznaczanie wartości bezwzględnej argumentu;
log	obliczanie wartości logarytmu naturalnego argumentu;
log10	obliczanie wartości logarytmu dziesiętnego;
sqrt	wyznaczanie pierwiastka kwadratowego argumentu;
floor	zaokrąglanie w dół (największa z możliwych wartości całkowitych, nie większa od argumentu);
ceil	zaokrąglanie w górę;
asin	oblicza wartość funkcji arcusa sinusa; wartość musi być z przedziału -1 .. 1;
acos	oblicza wartość funkcji arcusa cosinusa; wartość musi być z przedziału -1 .. 1;
atan	oblicza wartość funkcji arcusa tangensa;
sinh	oblicza wartość sinusa hiperbolicznego;
cosh	oblicza wartość cosinusa hiperbolicznego;
tanh	oblicza wartość tangensa hiperbolicznego;

MECHANIZM INTELIGENTNEGO UZGADNIANIA

Trójka *obiekt-atrybut-wartość* może zawierać operatory relacji „<”, „<=”, „=”, „>=”, „>” odnoszące się do wartości atrybutu. W systemie PC-Shell zastosowano rozwiązanie (wcześniej wprowadzone w systemie PC-Expert) określane tu jako *mechanizm inteligentnego uzgadniania*. Polega ono na tym, że podczas uz-

gadniania dwóch wyrażeń uzgadnianie następuje nie tylko wtedy, gdy są one identyczne lub gdy zawierają zmienne, lecz również wtedy, gdy naturalna interpretacja wyrażeń zawierających operatory relacji pozwala na ich uzgodnienie. Odróżnia to mechanizm uzgadniania systemu PC-Shell od prologowego mechanizmu uzgadniania, który ma charakter wyłącznie syntaktyczny.

Dla przykładu, następujące wyrażenia systemu PC-Shell będą uzgodnione:

```
warunek reguły lub hipoteza:
    miesięczny_obrót( firma_X ) >= 2500
fakt lub konkluzja reguły:
    miesięczny_obrót( firma_X ) = 2600
warunek reguły lub hipoteza:
    zadłużenie( firma_X ) >= 25
fakt lub konkluzja reguły:
    zadłużenie_mld( firma_X ) = 35
```

KILKA UWAG O ODWZOROWANIU WIEDZY W TRÓJKĘ OAW

Ten sam fragment wiedzy może być w różny sposób odwzorowany w określony język reprezentacji. Dotyczy to również języka opisu bazy wiedzy systemu PC-Shell, który jest niczym innym jak językiem służącym do reprezentowania wiedzy. Dla przykładu informacja, że:

„temperatura ciała pacjenta X wynosi 37 stopni Celsjusza”

może być zakodowana m.in. w następujący sposób:

```
a) temperatura_ciała_pacjenta_X_wynosi_37_stopni_Celsjusza
b) temperatura_ciała_pacjenta_X=37
c) temperatura_ciała(pacjent_X)=37
```

Wybór konkretnego sposobu reprezentacji wiedzy musi być bardzo przemyślany, z uwzględnieniem przynajmniej następujących kwestii:

1. Tekst zawarty w bazie wiedzy musi być czytelny, zrozumiały dla potencjalnych użytkowników systemu ekspertowego, bowiem jawna reprezentacja wiedzy stanowi jedną z najważniejszych cech tej technologii;
2. Należy wziąć pod uwagę, kto będzie użytkownikiem systemu. Jeśli symbole atrybutów i wartości będą bardzo skrótowe, mogą stać się nieczytelne dla użytkownika aplikacji. Pomocą mogą być objaśnienia typu *what is* i *metaphory*. Z drugiej strony zbyt długie teksty mogą być uciążliwe podczas pisania bazy wiedzy i co ważniejsze nie będą mieściły się na ekranie, wymagając „przewijania”;
3. Wybór konkretnego sposobu odwzorowania wpływa na sposób rozwiązywania problemów i możliwości uzyskiwania informacji. Można to zilustrować w oparciu o wyszczególnione w podanym wcześniej w przykładzie przypadki:

ad. a)

W trakcie wnioskowania nie można przekazywać wartości temperatury. Jedyna dozwolona postać celu wygląda następująco:

```
temperatura_ciała_pacjenta_X_wynosi_37_stopni_Celsjusza
```

ad. b)

W tym przypadku możliwe jest przekazywanie i testowanie wartości temperatury. Cel może zawierać zmienną reprezentującą tę wartość.

ad. c)

Podobnie jak w przypadku 2 możliwe jest uzyskanie i testowanie wartości temperatury. Ponadto w sposób jawny zapisany jest obiekt, którego dotyczy ta porcja wiedzy.

PRZYKŁADY POPRAWNIE ZBUDOWANYCH REGUŁ

Przykład A:

```
// Reguły zaczerpnięto z systemu diagnosta MC 14007
// (Gutt T., Michalik K. [1988]).
rules
1: ma_zbyt_wysoka_rezystancje( a1 ) if
    działa_wadliwie( a1 ),
    ma_wartości_wyższe_od_oczekiwanych( irc ),
    n = N,
    b = B,
    n_( Irc ) = W1,
    W2 := W1 / N,
    W3 := 1 - B,
    W2 > w3;
2: zbyt_duża_dawka_implantacji_E if
    działa_wadliwie( d7 ),
    ma_wartości_niższe_od_oczekiwanych( ute_1 ),
    ma_wartości_niższe_od_oczekiwanych( ute_2 ),
    ma_wartości_niższe_od_oczekiwanych( utf );
3: ma_zbyt_wysoka_rezystancje( e2 ) if
    ma_zbyt_wysoka_rezystancje( a1 ),
    działa_wadliwie( e2 ),
    ma_wartości_wyższe_od_oczekiwanych( rs );
4: zbyt_wąskie_ścieżki_dyfuzyjne if
    ma_zbyt_wysoka_rezystancje( e2 ),
    ma_wartości_niższe_od_oczekiwanych( skrl );
end;
```

Przykład B:

```
rules
1: decyzja_kredytowa = "przyznać kredyt" if
    profil_klienta = "ok" &
    gwarancje_kredytowe = "dostateczne" &
    sytuacja_finansowa = "dobra";
2: decyzja_kredytowa = "skonsultuj z przełożonym" if
    profil_klienta = "ok" & (
        ( gwarancje_kredytowe = "niedostateczne" &
          sytuacja_finansowa = "dobra" ) |
        ( gwarancje_kredytowe = "dostateczne" &
          sytuacja_finansowa = "zła" ) );
3: decyzja_kredytowa = "odmówić kredytu" if
    profil_klienta = "zły" |
    gwarancje_kredytowe = "niedostateczne" &
    sytuacja_finansowa = "zła" );
end;
```

Przykład C:

```
// Przykład reguł wykorzystujących zmienne parametryczne
rules
  atrybut1 = "wartość1" if
    atrybut2 = X &
    Y := sin( #PARMIN ) + #PARMIN &
    Z := sin( #PARMAX ) / sin( #PARMIN ) &
    Z := cos( #PARMAX ) &
    X >= Y &
    X <= #PARMAX;
  atrybut1 = "wartość2" if
    atr4 = X &
    X >= #PARMIN &
    X <= #PARMAX;
end;
```

PROGRAMOWANIE W SYSTEMIE PC-SHELL

WPROWADZENIE

Jak już wspomniano, systemy szkieletowe w porównaniu z językami programowania są znacznie wydajniejszymi narzędziami do budowy dziedzicznych systemów ekspertowych. Jednakże na ogół odbywa się to kosztem utraty elastyczności rozwiązań. W systemie PC-Shell 1.1 uczyniono pierwszy krok w stronę rozwiązania tego problemu. Istota tego rozwiązania polega na tym, że system PC-Shell (w wersji 1.1 i późniejszych) jest systemem hybrydowym, tj. łączącym deklaratywną reprezentacją wiedzy z proceduralnym językiem umożliwiającym programowanie działań systemu. Program w systemie PC-Shell składa się ze zbioru instrukcji zawartych w bloku *control* (Rys. 11). W ten sposób zachowana została zasada rozdzielania wiedzy eksperckiej oraz tzw. sterowania. Wydaje się, że rozwiązanie to powinno ułatwić budowę systemów ekspertowych działających w praktyce. Język programowania systemu PC-Shell będzie doskonałony i rozwijany również w przyszłych wersjach tego systemu, z uwzględnieniem specyficznych wymagań stawianych przez system ekspertowy. Rozwój języka będzie następował z zachowaniem zgodności z poprzednimi wersjami systemu.

```
control
  program
end;
```

RYS. 11. BLOK STEROWANIA

Program w systemie PC-Shell składa się z dwóch części: deklaracji zmiennych oraz zbioru instrukcji:

```
[deklaracje zmiennych]
instrukcje
```

Jeśli program nie wykorzystuje zmiennych, to deklaracje nie muszą oczywiście wystąpić. Nazwy zmiennych muszą rozpoczynać się od dużej litery, po której może (choć nie musi) następować ciąg liter, cyfr, znaków podkreślenia. Ta sama nazwa nie może się pojawić w dwóch różnych deklaracjach typów.

Specjalnym typem są tzw. *zmiennie systemowe*, niejawnie deklarowane w systemie. Ich identyfikatory nie mogą się pojawić w deklaracjach zmiennych. W obecnej wersji systemu wprowadzono jedną zmienną tego typu o nazwie *RETURN*. Wartość tej zmiennej jest modyfikowana użyciem niektórych instrukcji programowania, pojawiając się jako „efekt uboczny” ich działania. Daje to możliwość specjalnej obsługi niektórych zdarzeń w systemie.

Ostatnim typem zmiennych wprowadzonych w wersji 2.1 systemu są *zmiennie parametryczne*, deklarowane w bloku faset. Dokładny opis tych zmiennych znajduje się w rozdziale 8.

Nowością wprowadzoną w wersji 2.2 jest możliwość deklarowania *funkcji* (więcej informacji na ten temat zamieszczono w dalszej części niniejszego rozdziału).

TYPY ZMIENNYCH

Typy zmiennych w języku Sphinx można podzielić na typy proste (numeryczne oraz typ tekstowy) oraz typy złożone - tablice oraz rekordy.

TYPY PROSTE

Poza podstawowymi typami *char* oraz *float* (występującymi we wcześniejszych wersjach systemu), dostępne są również typy całkowite *int*, *long* oraz rozszerzony typ zmiennoprzecinkowy *double*. Typy te odpowiadają typom zdefiniowanym w języku C. Dokładne dane typów numerycznych zestawiono w tablicy poniżej. Od wersji 2.3 (32-bitowej) zakresy wartości poszczególnych typów przedstawione są w tabeli poniżej.

Typ	Ilość bajtów	Zakres
int	4	-2 147 483 648...2 147 483 648
long	4	-2 147 483 648...2 147 483 648
float	4	$3,4 \cdot 10^{-38} \dots 3,4 \cdot 10^{38}$
double	8	$1,7 \cdot 10^{-308} \dots 1,7 \cdot 10^{308}$

W związku z wprowadzeniem nowych typów numerycznych, zdefiniowano następujące zasady obliczania wyrażeń arytmetycznych:

- wyrażenie numeryczne jest obliczane w zakresie takiego typu jak typ zmiennej po lewej stronie wyrażenia;
- po prawej stronie wyrażenia mogą wystąpić tylko elementy o typie zgodnym z typem wyrażenia lub typem dającym się automatycznie rzutować;
- rzutowanie typów polega na automatycznym przypisaniu typu mniej dokładnego do typu bardziej dokładnego oraz na automatycznym przypisaniu typu całkowitego do typu zmiennoprzecinkowego;
- w systemie PC-Shell znakiem oddzielającym część całkowitą od części ułamkowej (symbol dziesiętny) w liczbach zmiennoprzecinkowych jest kropka, dlatego w celu zapewnienia zgodności z innymi systemami wprowadzono instrukcję *c_ntos* służącą do konwertowania liczb na inne sposoby zapisu liczb (w szczególności polski z przecinkiem).

Przykłady :

```
double D;
float F;
long L;
int I;
D := F + L * I;      // wyrażenie poprawne, wykonywane
                    // na typie double
L := F + I;          // wyrażenie błędne - float nie może być
                    // przekształcony automatycznie do typu
                    // całkowitego
F := L * I + 10.4;    // wyrażenie poprawne
F := D * 1.3;         // wyrażenie błędne - double jest typem
                    // bardziej dokładnym niż float
```

Do konwersji pomiędzy typami zmiennoprzecinkowymi a całkowitymi służy instrukcja *ftoi*.

TYP VARIANT

W związku z wprowadzeniem od wersji 4.0 obsługi mechanizmu OLE Automation wprowadzono nowy typ danych **variant**. Służący do przechowywania i obsługi elementów zarządzanych przez ten mechanizm. Zmienne tego typu mogą być używane jedynie w instrukcjach związanych z OLE Automation.

TABLICE

Deklarowane zmienne mogą być zmiennymi prostymi lub zmiennymi indeksowymi (tablicami). Od wersji 1.7 dozwolone jest deklarowanie i używanie tablic jedno i dwuwymiarowych. Wszystkie elementy tablicy są tego samego typu. Deklaracja tablicy określa poza typem i nazwą tablicy również jej wymiar i rozmiar. Tablice indeksowane są od 0, a nie od 1 jak np. w Fortranie lub PL/1.

Przykład deklaracji tablic:

```
float Tab1[ 10 ], Tab2[ 5, 10 ];
int Tab3[ 2, 2 ];
```

REKORDY

Rekord jest typem danych reprezentującym zdefiniowany przez użytkownika zbiór pól (zmiennych) dowolnego typu prostego. Rekord może zawierać dowolną ilość pól prostych, polami nie mogą być tablice, natomiast jest możliwe zdefiniowanie zmiennej rekordowej będącej tablicą rekordów określonego typu. Każdy rekord musi posiadać swoją unikatową nazwę określaną w momencie definiowania rekordu. Składnia definicji rekordu wygląda następująco:

```
record nazwa_rekordu
begin
    deklaracja_pola_1,
    ...
    deklaracja_pola_n
end
opcjonalna_lista_zmiennych;
```

Przykład:

```
record Time
begin
    int Hour, Min, Sec;
end Czas;

record Dane_personalne
begin
    char Nazwisko, Imie;
    int Wiek;
    int Pensja;
end Tablica[ 10 ];

record Dane_personalne Rekord1;
```

Dostęp do pól rekordu uzyskuje się przy użyciu operatora . (kropka), nazywanego operatorem wyluskania. Operator wyluskania pola występuje bezpośrednio po nazwie zmiennej a przed nazwą pola. Poniżej przedstawiamy przykłady poprawnych wyluskań pól:

```
Czas.Hour := 12;
Czas.Min := 10;
Czas.Sec := 0;
Czas.Min := Czas.Min + 10;
Tablica[0].Nazwisko := "Kowalski";
Tablica[0].Imie := "Jan";
Rekord1.Nazwisko := Tablica[0].Nazwisko;
```

ZMIENNE PARAMETRYCZNE

Zmienne parametryczne są deklarowane wyłącznie w bloku faset. W bloku *control* można je wykorzystywać, w szczególności do zmiany bieżącej wartości, podobnie jak zwykłych zmiennych. Zasadniczym miejscem użycia zmiennych parametrycznych jest blok reguł, gdzie identyfikatory zmiennych parametrycznych muszą być poprzedzone znakiem #. Szersze omówienie tych zmiennych można znaleźć w rozdziale 8 pt. „Automatyczna parametryzacja baz wiedzy”.

FUNKCJE

Jednym z najważniejszych elementów współczesnych języków programowania strukturalnego są funkcje i procedury. Również język Sphinx udostępnia możliwość deklarowania funkcji, które w rozumieniu języka Pascal są procedurami. Funkcje muszą być zadeklarowane na początku bloku sterowania przed jakąkolwiek inną instrukcją. Możliwe jest wcześniejsze zadeklarowanie nagłówków przed pełną definicją, co zapewnia możliwość wywołań rekurencyjnych (w tym rekurencję pośrednią). Jedynymi elementami języka dopuszczalnymi przed funkcjami są deklaracje zmiennych oraz definicje rekordów, w takim przypadku są one traktowane jak zmienne globalne, dostępne również wewnątrz funkcji.

DEKLAROWANIE FUNKCJI

Deklarowanie funkcji polega na podaniu słowa kluczowego *function*, po nim nazwy symbolicznej (tzn. z małej litery) funkcji oraz ewentualnie listy argumentów. Lista argumentów ujęta jest w nawiasy okrągłe (oraz), w przypadku jej braku nawiasy pomijamy. Definicja argumentu składa się z nazwy typu, który może być typem rekordowym, następnie ewentualnego znaku referencji, potem nazwy zmiennej (z dużej litery) a później ewentualnego dwuznaku [] oznaczającego tablice deklarowanego typu. Kolejne argumenty oddzielone są od siebie znakiem przecinka.

```
function <nazwa_funkcji> [ ( <deklaracje_argumentów> ) ] ;
```

Przykład:

```
function test( int X, int Y, char Tekst )  
function testTbl( int Pos[], char Tekst )  
function sqrt( double d, double &wynik )
```

Deklarowanie funkcji kończymy znakiem średnika lub bezpośrednio definiujemy ciało funkcji w bloku **begin ... end**.

DEFINIOWANIE FUNKCJI

Definicja funkcji rozpoczyna się od nagłówka funkcji, czyli podania jej nazwy i ewentualnie (o ile występują) listy argumentów. Następnie wewnątrz bloku *begin...end* definiujemy zachowanie funkcji. W definicji mogą się znajdować deklaracje zmiennych lokalnych, instrukcje języka Sphinx oraz wywołania funkcji zadeklarowanych wcześniej. W obrębie definicji funkcji widoczne są zmienne zadeklarowane w jej nagłówku i są one traktowane jako lokalne. Jedyną zmienną globalną w języku Sphinx widoczną wewnątrz każdej funkcji jest zmienna *RETURN* służąca do przekazywania wyniku wykonywania funkcji i instrukcji.

Zmienne referencyjne, wyróżniane symbolem „&”, oznaczają powiązanie zmiennej podanej w momencie wywołania ze zmienną lokalną istniejącą w ciele funkcji. Jakakolwiek zmiana zawartości zmiennej referencyjnej jest uwidoczniona w zmiennej zewnętrznej. Szczególnie zalecane jest podawanie jako zmiennych referencyjnych dużych struktur tablicowych i rekordów jako, że wywołanie referencyjne jest szybsze nie powoduje bowiem odkładania na stos dużej ilości zmiennych (argumentów).

Wewnątrz definicji zabronione są instrukcje *menu*, *exit* oraz *goto*. Możliwe jest wywoływanie rekurencyjne tej samej funkcji bezpośrednio jak również poprzez rekurencję pośrednią.

Przykład definicji funkcji rekurencyjnej:

```
function silnia( long X, long &Result )  
  begin  
    if ( X <= 1 )  
      begin  
        Result := 1;  
      end  
    else  
      begin  
        long L;  
        L := X - 1;  
        silnia( L, Result );  
        Result := Result * X;  
      end;  
  end;
```

Przykład wywołania funkcji *silnia*:

```
function wykonajSilnie( long X )
begin
    long Wynik;
    char Łańcuch;
    silnia( X, Wynik );
    sprintf( Łańcuch, " %ld! = %ld", X, Wynik );
    MessageBox( 0,0, "Silnia", Łańcuch );
end;
```

Wywołanie `wykonajSilnie(4)` spowoduje obliczenie i wyświetlenie wyniku na ekranie („4! = 24”).

KONWERSJA ARGUMENTÓW

Typy argumentów podane w wywołaniach funkcji muszą zgadzać się z typami argumentów w definicjach i deklaracjach funkcji. Konwersje następują w przypadku podania bezpośrednio jako argumentów wywołania stałych w postaci liczb.

TABLICE JAKO ARGUMENTY

Przy definiowaniu funkcji jako argumentów możemy użyć tablic, oznaczamy je przez podanie po nazwie zmiennej dwuznaku `[]`, bez jawnego określenia rozmiaru tablicy. Zwiększa to znacznie elastyczność operowania na tablicach. Rzeczywisty rozmiar tablicy można uzyskać za pomocą instrukcji `arraySize`.

Przykład:

```
function zerujTablice( int &Tablica[] )
begin
    int Rozmiar;
    int I1;
    tblSize( Tablica, 0, Rozmiar );
    while ( Rozmiar > 0 )
    begin
        Rozmiar := Rozmiar - 1;
        Tablica[ Rozmiar ] := 0;
    end;
end;
```

PRZEKAZYWANIE REZULTATU PRZEZ FUNKCJĘ

Funkcje definiowane przez użytkownika mogą przekazywać wartości przez zmienne referencyjne lub zmienną globalną *RETURN*, która zasadniczo służy do sygnalizowania prawidłowości wykonania instrukcji lub funkcji (1 – prawidłowe zakończenie, 0 – błąd). W funkcjach możliwe jest szybsze (i bardziej czytelne) przypisanie wartości tej zmiennej przez użycie słowa kluczowego *return* i podania po nim przekazywanej wartości. Instrukcja ta powoduje natychmiastowe przerwanie wykonywania funkcji i przypisanie odpowiedniej wartości zmiennej globalnej *RETURN*.

Przykład:

```
function test( int X )
begin
  if ( X < 0 )
    begin
      return 0;
    end;
    // ...
  return 1;
end;

int Y;
Y := 1;
test( Y );
if ( RETURN == 1 )
  begin
    MessageBox( 0, 0, "Wywołanie funkcji", "prawidłowe" );
  end
else
  begin
    MessageBox( 0, 0, "Wywołanie funkcji", "nieprawidłowe" );
  end;
```

INSTRUKCJE PROGRAMOWANIA

INSTRUKCJE PROSTE I ZŁOŻONE

Instrukcje w systemie PC-Shell podzielone są na proste i złożone. Instrukcje złożone zbudowane są w formie bloku rozpoczynającego się słowem *begin* i zakończone słowem *end*. Wewnątrz bloku zawarty jest niepusty zbiór instrukcji prostych (zob. Rys. 12). Instrukcje nie będące instrukcjami złożonymi będą nazywane instrukcjami prostymi. Wszystkie instrukcje omawiane w dalszej części tego rozdziału są instrukcjami prostymi.

```
begin
  instrukcja_prosta_1;
  ...
  instrukcja_prosta_n;
end;
```

RYS. 12. SKŁADNIA INSTRUKCJI ZŁOŻONYCH

Instrukcje złożone wykorzystywane są w powiązaniu z niektórymi instrukcjami prostymi, np. *for*, *while*, *if*.

INSTRUKCJA PRZYPISANIA

Instrukcja przypisania może przyjmować jedną z następujących postaci:

```
zmienna := wyrażenie_arytmetyczne;
zmienna := symbol
zmienna := łańcuch_znakowy
zmienna := zmienna
```

Zmienna może być zmienną dowolnego typu prostego (*char*, *float*, *double*, *int* lub *long*). Jeśli zmienna jest zadeklarowana jako *char*, to po prawej stronie operatora przypisania może pojawić się zmienna typu *char*, symbol lub łańcuch znakowy.

Przykład:

```
char C1, C2, C3[4,5];
C2 := wartość; //_typu_symbol;
C2 := "wartość typu łańcuch znakowy";
C1 := C2;
C3[2,3] := C1;
```

Jeśli po lewej stronie operatora `:=` występuje zmienna zadeklarowana jako zmienna numeryczna, to po prawej stronie operatora może pojawić się wyłącznie wyrażenie arytmetyczne, którego szczególnym przypadkiem jest stała liczbowa lub zmienna.

W wyrażeniu arytmetycznym mogą pojawić się:

- operatory arytmetyczne `+`, `-`, `*`, `/`;
- liczby;
- zmienne typu numerycznego (patrz uwagi poniżej);
- wywołania funkcji;
- nawiasy okrągłe.

W odróżnieniu od wyrażeń arytmetycznych stosowanych jako warunki reguł, wyrażenia w bloku *control* mogą zawierać większą liczbę argumentów.

Wyrażenie arytmetyczne dla operatorów o tych samych priorytetach jest wykonywane od strony lewej do prawej. Jeśli argument występuje między operatorami o różnych priorytetach, to następuje jego związanie z operatorem o priorytecie wyższym. Operatory `+` i `-` mają ten sam priorytet. Jest on mniejszy od takich samych priorytetów operatorów `*` i `/`. Kolejność obliczania wyrażeń może być zmieniona przez zastosowanie nawiasów okrągłych na zasadach powszechnie obowiązujących dla tego typu wyrażeń.

W przypadku używania zmiennych różnego typu obowiązuje zasada, że zmienna po lewej stronie musi być zmienną o największym zakresie spośród zmiennych występujących po prawej zmiennych (stałych). Zmienne o mniejszym zakresie są przypisywane automatycznie do postaci ogólniejszej i w tej postaci następuje obliczanie wyrażeń. Kolejność od najbardziej ogólnego (największego zakresu) do najmniejszego jest następująca:

```
double > float > long > int
```

Oznacza to, że np. zmiennej typu *float* może być przypisana zmienna typu *long* lub *int*, ale nie *double*.

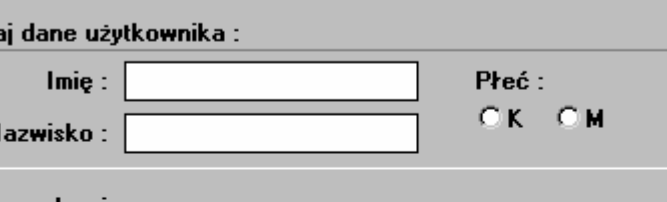
Przykład:

```
float X, Y, Z[ 2 ];
int I; long LONG;
double ZmiennaX;
X := ( 2 + 3 ) * ( 4 - 1 );
Y := 15 - X * 2; // (Y = -15)
Z[ 1 ] := 0;
Z[ 0 ] := 2 * sin( Z[ 1 ] ) + Y; // (Z = -15)
I := 1;
ZmiennaX := I + Y * 1.45;
X := I;
X := ZmiennaX; // błędne przypisanie typu bardziej dokładnego
// do typu mniej dokładnego
I := LONG; // błąd!
LONG := I; // poprawne wyrażenie
```

INSTRUKCJE JĘZYKA PROGRAMOWANIA PC-SHELL

Pełny, uporządkowany alfabetycznie, spis wszystkich instrukcji języka programowania systemu PC-Shell zamieszczono w części 3. dokumentacji pt. „Instrukcje języka Sphinx”.

Tworzenie tego typu zasobów wymaga od inżyniera wiedzy posiadania narzędzi umożliwiających ich tworzenie, takich jak Borland ResourceWorkshop. Możliwe jest także wykorzystanie okien standardowych systemu Windows. Zasadniczo okna dialogowe służą do wyświetlania oraz pobierania danych od użytkownika. Z tego względu nie wystarczy uaktywnić okna dialogowego, należy również określić rodzaj pobieranych danych oraz rodzaj danych, które należy wyświetlić w oknie dialogowym. W języku Sphinx zastosowano rozwiązanie polegające na powiązaniu pojedynczych elementów okna dialogowego z odpowiadającymi im zmiennymi bloku sterowania.



The screenshot shows a dialog box titled "Przykład okna dialogowego wykonanego w języku SPHINX". The dialog has a close button (X) in the top right corner. The main content area is divided into several sections:

- Podaj dane użytkownika :** This section contains two text input fields for "Imię :" and "Nazwisko :". To the right of these fields is a label "Płeć :" followed by two radio buttons labeled "K" and "M".
- Data urodzenia :** This section contains three input fields: "Dzień :", "Miesiąc :" (which is a dropdown menu), and "Rok :".
- Stan cywilny** and **Średni zarobek ostatnich 3 miesięcy :** These are labels for the bottom section, which includes a checkbox labeled "Wolny" and a text input field for the average income, followed by the symbol "zł".

At the bottom of the dialog are two buttons: "OK" and "Anuluj".

Obsługę okna dialogowego w języku Sphinx dzielimy na 3 etapy:

1. inicjalizację okna dialogowego;
2. powiązanie obiektów kontrolnych okna dialogowego ze zmiennymi;
3. uaktywnienie okna dialogowego – rozpoczęcie interakcji z użytkownikiem.

INICJALIZACJA OKNA DIALOGOWEGO

Uwaga! Nie należy zmieniać wartości identyfikatora okna dialogowego przed jego wyświetleniem.

Przykład:

```
int ID;
// zainicjowanie okna dialogowego "DIALOG_1"
// z biblioteki "custom.dll"
dlgCreate( ID, "custom.dll", "DIALOG_1" );
// od tego miejsca w programie utworzone okno będzie
// identyfikowane za pośrednictwem identyfikatora ID
```

POWIĄZANIE OBIEKTÓW KONTROLNYCH ZE ZMIENNYMI

Do powiązania kolejnych elementów kontrolnych (przyciski, pola edycyjne, itp.) okna dialogowego ze zmiennymi służą instrukcje typu *dlgBind...*, wśród których znajdują się: *dlgBindStatic*, *dlgBindEdit*, *dlgBindListBox*, *dlgBindComboBox*, *dlgBindRadioButton* oraz *dlgBindCheckBox*.

Element kontrolny	Instrukcja
pole statyczne (ang. <i>static</i>)	<i>dlgBindStatic</i>
pole edycyjne (ang. <i>edit-box</i>)	<i>dlgBindEdit</i>
lista prosta (ang. <i>list-box</i>)	<i>dlgBindListBox</i>
lista typu <i>combo-box</i>	<i>dlgBindComboBox</i>
przycisk typu <i>check-box</i>	<i>dlgBindCheckBox</i>
Przycisk typu <i>radio-button</i>	<i>dlgBindRadioButton</i>

Wszystkie powyższe instrukcje wymagają podania, jako pierwszego argumentu, identyfikatora okna dialogowego (utworzonego na etapie inicjowania) oraz jako drugiego argumentu – identyfikatora elementu kontrolnego.

Uwaga! Nie wolno jako identyfikatora pola podawać wartości *-1*, jaką posiadają często pola statyczne. Wartość ta oznaczałaby zakaz zmiany zawartości pola!

W zależności od typu elementu kontrolnego dalsza lista argumentów instrukcji jest różna i najczęściej wymaga jednej zmiennej (np. typu *char* dla pola edycji lub typu *int* dla przycisków). Dla list wymagane jest dodatkowo podanie tablicy wartości, którymi ma być zainicjowana lista. Wypełnienie listy polega na umieszczeniu wszystkich podanych wartości aż do napotkania łańcucha pustego.

Przykład:

```
int ID, RetVal;
char Nazw, Imię;
int PłećK, PłećM, StanCyw;
char Mies[ 12 ];
record Data
begin
    char Dzień;
    int Miesiac;
    char Rok;
end DataUrodz;
char Zarobek;
dlgCreate( ID, "custom.dll", "PLACE_DLG" );
// inicjalizacja kolejnych obiektów kontrolnych
// i powiązanie ich ze zmiennymi
Imię := "";
Nazw := "";
dlgBindEdit( ID, 101, Imię );
dlgBindEdit( ID, 102, Nazw );
PłećK := 0;
```

```

PłećM := 0;
dlgBindRadioButton( ID, 103, PłećK );
dlgBindRadioButton( ID, 104, PłećM );
DataUrodz.Dzien := "1";
DataUrodz.Miesiac := 1;
DataUrodz.Rok := "1980";
Mies[ 0 ] := "Styczeń";
Mies[ 1 ] := "Luty";
Mies[ 2 ] := "Marzec";
Mies[ 3 ] := "Kwiecień";
Mies[ 4 ] := "Maj";
Mies[ 5 ] := "Czerwiec";
Mies[ 6 ] := "Lipiec";
Mies[ 7 ] := "Sierpień";
Mies[ 8 ] := "Wrzesień";
Mies[ 9 ] := "Październik";
Mies[ 10 ] := "Listopad";
Mies[ 11 ] := "Grudzień";
dlgBindEdit( ID, 105, DataUrodz.Dzien );
dlgBindListBox( ID, 106, Mies, DataUrodz.Miesiac );
dlgBindEdit( ID, 107, DataUrodz.Rok );
StanCyw := 1; // załącz
dlgBindCheckBox( ID, 108, StanCyw );
Zarobek := "0";
dlgBindEdit( ID, 109, Zarobek );
// uaktywnienie utworzonego okna dialogowego
dlgExecute( ID, RetVal );

```

UAKTYWNIENIE OKNA DIALOGOWEGO

Uaktywnienie okna dialogowego następuje po wywołaniu instrukcji *dlgExecute*. Po utworzeniu obiektu okna i ustawieniu wszystkich jego elementów kontrolnych (zgodnie z wartościami powiązanych z nimi zmiennych), okno zostaje wyświetlone na ekranie. System przechodzi w stan oczekiwania na działanie ze strony użytkownika. Po zamknięciu okna dialogowego przez naciśnięcie przycisku „OK” wartości poszczególnych elementów kontrolnych są wpisywane do powiązanych z nimi zmiennych. Jeżeli zamknięcie okna nastąpiło w wyniku naciśnięcia przycisku „Anuluj”, wartości zmiennych, związanych z elementami kontrolnymi, pozostają niezmienione.

FUNKCJE DIALOGOWE

Funkcje dialogowe umożliwiają skojarzenie z dowolnym przyciskiem, znajdującym się w oknie dialogowym, określonej akcji, zapisanej w postaci funkcji. Instrukcja *dlgBindButton*, przypisująca akcję przyciskowi, wymaga podania jedynie nazwy funkcji, natomiast postać jej argumentów wynika z organizacji okna dialogowego oraz kolejności powiązania kolejnych elementów kontrolnych ze zmiennymi. I tak, kolejne argumenty powinny odpowiadać kolejnym polom okna dialogowego (kolejność jest warunkowana kolejnością wywoływania instrukcji typu *dlgBind...*), przy czym typ każdego argumentu musi być taki sam jak typ odpowiadający polu (czyli taki jak w odpowiedniej instrukcji *dlgBind...*). Ilość argumentów funkcji może być mniejsza niż ilość pól (ale nie większa!). W tym przypadku z wnętrza funkcji nie ma dostępu do niektórych pól.

Zainicjowanie przez użytkownika akcji, poprzez przyciśnięcie przycisku, powoduje wywołanie przypisanej funkcji, gdzie kolejnym argumentom przypisywane są bieżące wartości elementów kontrolnych okna dialogowego. Zmiana wartości parametrów wewnątrz funkcji spowoduje przepisanie nowych wartości do odpowiednich pól okna po zakończeniu wykonywania funkcji.

Pola statyczne nie posiadają odwzorowania do zmiennych, dlatego są one ignorowane w czasie wywoływania funkcji dialogowych i nie wolno ich uwzględniać na liście argumentów funkcji.

Przykład:

```
// Założenie:
// W bibliotece test.dll znajduje się definicja okna dialogowego
// DIALOG_1, zawierającego następujące elementy kontrolne:
// ID_NAZWISKO(101) - pole edycyjne (Nazwisko)
// ID_IMIE(102) - pole edycyjne (Imię)
// ID_PLECMEZ(103) - przycisk radio-button (Mężczyzna)
// ID_PLECKOBIETA(104) - przycisk radio-button (Kobieta)
// ID_CZYSC (200) - przycisk "Wyczyść"

// Funkcja czyści pola okna dialogowego
function wyczyśćPola( char &Nazw, char &Imie, int &M, int &K )
begin
    Nazw := "";
    Imie := "";
    M := 1; // domyślnie mężczyzna
    K := 0;
end;

int DLG, RET;
char Nazwisko, Imie;
int K,M;
dlgCreate( DLG, "test.dll", "DIALOG_1" );
dlgBindEdit( DLG, 101, Nazwisko );
dlgBindEdit( DLG, 102, Imie );
dlgBindRadioButton( DLG, 103, M );
dlgBindRadioButton( DLG, 104, K );
dlgBindButton( DLG, 200, "wyczyśćPola" );
Nazwisko := "Kowalski"; // domyślne wartości
Imie := "Jan";
M := 1;
K := 0;
dlgExecute( DLG, RET );
```


APLIKACJE O ARCHITEKTURZE TABLICOWEJ

OGÓLNE ZASADY BUDOWY APLIKACJI TABLICOWYCH

System PC-Shell zawiera elementy architektury tablicowej, co oznacza, że zapewnia obsługę tzw. *źródeł wiedzy*. Źródła wiedzy stanowią specjalistyczne bazy wiedzy, ujęte w osobne pliki, przeznaczone do rozwiązywania podproblemów wyodrębnionych w problemie głównym.

Aplikacja wykorzystująca architekturę tablicową musi składać się z modułu głównego, rozpoczynającego się wyrażeniem *knowledge base* oraz pewnej liczby źródeł (nieograniczonej), rozpoczynających się wyrażeniem *knowledge source*. W obecnej wersji systemu, każde źródło wiedzy musi mieć następującą strukturę:

```
knowledge source nazwa_źródła
  facets
    opis_faset
  end;
  rules
    opis_reguł
  end;
  facts
    opis_faktów
  end;
end;
```

Opis faset i reguł jest obowiązkowy natomiast opis faktów może być pominięty. Opis źródeł i sterowania może być zawarty jedynie w module głównym bazy wiedzy. W przypadku wykorzystywania źródeł wiedzy numeracja reguł musi być jawna. Jednocześnie należy pamiętać o tym, że numery reguł muszą być unikatowe w skali całej bazy wiedzy (włącznie ze źródłami). Proponuje się jako pierwszą cyfrę numerów reguł dawać identyfikator źródła, co zapobiega powstawaniu tych samych numerów w kilku różnych źródłach.

Wykorzystanie architektury tablicowej zilustrowano prostym przykładem do analizy wniosków kredytowych, zawartym w czterech plikach: *kredyt1.bw* (główny plik bazy wiedzy), *deckred.zw*, *profil.zw*, *gwaranc.zw*, *sytfm.zw* (źródła wiedzy).

Przykład 1. Wykorzystanie instrukcji *goal* i *addSolution*

```
// Główny plik bazy wiedzy zawierający deklaracje źródeł
// wiedzy oraz blok sterowania
knowledge base kredyt
  sources
    deckred:
      type kb
      file "deckred.zw";
    profil:
      type kb
      file "profil.zw";
    gwarancje:
      type kb
      file "gwaranc.zw";
    sytfm:
      type kb
      file "sytfm.zw";
  end;
  control
    int Odp;
    char S1, S2, S3;
    run;
    S1 := "KREDYT";
    S2 := "DEMONSTRACYJNY SYSTEM TABLICOWY\nKrzysztof
      Michalik\nAITECH Katowice";
    S3 := "Copyright (C)1995-99 AITECH Artificial Intelligence
      Laboratory";
    vignette( S1, S2, S3 );
    solutionWin( no );
```

```

Odp := 1;
while ( Odp == 1 )
begin
  setSysText( problem, "[BOcena profilu klienta[b" );
  solve( profil, "profil_klienta = Profil_klienta" );
  setSysText( problem, "[BOcena gwarancji kredytowych[b" );
  solve( gwarancje, "gwarancje_kredytowe =
                    Gwarancje_kredytowe" );
  setSysText( problem, "[BOcena sytuacji finansowej
                        kredytobiorcy[b" );
  solve( sytfina, "sytuacja_finansowa=Sytuacja_finansowa" );
  solutionWin( yes );
  setSysText( problem, "[BOcena wniosku kredytowego[b" );
  solve( deckred, "decyzja_kredytowa=DECYZJA_KREDYTOWA" );
  confirmBox( 0, 0, "KONTYNUOWAĆ KONSULTACJE ?", "Naciśnij 'OK' jeśli tak,
'ANULUJ' jeśli nie", Odp );
  delNewFacts;
  solutionWin( no );
end;
solutionWin( yes );
end;
end;
// --- koniec pliku "kredyt1.bw"

// Główne źródło wiedzy oceniające ostatecznie decyzję
// kredytową na podstawie oceny klienta, gwarancji posiadanych
// przez badany podmiot i jego sytuacji finansowej
knowledge source dec_kred
facets
  single yes;
  decyzja_kredytowa:
    val oneof { "przyznać kredyt", "odmówić kredytu",
               "skonsultuj z przełożonym" };
  profil_klienta:
    val oneof { "ok", "zły" };
  gwarancje_kredytowe:
    val oneof { "dostateczne", "niedostateczne" };
  sytuacja_finansowa:
    val oneof { "dobra", "zła" };
end;
rules
1: decyzja_kredytowa = "przyznać kredyt" if
  profil_klienta = "ok",
  gwarancje_kredytowe = "dostateczne",
  sytuacja_finansowa = "dobra";

```



```

2: decyzja_kredytowa = "skonsultuj z przełożonym" if
    profil_klienta = "ok",
    gwarancje_kredytowe = "niedostateczne",
    sytuacja_finansowa = "dobra";
3: decyzja_kredytowa = "skonsultuj z przełożonym" if
    profil_klienta = "ok",
    gwarancje_kredytowe = "dostateczne",
    sytuacja_finansowa = "zła";
4: decyzja_kredytowa = "skonsultuj z przełożonym" if
    profil_klienta="zły",
    gwarancje_kredytowe = "dostateczne",
    sytuacja_finansowa = "dobra";
5: decyzja_kredytowa = "odmówić kredytu" if
    gwarancje_kredytowe = "niedostateczne",
    sytuacja_finansowa = "zła";
6: decyzja_kredytowa = "odmówić kredytu" if
    profil_klienta = "zły",
    gwarancje_kredytowe = "niedostateczne";
7: decyzja_kredytowa = "odmówić kredytu" if
    profil_klienta = "zły",
    sytuacja_finansowa = "zła";
end;
end;
// --- koniec pliku "deckred.zw"

// Źródło wiedzy oceniające profil klienta
// ubiegającego się o kredyt
knowledge source prof_klienta
    facets
        single yes;
        profil_klienta:
            val oneof { "ok", "zły" };
        status_prawny_klienta:
            query "Podaj status prawny klienta"
            val oneof { "nie ma osobowości prawnej",
                "ma osobowość prawną" };
        dotychczasowe_kredyty:
            query "Podaj poziom spłaty dotychczasowych kredytów"
            val oneof { "spłacone", "niespłacone" };
    end;
    rules
        11: profil_klienta = "ok" if
            status_prawny_klienta = "ma osobowość prawną",
            dotychczasowe_kredyty = "spłacone";
        12: profil_klienta = "zły" if
            status_prawny_klienta = "nie ma osobowości prawnej";
        13: profil_klienta = "zły" if
            dotychczasowe_kredyty = "niespłacone";
    end;
end;
// --- koniec pliku "profil.zw"

// Źródło oceny gwarancji kredytowych na podstawie rodzaju
// zabezpieczenia oferowanego przez oceniany podmiot
knowledge source gwar_kred
    facets
        single yes;
        gwarancje_kredytowe:

```

```

    val oneof { "dostateczne", "niedostateczne" };
    rodzaj_zabezpieczenia:
    query "Podaj rodzaj zabezpieczenia"
    val oneof { "brak", "hipoteka", "papiery wartościowe",
                "trudno zbywalne" };

end;
rules
21: gwarancje_kredytowe = "dostateczne" if
    rodzaj_zabezpieczenia = "hipoteka";
22: gwarancje_kredytowe = "dostateczne" if
    rodzaj_zabezpieczenia = "papiery wartościowe";
23: gwarancje_kredytowe = "niedostateczne" if
    rodzaj_zabezpieczenia = "trudno zbywalne";
24: gwarancje_kredytowe = "niedostateczne" if
    rodzaj_zabezpieczenia = "brak";
end;
end;
// --- koniec pliku "gwaranc.zw"

// Źródło wiedzy oceniające sytuację
// finansową badanego podmiotu

knowledge source syt_finansowa
facets
    single yes;
    sytuacja_finansowa:
    val oneof { "dobra", "zła" };
    płynność_finansowa:
    query "Określ płynność finansową"
    val oneof { "dobra", "zachowana", "zła" };
    poziom_zadłużenia:
    query "Podaj poziom zadłużenia"
    val oneof { "dopuszczalny", "niedopuszczalny" };
    rentowność_sprzedaży:
    query "Jaka jest rentowność sprzedaży?"
    val oneof { "niska", "wysoka", "bardzo wysoka" };
end;
rules
31: sytuacja_finansowa = "dobra" if
    płynność_finansowa = "dobra",
    rentowność_sprzedaży = "wysoka";
32: sytuacja_finansowa = "dobra" if
    płynność_finansowa = "dobra",
    rentowność_sprzedaży = "bardzo wysoka";
35: sytuacja_finansowa = "dobra" if
    płynność_finansowa = "zachowana",
    rentowność_sprzedaży = "bardzo wysoka";
36: sytuacja_finansowa = "zła" if
    płynność_finansowa = "zła";
37: sytuacja_finansowa = "zła" if
    rentowność_sprzedaży = "niska";
38: sytuacja_finansowa = "dobra" if
    płynność_finansowa = "zachowana",
    rentowność_sprzedaży = "wysoka",
    poziom_zadłużenia = "dopuszczalny";

```

```

39: sytuacja_finansowa = "zła" if
    płynność_finansowa = "zachowana",
    rentowność_sprzedaży = "wysoka",
    poziom_zadłużenia = "niedopuszczalny";
end;
end;
// --- koniec pliku "sytfm.zw"

```

W systemie PC-Shell dostępne są instrukcje pozwalające zastosować inne warianty rozwiązania obsługi źródeł wiedzy z poziomu programu użytkownika.

Poniższy fragment kodu jest równoważny przedstawionemu w przykładzie 1 (dla uproszczenia zamieszczono jedynie blok sterujący – pozostałe bloki kodu należy pozostawić bez zmian w stosunku do przykładu 1).

Przykład 2. Użycie instrukcji *solve*

```

control
    int Odp;
    run;
    createAppWindow;
    Odp := 1;
    while( Odp == 1 )
    begin
        solutionWin( no );
        solve( profil, "profil_klienta = PK" );
        solve( gwarancje, "gwarancje_kredytowe = GK" );
        solve( sytfm, "sytuacja_finansowa = SF" );
        solutionWin( yes );
        solve( deckred, "decyzja kredytowa = DK" );
        confirmBox( 0, 0, "Kontynuować konsultacje?", "Naciśnij OK jeśli tak,
ANULUJ jeśli nie", Odp );
        delNewFacts;
    end;
end;

```


APLIKACJE HYBRYDOWE

WPROWADZENIE DO APLIKACJI HYBRYDOWYCH

Nowatorską cechą systemu PC-Shell jest to, że posiada on architekturę hybrydową, łączącą w sobie system ekspertowy i symulator sieci neuronowej. Daje to możliwość tworzenia tzw. aplikacji hybrydowych, tj. powstających w rezultacie wykorzystania tej dość rzadkiej, wśród spotykanych systemów ekspertowych, cechy. Uzasadnieniem dla budowy tego typu systemów mogą być następujące przesłanki:

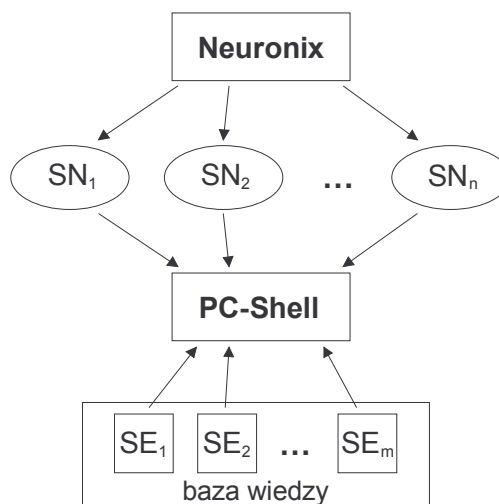
- nie zawsze możliwe jest pozyskanie reguł heurystycznych dla baz wiedzy systemu ekspertowego, istnieją natomiast dane mogące służyć do budowy sieci neuronowej;
- nie zawsze istnieje odpowiedni zbiór danych mogący być podstawą do uczenia sieci neuronowej, istnieje natomiast możliwość zbudowania bazy wiedzy;
- rozwiązywany problem ma naturę złożoną, a do rozwiązania każdego z podproblemów bardziej efektywna lub korzystna wydaje się inna technologia przetwarzania wiedzy;
- w przypadku gdy różne metody mogą być użyte do rozwiązania tego samego problemu, niektóre cechy jednej z nich odgrywają decydującą rolę (np. może to być zdolność systemów ekspertowych do wyjaśnień oraz kodyfikacji wiedzy).

Te i inne przesłanki inspirowały rozwój hybrydowej architektury systemu PC-Shell. Należy podkreślić, że integracja nastąpiła tu nie tylko na poziomie architektury systemu ale również w zakresie języka reprezentacji wiedzy. Istotny jest również fakt, że w zastosowanym rozwiązaniu nie ma potrzeby komunikowania się pomiędzy aplikacjami za pomocą plików lub przez uruchamianie procesów zewnętrznych.

Budowa hybrydowej aplikacji w systemie PC-Shell składa się z następujących etapów:

1. Stworzenie za pomocą systemu Neuronix aplikacji neuronowych (SN_1, \dots, SN_m), dla wybranych podproblemów.
2. Opracowanie baz wiedzy (w formie źródeł wiedzy) SE_1, \dots, SE_n .
3. Integracja opracowanych w poprzednich etapach źródeł wiedzy na poziomie języka reprezentacji wiedzy systemu PC-Shell.

Z punktu widzenia praktycznej realizacji rola systemu Neuronix sprowadza się do uczenia sieci i wygenerowania definicji aplikacji neuronowej w formie pliku definiującego (pliki z rozszerzeniem *npr*) oraz pliku zawierającego zbiór wag (pliki z rozszerzeniem *wag*).



RYS. 14. STRUKTURA APLIKACJI HYBRYDOWEJ W SYSTEMIE PC-SHELL

Zastosowane w systemie PC-Shell rozwiązanie jest na tyle elastyczne, że pozwala na swobodny przepływ wiedzy pomiędzy siecią neuronową a systemem ekspertowym w obu kierunkach: z sieci neuronowej do systemu ekspertowego i z systemu ekspertowego do sieci neuronowej. Oznacza to w praktyce, że rozwiązanie znalezione przez każde z tych źródeł wiedzy jest dostępne dla pozostałych. Schematycznie strukturę takich aplikacji i przepływ wiedzy pomiędzy nimi można ująć w następujący sposób:

a) przepływ wiedzy z sieci neuronowej do systemu ekspertowego:

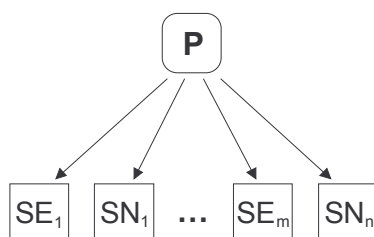
```
knowledge base szkic_aplikacji_hybrydowej
sources
  nn_appl:
    type neural_net
    file "nn_appl.npr";
end;
control
  char Problem;
  float WE[ <rozmiar_1> ], WY[ <rozmiar_2> ];
  // generacja sieci n zgodnie z "nn_appl"
  initNetwork( nn_appl );
  // uruchomienie sieci z wektorem wejsc. WE
  // rezultat jest przekazany w tablicy WY
  runNetwork( nn_appl, WE, WY );
  // usunięcie sieci neuronowej
  delNetwork( nn_appl );
  addFact( <obiekt_1>, <atrybut_1>, <wartość_1> );
  addFact( <obiekt_1>, <atrybut_m>, <wartość_rozmiar_2> );
  goal( Problem );
end;
end;
```

b) przepływ wiedzy z systemu ekspertowego do sieci neuronowej

```
knowledge base szkic_aplikacji_hybrydowej
sources
  nn_appl:
    type neural_net
    file "nn_appl.npr";
end;
control
  char Problem, M[ <rozmiar_1> ];
  // N - liczba rozwiązań zapamiętanych za pomocą
  // instrukcji saveSolution
  float WE[ <rozmiar_2> ], WY[ <rozmiar_3> ], N;
  goal( Problem );
  // Zapamiętanie rozwiązań w formie łańcuchów
  // tekstowych w tablicy "M"
  saveSolution( M, N );
  // miejsce na procedury konwersji łańcuchów na
  // liczby przekazane na wejście sieci n.
  runNetwork( WE, WY );
end;
end;
```

Sposób wiązania ze sobą poszczególnych podproblemów, z zastosowaniem różnych technologii do ich rozwiązywania, może być praktycznie dowolny. Ogólnie podstawowe struktury problemów można przedstawić następująco:

a)



b)



gdzie:

SE₁, ..., SE_m – podproblemy rozwiązywane w oparciu o bazy wiedzy;

SN₁, ..., SN_n – podproblemy rozwiązywane za pomocą sieci neuronowych;

P – problem główny;

R – rozwiązanie ostateczne.

Schemat a) przedstawia sytuację, gdy podproblemy są od siebie względnie niezależne, natomiast rysunek b) – sytuację, w której rozwiązania poszczególnych podproblemów zależą od rozwiązań wcześniejszych.

AUTOMATYCZNA PARAMETRIZACJA BAZ WIEDZY

WPROWADZENIE

Bardzo istotną cechą, wprowadzoną w wersji 2.1 systemu PC-Shell, jest *parametryzacja baz wiedzy*. Właściwość ta ułatwia inżynierowi wiedzy tworzenie baz wiedzy bez określania z góry lub znajomości niektórych wartości w warunkach reguł. Niezwykle ważną cechą przyjętego w systemie PC-Shell rozwiązania jest możliwość **dynamicznej zmiany wartości** określonych parametrów w trakcie pracy systemu. Z punktu widzenia inżyniera wiedzy istotne jest to, że dynamiczna zmiana pewnych parametrów bazy wiedzy może odbywać się automatycznie, **bez konieczności bezpośredniej ingerencji** ze strony użytkownika w teksty źródłowe baz wiedzy.

Parametryzacja w systemie PC-Shell jest mechanizmem uniwersalnym i bardzo prostym w użyciu, gdyż ogranicza się ona do wyodrębnienia pewnej liczby tzw. *zmiennych parametrycznych*, które mogą wystąpić w bloku reguł, a równocześnie są normalnymi zmiennymi występującymi w bloku *control*. Należy tutaj zauważyć, że zmienne parametryczne w odróżnieniu od zmiennych tymczasowych w regułach mają zawsze określoną wartość i w bloku reguł nie mogą jej zmieniać. Zmienne parametryczne odróżniają się od zmiennych tymczasowych wystąpieniem przed ich nazwą znaku # (hash) oraz tym, że muszą one być zadeklarowane w bloku faset.

Dobrym przykładem zastosowania parametryzacji mogą być bazy wiedzy, w których sprawdzane są pewne wskaźniki (np. finansowe) względem pewnych wartości progowych lub przedziałów. Jednocześnie niektóre wartości progowe mogą być zmienne zależnie od pewnego kontekstu, np. przedmiotu badania. Dla przykładu, inaczej należy oceniać pewne wskaźniki finansowe dla stoczni czy kopalni, a inaczej dla hurtowni lub firmy handlowej. Należy podkreślić, że tego typu problemy mogą czasami powstawać podczas budowy baz wiedzy w innych dziedzinach, np. technice. Mechanizm parametryzacji ułatwia również ocenę konkretnego wskaźnika względem podobnego dla branży (średnie w branży), które często muszą być zewnętrzne w stosunku do bazy wiedzy, czasami mogą być pozyskiwane z wyspecjalizowanych baz danych, zawierających dane statystyczne dotyczące branż.

Innym powodem może być konieczność parametryzacji bazy wiedzy dla różnych użytkowników tej samej aplikacji SE. Jeden z użytkowników będzie chciał zaostrożonych kryteriów oceny, podczas gdy inny będzie stosował łagodniejsze. Należy podkreślić, że prezentowana metoda parametryzacji umożliwia przeprowadzenie jej przez użytkownika końcowego, niekoniecznie inżyniera wiedzy.

DEKLAROWANIE ZMIENNYCH PARAMETRYCZNYCH

Deklaracja zmiennych parametrycznych w bloku faset wygląda następująco:

```
identyfikator_atrybutu:
...
  param {      zmienna_param_1 = wartość_domyślna_1, ... ,
              zmienna_param_n = wartość_domyślna_n }
...;
```

gdzie: wartość_domyślna_i – liczba lub łańcuch znakowy.

Przykład:

```
facets
  wskaźnik_X:
    ...
    param { MIN = 1.5, MAX = 2.3 };
  koszty:
    param { Minimum_kosztów = 0.4, Maximum_kosztów = 0.7 };
  wskaźnik_Y:
    val oneof { "ujemny", "zerowy", "dodatni" }
    param { PARAM_1 = "dodatni", PARAM_2 = "ujemny" };
end;
```

Faseta *param* zawiera listę zmiennych parametrycznych wraz z ich wartościami domyślnymi (podanie wartości domyślnej jest obowiązkowe). Wartość domyślna decyduje o typie zmiennej parametrycznej – czy jest ona łańcuchem znakowym (zmienną typu *char*) czy liczbą (zmienną typu *float*). Należy zaznaczyć, że w bloku reguł tzw. zmienne tymczasowe nie wymagają deklaracji i mogą przyjmować jedynie wartości typu *char* lub *float*. Dany atrybut może zawierać dowolną ilość zmiennych parametrycznych. Składnia nazwy zmiennej parametrycznej jest identyczna ze składnią zmiennych w bloku *control*. Nazwa zmiennej musi być oczywiście unikatowa w skali bazy wiedzy.

Jeżeli dany atrybut jest opisany fasetą *val*, to ta faseta musi poprzedzać fasetę *param*, a wartości domyślne zmiennych parametrycznych muszą być zgodne z typem wartości występujących w liście fasety *val*.

ZMIENNE PARAMETRYCZNE W BLOKU REGUŁ

Użycie zmiennych parametrycznych jest dozwolone wewnątrz warunków reguł i wymaga, aby zmienna taka została wyróżniona wystąpieniem znaku „#”. Opuszczenie znaku „#” powoduje, że podana dalej zmienna jest traktowana jako zmienna tymczasowa danej reguły. O ile dozwolone jest w regułach przypisanie zmiennej parametrycznej do zmiennej zwykłej (tymczasowej), to sytuacja odwrotna nie jest dozwolona. Wynika to z natury zmiennych parametrycznych (w gruncie rzeczy podczas konsultacji są one traktowane przez system bardziej jako stałe niż zmienne).

Poniższy przykład prezentuje różne sposoby użycia zmiennych parametrycznych w regułach.

Przykład:

```
1: atr1 = "c" if
    atr3 = X,
    Y := sin( #MIN4 ) + #MIN4,
    Z := sin( #MAX4 ) / sin( #MIN4 ),
    Z := cos( #MAX4 ),
    X >= Y,
    X <= #MAX4;

2: atr2 = "a" if
    atr3 = X,
    X >= #MIN4,
    X <= #MAX4;

234: poziom_kosztów = "w normie" if
    koszty >= #Minimum_kosztów &
    koszty <= #Maximum_kosztów;
```

ZMIENNE PARAMETRYCZNE W BLOKU STEROWANIA

Zmienne parametryczne wewnątrz bloku *control* traktowane są tak jak zwykłe zmienne zadeklarowane w tym bloku. Zmiennych parametrycznych nie deklaruje się w bloku *control* (wyłącznie w bloku faset). Ich typ jest zgodny z typem wartości domyślnej, zmienna taka jest wtedy typu *char* lub typu *float*. Jakakolwiek zmiana wartości zmiennej parametrycznej w bloku *control* powoduje, że podczas uaktywnienia (w procesie wnioskowania) warunku ze zmienną parametryczną, system potraktuje tę zmienną jak stałą o wcześniej przypisanej wartości. Jednocześnie, o ile w bloku faset, w odniesieniu do atrybutu, dla którego zdefiniowano zmienną parametryczną, wystąpiła faseta *val*, to wartość przypisana (w bloku sterowania) do zmiennej parametrycznej musi być jedną z zadeklarowanych w tej fascie.

KATEGORIE PARAMETRÓW

W systemie PC-Shell wprowadzono pojęcie kategorii zmiennych parametrycznych na oznaczenie różnych możliwych kontekstów ich użycia. W pewnych przypadkach ten sam zbiór zmiennych parametrycznych, w różnych dających się „z góry” przewidzieć kontekstach, powinien mieć różne wartości.

Dla przykładu, sytuacja taka może wystąpić w systemie ekspertowym do analizy wniosków kredytowych. Zakłada się, że pewne grupy (branże) klientów banków będą wymagały nieco innych zasad oceny. Może to dotyczyć w szczególności wartości progowych dla określonych wskaźników. Oczywiście w takiej sytuacji można utworzyć dodatkowe zbiory reguł lub warunków kwalifikujących klienta do określonej grupy

i oceniających go według odmiennych zasad. Jednakże takie rozwiązanie zmniejsza czytelność bazy wiedzy, zwiększa liczbę reguł, tym samym obciążając system. Dodatkowo pojawienie się lub wyodrębnienie nowej grupy klientów banku będzie wymagało ingerencji w tekst źródłowy bazy wiedzy, co rodzi pewne komplikacje związane z utrzymywaniem ważności baz wiedzy.

Jak już wspomniano, w systemie PC-Shell problem ten rozwiązano poprzez wprowadzenie kategorii zmiennych parametrycznych. Mechanizm ten umożliwia zdefiniowanie zbioru kategorii wraz z przypisanymi do każdej z nich zmiennymi parametrycznymi. Każda z tych zmiennych może mieć przypisane odpowiednie wartości (w szczególności inne od wartości domyślnych z bloku faset). Informacje te są pamiętane w pliku «.ini», który jest tworzony dla każdej aplikacji systemu PC-Shell. Kategorie i tym samym wartości zmiennych parametrycznych mogą być zmieniane programowo (patrz punkt następny) lub za pomocą opcji *NarzędziaParametryzacja* (patrz: „Podręcznik użytkownika”). Drugi z wymienionych sposobów jest tak prosty, że do parametryzacji bazy wiedzy, która bywa nieodłącznym elementem niektórych wdrożeń systemów ekspertowych, nie jest potrzebny inżynier wiedzy ani informatyk – może tego dokonać upoważniony użytkownik końcowy. W ten sposób, kontynuując przykład, bank po otrzymaniu systemu ekspertowego jest w stanie samodzielnie parametryzować bazy wiedzy, co w niektórych zastosowaniach można również rozumieć jako ich „strojenie”, zgodnie z własną polityką kredytową i kryteriami oceny.

Kategorie mogą być wykluczające się oraz nakładające się na siebie co oznacza możliwość istnienia dwóch kategorii zawierających wewnątrz siebie różne lub takie same zmienne. System PC-Shell zapamiętuje kategorie w pliku *ini* o nazwie identycznej z nazwą głównej bazy wiedzy.

Poniższy przykład dotyczy sytuacji, w której wyodrębniono trzy grupy klientów, ocenianych (w każdym przypadku) za pomocą przynajmniej trzech wskaźników odnoszonych do trzech wymienionych parametrów. Pominęto tu zaznaczenie związków parametrów z określonymi atrybutami.

Przykład:

Kategorie	Zmienne parametryczne	Wartości
Branża1	PAR1	1, 4
	PAR2	-5
	PAR3	2, 3
Branża2	PAR1	1, 6
	PAR2	-5
	PAR3	2, 1
Branża3	PAR1	1, 4
	PAR2	-8
	PAR3	1, 9

ZMIANA WARTOŚCI ZMIENNYCH PARAMETRYCZNYCH

O ile użytkownik aplikacji lub program w bloku *control*, zbudowany przez inżyniera wiedzy, nie zmienia wartości parametrów, to system przyjmuje automatycznie wartości domyślne zadeklarowane w bloku faset. Z reguły jednak, jeśli do bazy wiedzy wprowadzono zmienne parametryczne, to zakłada się, że w pewnych sytuacjach ich wartości będą zmieniane. Zmiana tych wartości odbywa się automatycznie i ogólnie rzecz ujmując następuje w sposób interakcyjny lub programowy.

Oto podstawowe sposoby zmiany parametrów:

1. interakcyjne przypisanie wartości określonych w pliku konfiguracyjnym;
2. przypisanie wartości do wybranych zmiennych w bloku *control* za pomocą operacji przypisania;
3. programowe przypisanie wartości określonych w pliku konfiguracyjnym;
4. programowe przywrócenie wartości domyślnych w bazie wiedzy.

Ad.1.

Metoda ta polega na wykorzystaniu okna dialogowego do parametryzacji wywoływanego za pomocą opcji *NarzędziaParametryzacja*. Narzędzie to pozwala użytkownikowi na dynamiczną zmianę wartości zmiennych parametrycznych w czasie interaktywnej pracy z systemem PC-Shell. Inżynier wiedzy lub użytkownik może

bez uruchamiania bloku sterującego zmienić wartości zmiennych i np. uruchomić wnioskowanie poprzez menu systemu PC-Shell. Metoda ta jest szczególnie przydatna użytkownikom końcowym aplikacji, pozwalając na dokonywanie zmian bez ingerencji w kod źródłowy bazy wiedzy. Inżynierowi wiedzy to narzędzie może być użyteczne na etapie testowania i „dostrajania” bazy wiedzy. Więcej na temat tego narzędzia można znaleźć w „Podręczniku użytkownika”.

Ad. 2.

Przypisanie wartości do zmiennej w bloku sterowania jest najprostszą metodą zmiany wartości zmiennej parametrycznej. Umożliwia ona dowolne wykorzystanie zmiennej tak jak normalnej zmiennej, umożliwia np. wykonanie obliczeń czy zbudowanie łańcucha wartości w sposób dynamiczny.

Przykład:

```
// W tym przykładzie następuje rozszerzenie
// dopuszczalnego przedziału, ustalonego przez wartości
// domyślne, z (0.4 .. 0.7) do (0.3 .. 0.8), za pomocą
// operacji przypisania w bloku sterowania
knowledge base przykład
  facets
    koszty:
      param { Min_kosztów = 0.4, Maks_kosztów = 0.7 };
  end;
  rules
    234: poziom_kosztów = "w normie" if
      koszty >= #Min_kosztów &
      koszty <= #Maks_kosztów;
  end;
  control
    Min_kosztów := 0.3;
    Maks_kosztów := 0.8;
end;
```

Ad. 3.

Programowe przypisanie zmiennym parametrycznym wartości z pliku konfiguracyjnego jest związane z użyciem mechanizmu kategoryzacji zmiennych parametrycznych.

Odczytanie wartości kategorii z pliku *ini* następuje automatycznie po użyciu instrukcji *changeCategory*, gdzie jako parametr podajemy nazwę kategorii do uaktywnienia. Działanie tej instrukcji sprowadza się do przypisania określonym w danej kategorii zmiennym, określonych wartości. Możliwe jest uaktywnienie po sobie dwu i większej ilości kategorii, w przypadku wystąpienia w dwu kategoriach tych samych zmiennych, wtedy aktualna jest wartość zmiennej z ostatnio uaktywnionej kategorii.

Przykład:

```
// Ten przykład pokazuje schemat jednego z możliwych
// sposobów programowej, dynamicznej parametryzacji,
// nawiązując do omawianego wcześniej problemu decyzji
// kredytowych.
// Z punktu widzenia użytkownika końcowego zmiana parametrów
// odbywa się niejawnie poprzez wybór określonej branży.
knowledge base przykład
  facets
    wskaźnik_1:
      param { MIN_Wsk1 = 1.2, MAX_Wsk2 = 1.4 };
    wskaźnik_2:
      param { MIN_Wsk2 = -2, MAX_Wsk2 = 2 };
    wskaźnik_n:
      param { PAR_Wskn = "ujemny", PAR_Wskn = "dodatni" };
  end;
  control
    menu "Wybierz klasę klienta"
    1. "Branża 1"
    2. "Branża 2"
    3. "Branża n"
    case 1:
      changeCategory( "Branża 1" );
    case 2:
      changeCategory( "Branża 2" );
    case n:
      changeCategory( "Branża n" );
  end;
end;
```

Ad. 4.

Uzupełnieniem działania instrukcji *changeCategory* jest możliwość globalnego przywrócenia wszystkim zmiennym parametrycznym wartości domyślnych, co jest możliwe po wywołaniu tej instrukcji z parametrem jako pusty łańcuch lub jako łańcuch `_`. W systemie PC-Shell istnieje także specjalnie do tego celu przeznaczona instrukcja *setDefParamValue*, która przypisuje podanej zmiennej parametrycznej wartość domyślną.

STRUKTURA OPISU KATEGORII W PLIKU INICJALIZACYJNYM

Kategorie parametrów dla danej bazy wiedzy są pamiętane w pliku *ini* o nazwie takiej, jak nazwa bazy wiedzy. W sekcji *[Global]*, pod kluczem *CategoryCount*, zapamiętana jest ilość zdefiniowanych kategorii, natomiast w kluczach o nazwach typu *catX*, gdzie *X* jest numerem kolejnej kategorii, są zapamiętane nazwy kategorii. Nazwa kategorii jest równocześnie nazwą sekcji, w której znajduje się definicja tej kategorii. W ramach sekcji opisującej kategorię są zapamiętane nazwy zmiennych parametrycznych (klucz) wraz z ich wartościami.

Przykład:

```
[ Global ]
CategoryCount = 3
    // określone są 3 kategorie użytkownika o nazwach:
Cat0 = Hutnictwo
Cat1 = Górnictwo
Cat2 = Handel

[Hutnictwo]          // definicje poszczególnych kategorii
Min = 0.45
Max = 1.2
Wskaźnik_X = 1.45
```

```
[Górnictwo]
Min = 0.5
Max = 1.5
Wskaźnik_X = 1.3

[Handel]
Min = 0.35
Max = 0.8
Wskaźnik_X = 1.7
```

WIZUALIZACJA ZMIENNYCH PARAMETRYCZNYCH

Zmienne parametryczne najczęściej pojawiają się w regułach, w związku z czym, w naturalny sposób stają się elementem prezentowanym przez system podczas wyjaśnień lub przeglądania bazy wiedzy. Przyjęto zasadę, że zmienne parametryczne są prezentowane wraz z bieżącymi wartościami tych zmiennych. Wartości bieżące umieszczane są po zmiennych i ujęte w nawiasy kwadratowe.

Kolejny przykład ukazuje sposób prezentacji w ramach wyjaśnień typu „jak?”. Zmienne parametryczne są prezentowane w taki sam sposób w pozostałych przypadkach.

Przykład:

```
2: atr2 = "a"  JEŚLI
    atr4 = X i
    X >= #MIN4[ 2.00 ] i
    X <= #MAX4[ 4.00 ];
2* Fakt: atr4 = 3.00
```

ZALETY PARAMETRYZACJI BAZ WIEDZY

Korzyści z zastosowania opisywanej metody parametryzacji są następujące:

- możliwość parametryzacji bazy wiedzy zarówno przez inżyniera wiedzy jak i użytkownika końcowego;
- obniżenie kosztów wdrożenia i utrzymywania aplikacji opartej o ten system;
- interakcyjna lub programowa (dynamiczna) zmiana wartości parametrów;
- możliwość lepszego dostosowania aplikacji do specyfiki wybranych klas problemów, dzięki zastosowaniu pojęcia *kategorii* zmiennych parametrycznych;
- zmiana wartości wybranych grup parametrów, ujętych w ramach *kategorii*, zarówno programowo, jak i interakcyjnie;
- zwiększenie czytelności oraz zmniejszenie rozmiaru bazy wiedzy niektórych aplikacji SE dzięki wykorzystaniu dostępnych narzędzi parametryzacji.

ARKUSZE KALKULACYJNE

ARKUSZE KALKULACYJNE I ICH STOSOWANIE

Język Sphinx, począwszy od wersji 2.2, został wyposażony w instrukcje umożliwiające wykorzystanie w aplikacjach arkuszy kalkulacyjnych, zgodnych między innymi z arkuszami programu MS-Excel. Można więc z poziomu aplikacji systemu PC-Shell otworzyć okno zawierające arkusz kalkulacyjny, a następnie wykorzystać możliwości arkuszy do gromadzenia danych, ich przeliczania i zapisywania w postaci plików. Język Sphinx jest wyposażony w instrukcje dostępu do pojedynczych komórek, zakresów, a także całych arkuszy. Użytkownik w trakcie pracy z arkuszem może operować na pojedynczych komórkach, wpisywać formuły obliczeniowe, formatować zawartość komórek oraz ustawiać dodatkowe atrybuty takie jak na przykład, ochrona arkusza.

NAJWAŻNIEJSZE INFORMACJE O ARKUSZU KALKULACYJNYM

Poniżej przedstawiono wycinek wiedzy dotyczący obsługi arkusza. Pełniejszych informacji można zasięgnąć w podręcznikach dotyczących obsługi arkuszy kalkulacyjnych, a w szczególności, w podręcznikach opisujących arkusz kalkulacyjny Microsoft Excel.

Klawisze stosowane w czasie interaktywnej pracy:

Enter – zatwierdzenie zawartości komórki;

Tab – przejście do następnej komórki;

F2 – przejście w tryb edycyjny;

F9 – natychmiastowe przeliczenie arkusza;

Del – wyczyszczenie zawartości komórek;

Esc – zaniechanie edycji;

Shift + dowolny klawisz ruchu – rozszerzenie selekcji;

Ctrl + dowolny klawisz ruchu – przeskok na krawędź arkusza (na przykład naciśnięcie *Ctrl* i \Rightarrow spowoduje przeskok kursora do lewego końca arkusza w aktualnym wierszu).

Obsługa arkusza za pomocą myszy:

lewy przycisk – wskazanie aktywnej komórki;

prawy przycisk – przywołanie podręcznego menu;

lewy przycisk wraz z przesunięciem kursora – zaznaczenie komórek;

Ctrl + lewy przycisk wraz z przesunięciem kursora – zaznaczenie nowej sekcji komórek;

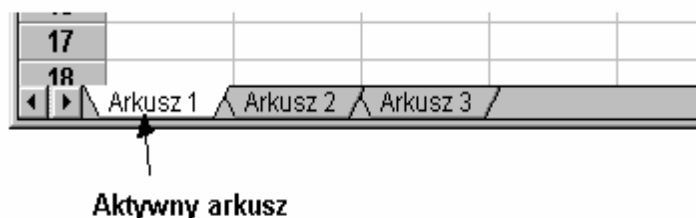
lewy przycisk, gdy kursor znajduje się na krawędzi zaznaczonej sekcji wraz z przesunięciem kursora – przeniesienie zaznaczonych komórek w wybrane miejsce;

Ctrl + lewy przycisk, gdy kursor znajduje się na krawędzi zaznaczonej sekcji wraz z przesunięciem kursora – skopiowanie zaznaczonych komórek w wybrane miejsce.

Dwukrotne naciśnięcie lewego klawisza na krawędzi między dwoma nagłówkami kolumn – dopasowanie szerokości kolumny do komórki zawierającej najszerszy napis (liczbę).

ARKUSZE KALKULACYJNE – POJĘCIA PODSTAWOWE

Podstawowym pojęciem związanym z arkuszami kalkulacyjnymi jest pojęcie *skoroszytu*. Skoroszyt to plik, który można otworzyć w pojedynczym oknie. Skoroszyt początkowo może nie być skojarzony z żadnym plikiem. W ramach skoroszytu zawarty może być jeden lub kilka *arkuszy*, w których przechowywane są dane. Najczęściej poszczególne arkusze zawierają dane powiązane ze sobą. Nazwy arkuszy w ramach pojedynczego skoroszytu wyświetlone są na dole okna w postaci listy (Rys. 15).



Aktywny arkusz

RYS. 15. AKTYWNE ARKUSZE PRZECHOWYWANE W SKOROSZYCIE

Każdy z arkuszy stanowi zbiór *komórek* pogrupowanych w *rzędy* (maksymalnie 16384) i *kolumny* (maksymalnie 256).

Komórki mogą zawierać stałe, formuły obliczeniowe, a także wyrażenia zbudowane w oparciu o stałe i formuły obliczeniowe oraz wykorzystujące dostępne operatory (patrz dalej).

STAŁE

Do stałych zalicza się:

- liczby, składające się z cyfr od 0 do 9 oraz znaków specjalnych takich jak „+”, „-”, „/”, „*”, „(”, „)”, „E”, „e”, „%”, „\$”;
- datę/czas, na przykład „28/09/97”, „17-Mar”, „21:55”;
- nazwy błędów takie jak „#N/A”, „#VALUE!”, „#REF!”, „#NULL!”, „#DIV/0!”, „#NUM!”, „#NAME?”;
- tekst złożony z dowolnego ciągu znaków.

FORMUŁY OBLICZENIOWE

Formuły obliczeniowe to ciągi znaków poprzedzone znakiem równości:

- odwołania do innych komórek, np. „=A1” (adresowanie względne), „=\$A1” (bezwzględny adres kolumny), „=A\$1” (bezwzględny adres wiersza), „=\$A\$1” (adresowanie bezwzględne), „=Dane1!A1” (komórka A1 z arkusza Dane1);
- nazwy, na przykład „=Wiek”, „=Miesiąc”, „=Zadłużenie”;
- funkcje, na przykład „=SIN(A1)”, „=PI()”.

OPERATORY

Wśród operatorów wykorzystywanych do budowania wyrażeń w arkuszach kalkulacyjnych znajdują się:

- operatory arytmetyczne:
 - + dodawanie,
 - odejmowanie,
 - / dzielenie,
 - * mnożenie,
 - % procent,
 - ^ potęgowanie;
- operatory tekstowe:
 - & połączenie;
- operatory relacji:
 - = równe,
 - <> różne od,
 - > większe niż,
 - < mniejsze niż,
 - >= większe lub równe,
 - <= mniejsze lub równe;

- operatory adresowania:
 - : zakres (na przykład „A1:C10” to obszar prostokątny o lewym, górnym narożniku w *A1* oraz prawym, dolnym narożniku w *C10*),
 - spacja* – część wspólna dwóch obszarów (na przykład „A1:C10 C10:F23” daje w wyniku pojedynczą komórkę *C10*),
 - , połączenie dwóch obszarów (na przykład „A1:C10,C10:F23”).

TRYBY ADRESOWANIA

W arkuszach kalkulacyjnych wyróżnia się następujące tryby adresowania:

- adresacja względna (np. „A1”) – odwołanie do komórki, której położenie określa się względem komórki bieżącej; jeśli komórka zawiera adres względny, to po jej skopiowaniu lub przesunięciu nastąpi automatyczne dopasowanie adresu tak, aby względne przesunięcie było identyczne do tego sprzed operacji;
- adresacja bezwzględna (np. „\$A\$1”) – odwołanie do komórki, której położenie jest określone dokładnie; jeśli komórka zawiera adres bezwzględny, to po jej skopiowaniu lub przesunięciu adres nie ulegnie zmianie (adresacja bezwzględna oznaczana jest znakiem \$ przed kolumną lub wierszem, do których następuje odwołanie);
- adresacja mieszana (np. „\$A1”, „A\$1”).

Aby odwołać się do komórek arkusza innego niż bieżący należy skorzystać z operatora adresowania zewnętrznego „!” (np. „Dane!A\$1”, „Zestawienie!A1”, „Raport!\$A\$1”).

NAZWY

W celu łatwiejszego i bardziej przejrzystego posługiwania się komórkami wprowadzono możliwość definiowania nazw (etykiet) wskazanych komórek, zakresów oraz stałych.

Przykładowo, zamiast posługiwać się formułą obliczeniową „=A1-F13”, można komórce *A1* przypisać nazwę *Przychód*, komórce *F13* nazwę *Rozchód*, a powyższą formułę zapisać jako „=Przychód-Rozchód” (zapis stanie się w ten sposób bardziej czytelny).

FUNKCJE

Arkusze kalkulacyjne dysponują dużym zbiorem predefiniowanych funkcji z takich dziedzin jak finanse, matematyka, statystyka i inne. Parametrami funkcji mogą być zarówno stałe jak i adresy komórek zawierających prawidłowe wartości. Dopuszczalne jest również stosowanie wywołań funkcji w miejsce parametrów. Warunkiem jest, aby funkcja podstawiana jako parametr zwracała dla niego poprawną wartość.

PROGRAMOWA OBSŁUGA ARKUSZY W JĘZYKU SPHINX

Język Sphinx dysponuje mechanizmami programowej obsługi arkuszy kalkulacyjnych oferując takie możliwości jak odczytywanie i zapisywanie skoroszytów, operowanie na pojedynczych komórkach lub grupach komórek, wyświetlanie arkuszy na ekranie (tworzenie tzw. widoków).

ROZPOCZĘCIE PRACY ZE SKOROSZYTEM

Każdy skoroszyt jest identyfikowany przez łańcuch znakowy, stanowiący nazwę skoroszytu (na przykład "bilans"). Aby móc skorzystać ze skoroszytu, należy najpierw powiązać jego nazwę (identyfikator) z odpowiednim plikiem wzorca za pomocą instrukcji *openSheet*, której parametrami są: łańcuch znakowy lub zmienna typu *char*, zawierająca nazwę-identyfikator skoroszytu oraz łańcuch znakowy (lub zmienna) określający nazwę pliku wzorca skoroszytu. Plik wzorca powinien zawierać definicję arkusza (tylko format, bez uwzględniania danych).

Przykładowo, wykonanie instrukcji

```
openSheet( "Bilans", "bilans.vts" );
```

spowoduje otwarcie i skojarzenie skoroszytu o identyfikatorze *Bilans* z plikiem wzorca *bilans.vts*.

Identyczny rezultat otrzymamy po wykonaniu poniższego fragmentu kodu:

```
char Nazwa;  
Nazwa := "Bilans";  
openSheet( Nazwa, "bilans.vts" );
```

WCZYTANIE DANYCH DO ARKUSZA

W pliku wzorca przechowywane są informacje dotyczące formatu komórek arkusza, nie służy on natomiast do przechowywania zawartych w arkuszu danych.

Istnieje jednak możliwość wczytania danych, zapisanych wcześniej w pliku, do już otwartych arkuszy. Zadanie to realizuje instrukcja *readSheet*, przy wywołaniu której należy podać identyfikator skoroszytu oraz nazwę pliku zawierającego potrzebne dane.

Tak więc, po wywołaniu instrukcji

```
readSheet( "Bilans", "bil.vts" );
```

arkusz o identyfikatorze *Bilans* wypełniony zostanie danymi zapisanymi w pliku *bil.vts*.

WYŚWIETLENIE ARKUSZA NA EKRANIE

Łaadowanie skoroszytu powoduje utworzenie nowego obiektu w pamięci, nie staje się on jednak automatycznie widoczny na ekranie. Wizualizacji arkusza dokonać można za pomocą instrukcji *showSheet*, wymagającej podania identyfikatora skoroszytu oraz określenia, czy ma być dozwolona edycja zawartości arkusza (wartość *1*), czy nie (wartość *0*). W pierwszym przypadku po wyświetleniu arkusza podręczne menu (wywoływane naciśnięciem prawego przycisku myszy) dostępne będzie w formie rozszerzonej.

Przykładowo, wywołanie instrukcji

```
showSheet( "Bilans", 1 );
```

otworzy na ekranie widok arkusza *Bilans* w trybie do edycji.

ZAMYKANIE WIDOKU ARKUSZA

Aby zamknąć widok arkusza z poziomu aplikacji należy wywołać instrukcję *closeWindow*, podając jako argument identyfikator skoroszytu. W odróżnieniu od instrukcji *closeSheet*, instrukcja *closeWindow* powoduje jedynie zamknięcie okna widoku, sam skoroszyt w dalszym ciągu pozostaje w pamięci operacyjnej. Możliwe jest więc wielokrotne otwieranie i zamykanie widoku arkusza bez konieczności każdorazowego ładowania skoroszytu z pliku.

ZAPAMIĘTANIE SKOROSZYTU

W trakcie działania aplikacji skoroszyt oraz zawarte w nim dane przechowywane są w pamięci operacyjnej. Aby zapisać zawartość skoroszytu na dysku należy posłużyć się instrukcją *writeSheet*, której parametrami są: identyfikator skoroszytu oraz nazwa pliku, w którym skoroszyt ma być zapisany.

Na przykład, po wykonaniu instrukcji

```
writeSheet( "Bilans", "nowy.vts" );
```

aktualna postać skoroszytu *Bilans* zostanie zapamiętana w pliku *nowy.vts*.

Oprócz zapisania skoroszytu w pliku binarnym (*.vts) istnieje możliwość jego eksportu do pliku tekstowego (*.txt), hipertekstowego (*.htm lub *.html) lub do pliku w formacie programu MS-Excel (*.xls). Format pliku docelowego uzależniony jest od podanego rozszerzenia nazwy pliku.

ZAMKNIĘCIE SKOROSZYTU

Wybrany skoroszyt można zamknąć (i jednocześnie usunąć z pamięci) instrukcją *closeSheet* podając jako parametr jego identyfikator. Przed zamknięciem skoroszytu zamknięty zostaje, o ile jest aktywny, jego widok.

Przykładowo, wywołanie instrukcji

```
closeSheet( "Bilans" );
```

spowoduje zamknięcie skoroszytu "Bilans" oraz jego ewentualnego widoku.

Skoroszyt jest zamykany automatycznie w chwili zakończenia aplikacji.

ODWOŁANIA DO KOMÓREK I ZAKRESÓW

Język Sphinx obejmuje zbiór instrukcji umożliwiających pobieranie lub zapisywanie wartości do pojedynczych komórek lub ich zakresów.

Do odczytu wartości zakresu komórek służy instrukcja *getSheetRange*, przy wywołaniu której, oprócz podania identyfikatorów skoroszytu oraz arkusza, należy określić współrzędne początku i końcażądanego zakresu danych, a także wskazać zmienną tablicową (wektor) dowolnego typu prostego, do której mają zostać wpisane wartości danych (tablica wypełniana jest wierszami). Ważne jest, by rozmiar wskazanej tablicy był wystarczający do pomieszczenia wszystkich danych z podanego zakresu.

Wypełnianie zakresu komórek danymi jest możliwe poprzez wywołanie instrukcji *setSheetRange*, której parametry są identyczne jak w przypadku polecenia *getSheetValue* (tym razem jednak tablica stanowi źródło danych). Ważne jest, by ilość danych w tablicy była wystarczająca do wypełnienia całego zakresu komórek arkusza.

Ilustrację działania omówionych instrukcji może stanowić poniższy krótki fragment kodu, wykonanie którego spowoduje przepisanie wartości z kolumny 2 do kolumny nr 3:

```
int Wskaźniki[ 10 ];
getSheetRange( "Bilans", "", 1, 2, 7, 2, Wskaźniki );
setSheetRange( "Bilans", "", 1, 3, 7, 3, Wskaźniki );
```

Odczyt wartości pojedynczej komórki arkusza realizuje instrukcja *getSheetValue*. Jej argumenty są analogiczne do instrukcji *getSheetRange*, zamiast nazwy tablicy należy jednak w tym wypadku podać identyfikator zmiennej typu prostego, do której zostanie wpisana wartość odczytanej komórki.

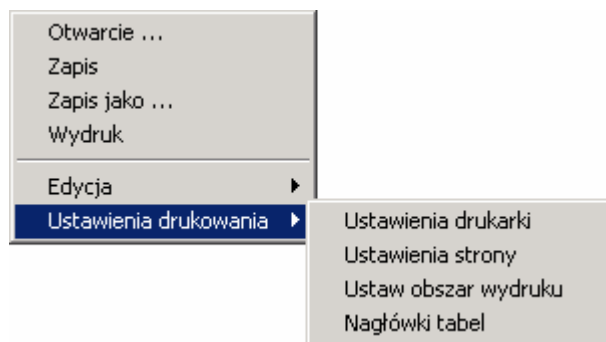
Aby wpisać określoną wartość do pojedynczej komórki arkusza, należy wykorzystać instrukcję *setSheetValue*. Parametry wywołania instrukcji są identyczne jak w przypadku polecenia *getSheetValue*, przy czym zmienna występuje tutaj w charakterze źródła danych.

Poniższy fragment kodu stanowi przykład użycia obu wyżej wymienionych instrukcji w celu skopiowania danych z jednej komórki do innej:

```
char Nazwisko;
getSheetValue( "Bilans", "", 4, 2, Nazwisko );
setSheetValue( "Bilans", "", 1, 1, Nazwisko );
```

PODRĘCZNE MENU ARKUSZA

Naciśnięcie prawego klawisza myszki w obrębie arkusza powoduje otwarcie podręcznego menu arkusza. Menu to może występować w dwóch postaciach, w zależności od wartości drugiego parametru instrukcji *showSheet* (patrz poprzedni rozdział).



RYS. 16. MENU PODRĘCZNE ARKUSZA – POSTAĆ SKRÓCONA



RYS. 17. MENU PODRĘCZNE ARKUSZA – POSTAĆ ROZSZERZONA

Poniżej zamieszczono krótki opis dostępnych opcji menu.

Otwarcie

Wczytanie pliku do skoroszytu.

Dostępne są następujące formaty plików:

- Formuła One (*.vts);
- Excel (*.xls);
- HTML (*.htm, *.html);
- inne (*.*)

Zapis

Zapisanie zawartości skoroszytu do pliku, który jest z nim skojarzony.

Zapis jako

Zapisanie zawartości arkusza do pliku o podanej nazwie. Dostępne formaty plików są takie jak w przypadku opcji *Otwarcie*, przy czym wybór opcji typu *Pliki inne (*.*)* powoduje zapisanie arkusza w formacie *Formuła One*. Format pliku uzależniony jest od rozszerzenia podanego przy nazwie arkusza, w razie jego pominięcia przyjmowane jest rozszerzenie wybrane w polu *Zapisz jako typ*. Aby zapisać arkusz w postaci zwykłego tekstu ASCII, jako rozszerzenie należy podać *.txt*.

Wydruk

Otwarcie okna dialogowego umożliwiającego wydruk zawartości skoroszytu. W oknie dostępne są dodatkowo opcje wyboru drukarki, ustawienia jej właściwości, określenia zakresu wydruku, liczby kopii oraz obejrzenie podglądu wydruku.

Edycja ⇒ Wytnij, Kopiuj, Wklej, Wklej specjalnie ...

Opcje obsługi schowka w zakresie wycianania, kopiowania i wstawiania do i z zaznaczonego obszaru.

Edycja ⇒ Wyczyść ⇒ Wszystko, Zawartość, Formaty

Opcje usuwania zawartości zaznaczonych komórek. Pierwsza opcja czyści zawartość i formaty (ustawienia komórek), druga tylko zawartość komórki pozostawiając niezmienione formatowanie, natomiast ostatnia czyści tylko formaty komórek pozostawiając zawartość.

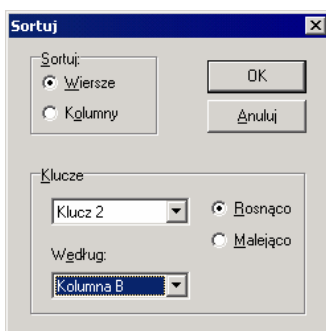
Skok do adresu

Przesunięcie arkusza do podanego celu. Celem może być pojedyncza komórka (np. „C23”), zakres komórek (np. „A3:B7”), nazwa komórki lub cały arkusz.

Sortowanie

Sortowanie wierszy lub kolumn w zaznaczonym zakresie. Jeśli sortujemy wiersze, każdy wiersz w zaznaczonym zakresie stanowi jeden rekord, sortowany w całości (analogicznie – podczas sortowania kolumn pojedynczym rekordem jest kolumna). Sortowanie może odbywać się według maksymalnie trzech kluczy (pół rekordu, czyli komórki wiersza lub kolumny). W pierwszej kolejności porównywane są klucze nr 1, w przypadku ich równości klucze nr 2, na końcu nr 3 (o ile wszystkie są ustawione). Ustawienia kluczy dokonujemy poprzez wybranie numeru klucza w polu *Klucze* (np. „Klucz 2”), a następnie przypisania mu żądanej kolumny (lub wiersza w przypadku sortowania kolumn) w polu *Według* (na przykład „Kolumna E”). Dla każdego klucza możemy określić, czy sortowanie ma być według rosnących, czy też malejących wartości tego klucza.

Na rysunku poniżej przedstawiono przykładowe ustawienia opcji sortowania, w tym przypadku sortowanie wierszami komórek z zakresu A1-C4 według dwóch kluczy: kolumna A (rosnąco) oraz kolumna B (malejąco).



RYS. 18. OKNO USTAWIEŃ OPCJI SORTOWANIA

Na kolejnych dwóch rysunkach widoczna jest postać arkusza przed rozpoczęciem i po zakończeniu operacji sortowania.

PASYWA				
	A	B	C	D
1	sigma	pi	22	
2	delta	gamma	2	
3	alfa	beta	10	
4	delta	epsilon	88	
5				

RYS. 19. POSTAĆ WYJŚCIOWA ARKUSZA

PASYWA				
	A	B	C	D
1	alfa	beta	10	
2	delta	gamma	2	
3	delta	epsilon	88	
4	sigma	pi	22	
5				

RYS. 20. ZAWARTOŚĆ ARKUSZA PO ZAKOŃCZENIU SORTOWANIA

Znajdź

Odszukiwanie komórek według zadanego wzorca. Przeszukiwanie może odbywać się wierszami bądź kolumnami, poszukiwany wzorzec może być interpretowany jako formuła lub wartość komórki (opcja *Przeglądaj*). Podczas poszukiwania może być uwzględniana wielkość liter oraz dokładne dopasowanie wzorca do zawartości komórki. Poszukiwanie może być kontynuowane po naciśnięciu przycisku *Następny*.

Dialog box "Znajdź" z następującymi elementami:

- Pole tekstowe "Znajdź:" z przyciskiem rozwijającym.
- Przycisk "Następny".
- Pole wyboru "Przeszukuj:" z menu rozwijanym (wybrano "Kolumnami").
- Przycisk "Poprzedni".
- Pole wyboru "Przeglądaj:" z menu rozwijanym (wybrano "Formuły").
- Przycisk "Zamknij".
- Przyciski opcjonalne: ☐ Uwzględniaj wielkość liter, ☐ Znajdź tylko całe komórki.

RYS. 21. OKNO USTAWIEŃ OPCJI PRZESZUKIWANIA ARKUSZA

Zastąp

Zastępowanie jednego wzorca innym. Przeszukiwanie może odbywać się wierszami bądź kolumnami. W trakcie przeszukiwania może być uwzględniana wielkość liter oraz dokładne dopasowanie poszukiwanego wzorca do zawartości komórki. Zastąpienie wzorca może być wykonywane pojedynczo dla poszczególnych wystąpień wzorca (przycisk *Zastąp*) lub automatycznie dla wszystkich wystąpień (przycisk *Wszystkie*). Przycisk *Następny* odszukuje następne wystąpienie podanego wzorca.

Dialog box "Zamień" z następującymi elementami:

- Pole tekstowe "Znajdź:" z przyciskiem rozwijającym.
- Przycisk "Następny".
- Pole tekstowe "Zamień na:" z przyciskiem rozwijającym.
- Przycisk "Zamień".
- Pole wyboru "Przeszukuj:" z menu rozwijanym (wybrano "Kolumnami").
- Przycisk "Zamień wszystkie".
- Przyciski opcjonalne: ☐ Uwzględniaj wielkość liter, ☐ Znajdź tylko całe komórki.
- Przycisk "Zamknij".

RYS. 22. OKNO USTAWIEŃ OPCJI ZASTĘPOWANIA WZORCA W ARKUSZU

Ustawienia ⇨ Komórek

Ustawienie właściwości zaznaczonych komórek (patrz tabela poniżej).

Symbol formatu	Opis
<i>General</i>	Wyświetlanie liczb w formacie ogólnym.
0	Znacznik pozycji. Jeśli liczba zawiera mniej cyfr niż jest znaczników w formacie, to zostaje ona uzupełniona zerami. Jeśli na prawo od przecinka jest więcej cyfr niż znaczników, to część ułamkowa zostaje zaokrąglona do tylu miejsc po przecinku, ile jest znaczników w formacie. Nadmiarowe cyfry na lewo od przecinka zostają zachowane.
#	Znacznik pozycji. Funkcjonuje podobnie jak znacznik „0”, z tym, że liczba nie jest uzupełniana zerami jeśli zawiera mniej cyfr niż jest znaczników w formacie.
?	Znacznik pozycji. Działa podobnie jak znacznik „0”, przy czym uzupełnia brakujące pozycje spacjami.
. (kropka)	Przecinek dziesiętny. Ustala liczbę cyfr (zer lub „#”) wyświetlanych po każdej stronie przecinka dziesiętnego. Jeśli format po lewej stronie przecinka zawiera tylko „#”, liczby mniejsze od 1 zaczynają się od przecinka. Jeśli format zawiera „0” na lewo od przecinka, liczby mniejsze od 1 zaczynają się od „0” przed przecinkiem.
%	Wyświetla liczbę jako procent. Liczba jest mnożona przez 100 oraz dopisywany jest znak „%”.
, (przecinek)	Separator tysięcy. Jeśli format zawiera przecinek między „#” lub „0”, liczba jest wyświetlana z podziałem na tysiące. Jeśli po przecinku występuje spacja, to liczba jest skalowana przez 1000. Np. format „0”, skaluje przez 1000 (10 000 zostanie wyświetlone jako 10).
<i>E- E+ e- e+</i>	Wyświetla liczbę w zapisie naukowym. Jeśli format zawiera symbol zapisu naukowego na lewo od „0” lub „#”, to liczba jest wyświetlana w zapisie naukowym z dodatkiem „E” lub „e”. Liczba „0” i „#” na prawo od przecinka dziesiętnego określa liczbę cyfr wykładnika. „E-” i „e-” umieszczają znak minusa przed wykładnikami ujemnymi. „E+” i „e+” umieszczają znak minusa przed

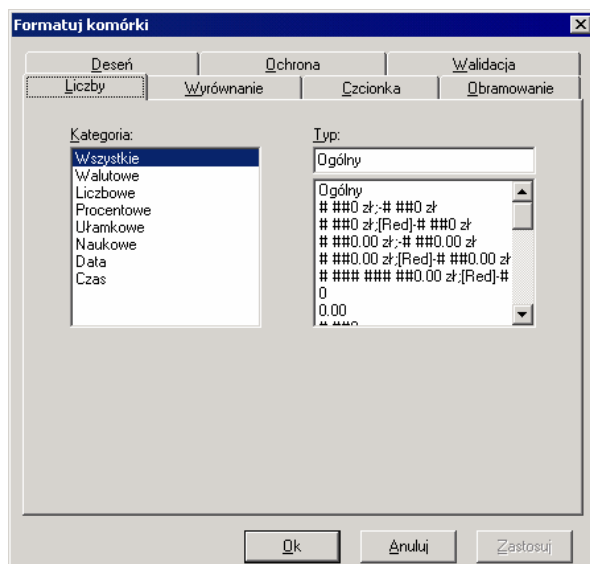
	wykładnikami ujemnymi oraz znak plusa przed wykładnikami dodatnimi.
\$ - + / () : spacja	Wyświetla dany znak. Aby wyświetlić znak inny niż wymienione należy poprzedzić go znakiem „\” lub umieścić w cudzysłowach (\" "). Można również użyć „/” dla formatów ułamkowych.
\	Wyświetla następny znak. Znak „\” nie jest wyświetlany. Znaki lub napisy można również wyświetlić przez umieszczenie ich w cudzysłowach (\" "). Znak „\” jest wstawiany automatycznie dla następujących znaków: „!”, „^”, „&”, „` ” (lewy pojedynczy cudzysłów), „' ” (prawy pojedynczy cudzysłów), „~”, „{”, „}”, „=”, „<” oraz „>”.
* (gwiazdka)	Powtarza następny znak aż do wypełnienia szerokości kolumny. W jednej sekcji formatu może być tylko jedna gwiazdka.
_ (podkreślenie)	Przeskakuje szerokość następnego znaku. Np. aby liczby ujemne otoczone „()” były wyrównane z liczbami dodatnimi, można wstawić format „_)” dla liczb dodatnich, żeby przeskoczyć szerokość nawiasu.
"tekst"	Wyświetla tekst zawarty wewnątrz cudzysłowów.
@	Znacznik pozycji tekstu. Jeśli w komórce jest tekst, zastępuje on znak „@”.
m	Numer miesiąca. Wyświetla miesiące jako liczby bez wiodących „0” (np. „1”-„12”). Może również reprezentować minuty, jeśli użyte z formatami „h” lub „hh”.
mm	Numer miesiąca. Wyświetla miesiące jako liczby z wiodącymi „0” (np. „01”-„12”). Może również reprezentować minuty, jeśli użyte z formatami „h” lub „hh”.
mmm	Skrót nazwy miesiąca. Wyświetla nazwy miesięcy w postaci skrótów (np. „sty”-„gru”).
mmmm	Pełna nazwa miesiąca. Wyświetla pełne nazwy miesięcy (np. „styczeń”-„grudzień”).
d	Numer dnia. Wyświetla dzień jako liczbę bez wiodącego „0” (np. „1”-„2”).
dd	Numer dnia. Wyświetla dzień jako liczbę z wiodącym „0” (np. „01”-„02”).
ddd	Skrót nazwy dnia. Wyświetla nazwy dni w

	postaci skrótów (np. „Pn”-„N”).
dddd	Pełna nazwa dnia. Wyświetla nazwy dni w postaci pełnej (np. „poniedziałek”-„niedziela”).
yy	Rok. Wyświetla rok jako liczbę 2-cyfrową (np. „00”-„99”).
yyyy	Rok. Wyświetla rok jako liczbę 4-cyfrową (np. „1998”).
h	Godzina. Wyświetla godzinę jako liczbę bez wiodącego „0” (np. „0”-„23”). Jeśli format zawiera któryś z formatów „AM/PM”, wykorzystywany jest zegar 12-godzinny, w przeciwnym razie - zegar 24-godzinny.
hh	Godzina. Wyświetla godzinę jako liczbę z wiodącym „0” (np. „01”-„23”). Jeśli format zawiera któryś z formatów „AM/PM”, wykorzystywany jest zegar 12-godzinny, w przeciwnym razie - 24-godzinny.
m	Liczba minut. Wyświetla minuty jako liczbę bez wiodącego „0” (np. „0”-„59”). Format „m” musi wystąpić bezpośrednio po symbolu „h” lub „hh”. W przeciwnym razie jest interpretowany jako numer miesiąca.
mm	Liczba minut. Wyświetla minuty jako liczbę z wiodącym „0” (np. „00”-„59”). Format „m” musi wystąpić bezpośrednio po symbolu „h” lub „hh”. W przeciwnym razie jest interpretowany jako numer miesiąca.
s	Liczba sekund. Wyświetla sekundy jako liczbę bez wiodącego „0” (np. „0”-„59”).
ss	Liczba sekund. Wyświetla sekundy jako liczbę z wiodącym „0” (np. „00”-„59”).
AM/PM A/P am/pm a/p	Zegar 12-godzinny. Wyświetla „AM”, „am”, „A”, lub „a” dla czasu pomiędzy północą a południem oraz „PM”, „pm”, „P” lub „p” dla czasu od południa do północy.
[h]	Całkowita liczba godzin.
[m]	Całkowita liczba minut.
[s]	Całkowita liczba sekund.
s.0 s.00 s.000 ss.0 ss.00 ss.000	Ułamkowa część sekund.
[Black]	Wyświetla tekst komórki w kolorze czarnym.

[Blue]	Wyświetla tekst komórki w kolorze niebieskim.
[Cyan]	Wyświetla tekst komórki w kolorze cyan.
[Green]	Wyświetla tekst komórki w kolorze zielonym.
[Magenta]	Wyświetla tekst komórki w kolorze magenta.
[Red]	Wyświetla tekst komórki w kolorze czerwonym.
[White]	Wyświetla tekst komórki w kolorze białym.
[Yellow]	Wyświetla tekst komórki w kolorze żółtym.
[COLOR n]	Wyświetla tekst komórki w kolorze o podanym numerze (n) w bieżącej palecie kolorów.

Każdy format może składać się z czterech sekcji: opcje dla liczb dodatnich, ujemnych, zer oraz dla tekstu. Używając nawiasów wartości warunkowej („[” i „]”), można wyznaczyć różne ustawienia poszczególnych sekcji, na przykład liczby dodatnie wyświetlać na czarno, ujemne na czerwono, zera na niebiesko (w tym przypadku zapis powinien mieć następującą postać:

```
„[>0] [Black]General; [<0] [Red]General; [Blue]General”).
```



RYS. 23. OKNO USTAWIEŃ FORMATU KOMÓREK

Przykłady wykorzystania opcji formatujących:

Format	Zawartość komórki	Wynik
#,##0.00	32002,00	32 002,00
0.00%	0,32	32,00%
# ??/??	2/26	1/13
0.00E+00	2/26	7,69E-02
ddd-mmm	1900-1-1	pon-sty
h:mm A/P	23:03	11:03 P

Wyrównanie – wyrównanie tekstu w komórce w pionie i w poziomie, możliwe włączenie opcji zawijania tekstu (tekst szerszy od kolumny nie przesłania następnej kolumny lecz jest przenoszony do następnego wiersza).

Czcionka – określenie czcionki, jej stylu i wielkości, efektów typu podkreślenie, przekreślenie, a także koloru.

Obramowanie – rodzaj, kolor, widoczność linii obramowujących komórki (dla grup komórek również linie wewnętrzne).

Deseń – rodzaj tła komórki.

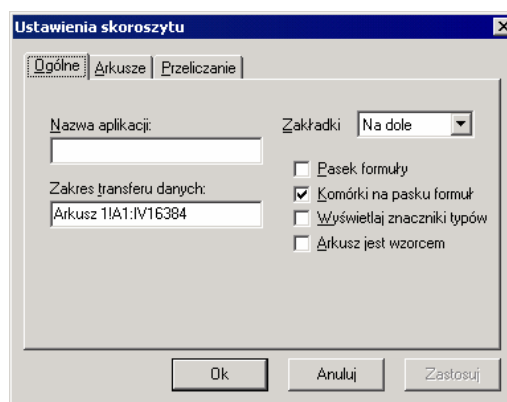
Ochrona – włączenie blokady komórki (uniemożliwienie zmiany zawartości) i/lub ukrycie komórki.

Walidacja – reguła kontrolująca poprawność zawartości komórki.

Ustawienia ⇔ Skoroszytu

Ustawienie właściwości skoroszytu.

Ogólne – nazwa aplikacji, zakres transferu danych, położenie zakładek arkuszy, wyświetlanie paska formuły, adresu aktywnej komórki, stosowanie znaczników typu (komórki o tym samym typie, na przykład zawierające liczby, mają obramowanie w tym samym kolorze), traktowanie arkusza jako wzorca.



RYS. 24. OKNO USTAWIEŃ OPCJI SKOROSZYTU

Arkusze – właściwości arkuszy wchodzących w skład skoroszytu: górna lewa komórka arkusza, aktywna komórka, wybrany zakres, włączenie ochrony.

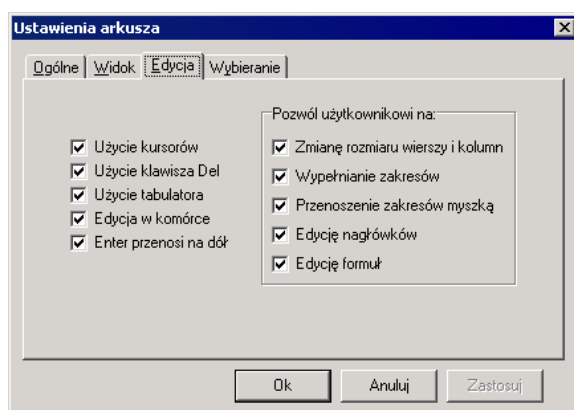
Przeliczanie – włączenie automatycznego przeliczania, włączenie przeliczania iteracyjnego (komórki w arkuszu są przeliczane liczbę razy podaną przez parametr *Maksymalna iteracja* lub aż do chwili, gdy wszystkie komórki nie zmieniają się bardziej niż *Maksymalna zmiana*). Zaznaczenie opcji *Precyzja jak wyświetlana* powoduje przechowywanie wartości z dokładnością podaną w formacie komórki.

Ustawienia ⇔ Arkusza ⇔ Właściwości

Ogólne – określenie nazwy arkusza (nazwę arkusza można również zmienić klikając dwukrotnie myszą na zakładce arkusza).

Widok – określenie widocznej części arkusza (min./maks. wiersz/kolumna), wyświetlanie formuł zamiast ich wartości, wyświetlanie linii siatki, wartości zerowych (jeśli ta pozycja nie jest zaznaczona, komórka jest wyświetlana jako pusta jeśli zawiera wartość zerową), nagłówków wierszy i kolumn, określenie, które suwaki mają być widoczne, wybór skali widoku, włączenie/wyłączenie wyświetlania wybranego zakresu komórek, praca automatyczna.

Edycja – określenie możliwości korzystania z wybranych klawiszy (klawisze kursora, klawisz *Del*, tabulator), ustalenie uprawnień użytkownika (zmiana rozmiaru wierszy i kolumn, wypełnianie zakresów, przenoszenie zakresów myszą, edycja nagłówków, edycja formuły).



RYS. 25. OKNO USTAWIEŃ WŁAŚCIWOŚCI ARKUSZA

Wybieranie – określenie, czy użytkownik może wybierać zakresy, komórki bądź obiekty, włączenie trybu wierszowego – po zaznaczeniu tej opcji wskazanie pojedynczej komórki powoduje zaznaczenie całego wiersza (kolumny, jeśli wskażemy nagłówek kolumny).

Ustawienia ⇒ Arkusza ⇒ Wstaw nowy

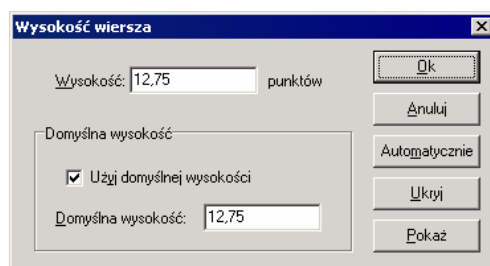
Dodanie nowego arkusza do bieżącego skoroszytu.

Ustawienia ⇒ Arkusza ⇒ Usuń aktywny arkusz

Usunięcie aktywnego arkusza z bieżącego skoroszytu.

Ustawienia ⇒ Wierszy ⇒ Wysokość wiersza

Ustalenie wysokości wiersza. Zaznaczenie opcji *Użyj domyślnej wysokości* spowoduje ustawienie wysokości wiersza zgodnie z wartością określoną w polu *Domyślna wysokość*.



RYS. 26. USTAWIENIA WYSOKOŚCI WIERSCZA

Ustawienia ⇒ Wierszy ⇒ Ukryj

Ukrycie wiersza, w którym znajduje się aktywna komórka (lub kilku wierszy, jeśli zaznaczono większy obszar).

Ustawienia ⇒ Wierszy ⇒ Odkryj

Wyświetlenie wcześniej ukrytych wierszy arkusza.

Ustawienia ⇒ Wierszy ⇒ Domyślna wysokość

Ustalenie wysokości wiersza aktywnej komórki (lub wierszy wchodzących w skład zaznaczonego obszaru) zgodnie z wartością domyślną.

Ustawienia ⇒ Kolumn ⇒ Szerokość kolumny

Ustalenie szerokości kolumny. Zaznaczenie opcji *Użyj domyślnej szerokości* spowoduje ustawienie szerokości kolumny zgodnie z wartością określoną w polu *Domyślna szerokość*.

Ustawienia ⇒ Kolumn ⇒ Autodopasowanie

Ustalenie szerokości kolumny w taki sposób, by uniknąć „obcięć” zawartości jakiejkolwiek komórki danej kolumny lub konieczności jej podzielenia na większą ilość wierszy. Opcję *Autodopasowanie* można również uaktywnić przez dwukrotne kliknięcie na linii rozdzielającej nagłówki kolumn.

Ustawienia ⇒ Kolumn ⇒ Ukryj

Ukrycie kolumny, w której znajduje się aktywna komórka (lub kilku kolumn, jeśli zaznaczono większy obszar).

Ustawienia ⇒ Kolumn ⇒ Odkryj

Wyświetlenie wcześniej ukrytych kolumn arkusza.

Ustawienia ⇒ Kolumn ⇒ Domyślna szerokość

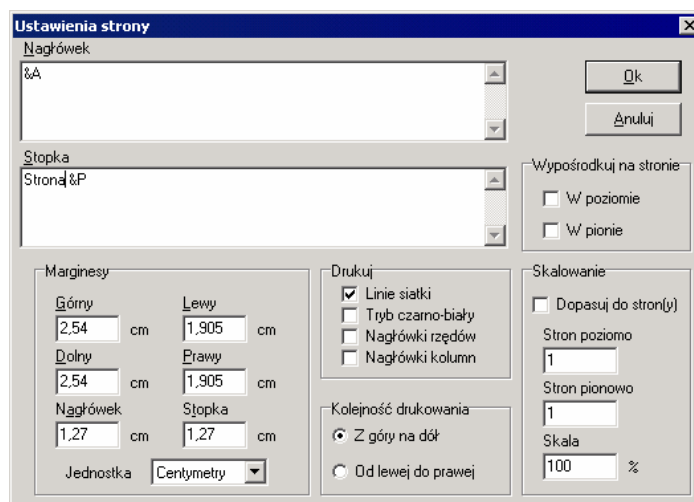
Ustalenie szerokości kolumny aktywnej komórki (lub kolumn wchodzących w skład zaznaczonego obszaru) zgodnie z wartością domyślną.

Ustawienia ⇒ Wydruku ⇒ Ustawienia drukarki

Wywołanie okna dialogowego umożliwiającego wybór drukarki, a także ustawienie jej właściwości, rodzaju papieru, orientacji wydruku.

Ustawienia ⇒ Wydruku ⇒ Ustawienia strony

Określenie formatu wydruku (nagłówki/stopka, rozmiar marginesów, zakres i rodzaj drukowanego obszaru).



RYS. 27. OKNO USTAWIEŃ STRONY

Zestawienie kodów formatujących nagłówki/stopkę zamieszczono w tabeli.

Kod	Opis
&L	Wyrównanie do lewej kolejnych znaków
&C	Wyśrodkowanie kolejnych znaków
&R	Wyrównanie do prawej kolejnych znaków

<code>&D</code>	Wydrukowanie aktualnej daty
<code>&T</code>	Wydrukowanie aktualnego czasu
<code>&F</code>	Wydrukowanie nazwy skoroszytu
<code>&A</code>	Wydrukowanie nazwy arkusza
<code>&P</code>	Wydrukowanie numeru strony
<code>&P+numer</code>	Wydrukowanie numeru strony powiększonego o numer
<code>&P-numer</code>	Wydrukowanie numeru strony zmniejszonego o numer
<code>&&</code>	Wydrukowanie znaku „&”
<code>&N</code>	Wydrukowanie całkowitej liczby stron w dokumencie

Jeśli nie podano „&L” lub „&R” – kody i tekst są wyśrodkowywane.

Kody czcionek (zestawione w tabeli poniżej) muszą wystąpić przed innymi kodami formatującymi – w przeciwnym razie są ignorowane. Wyjątek stanowią kody sposobu wyrównywania („&L”, „&C”, „&R”), oznaczające zarazem początek nowej sekcji wydruku; po kodzie wyrównania dozwolone jest wystąpienie kodu czcionki.

Kod	Opis
<code>&B</code>	Pogrubienie
<code>&I</code>	Kursywa
<code>&U</code>	Nagłówek podkreślony
<code>&S</code>	Nagłówek przekreślony
<code>&O</code>	Ignorowane
<code>&H</code>	Ignorowane
<code>&"czcionka</code>	Krój czcionki
<code>&nn</code>	Rozmiar czcionki (wyrażony liczbą dwucyfrową!)

Opcje sekcji *Skalowanie*, po zaznaczeniu opcji *Dopasuj do stron(y)*, umożliwiają wydrukowanie wybranego zakresu na określonej liczbie stron (w poziomie i w pionie). Ponadto istnieje możliwość określenia pożądanej skali (np. 50% – dwukrotne pomniejszenie).

Sekcja *Kolejność drukowania* umożliwia określenie sposobu dzielenia arkusza na strony – w pionie (strona 2 poniżej strony 1) lub w poziomie (strona 2 obok strony 1).

Ustawienia ⇒ Wydruku ⇒ Ustaw obszar wydruku

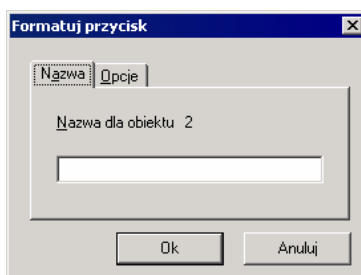
Ustalenie formatu wydruku (nagłówek/stopka, rozmiar marginesów, sposób rozmieszczenia na stronie, skalowanie, kolejność drukowania, dodatkowe opcje wydruku – drukowanie linii siatki, nagłówków wierszy, kolumn, druk czarno-biały).

Ustawienia ⇒ Wydruku ⇒ Nagłówki tabel

Definicja nagłówków tabel pojawiających się na każdej ze stron drukowanego arkusza.

Ustawienia ⇒ Obiektu

Okno definiowania właściwości zaznaczonego obiektu. W zależności od typu obiektu użytkownik może zdefiniować różne właściwości. Dla każdego z obiektów można zdefiniować nazwę (Rys. 27).

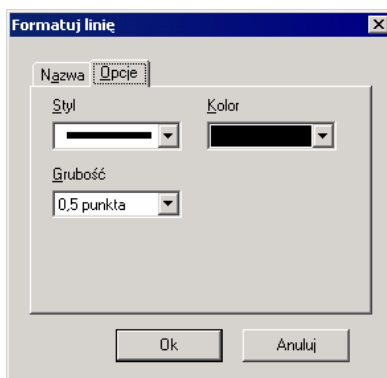


RYS. 27. NAZWA OBIEKTU

Uwaga ! Dla przycisku **musi być zdefiniowana nazwa** aby mógł on być później wiązany za pomocą instrukcji *sheetBindButton*.

Dopuszczalne opcje do poszczególnych obiektów :

Linia



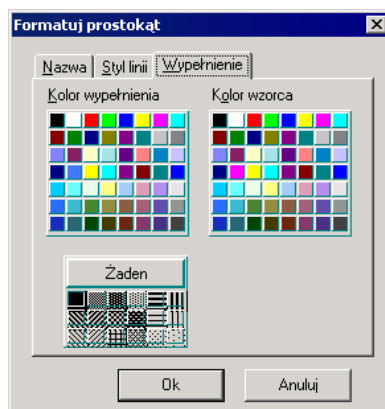
RYS. 28 FORMATOWANIE LINII

Styl – rodzaj linii,

Kolor – kolor linii,

Grubość – grubość linii.

Prostokąt, Elipsa, Wielokąt



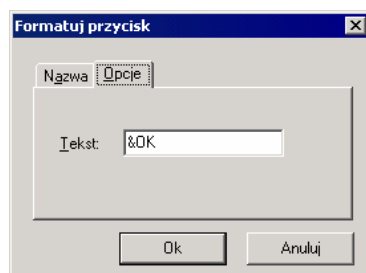
RYS. 29 FORMATOWANIE PROSTOKĄTA

Styl – rodzaj linii,

Kolor – kolor linii,

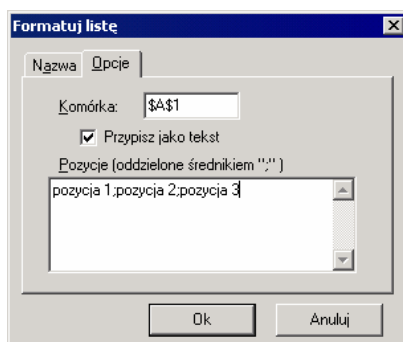
Grubość – grubość linii.

Wypełnienie – rodzaj i kolor wypełnienia figury.

Przycisk

RYS. 30 FORMATOWANIE PRZYCISKU

Tekst - tekst pojawiający na przycisku

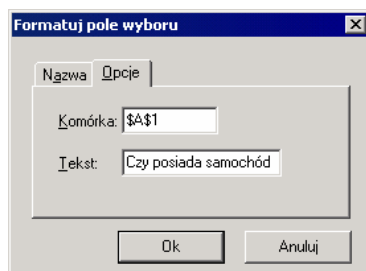
Lista

RYS. 31 FORMATOWANIE LISTY

Komórka – adres powiązanej komórki w której pamiętany jest rezultat wybór wartości z listy,

Przypisz jako tekst – określa czy w komórce jest pamiętany wybór jako tekst czy jako indeks,

Pozycje – lista pozycji na liście oddzielonych od siebie znakiem średnika ‘;’

Pole wyboru

RYS. 32 FORMATOWANIE POLA WYBORU

Komórka – adres powiązanej komórki w której pamiętany jest stan pola wyboru,

Tekst – treść przy polu wyboru.

Wstaw obiekt ⇒ Linia, Prostokąt, Elipsa, Łuk, Wielokąt, Przycisk, Lista, Pole wyboru

Opcje służą do wstawienia odpowiedniego obiektu na arkuszu. Po wybraniu opcji za pomocą myszki ‘rysujemy’ wstawiany obiekt.

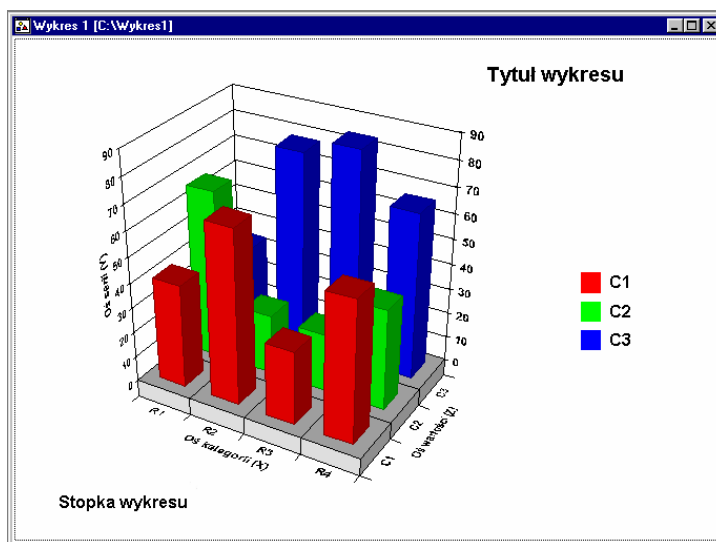
10

WYKRESY

NAJWAŻNIEJSZE INFORMACJE O WYKRESACH

Wraz z wersją 2.3 pakietu Sphinx wprowadzona została możliwość programowej obsługi wykresów.

Pod pojęciem wykresu rozumieć należy okno zawierające graficzną formę reprezentacji danych. Każdy wykres identyfikowany jest poprzez swoją unikalną nazwę, używaną w instrukcjach operujących na wykresach. Nazwa ta stanowi zarazem etykietę okna. Na rysunku poniżej przedstawiono przykładowe okno zawierające wykres z podstawowymi elementami, które mogą się na nim pojawić.



RYS. 28. PRZYKŁADOWE OKNO WYKRESU

Wykres składa się z takich elementów jak:

- pole wykresu,
- tytuł wykresu,
- stopka,
- legenda.

Poszczególne elementy wykresu (za wyjątkiem oczywiście samego wykresu) mogą być widoczne lub nie. Edycja wykresów może być wykonywana na przykład za pomocą bazy wiedzy *wykresy.bw* dostarczanej wraz z pakietem Sphinx. Dla każdego z elementów można zmienić treść informacji, rodzaj, wielkość i kolor czcionki oraz szereg innych parametrów. Część z opcji może być wykorzystana w sposób programowy (z poziomu języka Sphinx), dostęp do pozostałych możliwy jest za pośrednictwem menu podręcznego pojawiającego się po kliknięciu na wykresie prawym przyciskiem myszy. W trybie edycji wykresu pojawia się wtedy opcja *Designer*, która umożliwia wywołanie standardowego okna właściwości wykresu (ponieważ okno właściwości stanowi element biblioteki funkcji obsługujących wykres, nazwy poszczególnych opcji podane są w języku angielskim).

MECHANIZMY PROGRAMOWEJ OBSŁUGI WYKRESÓW

Podobnie jak to ma miejsce w przypadku arkuszy kalkulacyjnych, obsługa wykresów oparta jest na koncepcji wykresu oraz jego widoku. Wykres jest tworzony w pamięci i dopiero potem może być wizualizowany, dzięki czemu użytkownik może widok wykresu zamykać i otwierać bez utraty informacji aktualnie w nim zawartych. Możliwe jest odczytywanie i zapisywanie wykresów w plikach binarnych (definicje wykresów) lub w postaci tzw. map bitowych (w formacie *bmp*, *gif* lub *jpg*) celem późniejszego wykorzystania w innych aplikacjach.

TWORZENIE WYKRESU

Pierwszym krokiem powinno być utworzenie wykresu za pomocą instrukcji *openChart*. Pierwszy z jej parametrów stanowi nazwa wykresu, którą należy się posługiwać przy kolejnych odwołaniach do niego. Nazwa ta jest również częścią nagłówka pojawiającego się podczas wizualizacji wykresu. Musi być ona uni-

kalna, nie można więc utworzyć dwóch różnych wykresów o tej samej nazwie. Drugi parametr instrukcji określa nazwę pliku, w którym znajduje się wzorec początkowej postaci wykresu (szablon). Użytkownik może wcześniej zdefiniować postać, wygląd i wszelkie atrybuty wykresu i zapisać go w postaci pliku binarnego (*.vrc) na dysku, a następnie użyć go jako szablonu. Parametr ten może mieć postać pustego łańcucha tekstowego.

Przykład:

```
openChart( "Wykres", "" );
```

WSTAWIANIE DANYCH

Dane na wykresie mogą być ustawiane na trzy sposoby. Pierwszy z nich opiera się na formatowaniu wykresu za pomocą sekwencji instrukcji *setChartData* (a więc ustawianiu ilości wierszy i kolumn oraz wartości poszczególnych danych).

Przykład:

```
setChartData( "Wykres", "Rows", 2 ); // dwa wiersze
setChartData( "Wykres", "Cols", 3 ); // trzy kolumny
setChartData( "Wykres", "[1,1]", 1 ); // dane...
setChartData( "Wykres", "[1,2]", 2 );
setChartData( "Wykres", "[1,3]", 4 );
setChartData( "Wykres", "[2,1]", 1 );
setChartData( "Wykres", "[2,2]", 4 );
setChartData( "Wykres", "[2,3]", 3 );
```

Drugi sposób polega na bezpośrednim ustawieniu postaci wykresu i wartości danych przy użyciu tablicy (macierzy). Służy do tego instrukcja *setChartArray*. Instrukcja powoduje ustawienie wykresu zgodnie z rozmiarem podanej jako parametr tablicy i przepisanie wartości danych z tablicy do wykresu.

Przykład:

```
// definicja wykresu jest rownowazna poprzedniej
double Tablica[ 2, 3 ];
Tablica[ 0, 0 ] := 1;
Tablica[ 0, 1 ] := 2;
Tablica[ 0, 2 ] := 4;
Tablica[ 1, 0 ] := 1;
Tablica[ 1, 1 ] := 4;
Tablica[ 1, 2 ] := 3;
setChartArray( "Wykres", Tablica );
```

Ostatnią metodą wstawienia danych do wykresu jest wykorzystanie mechanizmu wiązania wykresu z arkuszem kalkulacyjnym. Bliższe informacje na ten temat zamieszczono na końcu rozdziału.

FORMATOWANIE WYKRESU

Postać wykresu może być ustawiana w sposób interakcyjny – w trakcie działania programu (przy użyciu menu dostępnego po naciśnięciu prawego przycisku myszy) lub określona przez inżyniera wiedzy „z góry”, za pomocą ogólnej instrukcji *setChartData*. Szczegółowy opis składni instrukcji oraz znaczenia jej parametrów zawarty jest w „Podręczniku inżyniera wiedzy” (3. część dokumentacji).

WIZUALIZACJA WYKRESU

Aby wyświetlić utworzony wykres na ekranie należy posłużyć się instrukcją *showChart*. Przy jej wywołaniu oprócz podania nazwy wykresu, należy wskazać czy ma być możliwa edycja wykresu (wartość *1*), czy nie (wartość *0*).

Przykład:

```
showChart( "Wykres", 1 ); // edycja wykresu dozwolona
```

ZAPAMIĘTYWANIE WYKRESU

Wykorzystując instrukcję *writeChart* można gotowy wykres zapamiętać na dysku w postaci pliku o podanej nazwie. Możliwe jest zapamiętanie definicji wykresu w formie binarnej (plik typu *.vtc) lub zapis postaci wykresu jako grafiki w jednym z następujących formatów:

- *bmp* – mapa bitowa bez kompresji;
- *gif* – skompresowany plik graficzny;
- *jpg* – skompresowany plik graficzny.

Przykład:

```
writeChart( "Wykres", "wykres.vtc" );
writeChart( "Wykres", "wykres.bmp" );
```

WCZYTANIE DEFINICJI WYKRESU

Zapamiętana wcześniej na dysku definicja wykresu (w postaci pliku binarnego *.vtc) może zostać zaimportowana do bieżącego wykresu za pomocą instrukcji *readChart*.

W tym miejscu należy wyjaśnić różnicę między wykorzystaniem instrukcji *readChart* a wczytaniem szablonu na etapie tworzenia wykresu. W pierwszym przypadku wykres jest automatycznie kojarzony z plikiem, co oznacza na przykład, że można zapisać wykres przy użyciu instrukcji *writeChart* pomijając nazwę pliku (podając łańcuch pusty). Pliki szablonowe, wczytywane w trakcie tworzenia wykresu za pomocą instrukcji *openChart*, stanowią jedynie początkowy wzorzec wykresu i nie są w z nim w żaden sposób powiązane.

ZAMYKANIE WYKRESU

W celu zamknięcia wykresu należy użyć instrukcji *closeChart*. Instrukcja powoduje zamknięcie widoku wykresu (o ile był aktywny) i usunięcie wykresu z pamięci. Aby zamknąć sam widok (bez usuwania wykresu) należy posłużyć się instrukcją *closeWindow*, służącą do zamykania dowolnego typu okien.

Przykład:

```
closeChart( "Płynność finansowa" );
```

ŁĄCZENIE WYKRESU Z ARKUSZEM KALKULACYJNYM

Język Sphinx daje możliwość kojarzenia danych zawartych w arkuszach kalkulacyjnych z wykresami. Służy do tego instrukcja *linkChart2Sheet*. Przy wywołaniu instrukcji podaje się nazwę okna wykresu, nazwę skoroszytu oraz żądany zakres danych, a także określa tryb połączenia wykresu z arkuszem (wartość *0* – zerwanie powiązania z arkuszem, *1* – powiązanie z arkuszem oraz aktualizacja wartości danych, *2* – powiązanie z arkuszem połączone z przeformatowaniem wykresu, a więc uaktualnieniem ilości wierszy i kolumn na wykresie zgodnie z wielkością wskazanego zakresu danych). Szczegółowe informacje na temat składni instrukcji *linkChart2Sheet* znaleźć można w tomie „Instrukcje języka Sphinx” (3. część dokumentacji), natomiast przykład łączenia wykresu z arkuszem znajduje się w demonstracyjnej bazie wiedzy *link.bw*.

DOSTĘP DO BAZ DANYCH

WPROWADZENIE

Wraz z wersją 2.1 systemu PC-Shell został wprowadzony mechanizm dostępu do konwencjonalnych baz danych. Zastosowane w systemie rozwiązanie opiera się na dostępie poprzez mechanizm ODBC (ang. *Open Database Connectivity*), promowanym przez firmę Microsoft. Mechanizm ODBC udostępnia dostęp do dowolnego systemu zarządzania bazą danych (ang. *DBMS*), pod warunkiem posiadania odpowiednich interfejsów (ang. *drivers*) dostarczanych z reguły przez producentów systemów zarządzania bazami danych.

Zalety mechanizmu ODBC to:

- niezależność od różnych baz danych (w taki sam sposób odwołujemy się do baz np. firmy Oracle i np. szeroko stosowanych baz typu dBase);
- pełna otwartość poprzez wykorzystywanie do komunikacji z bazami standardowego języka zapytań SQL;
- możliwość tworzenia zapytań w trakcie kompilacji lub w trakcie pracy programu.

W systemie PC-Shell dostęp do baz danych został zaimplementowany poprzez dodanie szeregu nowych instrukcji. Inżynier wiedzy może obecnie realizować takie operacje jak:

- inicjalizacja dostępu do bazy danych;
- przesłanie dowolnego zapytania SQL w tzw. trybie bezpośrednim wraz z możliwością pozyskania wyniku działania zapytania;
- sterowanie transakcjami za pomocą odpowiednich instrukcji programowania.

Proces komunikacji z bazą danych musi być obramowany etapem inicjacji dostępu oraz na końcu etapem zakończenia dostępu. Do tego celu służą instrukcje *sqlInit* oraz *sqlDone*. Instrukcją do przesyłania zapytań SQL jest instrukcja *sqlQuery*, natomiast instrukcjami służącymi do pobrania danych po wykonaniu zapytania są instrukcje *sqlInitBinding*, *sqlBind* oraz *sqlFetch*. Ostatnią instrukcją związaną z dostępem do baz danych jest instrukcja sterowania transakcjami *sqlTransact*.

W dalszej części rozdziału przedstawimy parę przykładów ilustrujących metody dostępu do baz danych. Pominęto przy tym omówienie składni języka SQL. Zainteresowanych tym językiem odsyłamy do bogatej literatury na ten temat.

POBIERANIE DANYCH Z BAZY DANYCH

Pobranie danych z bazy musi być poprzedzone utworzeniem bufora na dane. Dlatego należy najpierw oczyścić bufor przy użyciu funkcji *sqlInitBinding*, a następnie, za pomocą instrukcji *sqlBind*, dodać kolejne zmienne, odpowiadające kolejnym kolumnom w bazie danych. Należy podkreślić, że bardzo ważna jest kolejność wywoływania instrukcji *sqlBind* i kolejność nazw kolumn w zapytaniu SQL. Na przykład zapytanie dotyczące pobrania danych Nazwiska, Imienia i Pensji z bazy danych powinno być poprzedzone następującą sekwencją inicjującą bufory:

```
char Nazwisko, Imie;
int Pensja;
sqlInitBinding;
sqlBind( Nazwisko );
sqlBind( Imie );
sqlBind( Pensja );
sqlQuery( "SELECT NAZW, IMIE, PENSJA FROM ZAROBKI" );
```

Jako zmienne buforowe można także zastosować rekordy. W takim przypadku kolejne pola rekordu odpowiadają pojedynczej zmiennej buforowej. Raz zdefiniowany bufor może być używany do większej liczby zapytań. Należy pamiętać także o zgodności typów pomiędzy typem zmiennych a typem kolumny. System PC-Shell stosuje domyślne konwersje, np. gdy kolumna jest typu numerycznego, a odpowiadająca jej zmienna jest typu char, to nastąpi automatyczna konwersja liczby do postaci łańcucha znakowego. Pewnym udogodnieniem są konwersje kolumn typu „data”, „czas” lub kolumny złożonej „data-czas”. Dla tego typu kolumn należy jako bufor zastosować albo pojedynczą zmienną typu znakowego, albo odpowiednią ilość zmiennych numerycznych. Dla daty odpowiednio rok, miesiąc i dzień, a dla czasu godzina, minuta i sekunda. Dla złożonej kolejno: rok, miesiąc, dzień, godzina, minuta, sekunda i setna sekundy.

Przykład:

```
// Pobranie kolumny typu data do rekordu:
record Data
  begin
    int Rok, Mies, Dzień;
  end DataRozp;
sqlInitBinding;
sqlBind( DataRozp );
sqlQuery( "SELECT DATA FROM DANE WHERE NAZWISKO = 'Nowak'" );
```

Pobranie danych odbywa się dopiero w momencie wykonania instrukcji *sqlFetch*, która jako rezultat zwraca poprzez swój argument wartość 1, gdy są pobrane dane oraz 0, gdy brak już danych do pobrania. Instrukcję *sqlFetch* należy wykonywać w pętli, dopóki nie zostanie zwrócona wartość 0 oznaczająca koniec danych.

Przykład:

```
control
  char Name;
  int IsValue;
  // inicjalizacja dostępu do baz typu dBase
  sqlInit( "DSN=Pliki dBase", 1 );
  // inicjalizacja bufora na dane
  sqlInitBinding;      // wyczyść stary bufor
  sqlBind( Name );     // dodaj kolejną zmienną do bufora
  // przesłanie zapytania SQL
  sqlQuery( "SELECT NAME FROM ONE" );
  while ( 1 == 1 )
  begin
    sqlFetch( IsValue );
    if ( IsValue == 1 )
    begin
      // obsługa pobranych danych (kolejna pozycja
      // z pola NAME jest umieszczona w zmiennej Name)
      ...
    end
    else
    begin
      break;
    end;
  end;
  // zakończenie pracy z bazami danych
  sqlDone;
  ...
end;
```


ZAPIS DANYCH DO BAZY

Zapis danych do bazy danych polega na stworzeniu odpowiedniej instrukcji *INSERT* lub *UPDATE* języka SQL, w której należy odpowiednio wstawić właściwe polecenia i wartości. Do tworzenia tekstu zapytania w zmiennej typu char możemy użyć instrukcji *ntos*, *strcat* oraz *sprintf*.

Przykład:

```
control
// inicjalizacja dostępu do baz typu dBase
sqlInit( "DSN=Pliki dBase", 1 );
// dodanie kolejnego rekordu do bazy danych
sqlQuery( "INSERT INTO EMP (NAZWISKO,IMIE,PENSJA) VALUES
('Kowalski','Jan',832)" );
sqlDone;
end;
```

STEROWANIE TRANSAKCYJAMI

Problem transakcji w bloku *control* rozwiązano w ten sposób, że użytkownik w momencie inicjacji decyduje o tym, czy przekazuje kontrolę nad transakcjami systemowi PC-Shell lub sam nimi steruje za pomocą instrukcji *sqlTransact*. W pierwszym przypadku każde zapytanie SQL jest traktowane jako pojedyncza transakcja – każda instrukcja jest potwierdzana jako instrukcja poprawna (*commit*).

Przykład:

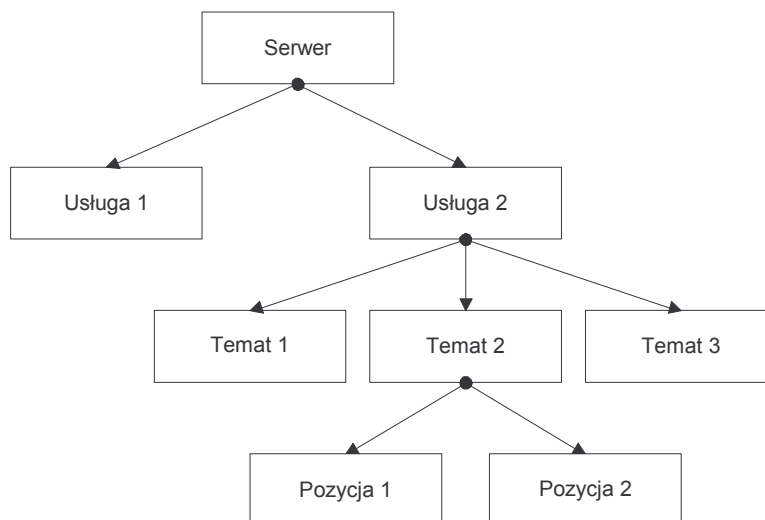
```
control
// inicjalizacja dostępu do baz typu dBase,
// parametr nr 2 równy 0 wskazuje, że transakcjami steruje
// program zbudowany przez inżyniera wiedzy
sqlInit( "DSN=Oracle DBMS", 0 );
// Początek transakcji nr 1
sqlQuery( "INSERT INTO EMP (NAZWISKO,IMIE,PENSJA) VALUES
('Kowalski','Jan',832)" );
sqlQuery( "INSERT INTO EMP (NAZWISKO,IMIE,PENSJA) VALUES
('Nowak','Tomasz',780)" );
if ( RETURN == 1 )
begin
// zatwierdzenie transakcji nr 1
sqlTransact( commit );
end
else
begin
// cofnięcie transakcji nr 1
sqlTransact( rollback );
end;
// początek transakcji nr 2
...
sqlDone;
end;
```


DYNAMICZNA WYMIANA DANYCH (DDE)

WPROWADZENIE

System PC-Shell może współpracować z innymi aplikacjami wykorzystując mechanizm dynamicznej wymiany danych DDE (ang. *Dynamic Data Exchange*). System ekspertowy PC-Shell może pracować zarówno jako klient DDE – wykorzystując zbiór instrukcji języka Sphinx w bloku sterowania, jak i serwer DDE udostępniając swoje usługi innym aplikacjom, obsługującym DDE.

Mechanizm DDE oparty jest na architekturze „klient–serwer”. Aplikacja, która udostępnia usługi innym nazywana jest serwerem DDE, natomiast aplikacja korzystająca z tych usług nazywana jest klientem DDE. Każdy serwer DDE ma określoną przez producenta strukturę umożliwiającą pracę z dowolnym klientem DDE. Struktura ta jest hierarchiczna i składa się z 3 poziomów. Pierwszy poziom to tzw. „usługa” (ang. *service*), kolejny poziom to zbiór „tematów” (ang. *topics*), w zakresie których dany serwer może nawiązać łączność. Każdy temat z kolei posiada zbiór „pozycji” (ang. *items*), reprezentujących pewne dane, które mogą być przedmiotem wymiany. Przykładowo dla arkuszy kalkulacyjnych tematami są otwarte arkusze, natomiast pozycjami jego komórki lub ich zakresy. Lista pozycji i tematów w trakcie pracy serwera może dynamicznie ulegać zmianie. Dokładne nazwy usług, tematów i pozycji udostępniane są przez producentów serwerów. Standardowo przyjmuje się, że każda usługa udostępnia temat o nazwie *System*, w ramach którego znajdują się m.in. takie pozycje jak *Topic* (udostępnia nazwy tematów udostępnianych przez usługę) oraz *TopicItemList* (zawiera listę pozycji dostępnych w danym temacie). Lista usług z reguły ogranicza się do jednej o nazwie takiej, jak nazwa aplikacji np. Excel, Progman lub PCShell.



RYS. 29. HIERARCHICZNA STRUKTURA WYMIANY DANYCH (DDE)

Komunikacja w ramach DDE opiera się na wykorzystaniu tzw. **kanałów**. Klient, chcąc nawiązać konwersację z serwerem, musi w pierwszym rzędzie zainicjować kanał, podając jako argumenty nazwę usługi oraz nazwę tematu. W przypadku nawiązania kontaktu pomiędzy aplikacjami ustalany jest uchwyt kanału, którym należy się posługiwać przy każdej kolejnej operacji dotyczącej nawiązanej „konwersacji”. Możliwe jest otwieranie dowolnej ilości kanałów jednocześnie, przy czym zwiększenie ilości otwartych kanałów powoduje zmniejszenie wydajności systemu Windows. Obsługa kanałów od strony serwera DDE jest automatyczna i nie wymaga oprogramowania jej przez użytkownika.

Mając nawiązaną komunikację, aplikacja może wykonywać trzy rodzaje operacji:

- pobieranie danych z serwera (komenda *request*);
- wysyłanie danych do serwera (komenda *poke*);
- wykonywanie poleceń na serwerze (komenda *execute*).

Pobieranie danych może polegać na przykład na pobraniu zawartości komórki w arkuszu kalkulacyjnym, pobraniu przez klienta PC-Shell’a zawartości zmiennej bloku sterowania (*control*), pobraniu wartości z aplikacji odczytującej stan urządzeń zewnętrznych (np. jakiegoś miernika bądź sterownika przemysłowego). Podobnie wysyłanie danych od klienta do serwera może służyć parametryzacji pracy serwera, przygotowaniu

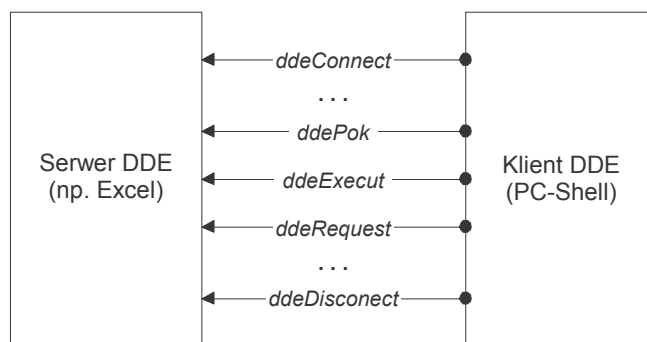
wartości pracy serwera itp. Dane pobierane i wysyłane w ramach DDE mogą być zarówno danymi tekstowymi jak i numerycznymi. Wykonywanie komend na serwerze DDE umożliwia zainicjowanie pewnych procesów, makr na serwerze DDE, dzięki czemu można dokonywać obliczeń i innych operacji przewidzianych przez producenta serwera.

PC-SHELL JAKO KLIENT DDE

Praca systemu PC-Shell jako klienta DDE polega na możliwości programowego wykorzystania możliwości komunikowania się z innymi aplikacjami, pracującymi jako serwery DDE. Przykładowymi aplikacjami z którymi może połączyć się PC-Shell są arkusze kalkulacyjne (np. Microsoft Excel), edytory tekstów i inne. Poszczególne etapy implementacji obsługi klienta DDE w języku Sphinx opisane zostaną poniżej i zobrazowane przykładem komunikacji z arkuszem kalkulacyjnym Microsoft Excel. Aplikacja Excel pracuje zarówno jako klient jak i serwer DDE, dlatego można wykorzystać ją do współpracy z systemem PC-Shell. Może się ona stać dla nas źródłem danych jak również może stanowić aplikację, do której przesyłamy dane.

Język Sphinx udostępnia pięć instrukcji które służą do implementacji funkcji klienta DDE:

- instrukcja inicjująca nawiązanie komunikacji (*ddeConnect*);
- instrukcja kończąca komunikację (*ddeDisconnect*);
- instrukcja przesłania danych do serwera DDE (*ddePoke*);
- instrukcja pobrania danych z serwera (*ddeRequest*);
- instrukcja zainicjowania komendy (*ddeExecute*).



RYS. 30. KOMUNIKACJA POMIĘDZY SERWEREM I KLIENTEM

Chcąc zainicjować komunikację DDE, musimy mieć pewność, że aplikacja, z którą chcemy się komunikować jest uruchomiona w systemie (jeżeli nie jest – możemy ją uruchomić instrukcją wykorzystując instrukcję *system*). Fragment kodu sprawdzający, czy jest uruchomiony Excel, może wyglądać następująco:

```

int TEST;
isAppRunning( "Microsoft Excel", TEST );
if ( TEST == 0 )
begin
    system( "C:\\EXCEL\\EXCEL.EXE" );
end;

```

Po upewnieniu się, że interesująca nas aplikacja jest uruchomiona, można przystąpić do próby nawiązania konwersacji. Jak już wspomniano, do tego celu służy instrukcja *ddeConnect*. Wymaga ona podania trzech parametrów: zmiennej typu *int*, której przypisany zostanie numer otwartego kanału (w przypadku pozytywnego nawiązania konwersacji) oraz dwóch parametrów określających nawiązywaną konwersację – nazwę usługi oraz nazwę tematu. W przypadku udanej próby nawiązania konwersacji z aplikacją Microsoft Excel, nazwą usługi jest słowo „Excel” natomiast tematami są nazwy arkuszy oraz temat „System”. Temat „System” służy do sterowania całym programem czyli np. otwierania nowych arkuszy, ich zamykania itp. Natomiast bezpośrednie połączenie się z arkuszem daje nam dostęp do jego zawartości.

```
int ID1, ID;
ddeConnect( ID1, "Excel", "System" );
ddeExecute( ID1, "[OPEN(\\\"C:\\\\WINSHLL\\BW\\KREDYT.XLS\\\")]" );
ddeConnect( ID2, "[KREDYT.XLS]Arkusz1" );
```

Powyższy przykład ukazuje sposób otwarcia konkretnego arkusza i nawiązania z nim bezpośredniego połączenia. Po nawiązaniu połączenia, posługując się identyfikatorem ID2, można przysyłać, pobierać dane oraz inicjować wykonywanie określonych poleceń.

Kończąc wymianę danych, należy pamiętać o zamknięciu każdej nawiązanej konwersacji za pomocą instrukcji *ddeDisconnect*.

```
ddeDisconnect( ID2 );
ddeDisconnect( ID1 );
```

Po nawiązaniu konwersacji możemy przystąpić do właściwej wymiany informacji. W przypadku przysyłania i pobierania danych należy zawsze określić pozycję, pod jaką przysyłamy dane. W przypadku arkusza kalkulacyjnego jest to adres komórki. Poniżej przedstawiamy przykład w którym zapisujemy i odczytujemy dane:

```
ddePoke( ID2, "W1K1", "Tekst z PC-Shell'a" );
ddePoke( ID2, "W2K1", ZmiennaX );
ddeRequest( ID2, "W1K1", ZmiennaZnakowa );
```

Adresowanie komórek w arkuszu może odbywać się także poprzez nazwę komórki nadanej wcześniej w arkuszu (np. *ddePoke(ID2, Wynik, 100)*).

Wykonywaniu komend służy instrukcja *ddeExecute*.

```
ddeExecute( ID1, "[SAVE]" ); // zapisanie zawartości arkusza
```

Wykonywanie instrukcji *ddePoke*, *ddeRequest* lub *ddeExecute* może zakończyć się pojawieniem okienka dialogowego o zajętości serwera. Taka sytuacja może być spowodowana na przykład wyświetleniem przez aplikację serwera okna dialogowego. W tym przypadku użytkownik powinien zamknąć okno dialogowe serwera i w aplikacji serwera wydać komendę „powtórz”.

Komunikacja pomiędzy PC-Shell'em a serwerami DDE odbywa się zawsze poprzez łańcuchy znakowe, co oznacza, że wszelkie liczby są automatycznie konwertowane przed wysłaniem na łańcuchy. W przypadku wersji narodowych programów i stosowania innej konwencji zapisu liczb zmiennoprzecinkowych może to spowodować niepożądane efekty. Przykładowo, polska wersja arkusza kalkulacyjnego Microsoft Excel rozpoznaje liczby zmiennoprzecinkowe po wystąpieniu przecinka, natomiast w systemie PC-Shell przyjęte jest, że znakiem oddzielającym część całkowitą od części dziesiętnej jest kropka (jak w każdym języku programowania). Inżynier wiedzy musi zatem dokonać odpowiedniej konwersji liczby przed wysłaniem jej do arkusza, aby umożliwić jej prawidłowe rozpoznanie.

Wykonanie fragmentu kodu:

```
float F;
F := 1.567;
ddePoke( ID, "Wynik", F );
```

spowoduje wstawienie do komórki o nazwie „Wynik” łańcucha tekstowego „1.567” (w polskiej wersji arkusza).

Poprawne wywołanie powinno wyglądać:

```
float F;
char Str;
F := 1.567;
c_ntos( F, Str, "," );
ddePoke( ID, "Wynik", Str );
```

Instrukcja `c_ntos` konwertuje liczbę 1.567 na łańcuch znakowy zamieniając znak kropki na wymagany znak przecinka.

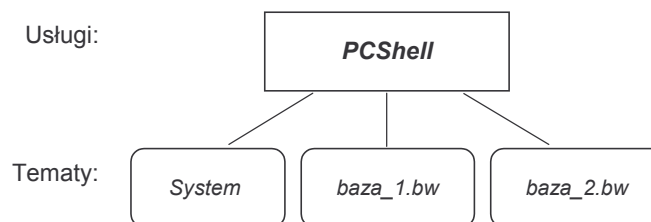
Konwersje liczb w przeciwnym kierunku odbywają się automatycznie z rozpoznaniem przecinka i kropki, nie jest zatem konieczna ich konwersja.

```
// Założenie:
// komórka Wynik1 = "1.345" (łańcuch tekstowy)
// komórka Wynik2 = 1,345 (liczba)
float F;
char Str;
ddeRequest( ID, "Wynik1", Str ); // tekst "1.345"
ddeRequest( ID, "Wynik1", F ); // liczba 1.345
ddeRequest( ID, "Wynik2", Str ); // tekst "1,345"
ddeRequest( ID, "Wynik2", F ); // liczba 1.345
```

Zachęcamy do zapoznania się z bazą wiedzy excel.bw, ilustrującą sposób wykorzystania funkcji dynamicznej wymiany danych na poziomie klienta DDE.

PC-SHELL JAKO SERWER DDE

Począwszy od wersji 2.2 system ekspertowy PC-Shell może pracować jako serwer DDE, co umożliwia połączenie się z nim i wykorzystanie jego możliwości przez aplikacje spełniające rolę klientów DDE. Teoretycznie możliwe jest otwarcie dowolnej ilości kanałów konwersacji jednocześnie (praktycznie ilość ta ograniczona jest zasobami systemu Windows). PC-Shell udostępnia jedną usługę o nazwie „PCShell” w ramach której istnieje jeden standardowy temat „System” oraz dowolna ilość tematów o nazwie identycznej z bazą wiedzy z którą ma nastąpić połączenie.



RYS. 31. STRUKTURA USŁUG I TEMATÓW SYSTEMIE PC-SHELL

Temat *System* udostępnia trzy podstawowe pozycje: *Topics*, *SysItems* oraz *Formats*. Najważniejszymi, z punktu widzenia innych aplikacji, tematami są oczywiście tematy związane z bazami wiedzy. Nawiązanie konwersacji powiązane jest w systemie PC-Shell z konkretną bazą wiedzy – w momencie nawiązania konwersacji jest odczytywana baza wiedzy i natychmiast jest dokonywana jej translacja. W przypadku pozytywnego wczytania bazy wiedzy można przystąpić do komunikacji.

WSTAWIANIE DANYCH (POKE)

Możliwe jest przesyłanie danych bezpośrednio do istniejących zmiennych zadeklarowanych w bloku sterowania.

Przykład:

```
channelNumber =
    DDEInitiate( _app:"PCShell", topic:"C:\BW\KREDYT.BW" )
DDEPoke channelNumber, "X", "10"
DDETerminate channelNumber
```

WYKONYWANIE POLECEŃ (EXECUTE)

System PC-Shell udostępnia szereg poleceń umożliwiających sterowanie aplikacjami. Polecenia są podawane w postaci łańcuchów tekstowych, przy czym możliwe jest podawanie ciągów poleceń. W tym przypadku, każde z poleceń należy ująć w nawiasy kwadratowe („[]”). Parametry wewnątrz poleceń podawane są w postaci łańcuchów ograniczonych znakami apostrofu („' ’”).

Przykład:

```
"[ADDFACTSTR('wskaznik=1')] [GOAL('analiza=X')]"
```

Wykaz poleceń dostępnych w obecnej wersji systemu PC-Shell:

RUN – uruchomienie programu zawartego w bloku sterowania;

GOAL(*hipoteza*) – uruchomienie wnioskowania z podaną hipotezą;

ADDFACT(*O,A,W*) – dodanie do bazy wiedzy faktu opisanego trójką OAW, składnia zgodna z instrukcją *addFact* języka Sphinx;

ADDFACTSTR(*fakt*) – dodanie do bazy wiedzy faktu podanego w postaci tekstowej, np. **ADDFACT**('objaw = gorączka');

CALL(*funkcja(argumenty...)*) – wywołanie funkcji zdefiniowanej przez inżyniera wiedzy w bloku sterowania wraz z listą argumentów, przy czym argumentami mogą być zmienne istniejące w bloku sterowania, wcześniej ustawione na przykład za pomocą instrukcji *ddePoke*;

SETMODE(*0 lub 1*) – ustawienie trybu pracy konsultacji (1 – konsultacja standardowa – interakcja zapewniona przez system PC-Shell; 0 – konsultacja obsługiwana przez klienta DDE). Domyślnym trybem jest tryb pracy interakcyjnej. Przełączenie na tryb nie interakcyjny umożliwia klientowi pełną obsługę konsultacji. Służą do tego, opisane poniżej, polecenia *ANSWER*, *ANSWERIDX*, *RESETGOAL* oraz dodatkowe pozycje udostępnione do pobrania za pomocą instrukcji *ddeRequest* ([*status*], [*question*], [*answer_type*], [*variable*], [*solution*], [*values*], [*how*] oraz [*attribute*]). Po omówieniu poleceń i pozycji do pobrania danych podamy wyczerpujący przykład praktycznego wykorzystania trybu nieinterakcyjnego;

ANSWER(*wartość*) – przypisanie odpowiedzi na zapytanie o wartość zmiennej (polecenia należy użyć w przypadku, gdy zwróconą wartością [*answer_type*] jest [*VARIABLE*]). Po udzieleniu odpowiedzi należy wznowić wnioskowanie;

ANSWERIDX(*indeks*) – odpowiedź w postaci indeksu wartości z listy odpowiedzi wygenerowanych przez system. Po udzieleniu odpowiedzi należy wznowić wnioskowanie;

REGOAL – wznowienie wnioskowania;

RESETGOAL – zaniechanie wnioskowania;

GETSOURCE(*nazwa_źródła*) – wczytanie źródła wiedzy do bazy wiedzy;

FREESOURCE(*nazwa_źródła*) – zwolnienie źródła wiedzy.

POBIERANIE DANYCH (REQUEST)

Klient DDE ma możliwość pobierania od systemu PC-Shell zawartości zmiennych oraz następujące rodzaje danych:

[**status**] – zwracana jest wartość określająca stan systemu. Możliwe wartości to „[*READY*]”, „[*WAITING_FOR_ANSWER*]” lub „[*PROBLEM_RESOLVED*]”;

[**error**] – tekst wyjaśniający błąd;

[**question**] – tekst zapytania w trakcie konsultacji;

[**answer_type**] – zwraca typ zapytania: [*LIST_ONEOF*] (lista odpowiedzi dla atrybutu typu *oneof*), [*LIST_SOMEOF*] (lista wartości dla atrybutu typu *someof*) lub [*VARIABLE*] (zapytanie o zawartość zmiennej);

[**variable**] – zwracana jest nazwa zmiennej;

[**solution**] – kolejny element rozwiązania (pełna trójka OAW), lista kończona jest wartością [*END*];

[**values**] – służy do pobrania kolejnego elementu z listy możliwych odpowiedzi na zapytanie;

[**attribute**] – nazwa atrybutu którego dotyczy zapytanie;

[**how**] – pobranie kolejnej linii tekstu wyjaśnień typu „jak?”.

Poniżej zamieszczono przykład konsultacji z interakcją zrealizowaną po stronie klienta DDE, w tym przypadku programu Microsoft Word:

```
' Uwaga!
' Ask1 oraz Solution1 są formularzami
' (patrz przykład konsult.doc)
Sub TestDDE2()
    ' Zainicjuj kanał komunikacji
    ch = DDEInitiate( "PCShell", "C:\WINSHELL\BW\KREDYT1.BW" )
    ' Ustaw tryb pracy na nieinterakcyjny
    DDEExecute Channel:=ch, Command:="SETMODE(0)"
    ' Załaduj do pamięci źródło wiedzy "deckred"
    DDEExecute Channel:=ch, Command:="GETSOURCE('deckred')"
    ' uruchom wnioskowanie z hipotezą
    ' "decyzja kredytowa=DECYZJA KREDYTOWA"
    DDEExecute Channel:=ch, Command:="GOAL('decyzja kredytowa =
DECYZJA KREDYTOWA')"
    ' Sprawdź stan status wnioskowania
    M = DDERequest( Channel:=ch, Item:="[status]" )
    While ( M = "[WAITING FOR ANSWER]" )
        ' System zadał zapytanie, pobieramy typ zapytania
        TYP = DDERequest( Channel:=ch, Item:="[answer_type]" )
        If TYP = "[LIST_ONEOF]" Then
            ' Jest lista do wyboru
            UserForm1.ListBox1.Clear
            ' Pobieramy listę aż dostaniemy łańcuch [END]
            ' kończący listę
            Do
                S = DDERequest( Channel:=ch, Item:="[values]" )
                If S <> "[END]" Then
                    Ask1.ListBox1.AddItem S
                End If
            Loop Until S = "[END]"
        End If
        ' Pobieramy treść zapytania systemu
        Q = DDERequest( Channel:=ch, Item:="[question]" )
        Ask1.Label2.Caption = Q
        ' Wyświetlamy okno konsultacji w którym z przyciskiem
        ' skojarzone jest makro CommandButton1_Click()
        Ask1.Show
        ' Wznawiamy wnioskowanie
        DDEExecute Channel:=ch, Command:="REGOAL"
        M = DDERequest( Channel:=ch, Item:="[status]" )
    Wend
    ' Sprawdź czy jest i pokaż rozwiązanie
    If M = "[PROBLEM_RESOLVED]" Then
        Solution1.ListBox1.Clear
        Do
            ' pobieramy listę rozwiązań
            S = DDERequest( Channel:=ch, Item:="[solution]" )
            If S <> "[END]" Then
                UserForm2.ListBox1.AddItem S
            End If
        Loop Until S = "[END]"
        ' Wyświetlamy rozwiązania
        Solution1.Show
        ' pobranie i wyświetlenie wyjaśnienia "Jak?"
        Solution1.ListBox1.Clear
        Do
            S = DDERequest( Channel:=ch, Item:="[how]" )
            If S <> "[END]" Then
```

```
        Solution1.ListBox1.AddItem S
    End If
    Loop Until S = "[END]"
    Solution1.Show
End If
' Zwolnij źródło wiedzy
DDEExecute Channel:=ch, Command:="FREESOURCE('deckred')"
DDETerminate( ch )
End Sub

Private Sub CommandButton1_Click()
    l = ListBox1.ListIndex
    If ( l = -1 ) Then
        MsgBox ("Proszę odpowiedzieć na zapytanie")
    Else
        S = Str( l )
        ST = "ANSWERIDX(" + S + ")"
        DDEExecute Channel:=ch, Command:=ST
        Ask1.Hide
    End If
End Sub

Private Sub
    ListBox1_DblClick( ByVal Cancel As MSForms.ReturnBoolean )
    Call CommandButton1_Click
End Sub
```

PRZYKŁAD WYKORZYSTANIA MECHANIZMU DDE

Ważną zaletą edytorów tekstu, pracujących w systemie Windows, jest ich otwartość na współpracę z innymi aplikacjami, realizowana między innymi za pośrednictwem protokołu dynamicznej wymiany danych DDE (ang. *dynamic data exchange*). Dzięki wykorzystaniu protokołu DDE istnieje możliwość sterowania pracą, na przykład, edytora tekstu (w zakresie edycji, formatowania i wydruku tekstu). Poniżej przedstawiono sposób automatycznego tworzenia i wydruku prostego raportu za pośrednictwem edytora Microsoft Word (w wersji 6.0).

NAWIĄZANIE KONWERSACJI Z EDYTOREM

Aby móc skorzystać z mechanizmów DDE, w pierwszej kolejności należy nawiązać „konwersację” z serwerem (w naszym przypadku jest nim MS-Word). Nazwę serwera w przypadku programu MS-Word stanowi łańcuch tekstowy „WinWord”, natomiast interesujący nas „temat” identyfikuje łańcuch tekstowy „System”.

```
int ID;
ddeConnect( ID, "WinWord", "System" );
```

TWORZENIE PLIKU RAPORTU

Tworzenie raportu należy rozpocząć od polecenia otwarcia nowego dokumentu (komenda *[FileNew]*):

```
ddeExecute( ID, "[FileNew]" );
```

Mając nawiązaną konwersację i otwarty nowy dokument możemy przystąpić do tworzenia właściwego raportu. Tworzenie raportu polega na wypełnieniu dokumentu treścią i odpowiednim jego sformatowaniu (w sposób identyczny jak podczas interakcyjnej pracy z edytorem). W naszym przypadku, zamiast posługiwania się klawiaturą i myszką, wszelkie operacje na edytorze zastępujemy komendami akceptowanymi przez edytor a wysyłanymi jako rozkazy w instrukcji *ddeExecute*. Format poleceń jest zgodny z konwencjami DDE – komendy są podawane w nawiasach kwadratowych „[]” natomiast składnia komend jest identyczna ze

składnią WordBasic'a stosowanego w edytorze MS-Word. W celu lepszego zorientowania się w sposobie użycia poszczególnych poleceń należy zapoznać się z poniższym przykładem oraz przykładową bazą wiedzy *raport.bw*, dostarczoną z systemem PC-Shell. Warto również przeanalizować składnię makr zarejestrowanych, na przykład, w trakcie pracy, składnia makr opiera się bowiem właśnie na WordBasic'u.

Ograniczeniem wymiany za pomocą komend DDE jest wielkość tekstu – w zależności od programu narzucane są pewne ograniczenia dotyczące długości przesyłanych komend, i tak na przykład edytor MS-Word akceptuje maksymalnie 512 znaków. Aby ominąć powyższe ograniczenia należy posługiwać się krótszymi komendami lub do wymiany danych posłużyć się „schowkiem” (ang. *clipboard*). W celu wymiany danych można również wykorzystać pliki. W poniższym przykładzie przedstawiono sposób przesłania treści wyjaśnień „jak?” poprzez „schowek”.

WYDRUK RAPORTU

Edytor Word umożliwia wydrukowanie dokumentu po wydaniu komendy:

```
ddeExecute( ID, "[FilePrint]" );
```

PRZYKŁAD TWORZENIA RAPORTU

```
control
int ID;
char Dest;
// ustalenie połączenia z edytorem Microsoft Word
ddeConnect( ID, "WinWord", "System" );
// otwarcie nowego dokumentu
ddeExecute( ID, "[FileNew]" );
// WYPEŁNIENIE RAPORTU
// - utworzenie nagłówka
ddeExecute( ID, "[TableInsertTable .ConvertFrom = \"\",
    .NumColumns = \"2\", .NumRows = \"1\", .InitialColWidth =
    \"Auto\", .Format = \"0\", .Apply = \"167\" ]" );
// b) zmiana czcionki
ddeExecute( ID, "[FormatFont .Points = \"18\" ]" );
ddeExecute( ID, "[FormatFont .Font = \"Braggadocio\" ]" );
ddeExecute( ID, "[FormatFont .Color = 2 ]" );
// - zmiana wyrównania na centrowanie tekstu
ddeExecute( ID, "[CenterPara]" );
// - wstawienie tekstu "AI"
ddeExecute( ID, "[Insert \"AI\"]" );
// - zmiana koloru na granatowy
ddeExecute( ID, "[FormatFont .Color = 6 ]" );
ddeExecute( ID, "[Insert \"TECH\"]" );
// - nowa linia
ddeExecute( ID, "[InsertPara]" );
// - przywrócenie czcionki
ddeExecute( ID, "[FormatFont .Points = \"12\" ]" );
ddeExecute( ID, "[FormatFont .Font = \"Arial CE\"]" );
ddeExecute( ID, "[FormatFont .Color = 0 ]" );
ddeExecute( ID, "[Insert \"\nArtificial Intelligence
    Laboratory\"]" );
ddeExecute( ID, "[InsertPara]" );
ddeExecute( ID, "[Insert \"KATOWICE\" ]" );
// - przesunięcie do prawej kolumny
ddeExecute( ID, "[CharRight 1]" );
// zmiana wyrównania na 'do prawej'
ddeExecute( ID, "[RightPara]" );
ddeExecute( ID, "[Insert \"KATOWICE, \" ]" );
// - wstawienie bieżącej daty
```

```
ddeExecute( ID, "[InsertDateTime .DateTimePic = \"d MMMM
rrrr\" ]" );
// - opuszczenie tabeli
ddeExecute( ID, "[CharRight 1]" );
ddeExecute( ID, "[CharRight 1]" );
ddeExecute( ID, "[InsertPara][InsertPara]" );
// - utworzenie nagłówka raportu
ddeExecute( ID, "[CenterPara]" );
ddeExecute( ID, "[FormatFont .Bold = 1 ]" );
ddeExecute( ID, "[Insert \"Raport systemu PC-Shell\" ]" );
ddeExecute( ID, "[FormatFont .Bold = 0 ]" );
ddeExecute( ID, "[InsertPara]" );
// - właściwa treść raportu:
ddeExecute( ID, "[Insert \"Wyjaśnienia dla rozwiązania
rodzaj = pieczarka\"]" );
ddeExecute( ID, "[InsertPara]" );
saveExplan( Dest, _, "rodzaj", "pieczarka", 0 );
if ( RETURN == 1 )
begin
    toClipboard( Dest );
    ddeExecute( ID, "[EditPaste]" );
end
else
begin
    ddeExecute( ID, "[Insert \"Brak.\"][InsertPara]" );
end;
// WYDRUK RAPORTU NA DRUKARCE DOMYŚLNEJ
ddeExecute( ID, "[FilePrint]" );
ddeDisconnect( ID );
end;
```

AUTOMATYZACJA OLE

Począwszy od wersji 4.0 systemu Sphinx system PC-Shell oraz język został wyposażony w obsługę mechanizmu OLE (Object Linking and Embedding). Obsługa sprowadza się z jednej strony do udostępniania swoich usług poprzez serwer automatyzacji OLE, jak również język Sphinx został rozszerzony o instrukcje do obsługi obiektów OLE Automation.

SERWER AUTOMATYZACJI OLE

System PC-Shell jest rejestrowany w systemie obsługi OLE jako obiekt "PCShell" klasy "Sphinx". W ramach tego obiektu udostępniony jest zbiór metod umożliwiających wczytywanie i pracę z istniejącymi bazami lub źródłami wiedzy. Poniżej przedstawione są wszystkie metody obiektu "Sphinx.PCShell".

Funkcja `kbOpen` (long Instance, BSTR Path, long Parent) int

Funkcja tworzy nowy wskaźnik do bazy wiedzy zapisanej w pliku określonym parametrem *Path* i zwraca jego identyfikator. Parametr *Path* powinien wskazywać do istniejącej i prawidłowej bazy lub źródła wiedzy. Funkcja ta automatycznie wczytuje bazę wiedzy do pamięci. Parametr *Instance* w obecnej wersji jest ignorowany – powinien przyjąć wartość 0, natomiast parametr *Parent* umożliwia wskazanie uchwytu okna rodzica, może być zignorowany (wartość 0) wtedy oknem macierzystym staje się aktualnie aktywne okno.

Parametr zwracany przez funkcję to uchwyt bazy wiedzy którym posługujemy się przy wywoływaniu pozostałych metod. Gdy wartość jest ujemna – nastąpił błąd.

Otwartą bazę wiedzy należy na koniec zamknąć procedurą *kbClose*.

Procedura `kbInitialize` (long Instance)

Procedura uruchamia inicjalizację systemu PC-Shell. Metoda ta musi być wywołana po utworzeniu obiektu. Przed zakończeniem należy wywołać metodę *kbDone*

Procedura `kbDone` ()

Procedura zwalnia bibliotekę obsługującą system PC-Shell. Powinna być wywoływana jako ostatnia przed zwolnieniem obiektu.

Procedura `kbClose` (long kbID)

Procedura zamyka (zwalnia) wczytaną bazę wiedzy. Po jej wywołaniu identyfikator kbID wskazuje na nie zainicjowaną bazę wiedzy.

Procedura `kbRunProgram` (long kbID)

Procedura uruchamia blok programowy zdefiniowany w załadowanej bazie wiedzy.

Procedura `kbSetParent` (long kbID, long Parent)

Procedura umożliwia ustawienie uchwytu okna macierzystego. Przekazywany parametr *Parent* musi być prawidłowym uchwytem typu HWND okna.

Procedura `kbCallFunction` (long kbID, BSTR FunctionCall)

Procedura umożliwia uruchomienie wybranej funkcji z bloku control wczytanej bazy wiedzy. Parametr *FunctionCall* jest pełną postacią wywołania funkcji wraz z parametrami podanymi wewnątrz nawiasów np. "zapisz("test.txt")".

Procedura `kbSetVariable` (long kbID, BSTR VarName, BSTR Value)

Procedura umożliwia ustawienie wartości zmiennej globalnej zdefiniowanej w bloku control.

Procedura **kbGetVariable** (long kbID, BSTR VarName, Variant Dest)

Procedura umożliwia pobranie wartości zmiennej globalnej bloku control.

Procedura **kbGoal** (long kbID, BSTR Goal)

Procedura uruchamia proces wnioskowania wstecz o celu zdefiniowanym w parametrze *Goal*. Przebieg procesu wnioskowania uzależniony jest od trybu pracy określonego przez procedurę *kbSetInteractive*. Domyślnie po wczytaniu tryb jest tzw. interaktywny tzn. że w momencie zapytania system PC-Shell otwiera okno zapytania w którym użytkownik udziela interakcyjnie odpowiedzi. W trybie nie interakcyjnym procedura nie generuje jakichkolwiek okien (zapytania lub odpowiedzi). Po jej wywołaniu można sprawdzić wtedy stan procesu wnioskowania za pomocą procedury *kbGetRequest* z parametrem *[status]* – system PC-Shell zwróci wartość *[WAITING_FOR_ANSWER]* lub wartość *[PROBLEM_RESOLVED]*. W przypadku gdy system czeka na odpowiedź możemy jej udzielić za pomocą procedur *kbAnswerIDX* lub *kbAnswerStr* i wznowić proces wnioskowania procedurą *kbReGoal*. Parametry zapytania można pobrać również za pomocą procedury *kbGetRequest*.

Procedura **kbSolve** (long kbID, BSTR Source, BSTR Goal)

Procedura podobnie jak *kbGoal* uruchamia proces wnioskowania lecz na podstawie źródła określonego w parametrze *Source*. Procedura ta uruchamia proces wnioskowania jedynie w sposób interakcyjny.

Procedura **kbReGoal** (long kbID)

Procedura wznowia przerwany proces wnioskowania. Aktywna jedynie w trakcie uruchamiania procesu wnioskowania w trybie nie interakcyjnym.

Procedura **kbResetGoal** (long kbID)

Procedura przerywa proces wnioskowania uruchomiony poprzednio w trybie nie interakcyjnym.

Procedura **kbAddFactStr** (long kbID, BSTR Fact)

Procedura dodaje do bazy wiedzy fakt *Fact*, podany w postaci trójki OAW.

Procedura **kbAnswerIDX** (long kbID, long Index)

Procedura **kbAnswerStr** (long kbID, BSTR Answer)

Procedury umożliwiają symulację odpowiedzi w trybie nie interakcyjnym. Służą wybraniu odpowiedzi odpowiednio – indeksu lub przypisania wartości o którą zapytał się system ekspertowy.

Procedura **kbSetInteractive** (long kbID, long Mode)

Procedura ustawia tryb pracy wnioskowania dla procedury *kbGoal*. Wartość 1 oznacza interaktywny tzn. pojawiają się standardowe okna systemu PC-Shell, 0 – brak interakcji.

Procedura **kbGetRequest** (long kbID, BSTR Key, Variant Dest)

Procedura umożliwia pobranie wartości parametrów stanu systemu, wartości zapytań, możliwych odpowiedzi i innych informacji.

Dopuszczalne wartości o które system może zapytać to:

[status] – zwracana jest wartość określająca stan systemu (możliwe wartości to *[READY]*, *[WAITING_FOR_ANSWER]* lub *[PROBLEM_RESOLVED]*);

[error] – tekst wyjaśniający błąd;

[**question**] – tekst zapytania w trakcie konsultacji;

[**answer_type**] – zwraca typ zapytania ([*LIST_ONEOF*] – lista odpowiedzi dla atrybutu typu *oneof*, [*LIST_SOMEOF*] – lista wartości dla atrybutu typu *someof* lub [*VARIABLE*] – zapytanie o zawartość zmiennej);

[**variable**] – zwracana jest nazwa zmiennej tymczasowej użytej w czasie wnioskowania o wartość której system wygenerował zapytanie;

[**solution**] – kolejny element rozwiązania (pełna trójka OAW), lista kończona jest wartością [*END*];

[**values**] – służy do pobrania kolejnego elementu z listy możliwych odpowiedzi na zapytanie;

[**attribute**] – nazwa atrybutu którego dotyczy zapytanie;

[**how**] – pobranie kolejnej linii tekstu wyjaśnień typu „jak?”.

```
Procedura kbGetSource ( long kbID, BSTR Source )
```

```
Procedura kbFreeSource ( long kbID, BSTR Source )
```

Procedury umożliwiają wczytanie i zwolnienie z pamięci źródeł wiedzy.

```
Funkcja kbGetSolutionsCount ( long kbID ) int
```

Procedura zwraca ilość rozwiązań wygenerowanych podczas ostatniej konsultacji.

```
Procedura kbAttributeList ( long kbID, Variant List )
```

Procedura zwraca w postaci tablicy nazwy atrybutów zdefiniowanych w bazie wiedzy.

```
Procedura kbGetValueList (long kbID, BSTR Attribute, Variant List)
```

Procedura zwraca tablicę (jeżeli jest zdefiniowana) dopuszczalnych wartości dla podanego atrybutu.

```
Procedura kbAddFFact ( long kbID, BSTR Obj, BSTR Atr, Float Val )
```

Procedura dodaje numeryczny fakt do bazy wiedzy

```
Procedura kbCatchFact ( long kbID, BSTR Obj, BSTR Atr,  
                        long Which, Variant Value )
```

Procedura umożliwia przechwytywanie kolejnych faktów według określonego klucza. Sposób wywołania jest analogiczny jak instrukcja **catchFact** języka Sphinx.

```
Procedura kbDelFact ( long kbID, BSTR Obj, BSTR Atr, BSTR Val )
```

Procedura umożliwia usunięcie wybranych faktów z bazy wiedzy.

```
Procedura kbDelMewFacts ( long kbID )
```

Procedura usuwa wszystkie nowe fakty z bazy wiedzy.

```
Funkcja kbIsControlBlock ( long kbID ) int
```

Funkcja wraca wartość różną od 0 gdy podana baza wiedzy posiada nie pusty blok sterowania.

```
Procedura kbCatchFacts ( long kbID, BSTR Obj, BSTR Atr,  
                        Variant List )
```

Procedura zwraca w postaci tablicy wszystkie fakty z bazy wiedzy.

```
Procedura kbSplitOAV ( long kbID, BSTR OAV, Variant Obj,  
Variant Atr, Variant Value )
```

Procedura rozdziela łańcuch tekstowy – trójkę OAW na elementy składowe – obiekt, atrybut i wartość.

PRZYKŁAD WYWOŁANIA PC-SHELLA JAKO SERWERA OLE

Poniżej przedstawiony jest przykład w języku Visual Basic for Application (VBA) czyli języku wykorzystywanym np. w makrach pakietu Microsoft Office wywołania systemu eksperowe00go. Program wczytuje i uruchamia bazę wiedzy grzybki. Poniższy przykład można uruchomić jako makro np. w programie Microsoft Word.

```
' Przykład wykorzystania systemu PC-Shell jako obiektu COM
'
Dim v As Object
Dim kbID As Integer

'Inicjalizacja obiektu
Set v = CreateObject("Sphinx.PCShell")

' Inicjalizacja bibliotek
v.kbInitialize(0)

'Otwarcie bazy wiedzy
kbID = v.kbOpen(0, "C:\Sphinx 4.0\BW\grzyby1.bw", 0)

'Uruchomienie programu
v.kbRunProgram (kbID)
'zamknięcie bazy wiedzy
v.kbClose(kbID)
'zwolnienie zasobów
v.kbDone
```

OBSŁUGA AUTOMATYZACJI W JĘZYKU SPHINX

Język Sphinx został wyposażony w nowy typ danych **variant** oraz instrukcje umożliwiające komunikację z dowolnymi serwerami automatyzacji OLE. Poniżej przedstawimy opisy 5 instrukcji zapewniających komunikację OLE

```
oleCreateObject( V, Name );
```

Instrukcja **oleCreateObject** umożliwia utworzenie nowej instancji obiektu o nazwie i klasie określonej w parametrze *Name*. W wyniku jej wykonania tworzony jest obiekt (instancja tego obiektu), którego identyfikator zapisywany jest w parametrze *V* – zmiennej typu *variant*.

Przykład :

```
// Utworzenie obiektu automatyzacji do arkusza kalkulacyjnego
variant ExcelApplication;
oleCreateObject( ExcelApplication, "Excel.Sheet" );
```

```
oleFunction( V, FnName, Dst, .... );
```

Instrukcja **oleFunction** wywołuje funkcję obiektu automatyzacji OLE (*V*) o nazwie określonej w parametrze *FnName* i po jej prawidłowym wykonaniu zwraca rezultat wykonania funkcji do zmiennej *Dst*. Zmienna *Dst* może być dowolnego typu – w zależności od tego jaki wynik zwraca wywoływana funkcja. Pierwsze trzy parametry są wymagane, natomiast kolejne mogą wystąpić jeżeli funkcja wymaga większej ilości parametrów.

Przykład :

```

variant WordApplication, WordDocuments, WordDocument;
oleCreateObject( WordApplication, "Word.Application" );
olePropertyGet( WordApplication, "Documents", WordDocuments );

// Dodanie nowego - pustego dokumentu
oleFunction( WordApplication, "Add", WordDocument );

```

```

oleProcedure( V, FnName, .... );

```

Instrukcja **oleProcedure** wywołuje procedurę obiektu automatyzacji OLE (*V*) o nazwie określonej w parametrze *FnName*. Kolejne parametry są opcjonalne i zależą od wywoływanej procedury serwera automatyzacji OLE.

Przykład :

```

// ustaw aktywną komórkę w arkuszu na B3
oleProcedure( ExcelApplication, "Goto", "W3K2" );

```

```

olePropertSet( V, FnName, Value, ... );
olePropertyGet( V, FnName, Value, ... )

```

Instrukcje służą odpowiednio do ustawiania lub pobierania właściwości obiektu OLE. Parametr *Value* może być dowolnego typu zgodnego z wywoływaną właściwością. Pozostałe parametry po *Value* są opcjonalne.

Przykład :

```

// ustaw w aktywnej komórce formułę "=A1*3,14"
olePropertyGet(ExcelApplication,"ActiveCell",Range );
olePropertySet(Range,"FormulaR1C1","=W1K1*3,14" );

```

PRZYKŁAD OBSŁUGI AUTOMATYZACJI OLE

Poniżej przedstawiamy instrukcje bloku control które tworzą nowy arkusz kalkulacyjny pakietu Microsoft Office, wpisują wartość 4 do komórki A1 oraz formułę do komórki A2 a następnie pobierają wynik formuły z arkusza i prezentują na ekranie.

```

variant ExcelApplication,ExcelWorkbooks,ExcelWorkbook,Range;

// utwórz obiekt Excel.Application
oleCreateObject( ExcelApplication, "Excel.Application" );

// pokaż aplikację na ekranie 0 - wyłącza
olePropertySet(ExcelApplication,"Visible", 1);

olePropertyGet(ExcelApplication,"Workbooks",ExcelWorkbooks);
// dodaj nowy pusty arkusz
oleFunction(ExcelWorkbooks,"Add", ExcelWorkbook );

//
oleProcedure( ExcelApplication, "Goto", "W1K1" );
olePropertyGet(ExcelApplication,"ActiveCell",Range );
olePropertySet(Range,"Value","4" );

oleProcedure( ExcelApplication, "Goto", "W1K2" );
olePropertyGet(ExcelApplication,"ActiveCell",Range );
olePropertySet(Range,"FormulaR1C1","=W1K1*3,14" );
double D;
olePropertyGet(Range,"Value",D);
char S;

```

```
ntos( D, S );  
messageBox(0,0,S,S);
```

INTEGRACJA SYSTEMU PC-SHELL Z INNYMI APLIKACJAMI

System PC-Shell oferuje innym aplikacjom (nie będącym elementami pakietu Sphinx), pracującym w środowisku MS-Windows, możliwość dostępu do funkcji eksportowalnych, zawartych w bibliotekach DLL. Poniżej przedstawiony zostanie szczegółowy opis poszczególnych funkcji bibliotecznych oraz sposób ich wykorzystania z poziomu dowolnego języka programowania. Wraz z systemem dostarczane są nagłówki funkcji w językach C oraz Pascal – do wykorzystania m.in. w Delphi. W przypadku języka C, aby móc wykorzystać wyżej wymienione funkcje, należy dołączyć do projektu aplikacji plik importowy *cls30.lib*. W przypadku aplikacji tworzonych w języku Pascal lub Delphi dołączenie biblioteki *pcshell.pas* spowoduje automatyczne zaimportowanie funkcji. Aby aplikacja działała poprawnie, poza biblioteką *cls30.dll* potrzebne będą również następujące biblioteki: *tr30.dll*, *r30pl.dll*, *files.dll*, *bckp.dll*, *aitool30.dll*, *app30.dll*, *cedit30.dll*.

FUNKCJE BIBLIOTECZNE SYSTEMU PC-SHELL

```
(C)    int kbInitialize( HINSTANCE hInstance );
(Pascal) function kbInitialize( hInstance: integer ): integer;
```

Funkcja służy do zainicjowania systemu PC-Shell i musi być wywołana z aplikacji klienta najczęściej na początku działania programu. Dokonuje ona procesu inicjalizacji, rejestracji i rezerwacji zasobów. Na zakończenie działania aplikacji należy wywołać funkcję *kbDone*.

```
(C)    int kbDone();
(Pascal) function kbDone: integer;
```

Funkcja dokonuje zwolnienia z pamięci zasobów systemu PC-Shell. Powinna być wywołana na koniec działania programu klienta.

```
(C)    int kbOpen( HINSTANCE hInstance, char *pFileName, HWND hwnd );
(Pascal) function kbOpen( hInstance: integer; pFileName: pchar; wnd: integer ): integer;
```

Funkcja powoduje załadowanie nowej bazy wiedzy o nazwie *pFileName*. Ostatni parametr jest uchwytem okna macierzystego aplikacji klienta (będzie on „rodzicem” dla okien pojawiających się w trakcie pracy systemu). Wywołanie tej funkcji powoduje przydzielenie identyfikatora załadowanej bazy (zwracany jako rezultat wywołania funkcji), którym należy posługiwać się we wszelkich wywołaniach podanych poniżej funkcji. W przypadku błędnego załadowania bazy zwracany jest identyfikator *-1*. Otwarcie bazy wiedzy nie powoduje jej uruchomienia! Otwartą bazę na koniec należy zawsze zamknąć za pomocą funkcji *kbClose*.

Przykład:

```
ID = kbOpen( hInst, "C:\\baza.bw", hWindow );
```

```
(C)    int kbClose( int bwId );
(Pascal) function kbClose( bwId: integer ): integer;
```

Funkcja zamyka bazę wiedzy identyfikowaną przez identyfikator *bwId*, usuwa ją z pamięci zwalniając wszelkie przydzielone zasoby. Po wywołaniu tej funkcji identyfikator *bwId* staje się nieaktualny.

```
(C)    int kbSetParent( int bwId, HWND hWnd );
(Pascal) function kbSetParent( bwId: integer; hWnd: integer ): integer;
```

Funkcja ta służy do zmiany uchwytu okna „rodzica” dla okien systemu PC-Shell (domyślnie rodzicem jest okno podane przy wywołaniu funkcji *kbOpen*, jednak podczas pracy aplikacji można dynamicznie zmieniać identyfikator rodzica właśnie przy użyciu funkcji *kbSetParent*).

```
(C)    int kbRunProgram( int bwId );
(Pascal) function kbRunProgram( bwId: integer ): integer;
```

Funkcja powoduje wykonanie bloku sterowania (jeżeli istnieje) załadowanej bazy wiedzy identyfikowanej przez numer *bwId*. Należy zastrzec, że deklaracja *RUN*, zawarta w bloku sterowania, nie ma wpływu na jego automatyczne uruchomienie w wyniku wywołania funkcji *kbOpen*, musi on być zawsze uruchomiony jawnie, tj. za pomocą funkcji *kbRunProgram*. Program może być uruchamiany dowolną ilość razy bez konieczności ponownego ładowania bazy wiedzy.

```
(C)    int kbCallFunction( int bwId, char *pFunctionCall );
(Pascal) function kbCallFunction( bwId: integer; pFunctionCall: pchar ): integer;
```

Funkcja umożliwia wywołanie dowolnej funkcji użytkownika zdefiniowanej w bloku sterowania. Jako drugi parametr należy podać postać wywołania funkcji wraz z listą argumentów. Argumenty funkcji mogą być użyte w postaci jawnej lub w postaci nazw zmiennych, zadeklarowanych w bloku sterowania. Wartości zmiennych można wcześniej ustawić za pomocą funkcji *kbSetVariable*. Poprzez wywołania funkcji bloku sterowania twórca aplikacji ma praktycznie pełny dostęp do dowolnych instrukcji języka Sphinx.

Przykład:

```
// blok sterowania bazy wiedzy:
char S; // zmienna globalna
function zapiszParametry( char NazwaPliku )
begin
    // definicja funkcji
    ...
end;
```

Wywołanie z programu użytkownika:

```
kbCallFunction( bwID, "zapiszParametry(\"konfig1.ini\")" );
kbSetVariable( bwID, "S", "konfig2.ini" );
kbCallFunction( bwID, "zapiszParametry( S )" );
```

```
(C)    int kbSetVariable( int bwId, char *pVarName, char *pValue );
        int kbGetVariable( int bwId, char *pVarName, char *pDest, int iSize );
(Pascal) function kbSetVariable
( bwId: integer; pVarName: pchar; pValue: pchar ): integer;
function kbGetVariable
( bwId: integer; pVarName: pchar; pDest: pchar; iSize: integer ): integer;
```

Funkcje *kbSetVariable* i *kbGetVariable* służą do ustawiania i pobierania zawartości zmiennych z bloku sterowania. W przypadku zmiennych numerycznych następuje automatyczna konwersja łańcucha na liczbę, w przypadku liczb rzeczywistych akceptowany jest również przecinek jako separator części ułamkowej. Nazwa zmiennej podawana w wywołaniu tych funkcji może dotyczyć zmiennych prostych, pojedynczych elementów tablicy lub pól rekordów.

Funkcja *kbGetVariable* zwraca długość wartości tekstowej zmiennej lub wartość *-1*, która oznacza, że bufor na wartość zmiennej jest za mały. Podanie jako adresu bufora wartości *NULL* (C) lub *nil* (Pascal) spowoduje zwrócenie długości wartości zmiennej – bez znaku kończącego łańcuch *"\0"*.

Przykład:

```
char *ptr;
int len;
kbSetVariable( bwId, "Variable", "Nowa wartość zmiennej" );
len = kbGetVariable( bwId, "Variable", NULL, 0 );
ptr = new char[ len + 1 ];
kbGetVariable( bwId, "Variable", ptr, len + 1 );
...
delete ptr;
```

```
(C)    int kbGoal( int bwId, char *pHypothesis );
(Pascal) function kbGoal( bwId: integer; pHypothesis: pchar ): integer;
```

Funkcja służy uruchomieniu procesu wnioskowania wstecz i próby potwierdzenia postawionej hipotezy. Hipotezę podajemy w postaci trójki OAW. System PC-Shell wnioskuje na zawartych w pamięci regułach i faktach, w przypadku braku wystarczających danych w przypadku wnioskowania interaktywnego system generuje zapytania do użytkownika, za pomocą standardowych okien systemu PC-Shell, a po zakończeniu procesu wnioskowania – wyświetla okno rozwiązania. W przypadku pracy nieinteraktywnej funkcja *kbGoal* w momencie zadania pytania lub wygenerowania rozwiązania powraca. Stan procesu wnioskowania należy sprawdzić za pomocą funkcji *kbGetRequest* z parametrem *[status]* – system PC-Shell zwróci wartość *[WAITING_FOR_ANSWER]* lub wartość *[PROBLEM_RESOLVED]*. Sposób konstruowania pętli zadawania pytań, treści zapytania oraz pobierania wartości, jest opisany w przykładach poniżej. Proces nieinteraktywny wymaga wywołania funkcji *kbAnswerIDX* lub *kbAnswerStr* i wznowienia procesu funkcją *kbReGoal*.

Przykład:

```
bwId := kbOpen( hInstance, 'C:\BW\GRZYBY1.BW', Handle );
kbSetParent( bwId, Handle );
if ( bwId <> -1 ) then
begin
    kbGoal( bwId, 'rodzaj=X' );
    kbClose( bwId );
end;
```

```
(C)    int kbReGoal( int bwId );
(Pascal) function kbReGoal( bwId: integer ): integer;
```

Funkcja służy do wznowienia procesu wnioskowania w trybie nieinteraktywnym. Po podaniu odpowiedzi na zapytanie za pomocą funkcji *kbAnswerIDX* lub *kbAnswerStr*, proces wnioskowania musi być wznowiony za pomocą opisywanej funkcji.

```
(C)    int kbResetGoal( int bwId );
(Pascal) function kbResetGoal( bwId: integer ): integer;
```

Funkcja *kbResetGoal* przerywa nieinterakcyjny proces wnioskowania. Powinna być wywołana przed uruchomieniem nowego procesu wnioskowania, ale jedynie w przypadku gdy proces jest w trakcie zadawania pytań. Po wygenerowaniu rozwiązania nie potrzeba przerywać wnioskowania.

```
(C)    int kbAnswerIDX( int bwId, int iIdx );
        int kbAnswerStr( int bwId, char *pAnswer );
(Pascal) function kbAnswerIDX( bwId: integer; iIdx: integer ): integer;
        function kbAnswerStr( bwId: integer; pAnswer: pchar ): integer;
```

Funkcje służą do podania odpowiedzi na zapytanie w czasie wnioskowania nieinteraktywnego. Pierwsza postać używana jest do wyboru odpowiedzi z listy wartości wygenerowanych przez system PC-Shell, może ona być użyta wielokrotnie w przypadku wartości typu *someof* (dopuszczalna jest wtedy również odpowiedź *-1* równoznaczna z wartością *none*). Druga funkcja służy do podania wartości zmiennej.

Po podaniu odpowiedzi na zapytanie należy wznowić proces wnioskowania funkcją *kbReGoal*.

```
(C)    int kbSetInteractive( int bwId, int iMode );
(Pascal) function kbSetInteractive( bwId: integer; iMode: integer ): integer;
```

Funkcja zmienia tryb wnioskowania na tryb interakcyjny (*iMode = 1*) tzn. na taki, w którym PC-Shell zadaje pytania i wyświetla rozwiązania w standardowych oknach systemu. Tryb nieinterakcyjny (*iMode = 0*) umożliwia przechwytywanie zapytań i rozwiązań do własnych okien dialogowych. Przykładowe sesje konsultacji pokazane są poniżej oraz w plikach demonstracyjnych dostarczanych wraz z systemem Sphinx.

```
(C)    int kbSolve (int bwId, char * pSource, char * pHypothesis );
(Pascal) function kbSolve( bwId: integer; pSource: pchar; pHypothesis: pchar ):
integer;
```

Funkcja analogiczna do instrukcji *solve* języka Sphinx, powoduje załadowanie źródła wiedzy o nazwie określonej parametrem *pSource* i uruchomienie procesu wnioskowania z podaną hipotezą (*pHypothesis*). Po zakończeniu wnioskowania źródło jest usuwane z pamięci. Funkcja *kbSolve* pracuje zawsze w trybie **interakcyjnym**, nie ma możliwości uruchomienia za pomocą tej instrukcji trybu nieinterakcyjnego.

Przykład:

```
kbSolve( bwId, 'profil', 'profil_klienta = PK' );
kbSolve( bwId, 'gwarancje', 'gwarancje_kredytowe = GK' );
kbSolve( bwId, 'sytfina', 'sytuacja finansowa = SF' );
kbSolve( bwId, 'deckred', 'decyzja_kredytowa = DK' );
```

```
(C)    int kbAddFactStr( int bwId, char *fact );
        int kbAddFact( int bwId, char *pO, char *pA, char *pW );
(Pascal) function kbAddFactStr( bwId: integer; fact: pchar ): integer;
        function kbAddFact( bwId: integer; pO: pchar; pA: pchar; pW: pchar ):
integer;
```

Funkcje umożliwiają dodanie faktów do bazy wiedzy czy to w postaci pełnej trójki OAW (*kbAddFactStr*) lub podając poszczególne nazwy obiektu, atrybutu i wartości (*kbAddFact*).

Przykład:

```
kbAddFactStr( bwId, "wskaźnik_szybki = 3.4" );
kbAddFactStr( bwId, "obrót = 2000" );
kbGoal( bwId, "kondycja_financeowa = X" );
```

```
(C)    int kbGetSource( int bwId, char *pSrcName );
        int kbFreeSource( int bwId, char *pSrcName );
(Pascal) function kbGetSource( bwId: integer; pSrcName: pchar ): integer;
        function kbFreeSource( bwId: integer; pSrcName: pchar ): integer;
```

Funkcja *kbGetSource* umożliwia załadowanie eksperckiego źródła wiedzy do pamięci. Nazwa źródła podawana jako drugi argument jest nazwą symboliczną zdefiniowaną w bloku *sources* głównej bazy wiedzy. Jednocześnie w pamięci może być załadowane tylko jedno źródło, do zwolnienia źródła już zbędnego służy funkcja *kbFreeSource*.

Przykład:

```
kbGetSource( bwId, profil );
kbGoal( bwId, "profil_klienta = X" );
kbFreeSource( bwId, profil );
kbGetSource( bwId, gwarancje );
kbGoal( bwId, "gwarancje_kredytowe = X" );
kbFreeSource( bwId, gwarancje );
```

```
(C)    char *kbGetRequest( int bwId, char *pKey );
(Pascal) function kbGetRequest( bwId: integer; pKey: pchar ): pchar;
```

Funkcja *kbGetRequest* jest funkcją, która umożliwia kontrolowanie stanu systemu PC-Shell, pobieranie wartości zapytań, wartości możliwych odpowiedzi i innych informacji potrzebnych do obsługi m.in. procesu wnioskowania. Funkcja zawsze zwraca wartość jako łańcuch tekstowy, który należy zdublować w swoim systemie ponieważ każde następne wywołanie tej funkcji dla konkretnej bazy powoduje zniszczenie poprzedniej wartości.

Dopuszczalne wartości o które system może zapytać to:

[status] – zwracana jest wartość określająca stan systemu (możliwe wartości to *[READY]*, *[WAITING_FOR_ANSWER]* lub *[PROBLEM_RESOLVED]*);

[error] – tekst wyjaśniający błąd;

[question] – tekst zapytania w trakcie konsultacji;

[answer_type] – zwraca typ zapytania (*[LIST_ONEOF]* – lista odpowiedzi dla atrybutu typu *oneof*, *[LIST_SOMEOF]* – lista wartości dla atrybutu typu *someof* lub *[VARIABLE]* – zapytanie o zawartość zmiennej);

[variable] – zwracana jest nazwa zmiennej tymczasowej użytej w czasie wnioskowania o wartość której system wygenerował zapytanie;

[solution] – kolejny element rozwiązania (pełna trójka OAW), lista kończona jest wartością *[END]*;

[values] – służy do pobrania kolejnego elementu z listy możliwych odpowiedzi na zapytanie;

[attribute] – nazwa atrybutu którego dotyczy zapytanie;

[how] – pobranie kolejnej linii tekstu wyjaśnień typu „jak?”.

PRZYKŁADY WYKORZYSTANIA FUNKCJI BIBLIOTECZNYCH

Przykład 1. Procedura ładująca i uruchamiająca bazę wiedzy *kredyt1.bw*

```
procedure Przyklad1;
var
  bwId: integer;
begin
  chdir( 'C:\\ Sphinx 4.0\\BW' );
  { 1. Inicjalizacja bibliotek. Wywołanie to powinno
    być dokonane jeden raz, np. w trakcie uruchamiania
    programu. }
  kbInitialize( hInst );
  { 2. Otwarcie bazy wiedzy }
  bwId := kbOpen( hInst, 'C:\\SPHINX95\\BW\\KREDYT1.BW',
                  Handle );
  if ( bwId <> -1 ) then
  begin
    { 3. Uruchomienie programu zawartego w bloku sterowania }
    kbRunProgram( bwId );
    { 4. Zamknięcie bazy wiedzy }
    kbClose( bwId )
  end;
  { 4. Deinicjalizacja bibliotek DLL systemu PC-Shell.
    Powinna być wywołana np. w momencie zamykania aplikacji
    klienta }
  kbDone;
end;
```

Przykład 2. Sposób wywołania funkcji zdefiniowanych w bloku sterowania bazy wiedzy oraz ustawiania wartości zmiennych bloku sterowania.

```
procedure Przyklad2;
var
  bwId: integer;
  zmienna: array [0..100] of char;
begin
  chdir( 'C:\\Sphinx 4.0\\BW' );
  kbInitialize( hInst );
  bwId := kbOpen(hInst, 'C:\\Sphinx 4.0\\BW\\EXPORT.BW', Handle );
  if ( bwId <> -1 ) then
  begin
    { Wywołanie funkcji helloWorld z bloku sterowania }
    kbCallFunction( bwId, 'helloWorld' );
    { Ustawienie wartości zmiennej Variable na wartość
      kbSetVariable }
    kbSetVariable( bwId, 'Variable', 'kbSetVariable' );
    { Wywołanie funkcji showValue z parametrem Variable }
    kbCallFunction( bwId, 'showValue(Variable)' );
    { Pobranie wartości zmiennej Variable }
    kbGetVariable( bwId, 'Variable', zmienna, 100 );
    messagebox( Handle, zmienna, '<Variable>', MB_OK );
    kbClose( bwId )
  end;
  kbDone
end;
```

Przykład 3. Uruchomienie procesu wnioskowania w trybie interakcyjnym.

```
procedure Przyklad3;
var
  bwId: integer;
begin
  chdir( 'C:\\Sphinx 4.0\\BW' );
  kbInitialize( hInst );
  bwId := kbOpen( hInst, 'C:\\Sphinx 4.0\\BW\\GRZYBY1.BW',
                  Handle );
  if ( bwId <> -1 ) then
  begin
    { Uruchomienie wnioskowania z hipotezą "rodzaj=X" }
    kbGoal( bwId, 'rodzaj=X' );
    kbClose( bwId )
  end;
  kbDone
end;
```

Przykład 4. Sposób implementacji wnioskowania z wykorzystaniem okien dialogowych zdefiniowanych przez użytkownika. *AskDlg* oraz *SolutionDlg* to okna dialogowe użytkownika (proszę zwrócić uwagę na wywołania funkcji *kbGetRequest*).

```

procedure Przyklad4( base: pchar; hypothesis: pchar );
var
  bwId, SolCount, Res: integer;
  Ptr, Question, Answer, Value: pchar;
begin
  chdir( 'C:\Sphinx 4.0\BW' );
  kbInitialize( hInst );
  bwId := kbOpen( hInst, base, Handle );
  { 1. Ustawienie wnioskowania na tryb nieinterakcyjny }
  if ( bwId <> -1 ) then
    begin
      kbSetInteractive( bwId, 0 );
      { 2. Uruchomienie procesu wnioskowania }
      Res := kbGoal( bwId, hypothesis );
      { 3. Badanie stanu procesu wnioskowania }
      Ptr := kbGetRequest( bwId, '[status]' );
      while ( strcmp( Ptr, '[WAITING_FOR_ANSWER]' ) = 0 ) do
        begin
          { 4. Pobranie treści zapytania }
          Question := kbGetRequest( bwId, '[question]' );
          AskDlg.Pytanie.SetTextBuf( Question );
          { 5. Badanie typu odpowiedzi }
          Answer := kbGetRequest( bwId, '[answer_type]' );
          if ( strcmp( Answer, '[LIST_ONEOF]' ) = 0 ) then
            begin
              { 6. Pobieranie listy wartości/odpowiedzi }
              AskDlg.ListaOdpowiedzi.Items.Clear;
              repeat
                Value := kbGetRequest( bwId, '[values]' );
                if ( strcmp( Value, '[END]' ) <> 0 ) then
                  AskDlg.ListaOdpowiedzi.Items.Add( StrPas( Value ) )
              until ( strcmp( Value, '[END]' ) = 0 );
              { Wyświetlenie okna z zapytaniem }
              AskDlg.ShowModal;
              { 7. Udzielenie odpowiedzi na zapytanie }
              { ... }
              kbAnswerIDX( bwId, AskDlg.ListaOdpowiedzi.ItemIndex );
              { 8. Wznowienie procesu wnioskowania }
              kbReGoal( bwId );
            end;
          { 3. ... }
          Ptr := kbGetRequest( bwId, '[status]' );
        end;
      if ( strcmp( Ptr, '[PROBLEM_RESOLVED]' ) = 0 ) then
        begin
          SolutionDlg.ListaRozwiazan.Items.Clear;
          SolCount := 0;
          { Pobieramy listę rozwiązań }
          repeat
            Value := kbGetRequest( bwId, '[solution]' );
            if ( strcmp( Value, '[END]' ) <> 0 ) then
              begin
                SolutionDlg.ListaRozwiazan.Items.Add( StrPas( Value ) );
                SolCount := SolCount + 1
              end;
            until ( strcmp( Value, '[END]' ) = 0 );
        end;
    end;

```

```
if ( SolCount > 0 ) then
begin
  { Pobieramy listę rozwiązań "jak?" }
  SolutionDlg.ListaRozwiazan.Items.Add('Wyjaśnienia JAK?');
  repeat
    Value := kbGetRequest( bwId, '[how]' );
    if ( strcmp( Value, '[END]' ) <> 0 ) then
      SolutionDlg.ListaRozwiazan.Items.Add( StrPas(Value) )
    until ( strcmp( Value, '[END]' ) = 0 )
  end
  else
    SolutionDlg.ListaRozwiazan.Items.Add( 'Brak rozwiązań' );
  SolutionDlg.ShowModal
end;
kbClose( bwId )
end;
kbDone
end;
```