1. Wprowadzenie do programowania logicznego

We współczesnych teoriach rozwiązywania zadań i programowania komputerów stosuje się tradycyjne metody logiki, posługując się **klauzulową** odmianą języka logiki. Język klauzul jest bowiem prostszy, niż standardowy język logiki, dorównując mu jednocześnie siłą wyrazu i w większym stopniu zbliżony jest do innych formalizmów stosowanych w przetwarzaniu danych i programowaniu komputerów. Pojęcie klauzuli wprowadzimy najpierw na przykładach, a następnie sprecyzujemy go, wykorzystując zarówno przykłady, jak i formalizm zaproponowane przez R. Kowalskiego[1] w 1979 r.

Logika bada związki implikacji zachodzące między założeniami a konkluzjami. Przykładowo, na mocy praw logiki z założeń:

Robert lubi logikę

oraz

Robert lubi każdego, kto lubi logikę

wynika konkluzja

Robert lubi samego siebie

natomiast nie wynika konkluzja

Robert lubi tylko tych ludzi, którzy lubią logikę.

Przedmiotem zainteresowania logiki jest nie prawdziwość, fałszywość lub akceptowalność poszczególnych zdań, lecz zależności zachodzące między nimi. Jeśli pewna teza jest implikowana przez założenia prawdziwe czy też akceptowalne według jakiegoś kryterium, logika każe nam przyjąć ją jako konkluzję. Jeśli natomiast założenia implikują niemożliwą do zaakceptowania lub fałszywą tezę, to — chcąc pozostać w zgodzie z logiką — powinniśmy odrzucić przynajmniej jedno z założeń. Tak więc jeśli nie godzimy się z konkluzją, że Robert lubi samego siebie, logika zmusza nas do rezygnacji z założenia, że Robert lubi logikę, albo z założenia, że Robert lubi każdego, kto lubi logikę.

W celu wykazania, że założenia implikują pewną konkluzję, konstruuje się dowód złożony z kroków wnioskowania. Aby dowód był przekonywający, poszczególne jego kroki muszą być proste i oczywiste oraz powinny poprawnie łączyć się ze sobą. Dla osiągnięcia tego celu jest niezbędne, by zdania były jednoznaczne; jest też pożądane, by ich składnia była możliwie prosta. Wymaganie prostoty i jednoznaczności języka dowodów uzasadnia zastosowanie specjalnego języka symbolicznego w miejsce języka naturalnego.

[1] R. Kowalski, *Logika w rozwiązywaniu zadań*, WNT, Warszawa 1989.

Symboliczny język klauzul jest nader nieskomplikowany. Najprostszymi jego wyrażeniami są *zdania atomowe*, określające zależności między obiektami indywiduowymi:

Robert lubi, logikę.

Jan lubi Marię.

Jan jest o 2 lata starszy niż Maria.

(Wyrazy podkreślone składają się na nazwę zależności, nie podkreślone są nazwami indywiduów.) Bardziej złożone zdania wyrażają związek implikacji między atomowym warunkiem a atomową konkluzją:

Maria lubi Jana, jeśli Jan lubi Marię. Robert lubi ***x****, jeśli* ***x*** *lubi logikę.*

Zmienna ***x*** oznacza dowolne indywiduum. Zdanie może mieć kilka warunków, które muszą zachodzić jednocześnie, lub kilka możliwych konkluzji:

Maria lubi Jana lub Maria lubi Roberta, jeśli Maria lubi ***x****.*

(Maria lubi Jana lub Roberta, jeśli w ogóle kogoś lubi.)

x *lubi Roberta, jeśli* ***x*** *jest studentem Roberta i* ***x*** *lubi logikę.*

Zdania takie są również zwane **klauzulami**. Ogólnie, każda klauzula wyraża fakt, iż pewna liczba (być może zero) warunków łącznie implikuje alternatywę pewnej liczby (być może zera) konkluzji. Poszczególne warunki i konkluzje wyrażają zależności między indywiduami. Indywidua mogą być ustalone i oznaczone takimi słowami, jak

Robert, Jan, logika czy *2,*

które nazywamy (może nieco niefortunnie) **stałymi**; mogą też być nieokreślone i oznaczone zmiennymi:

u, v, w, x, y, z

Później rozważymy bardziej złożone nazwy indywiduów, które można konstruować za pomocą **symboli funkcyjnych**

ojciec(Jan) (tzn. ojciec Jana)

ułamek (3, 4) (tzn. ułamek ¾).

Na podstawie tych przykładów powinna już być widoczna wielka prostota języka klauzul logicznych w porównaniu z językiem naturalnym. Zaskakujące, że pomimo to język klauzul nie traci wiele z siły wyrazu języka naturalnego.

1.1. Język klauzul na przykładzie języka służącego opisowi związków rodzinnych.

Formuły atomowe, które występują jako warunki i konkluzje klauzul, wygodniej będzie zapisywać w uproszczonej, aczkolwiek nieco mniej naturalnej formie[2].

Formułę atomową rozpoczyna nazwa **relacji**, po której następuje ciąg nazw indywiduów pozostających ze sobą w tej relacji. Piszemy zatem:

[2] R. Kowalski, *Logika w rozwiązywaniu zadań*, WNT, Warszawa 1989.

Ojciec (Zeus, Ares) zamiast *Zeus jest ojcem Aresa*, oraz

Królowa (Harmonia) zamiast *Harmonia jest królową*.

Mówiąc ściśle, *Królowa* jest nazwą własności indywiduum, a nie relacji między indywiduami. Jednakże dla uproszczenia terminologii, również własności (zwane też **predykatami**) będziemy uważali za relacje. Ponadto, mówiąc o nazwie relacji będziemy używali terminu **symbol relacyjny** (predicate symbol). Do oznaczenia **implikacji** zastosujemy symbol , co należy czytać „jeśli". Na przykład zapis

*Płeć-żeńska (**x**) Matka (**x** , **y**)*

stwierdza, że ***x*** jest płci żeńskiej, jeśli ***x*** jest matką ***y***.

W celu ujednolicenia notacji, a w przyszłości także reguł wnioskowania, wygodniej będzie uważać wszystkie klauzule za implikacje, jeśli nawet nie mają żadnych warunków lub żadnych konkluzji. Napiszemy zatem:

Ojciec{Zeus, Ares)

zamiast

Ojciec {Zeus, Ares)

Implikacja bez konkluzji to negacja. Klauzula

Kobieta(Zeus)

wyraża fakt, że Zeus nie jest kobietą.

Poniższe klauzule opisują niektóre własności i związki rodzinne postaci z mitologii:

(F1) *Ojciec {Zeus, Ares)*
(F2) *Matka {Hera, Ares)*
(F3) *Ojciec (Ares, Harmonia)*
(F4) *Matka (Afrodyta, Harmonia)*
(F5) *Ojciec (Kadmos, Semele)*
(F6) *Matka (Harmonia, Semele)*
(F7) *Ojciec (Zeus, Dionizos)*
(F8) *Matka (Semele, Dionizos)*
(F9) *Bóg (Zeus)*
(F10) *Bóg (Hera)*
(F11) *Bóg (Ares)*
(F12) *Bóg (Afrodyta)*
(F13) *Królowa (Harmonia)*

Znaczenie tych klauzul powinno być intuicyjnie oczywiste. Następne cztery klauzule pozwalają uściślić to znaczenie w drodze nałożenia pewnych ograniczeń:

(F14) *Pleć-żeńska (x) Matka (x, y)*
(F15) *Płeć-męska (x) Ojciec (x, y)*
(F16) *Dziecko (y, x) Matka (x, y)*
(F17) *Dziecko (y, x) Ojciec (x, y)*

Klauzule te stwierdzają, że dla dowolnych x i y:

x jest płci żeńskiej, jeśli x jest matką y
x jest płci męskiej, jeśli x jest ojcem y
y jest dzieckiem x, jeśli x jest matką y
y jest dzieckiem x, jeśli x jest ojcem y

Zmienne w odrębnych klauzulach są różne, jeśli nawet mają takie same nazwy. Tak więc zmienna ***x*** w klauzuli (F14) nie ma żadnego związku ze zmienną ***x*** w klauzuli (F15). Nazwa zmiennej jest istotna tylko w kontekście

klauzuli, w której występuje. Dwie klauzule różniące się tylko nazwami występujących w nich zmiennych są sobie równoważne i mówimy o nich, że jedna stanowi **wariant** drugiej.

W języku klauzul wszystkie warunki klauzuli stanowią koniunkcję (tzn. są połączone spójnikami „i"), natomiast wszystkie konkluzje klauzuli stanowią alternatywę (tzn. są połączone spójnikami „lub"). Spójniki „i" oraz „lub" można więc bez obaw zastąpić przecinkami. Przecinki między warunkami należy czytać jako „i", a przecinki między konkluzjami jako „lub". Stąd klauzule

(F18) *Wnuk (x, y)* *Dziecko (x, z)* , Dziecko (z, y)
(F19) *Płeć-męska (x)* , *Płeć-żeńska (x)* *Człowiek (x)*

w których x, y i z są zmiennymi, stwierdzają, że dla dowolnych x, y, z

x jest wnukiem y, jeśli x jest dzieckiem z **i** *x jest dzieckiem y*
x jest płci męskiej **lub** *x jest płci żeńskiej, jeśli x jest człowiekiem*

Jeśli te same warunki implikują równocześnie kilka różnych konkluzji, dla każdej konkluzji będzie potrzebna osobna klauzula. Podobnie, jeśli ta sama konkluzja jest implikowana przez każdy z kilku warunków, dla każdego warunku będzie potrzebna osobna klauzula.

Na przykład zdanie:

Płeć-żeńska (x) **i** *Dziecko (y, x)* *Matka (x, y)*

ma równoważną postać klauzulową

Płeć-żeńska (x) *Matka (x, y)*
Dziecko (y, x) *Matka (x, y)*

Te dwie klauzule są domyślnie połączone spójnikiem „**i**"; czyli

x jest płci żeńskiej, jeśli x jest matką y **i** *y jest dzieckiem x, jeśli x jest matką y*

Podobnie zdanie:

Dziecko (y, x) *Matka(x, y)* ***lub*** *Ojciec (x, y)*

można wyrazić za pomocą klauzul

Dziecko (y, x) *Matka (x, y)*
Dziecko (y, x) *Ojciec (x, y)*

czyli

y jest dzieckiem x, jeśli x jest matką y **i** y jest dzieckiem x, jeśli x jest ojcem y.

Symbole relacyjne mogą oznaczać zależności między więcej niż dwoma indywiduami.

Na przykład, formuły atomowej

Rodzice(x, y, z)

użyjemy w celu wyrażenia faktu, że x jest ojcem z, a y jest matką z, tj.

Rodzice(x, y, z) *Ojciec (x, z) , Matka (y, z)*

1.2. Dokładniejsza definicja języka klauzul.

Zdefiniujemy teraz bardziej precyzyjnie **składnię** (gramatykę) języka klauzul, wskazując jednocześnie na jego powiązania z językiem naturalnym[3].

Klauzula jest wyrażeniem postaci

$B_1, B_2, ..., B_m \quad A_1, A_2, ..., A_n$

przy czym $B_1, B_2, ..., B_m, A_1, A_2, ..., A_n$ są formułami atomowymi, $n \geq 0$ i $m \geq 0$. Formuły atomowe $A_1, A_2, ..., A_n$ stanowią koniunkcję warunków klauzuli, a $B_1, B_2, ..., B_m$– alternatywę konkluzji.

Jeśli klauzula zawiera zmienne $x_1, ..., x_k$, należy ja interpretować jako stwierdzenie, że dla wszystkich $x_1, ..., x_k$: B_1 lub ... lub B_m jeśli A_1 i ... i A_n.

Jeśli n = 0, klauzulę należy interpretować jako bezwarunkowe stwierdzenie, że dla wszystkich $x_1, ..., x_k$: B_1 lub ... lub B_m .

Jeśli m = 0, klauzulę należy interpretować jako stwierdzenie , że dla wszystkich $x_1, ..., x_k$ nieprawda, że zachodzi A_1 i ... i A_n.

Jeśli m = n = 0, klauzulę zapisuje się w postaci □ (klauzula pusta) i interpretuje jako zdanie zawsze fałszywe.

Atom (lub formuła atomowa) to wyrażenie postaci: $R (t_1, ..., t_m)$, przy czym R jest m-argumentowym symbolem relacyjnym, $t_1, ..., t_m$ są termami oraz $m \geq l$.

Atom należy interpretować jako stwierdzenie, że relacja o nazwie R zachodzi między indywiduami o nazwach $t_1, ..., t_m$.

Term jest zmienną, stałą lub wyrażeniem postaci $f (t_1, ..., t_m)$, przy czym f jest m - argumentowym symbolem funkcyjnym, $t_1, ..., t_m$ są termami oraz $m \geq l$. (patrz: 2.2).

Zbiory symboli relacyjnych, symboli funkcyjnych, stałych i zmiennych mogą być dowolnymi zbiorami wzajemnie rozłącznymi. Przyjmiemy konwencję, w której małe litery

u, v, w, x, y, z oznaczają zmienne. Znaczenie innego rodzaju symboli można rozpoznać po ich pozycji w klauzuli.

Symbol implikacji w języku klauzul jest skierowany w kierunku odwrotnym niż w klasycznym języku logiki. Przyzwyczajenie każe pisać raczej: A B (jeśli A to B), a nie

B A (B jeśli A). Jednakże różnica jest nieistotna. Notację: B A stosuje się w celu wyeksponowania konkluzji klauzuli.

Poszczególne pozycje związane z symbolem relacyjnym lub funkcyjnym noszą nazwę jego **argumentów**. W atomie $R (t_1, ..., t_m)$ t_1 jest argumentem pierwszym, a t_m – argumentem ostatnim.

Aby móc odwoływać się do nieskończonej liczby indywiduów używając skończonej liczby klauzul, wprowadza się termy złożone. Na przykład liczby całkowite nieujemne mogą być reprezentowane termami

$$0,\ n (0),\ n (n (0) ... n (\underbrace{n (... n (0) ...)}_{\text{k razy}}), ...$$

0 jest stałą, zaś n jest jednoargumentowym symbolem funkcyjnym („n” to skrót od „następnik”). Term n(t) oznacza liczbę o jeden większą niż liczba, którą oznacza term t.

W porządku liczb całkowitych jest to następnik liczby t. Klauzule

(L1) *Liczba(0)*

(L2) *Liczba(n(x))* *Liczba(x)*

stwierdzają, że *0* jest liczbą oraz *n(x)* jest liczbą, jeśli jest nią *x*.

1.3. Zstępujący a wstępujący porządek definicji.

Definicje języka klauzul przedstawiono w porządku zstępującym. Pierwsza definicja objaśnia docelowe pojęcie klauzuli, odwołując się do pojęcia formuły atomowej, które staje się nowym pojęciem docelowym, a następna definicja sprowadza je do dwóch pojęć niższego rzędu – symbolu relacyjnego i termu. Pojęcie termu jest zdefiniowane rekurencyjnie i sprowadza się ostatecznie do pojęć: zmiennej, stałej i symbolu funkcyjnego. Postać tych symboli jest bez znaczenia pod warunkiem, że są od siebie wzajemnie odróżnialne i nie mogą być pomylone z symbolami „zastrzeżonymi":

, ( )

Przyjmiemy zatem, że symbole zastrzeżone nie występują wewnątrz innych symboli.

Zstępujący porządek definicji ma tę zaletę, że potrzeba każdej kolejnej definicji jest dobrze umotywowana. Wadą natomiast jest fakt, że definicje odwołują się do pojęć jeszcze nie zdefiniowanych, nie są więc w pełni zrozumiałe przy pierwszym ich czytaniu.

Wstępujący porządek definicji ma cechy przeciwne. Zaczyna się wówczas od pojęć, których się nie definiuje – „pojęć pierwotnych", a zatem niedefiniowalnych, albo pojęć wystarczająco zrozumiałych. Kolejne pojęcia są definiowane przez odwołanie do pojęć już wprowadzonych. Postępowanie kończy się po zdefiniowaniu pojęcia docelowego. Każda kolejna definicja jest od razu całkowicie zrozumiała, natomiast sens jej wprowadzenia może stać się jasny dopiero po osiągnięciu celu.

Rozróżnienie między podejściem wstępującym i zstępującym jest istotne nie tylko w przypadku definicji, ale także przy konstruowaniu i formułowaniu dowodów oraz pisaniu programów komputerowych. Dowód może być sformułowany tradycyjnym w matematyce sposobem wstępującym, tzn. rozumowanie wychodzi od danych założeń, z konkluzji są wyprowadzane dalsze konkluzje i proces ten kończy się po wyprowadzeniu tezy, która należało udowodnić. Dowód można także sformułować sposobem zstępującym, który odzwierciedla proces jego konstruowania, tzn. rozumowanie wychodzi od końcowej tezy, sprowadza ją do tez pomocniczych i tak dalej, do chwili, gdy wszystkie tezy pomocnicze będą rozpoznane jako prawdziwe.

Programy komputerowe również można pisać metodą wstępującą, zaczynając od elementarnych programów bezpośrednio zrozumiałych dla komputera i formułując nowe programy w kategoriach programów już istniejących. Na każdym etapie rozwoju programy takie mogą być wykonywane przez komputer, a zatem można je testować. Jeśli napisane wcześniej programy niższego rzędu nie dają możliwości konstruowania odpowiednich programów wyższego rzędu, muszą one ulec modyfikacji. Doświadczenie uczy, że lepiej jest pisać programy metodą zstępującą, formułując najpierw programy najwyższego rzędu w kategoriach nie istniejących jeszcze programów niższych rzędów. Te ostatnie pisze się później, co gwarantuje, że będą dobrze pasowały do danego

[3] R. Kowalski, *Logika w rozwiązywaniu zadań*, WNT, Warszawa 1989.

zagadnienia. Co więcej, programy niższego rzędu mogą być w przyszłości modyfikowane i ulepszane, i nie wpłynie to na resztę programu.

Obok problemu zastosowania symbolicznego języka logiki do reprezentowania informacji, ważne jest rozróżnienie między rozumowaniem wstępującym i zstępującym. Jest to rozróżnienie między analizą (postępowanie zstępujące) a syntezą (postępowanie wstępujące). Ponadto wprowadzenie wnioskowania zstępującego w miejsce wstępującego godzi ze sobą sposób rozumowania postulowany przez logikę i opinię psychologii co do rzeczywistego sposobu rozumowania człowieka.

Pojęcie rozumowania zstępującego ustala związek między strategią rozwiązywania zadań przez człowieka, polegającą na podziale zadania na mniejsze podzadania oraz metodą wykonywania programów komputerowych przez zastępowanie wywołania procedury treścią tej procedury. Stanowi łącznik między trzema dziedzinami badań: logiką, analizą sposobu myślenia człowieka i programowaniem komputerów.

1.4. Semantyka języka klauzul.

Pojęcie **składni** odnosi się do budowy zdań. W kontekście historycznym jest także związane z regułami wnioskowania i dowodami. **Semantyka** natomiast zajmuje się znaczeniem zdań. Tłumaczenie klauzul na język naturalny to jedynie nieformalny sposób badania ich semantyki.

W językach naturalnych odnosimy się do znaczenia słów i zdań dość niefrasobliwie.

W przypadku języka logiki jesteśmy ostrożniejsi. Jakiekolwiek znaczenie mogłoby być przypisane symbolowi relacyjnemu, stałej, symbolowi funkcyjnemu czy zdaniu, błędzie ono zawsze zrelatywizowane do zbioru zdań wyrażających wszystkie stosowne założenia. Przykładowo w przypadku związków rodzinnych, jeśli klauzule (F1) – (F19) wyrażają wszystkie założenia, nic nie wyklucza interpretacji, w której prawdziwe będzie twierdzenie

(F) Matka (Zeus, Ares)

Jest ono niesprzeczne z ustanowionymi założeniami (F1) – (F19), które są jedynym czynnikiem ograniczającym znaczenia, jakie wolno przypisać symbolom

Matka, *Ojciec*, *Zeus*, itd.

Aby wykluczyć możliwość (F), musimy wprowadzić dodatkowe założenia, np.

(F20) *Płeć-męska(x), Płeć-żeńska(x)*

Zdanie (F) jest niesprzeczne z (F1) – (F19), ale sprzeczne z (F1) – (F20).

Gdy dany jest zbiór klauzul wyrażający wszystkie założenia odnoszące się do pewnej dziedziny, aby ustalić sens któregokolwiek symbolu lub klauzuli, należy najpierw ustalić, jakie są logiczne konsekwencje założeń. Znaczenie symbolu relacyjnego takiego jak *Matka*, może być utożsamiane ze zbiorem wszystkich zdań będących logiczną konsekwencją założeń i zawierających ten symbol. Zatem znaczenie symbolu *Matka* w kontekście zdań (F1) – (F20) obejmuje negację

(F*) *Matka (Zeus, Ares)*

której nie obejmuje znaczenie symbolu *Matka* w kontekście (F1) – (F19).

Z powyższego wynika, że można w ogóle nie mówić o znaczeniu. Każda wypowiedz na temat znaczenia może być wyrażona równoważnie za pomocą pojęcia konsekwencji logicznej. Aby zdefiniować semantykę języka klauzul, wystarczy więc podać definicję konsekwencji.

Jeśli chcemy w języku klauzul wykazać, że zbiór założeń implikuje pewną konkluzję, to zakładamy, że konkluzja nie jest prawdziwa i pokazujemy, że negacja konkluzji jest sprzeczna z założeniami. Semantyka języka klauzul sprowadza się zatem do pojęcia sprzeczności. Na przykład, aby udowodnić, że zdanie (F*) jest częścią ustanowionego przez klauzule (F1) – (F20) znaczenia symbolu *Matka*, pokazujemy, że negacja (F*), czyli właśnie asercja (F), jest sprzeczna z (F1) – (F20). Sprowadzenie semantyki do pojęcia sprzeczności może wydawać się dziwaczne, ale ma niebagatelne zalety np. dla przetwarzania komputerowego.

Semantyka języka logiki oparta na pojęciu interpretacji nie zależy od reguł wnioskowania służących do manipulowania wyrażeniami języka. To odróżnia język logiki od ogromnej większości formalizmów stworzonych na gruncie informatyki, a w szczególności dla badań nad sztuczną inteligencją. Programy napisane w typowych językach programowania interpretuje się, określając ich wpływ na zachowanie się komputera. Ciężar przekształcania informacji spoczywa na programiście, który musi to zachowanie opisać. Natomiast kiedy programy są zapisane w języku logiki, można je interpretować, posługując się równoważnymi im zdaniami języka naturalnego, zrozumiałymi dla człowieka. W tym przypadku główny ciężar pracy spada na maszynę, która wykonując operacje (odpowiadające krokom wnioskowania) czysto mechanicznie, musi sprawdzić, czy informacja, którą wyraża program, implikuje logicznie istnienie rozwiązania danego zadania. Maszyna występuje w roli osoby rozwiązującej zadanie. Skonstruowanie dowodu, wykonanie programu i rozwiązanie zadania utożsamiają się ze sobą. Co więcej, strategia rozwiązywania przez człowieka zadań sformułowanych w języku naturalnym jest zbliżona do strategii, jaką stosuje komputer do zadań sformułowanych w języku logiki.
Przed precyzyjnym semantycznym zdefiniowaniem sprzeczności i interpretacji, warto przykładami zilustrować siłę wyrazu języka klauzul i niektóre cechy jego semantyki.

Przykład 1.4.1.

Przykład omylnego Greka.

Aby wykazać, że założenia

(G1) *Człowiek (Turing)*
(G2) *Człowiek (Sokrates)*
(G3) *Grek (Sokrates)*
(G4) *Omylny(x) Człowiek (x)*

implikują konkluzję, iż istnieje omylny Grek, dołączamy do nich negację konkluzji

(G5) *Omylny (u), Grek(u)*

i wykazujemy, że otrzymany w wyniku zbiór klauzul jest sprzeczny. Co więcej, analizując sprzeczność klauzuli (G5) z klauzulami (G1) – (G4), można ustalić jej przyczynę, a więc podstawienie: *u = Sokrates*, które określa obiekt indywiduowy będący zarówno omylnym, jak i Grekiem. Można zatem przyjąć, że klauzula (G5) wyraża zadanie znalezienia obiektu *u*, który jest omylnym Grekiem. Podstawienie *u = Sokrates* wyodrębnione z dowodu możemy traktować jako rozwiązanie tego zadania.

Przykład omylnego Greka wprowadzono po raz pierwszy w celu objaśnienia działania programów napisanych w języku programowania *Planner* (Hewitt C. 1969). W tym miejscu

nasze intencje są akurat odwrotne: chcemy pokazać, że informacja wyrażona w języku logiki może być zrozumiana bez znajomości procesów, jakie wywołuje ona wewnątrz komputera.

Przykład 1.4.2.

Silnia.

Przykład omylnego Greka nie jest typowy dla programów pisanych w konwencjonalnych językach programowania. Typowy będzie natomiast przykład silni.

Silnią liczby 0 jest 1.

Silnią x + l jest x + l pomnożone przez silnię x.

W najprostszym sformułowaniu definicji korzysta się z symboli funkcyjnych:

silnia (x)	oznacza silnię liczby x
razy (x, y)	oznacza iloczyn liczb x i y
n (x)	oznacza liczbę x + 1.

Dla oznaczenia równości przyjmiemy dwuargumentowy symbol relacyjny. *Równe (x, y)* zachodzi wówczas, gdy x jest „tym samym" co y.

Równe (silnia (0), l)
Równe (silnia (n(x)), razy (n (x), silnia (x)))

Są jeszcze potrzebne definicje charakteryzujące symbole *Równe* i *razy*. Dla równości przyjmuje się często następujące klauzule:

(1) *Równe (x, x)*
(2) *Równe (x, y)* *Równe (x, z), Równe (z, y)*
(3) *Równe (silnia(x), silnia(y))* *Równe (x, y)*

Aby teraz znaleźć na przykład silnię liczby 2, zaprzeczamy jej istnieniu

(4) *Równe (silnia (n (n (0))), w)*

Ale same tylko klauzule (l) i (4) są już sprzeczne i można stwierdzić, że przyczyną sprzeczności jest podstawienie

w = silnia(n(n(0)))

Niestety, niewiele to wnosi informacji.

Problem polega na tym, że symbole funkcyjne *silnia*, *razy* oraz *n* pozwalają odwoływać się do tej samej liczby za pomocą wielu różnych nazw. Nie zawierające zmiennych termy

n(n(0)), n(1), n(silnia(0)), n(silnia(razy(0, n(0))))

wszystkie oznaczają tę samą liczbę 2 i są sobie równe (w sensie definicji relacji *Równe*). Trudność tę można usunąć, jeśli nazwy indywiduów będą unikatowe. W naszym przykładzie wystarczy użyć jednej tylko stałej 0 i jednego symbolu funkcyjnego *n*. Funkcje silni i mnożenia można potraktować jako relacje:

Silnia (x, y)	zachodzi, jeśli y jest silnią x.
Razy(x, y, z)	zachodzi, jeśli x razy y wynosi z.

Teraz klauzule

(Sil 1) *Silnia (0, n(0))*
(Sil 2) *Silnia (n(x), u)* *Silnia (x, v), Razy (n(x), v, u)*

stanowią w kontekście odpowiedniej definicji mnożenia kompletną definicję silni. Relacja równości nie pojawia się w nich i jej definicja jest zbyteczna. Załóżmy, że dana jest definicja mnożenia, obejmująca takie klauzule, jak

Razy(0, x, 0)
Razy(n(0), y, y)

itd.

Aby rozwiązać zadanie znalezienia silni liczby 2, zaprzeczamy jej istnieniu

(Sil 3) *Silnia(n(n(0)) , w)*

Otrzymany zbiór klauzul (Si11) – (Si13) jest sprzeczny z dowolną definicją mnożenia, jeśli tylko implikuje ona asercje

Razy (n(n(0)), n(0), n(n(0)))

Razy(n(0), n(0), n(0))

Jeśli znajdziemy przykład ujawniający tę sprzeczność, uzyskamy tym samym jedyne podstawienie stanowiące rozwiązanie zadania

y = n(n(0))

W ten sposób definicja relacji *Silnia* uzupełniona definicją relacji *Razy* spełnia rolę programu, który może być przez komputer zastosowany do obliczania wartości silni. Program ten jest zrozumiały również dla osoby nie znającej zasad działania komputera.

1.5. Uniwersum zbioru klauzul i interpretacje.

Przytoczone dwa sformułowania definicji silni ilustrują ogólną zasadę posługiwania się językiem klauzul. Aby uniknąć problemów związanych z indywiduami, które mają więcej niż jedną nazwę, należy oszczędnie używać stałych i symboli funkcyjnych. Jeśli każde indywiduum jest oznaczone dokładnie jednym termem nie zawierającym zmiennych, to uniwersum zbioru klauzul, które intuicyjnie stanowi zbiór wszystkich indywiduów opisanych przez te klauzule, można utożsamiać ze zbiorem wszystkich termów ustalonych, które daje się zbudować ze stałych i symboli funkcyjnych występujących w klauzulach. Za możliwą interpretację zbioru klauzul będziemy teraz uważać przypisanie każdemu n – argumentowemu symbolowi relacyjnemu występującemu w klauzulach pewnej n – argumentowej relacji między indywiduami należącymi do uniwersum.
Prostym przykładem są założenia (G1) – (G4) zagadnienia omylnego Greka. Mają one niewielkie, skończone uniwersum złożone z dwóch stałych: *Turing* i *Sokrates*.

Podać możliwą interpretację znaczy tyle, co podać dla każdego z trzech symboli relacyjnych występujących w klauzulach relację w zbiorze uniwersum. Każdemu symbolowi relacyjnemu może być przypisana jedna z czterech różnych interpretacji, zatem zbiór klauzul jako całość ma $4 \cdot 4 \cdot 4 = 64$ możliwe interpretacje .

Jednak tylko w dwóch interpretacjach wszystkie klauzule będą prawdziwe. Jedna z nich zakłada, że wszystkie atomy ustalone są prawdziwe.

Człowiek (Sokrates), Człowiek (Turing)
Omylny (Sokrates), Omylny (Turing)
Grek (Sokrates), Grek (Turing)

W drugiej są prawdziwe atomy

Człowiek (Sokrates), Człowiek (Turing)
Omylny (Sokrates), Omylny (Turing)

Grek (Sokrates)

natomiast fałszywy jest atom

Grek (Turing)

Rozszerzony zbiór klauzul (G1) – (G5) ma to samo uniwersum i ten sam zbiór 64 możliwych interpretacji. Jednak nie ma żadnej interpretacji, w której wszystkie klauzule (G1) – (G5) byłyby jednocześnie prawdziwe. Dwie interpretacje, które czyniły prawdziwymi klauzule (G1) – (G4), czynią fałszywą klauzulę (G5). W szczególności konkretyzacja[4] klauzuli (G5)

(G'5) *Omylny (Sokrates}, Grek (Sokrates)*

w której *u = Sokrates*, jest fałszywa w obu interpretacjach, gdyż dwa warunki

Omylny(Sokrates) oraz *Grek(Sokrates)*

którym klauzula (G'5) zaprzecza, są prawdziwe w obu interpretacjach. Ponieważ klauzula (G'5) w obu interpretacjach jest fałszywa, fałszywa jest także klauzula (G5) (gdyż klauzula zawierająca zmienne jest prawdziwa w pewnej interpretacji wtedy i tylko wtedy, gdy wszystkie jej konkretyzacje są prawdziwe, a fałszywa, gdy przynajmniej jedna konkretyzacja jest fałszywa). Zatem klauzule (G1) – (G5) są sprzeczne, gdyż nie istnieje interpretacja, w której wszystkie one byłyby prawdziwe. Analiza dowodu sprzeczności pozwala zidentyfikować indywiduum *u = Sokrates*, którego istnienie jest negowane przez sprzeczną z pozostałymi klauzulę (G5).

Semantyczna metoda dowodzenia sprzeczności zbioru klauzul, polegająca na zademonstrowaniu, że żadna interpretacja nie zapewnia prawdziwości wszystkich klauzul zbioru, jest metodą ogólną, dającą się zastosować do każdego zbioru klauzul. Co więcej, wystarczy rozważyć tylko te interpretacje, w których zbiór indywiduów ogranicza się do uniwersum danego zbioru klauzul. Jeśli klauzule nie zawierają żadnych stałych, jest konieczne włączenie do uniwersum jednej dowolnie wybranej stałej. W tym przypadku uniwersum będzie zawierało wszystkie termy ustalone, jakie można utworzyć za pomocą tej stałej oraz symboli funkcyjnych występujących w zbiorze klauzul.

Włączenie do uniwersum dowolnej stałej, gdy same klauzule nie zawierają stałych, jest formalizacją założenia, iż istnieje co najmniej jedno indywiduum. Przy tym założeniu z klauzuli

(1) *Dobre(x)*

która stwierdza, że wszystko jest dobre, wynika, że co najmniej jedna rzecz jest dobra. Klauzula ta jest sprzeczna z założeniem, że nie ma nic dobrego

(2) Dobre(x)

Uniwersum zawiera w tym przypadku jedną, bliżej nie określoną stałą, dajmy na to ☼. Możliwe są dwie interpretacje – w pierwszej

Dobre (☼)

jest prawdziwe, w drugiej

[4] Konkretyzacje klauzuli otrzymujemy, zastępując w niej niektóre lub wszystkie zmienne dowolnymi termami.

Dobre (☼)

jest fałszywe.

Pierwsza z interpretacji zaprzecza klauzuli (2). Druga zaprzecza klauzuli (l). W żadnej interpretacji (l) i (2) nie są jednocześnie prawdziwe, są zatem ze sobą sprzeczne. Zwróćmy uwagę, że na zademonstrowaną sprzeczność nie ma wpływu nazwa hipotetycznego elementu uniwersum. Argumentacja będzie taka sama, niezależnie od tego, jakiej stałej użyjemy.

Samo pojęcie interpretacji daje się uprościć. Dla wyspecyfikowania interpretacji wystarczy podać, jaki ma ona wpływ na prawdziwość lub fałszywość ustalonych formuł atomowych. *Interpretację* można zatem uważać za przypisanie każdemu z ustalonych atomów, jakie daje się zbudować z elementów uniwersum oraz symboli relacyjnych występujących w klauzulach, jednej z dwóch wartości, logicznych:

prawda lub *fałsz.*

1.6. Bardziej precyzyjna definicja sprzeczności.

Pora teraz przedstawić bardziej precyzyjną definicję sprzeczności.

Zbiór klauzul S jest **sprzeczny** wtedy i tylko wtedy, gdy nie jest niesprzeczny. Zbiór klauzul jest niesprzeczny, jeśli w pewnej interpretacji wszystkie klauzule zbioru są prawdziwe.

Klauzula jest prawdziwa w pewnej interpretacji zbioru klauzul S wtedy i tylko wtedy, gdy każda nie zawierająca zmiennych konkretyzacja klauzuli, uzyskana przez zastąpienie zmiennych termami należącymi do uniwersum zbioru S, jest prawdziwa w tej interpretacji. W przeciwnym wypadku klauzula jest fałszywa w tej interpretacji.

Klauzula nie zawierająca zmiennych **jest prawdziwa w interpretacji I** wtedy i tylko wtedy, gdy prawdą jest, że jeśli wszystkie jej warunki są prawdziwe w interpretacji I, to przynajmniej jedna z jej konkluzji jest prawdziwa w I. W innym sformułowaniu, klauzula taka jest prawdziwa w interpretacji I wtedy i tylko wtedy, gdy co najmniej jeden z jej warunków jest fałszywy w I lub przynajmniej jedna z konkluzji jest prawdziwa w I. W przeciwnym wypadku klauzula jest fałszywa w interpretacji I.

Ścisła definicja sprzeczności wyjaśnia semantykę klauzuli pustej □ (patrz: 1.2). Ponieważ klauzula pusta nie ma ani warunków, ani konkluzji, w żaden sposób nie może być prawdziwa w jakiejkolwiek interpretacji. Jest to jedyny przykład klauzuli samosprzecznej. Aby wykazać sprzeczność zbioru klauzul, wystarczy wykazać, że implikuje on logicznie klauzulę pustą, w oczywisty sposób sprzeczną. Pusty zbiór klauzul jest niesprzeczny. Wszystkie należące do niego klauzule są prawdziwe we wszystkich interpretacjach, gdyż nie zawiera on w ogóle żadnych klauzul, które mogłyby być fałszywe .

Pojęcia konkretyzacji i podstawienia odgrywają ważną rolę nie tylko przy definiowaniu semantyki języka klauzul, lecz także podczas formułowania reguł wnioskowania, o czym później. Konkretyzację klauzuli otrzymujemy stosując do niej podstawienia. Podstawienie polega na przypisaniu zmiennym termów. Każdej konkretnej zmiennej może być przypisany tylko jeden term. Podstawienie praktycznie jest przedstawiać jako zbiór niezależnych jego elementów

$x_1 = t_1$, $x_2 = t_2$, … , $x_m = t_m$.

Element $x_i = t_i$ podstawienia przypisuje term t_i zmiennej x_i.

Zastosowanie podstawienia σ do wyrażenia E daje u wyniku nowe wyrażenie Eσ, które ma taką samą postać jak E, z tym że dla każdego elementu podstawienia $x_i = t_i$, jeśli wyrażenie E zawiera wystąpienie zmiennej x_i, to nowe wyrażenie będzie zawierało wystąpienie termu t_i. Operacja podstawienia zastępuje wszystkie wystąpienia tej samej zmiennej tym samym termem. Wyrażenie E może być dowolnym termem, atomem, klauzulą lub zbiorem klauzul. Różne zmienne mogą być zastąpione tym samym termem.

Z powyższego wynika, że różne zmienne niekoniecznie muszą odnosić się do różnych indywiduów. Na przykład założenia

(L1) *Lubi (Robert, logika)*
(L2) *Lubi (Robert, x) Lubi (x, logika)*
(L3) *Lubi(x, y), Lubi(y, y)*

(Nikt nie lubi takiego, kto lubi sam siebie.)

są sprzeczne, ponieważ (LI) i (L2) są sprzeczne z klauzulą

Lubi(Robert, Robert), Lubi(Robert, Robert)

która jest konkretyzacją (L3) dla x = Robert i y = Robert.

1.7. Semantyka alternatywy konkluzji.

Ścisła definicja sprzeczności wyjaśnia semantykę alternatywy konkluzji. Jeśli klauzula ma kilka konkluzji, należy ją interpretować jako stwierdzenie, że jeśli wszystkie warunki są spełnione, to spełniona jest przynajmniej jedna z konkluzji (być może więcej). Jest to **inkluzywna** interpretacja spójnika „lub", w przeciwieństwie do interpretacji **ekskluzywnej**, w której „A lub B" interpretuje się w ten sposób, że jest spełnione albo A, albo B, ale nie oba jednocześnie.

Inkluzywna interpretacja spójnika „lub" sprawia na przykład, że zbiór założeń

(B1) *Zwierzę(x), Minerał(x), Roślina(x)*
(B2) *Zwierzę(x) Ostryga(x)*
(B3) *Minerał(x) Cegła(x)*
(B4) *Roślina(x) Kapusta(x)*

nie jest sprzeczny z przypuszczeniem, że coś jest jednocześnie zwierzęciem i rośliną.

(B5) *Zwierzę(x) Bakteria(x)*
(B6) *Roślina(x) Bakteria(x)*
(B7) *Bakteria( )*

„Lub" interpretowane ekskluzywnie można ująć w kategoriach „lub" inkluzywnego i negacji. Na przykład dla wyrażenia faktu, że każdy człowiek jest albo mężczyzną, albo kobietą, ale nie jednocześnie jednym i drugim, potrzeba dwóch klauzul:

Kobieta(x), Mężczyzna(x) Człowiek(x)
Kobieta(x), Mężczyzna(x), Człowiek(x)

1.8. Klauzule hornowskie.

W wielu zastosowaniach logiki formę klauzul można ograniczyć, dopuszczając tylko te, które mają co najwyżej jedną konkluzję[5]. Klauzule z co najwyżej jedną konkluzją nazywamy **klauzulami hornowskimi** na cześć logika Alfreda Horna, który jako pierwszy zajął się badaniem ich własności (1951). W istocie można udowodnić, że każde zagadnienie, które może być wyrażone w języku logiki, daje się wyrazić za pomocą samych klauzul hornowskich.

Większość formalizmów stosowanych w programowaniu komputerów wykazuje podobieństwo raczej do klauzul hornowskich niż do „niehornowskich". Również większość modeli rozwiązywania zadań stworzonych dla badań nad sztuczną inteligencją okazuje się przydatna do zadań sformułowanych z użyciem klauzul hornowskich.

Klauzule hornowskie są więc bardzo ważnym podzbiorem języka klauzul. Ponadto metody wnioskowania dla klauzul hornowskich mają prostą interpretację zarówno w teorii rozwiązywania zadań, jak w programowaniu komputerów.

Jakkolwiek teoretycznie można by obejść się bez klauzul niehornowskich, w praktyce są one niezastąpione. Rozszerzenie metod rozwiązywania zadań hornowskich na pełny język klauzul pozwala na istotne wzbogacenie dominujących dzisiaj prostszych koncepcji rozwiązywania zadań.

Kilka podanych niżej potocznych opinii o grzybach jadalnych i muchomorach stanowi prosty przykład twierdzeń, które dają się w sposób naturalny wyrazić jedynie za pomocą klauzul niehornowskich.

Przykład 1.8.1.

Dajmy na to, że

(1) *Każdy grzyb jest jadalny lub jest muchomorem.*
(2) *Każdy borowik jest grzybem.*
(3) *Wszystkie muchomory są trujące.*
(4) *Żaden borowik nie jest grzybem jadalnym.*

Symbolicznie zapiszemy to jako

(GR1) *Jadalny(x), Muchomor(x) Grzyb(x)*
(GR2) *Grzyb(x) Borowik(x)*
(GR3) *Trujący(x) Muchomor(x)*
(GR4) *Borowik(x), Jadalny(x)*

Logiczną konsekwencją tych poglądów jest stwierdzenie, że:

Wszystkie borowiki są trujące.

(GR5) *Trujący(x) Borowik(x)*

Jednakże każdy grzybiarz wie, iż większość borowików jest bardzo smaczna, a tylko nieliczne są trujące. Jeśli odrzucamy konkluzję (GR5) i chcemy zachować wiarę w logikę, musimy odrzucić przynajmniej jedno z początkowych założeń (GR1) - (GR4). Zadziwiające, jak wielu ludzi w takiej sytuacji woli raczej zrezygnować z logiki.

[5] R. Kowalski, *Logika w rozwiązywaniu zadań*, WNT, Warszawa 1989.

2. Turbo Prolog – komputerowy język programowania logicznego

Język Prolog (skrót od stów *programming in logie*) w zasadniczej swojej części jest podzbiorem języka klauzulowej postaci logiki. Ponieważ koncepcja Prologu, w tym jego terminologia, opiera się na logice predykatów, programowanie w tak zwanym czystym Prologu (ang. *pure Prolog*) jest równocześnie konstruowaniem pewnej teorii formalnej. Rozpatrując konkretną dziedzinę zastosowań, programista piszący program w języku Prolog stara się wyodrębnić i nazwać określone, niepodzielne obiekty oraz sprecyzować i nazwać funkcje i relacje zachodzące między nimi, tj. opisać stany obiektu. Po wprowadzeniu podstawowych wyrażeń języka: stałych, termów i predykatów, będących nazwami obiektów, funkcji i relacji, następuje wypisanie odpowiednich aksjomatów.

Na podstawie aksjomatów, w czysto mechaniczny sposób, można wydedukować istotne wnioski, dotyczące rozpatrywanej dziedziny wiedzy. W logice klauzul, a tym samym w Turbo Prologu, stosuje się tylko jedną regułę wnioskowania, zwaną **zasadą rezolucji** (p. 2.4.).

2.1. Słownik klauzulowej postaci logiki.

Słownik języka klauzul, zgodny ze składnią Turbo Prologa[6], zawiera następujące symbole:

- stałe,
- zmienne,
- n – argumentowe symbole funkcyjne, n > 0,
- n – argumentowe symbole predykatowe, n > = 0,
- wyróżniony dwuargumentowy symbol predykatowy =,
- symbole logiczne *and* i *or*,
- wyróżniony symbol logiczny implikacji (klauzuli warunkowej) **:-** lub *if* ,
- symbole interpunkcyjne i pomocnicze: ( ) , ; : . [ ] .

Za pomocą wprowadzonych symboli tworzy się abstrakcyjne napisy, będące wyrażeniami zbudowanymi zgodnie z pewnymi ogólnymi zasadami syntaktycznymi (składniowymi), podanymi w p. 2.2. Semantykę (znaczenie) poszczególnych symboli i wyrażeń wyjaśniono w p. 2.3.

Stałe są reprezentowane przez:

- symbole zbudowane z liter, cyfr i znaku podkreślenia, rozpoczynające się od małej litery,
- symbole zbudowane z liter, cyfr i znaku podkreślenia, ujęte w cudzysłów,
- ciągi cyfr.

Przykładami poprawnie utworzonych symboli stałych są:

al, alfa, beta_2, miasto_Poznan, piotr, „Piotr”, „1990”, 1999.

[6] J. Szajna, M. Adamski, T. Kozłowski, *Turbo Prolog. Programowanie w języku logiki*, WNT, Warszawa 1991.

Zmienne są reprezentowane przez symbole zbudowane z liter, cyfr i znaku podkreślenia. Symbol zmiennej rozpoczyna się od dużej litery.

Przykładami poprawnie zbudowanych symboli zmiennych są:

Al, Alfa, Beta_2, Miasto, MIASTO, Miejsce_pracy.

Symbole funkcyjne są reprezentowane przez małe litery lub ciągi liter, cyfr i znaków podkreślenia, rozpoczynające się od małej litery. Zero-argumentowe symbole funkcyjne są szczególnym przypadkiem stałych. Symbole funkcyjne dzieli się na **funktory** (symbole prefiksowe) oraz **operatory** (symbole infiksowe).

Symbole predykatowe tworzy się z liter i cyfr oraz znaku podkreślenia.

Zero-argumentowy symbol predykatowy nosi nazwę **symbolu zdaniowego**. Symbol predykatowy, zwłaszcza dla n > 0, jest również nazywany **symbolem relacyjnym**. Przykładami poprawnie zbudowanych symboli predykatowych są:

p, p1, p_1, jest_ojcem, jestSynem.

W rozpatrywanym języku klauzul występują tylko dwa spójniki logiczne *and* oraz *or*, oznaczające odpowiednio koniunkcję oraz alternatywę. Symbol *or* w Turbo Prologu można zastąpić średnikiem, a *and* - przecinkiem.

W wielu programach oprócz działań symbolicznych często występuje także potrzeba wykonywania obliczeń numerycznych, więc wprowadzono również możliwość realizacji operacji arytmetycznych. W Turbo Prologu obliczenia numeryczne są oparte na trzech następujących założeniach:

- istnieją stałe o traktowane jako liczby, odróżniane od innych stałych o charakterze czysto symbolicznym;
- pewne operatory i funkcje są zdefiniowane jako **realizowalne** (ang. *exacutable*) w sensie numerycznym, np.: + * / – 1og sin ;

  podobną rolę, jak wyrażenia arytmetyczne w klasycznych językach programowania, w Turbo Prologu odgrywają tzw. termy realizowalne (ang. *exacutable terms*), zbudowane wyłącznie z realizowalnych funkcji i operatorów, liczb, zmiennych oraz innych realizowalnych termów;
- standardowy operator = (w niektórych odmianach Prologu zapisywany jako *is*) przekształca realizowalny term na liczbę stanowiącą wynik wykonywanych obliczeń oraz uzgadnia ją z innym termem, zwykle ze zmienną.

Wyrażenie X=T, w którym X jest zmienną, a T jest realizowalnym termem, jest równoważne instrukcji przypisania w Pascalu X **:=** T.

Semantyka predykatu T1=T2 jest następująca[7]:

- w pierwszym kroku są wykonywane obliczenia numeryczne zdefiniowane po prawej stronie wyrażenia, czyli term T2 jest interpretowany jako wyrażenie algebraiczne;
- predykat T1=T2 jest prawdziwy, jeżeli Tl jest zmienną, która przyjmuje wartość obliczoną na podstawie T2, albo jest stałą równą tej wartości. W przeciwnym razie predykat jest fałszywy.

[7] A. Thayse, *From Standard Logic to Logic Programming*, John Willey & Sons, Chichester, 1988.

2.2. Reguły tworzenia wyrażeń języka.

Język klauzul składa się z następujących wyrażeń: termów, formuł i klauzul. Zbiór termów jest zbiorem wyrażeń spełniającym następujące warunki:

- stałe są termami,
- zmienne są termami,
- jeżeli f jest n – argumentowym symbolem funkcyjnym, a tl, t2,..., tn są termami, to f (tl, t2,...,tn) jest również termem.

Term składa się z funktora f, po którym następuje ujęty w nawiasy ciąg oddzielonych przecinkami argumentów – dowolnych termów. Jeżeli n = 0, to pisze się f zamiast f ( ). W tym przypadku funktor oznacza stałą.

Predykat składa się z symbolu predykatowego, po którym następuje ujęty w nawiasy ciąg oddzielonych przecinkami argumentów – dowolnych termów. W predykatach zeroargumentowych nawiasy pomija się. Funktor jest nazwą funkcji, a predykat nazwą relacji. Jeżeli p jest symbolem n – argumentowej relacji ($n \geq 0$), to predykat p (tl, t2, ...,tn) jest nazywany **formułą atomową**. W klauzulach warunkowych Prologu mogą też występować formuły molekularne, zbudowane z formuł atomowych i spójników logicznych *and* i *or*:

- koniunkcji *A1 and A2 and ... and An*,
- alternatywy *B l or B2 or... or Bm*.

Klauzula warunkowa jest wyrażeniem o postaci:

B1, B2, ..., Bm **:-** *A1, A2,..., An* n > = 0, m > = 0,

w którym *B1, ..., Bm* oraz *A1, ..., An* są formułami atomowymi (predykatami). Przecinki po lewej stronie klauzuli są interpretowane jako spójniki *or*, a po prawej – jako spójniki *and*. Ciąg formuł *B1, ..., Bm* nosi nazwę następnika klauzuli, a ciąg *Al, ..., An* – poprzednika klauzuli. Formuły *B1, ..., Bm* w następniku tworzą alternatywę konkluzji klauzuli. Formuły atomowe *A1, ..., An* traktowane łącznie są uważane za koniunkcję warunków (przesłanek) klauzuli warunkowej.

Rolę podobną do spójnika *not* odgrywa w Turbo Prologu predykat wyższego rzędu *not( )*. Argumentami predykatu wyższego rzędu mogą być inne predykaty.

W Turbo Prologu korzysta się wyłącznie z klauzul Horna (patrz: 1.8), w których m = 0 lub m = l. Jeśli m = l i n = 0:

B1 **:-**

to klauzula nosi nazwę **faktu**. W terminologii Turbo Prologu znak **:-** jest zastępowany w tym przypadku kropką

*B1***.**

Warunkowa klauzula Horna dla m = l i n > 0 przyjmuje postać zwaną **regułą**:

B1 **:-** *Al, A2, ..., An*

lub

B1 if Al and A2 and … and An

Obydwa zapisy reguły są akceptowane przez Turbo Prolog. Jeżeli m = 0, to klauzula jest pytaniem

:- Al, ..., An

W Turbo Prologu pytanie podaje się w postaci:

Goal: Al, ..., An

lub

Goal: Al and ... and An

W szerzej pojętej logice klauzul, tak jak w Turbo Prologu, są dopuszczalne pewne rozszerzenia, umożliwiające bardziej zwarty zapis zestawu klauzul o identycznych lewych stronach.

Zamiast klauzul:

B1 **:-** *A1*
B1 **:-** *A2*
…
B1 **:-** *An*

podaje się jedną klauzulę

B1 **:-** *A1 or A2 or … or An*

Zamiast symbolu *or* stosuje się również średnik

B1 **:-** *A1 ; A2 ; … ; An*

Klauzula dla n = 0 i m = 0

:-

nosi nazwę **klauzuli pustej** (patrz: 1.2) i oznacza sprzeczność (jest zawsze fałszywa). W sposób jawny w Turbo Prologu klauzula pusta nie występuje.

2.3. Semantyka języka klauzul w Prologu

Poznaliśmy już (w p. 1.4) semantykę klauzul w ujęciu tradycyjnym. W logice klasycznej szuka się interpretacji (modelu) dla pewnych już istniejących abstrakcyjnych wyrażeń. W tym podrozdziale przedstawimy semantykę języka klauzul w ujęciu stosowanym w Prologu[8].

Dla danej struktury interpretacyjnej (tj. abstrakcyjnej dziedziny obiektów wraz ze zdefiniowanymi w niej relacjami i funkcjami) tworzy się opis w języku logiki, będący specyficzną logiczną teorią formalną. Opis ten składa się z pewnych stwierdzeń, przyjętych za prawdziwe, zwanych aksjomatami specyficznymi teorii. W języku klauzul wszystkie aksjomaty są zapisane za pomocą klauzul: faktów i reguł. Zbiór klauzul - aksjomatów, tworzący formalną teorię specyficzną rozpatrywanego problemu, w terminologii Prologu nosi nazwę **bazy danych**.

Przez interpretację teorii rozumie się przyporządkowanie każdej stałej, zmiennej, każdemu termowi i predykatowi pewnego składnika (obiektu, funkcji, relacji) występującego w systemie. Programując w języku logiki, zazwyczaj na bieżąco tworzy się teorię dla analizowanego lub projektowanego systemu.

Zastosowanie logiki klauzul do określonej abstrakcyjnej dziedziny rozpoczyna się od wyodrębnienia jej pewnych niepodzielnych, najprostszych elementów (obiektów), którym za nazwy służą stałe. Obiekty złożone są przedstawione za pomocą termów złożonych. Zapis termu złożonego jest zbliżony do przyjętego w matematyce zapisu funkcji, w którym n-tce obiektów składowych przypisuje się pewien obiekt złożony. W takim ujęciu obiekt prosty jest szczególnym przypadkiem obiektu złożonego, przedstawionego funktorem zeroargumentowym.

[8] J. Szajna, M. Adamski, T. Kozłowski, *Turbo Prolog. Programowanie w języku logiki*, WNT, Warszawa 1991. patrz także: F. Kluźniak, F. Szpakowicz, *Prolog*, WNT, Warszawa 1983.

Relacje zachodzące między poszczególnymi obiektami są odzwierciedlone za pomocą odpowiednich predykatów, przy czym zwyczajowo nazwa relacji jest równocześnie symbolem mnemonicznym związanego z nią predykatu (symbolem relacyjnym). Tworzenie teorii (logicznej bazy danych) jest zakończone po wypisaniu stwierdzeń uważanych za prawdziwe, w postaci klauzul (reguł i faktów).

Przykład 2.3.1

Przykładem dziedziny może być pewna grupa osób, pomiędzy którymi istnieją określone związki rodzinne (por. p. 1.1). W naszym przykładzie będą to: chłopiec o imieniu Piotr, zbiór osób z nim spokrewnionych oraz pies Kalif. W języku potocznym elementy tej dziedziny można określić następująco:

Piotr
Agnieszka, siostra Piotra
Ewa, matka Piotra
Ojciec Piotra
Kalif, pies Piotra.

Zgodnie z przedstawioną konwencją notacyjną obiekty są oznaczane nazwami rozpoczynającymi się od małej litery.

piotr
agnieszka
ewa
ojciec (piotr)
kalif

Przedostatni z obiektów przedstawiono za pomocą termu, w którym *ojciec* jest symbolem funkcyjnym, a *piotr* stałą.

W rozpatrywanej dziedzinie zachodzą pewne relacje, które można przedstawić za pomocą predykatów. Spośród wielu powiązań typu relacyjnego możemy wybrać:

a) osoba nazwana termem < *osoba* > jest członkiem rodziny:
jest_członkiem_rodziny (< osoba >)

b) osoba nazwana termem < *siostra* > jest siostrą osoby nazwanej termem < *osoba* >:
jest_siostrą (< osoba >, < siostra >)

c) osoba nazwana termem < *matka* > jest matką osoby nazwanej termem < *dziecko* >:
jest_matką (< dziecko >, < matka >)

d) osoba nazwana termem < ojciec > jest ojcem osoby nazwanej termem < dziecko >:
jest_ojcem (< dziecko >, < ojciec >)

e) zwierzę nazwane termem < *zwierzę* > jest psem:
jest_psem (< zwierzę >)

f) osoby nazwane termami < *ojciec* >, < *matka* > są małżeństwem:
są_małżeństwem (< ojciec >, < matka >)

g) osobami są matka, ojciec, dziecko, siostra:
< osoba > = < matka >; < ojciec >; < dziecko; < siostra >

W celu wskazania miejsc sensownego podstawiania poszczególnych termów jako argumentów rozpatrywanych predykatów, zastosowano tzw. **ramki syntaktyczne**.

Ramki syntaktyczne w Turbo Prologu są przedstawiane w sposób pośredni za pomocą deklaracji typu obiektu (dziedziny) i wskazania jego miejsca w odpowiednim predykacie. Konieczność deklarowania przynależności poszczególnych obiektów, reprezentowanych przez termy, do określonego typu, a tym samym klasyfikacja termów, jest charakterystyczną cechą Turbo Prologu, odróżniającą go od innych odmian Prologu.

W predykatach o liczbie argumentów większej lub równej dwa pojawia się problem interpretacji roli poszczególnych argumentów. W predykacie *jest_matką (agnieszka, ewa)* trudno bez komentarza odgadnąć, czy Agnieszka jest matką Ewy, czy odwrotnie. Wyjściem z sytuacji jest wskazanie roli poszczególnych termów występujących jako argumenty predykatu. Opis termu umieszcza się w nawiasach <...>, aby wskazać Czytelnikowi, że jest to opis typu wyrażenia, a nie samo wyrażenie. Wyrażenia w nawiasach < > są nazywane **deskryptorami**. W rozpatrywanym przypadku predykat zapisuje się np. jako:

jest_matką (< dziecko >, < matka >)

Relacja przedstawiona predykatem *p (tl, ..., tn)* jest zdefiniowana przez podanie wszystkich n – elementowych podzbiorów elementów dziedziny, między którymi zachodzi ta relacja. Relację przedstawioną predykatem *p* definiuje się również jako odwzorowanie między każdym n – elementowym podzbiorem elementów dziedziny a wartościami logicznymi należącymi do zbioru {*true, false*}, czyli {prawda, fałsz}. Obydwa te podejścia są równoważne, jeśli założyć, że nie wymienionym n – elementowym podzbiorom zbioru elementów dziedziny odpowiada wartość logiczna *false*; jest to tzw. założenie o zamkniętym świecie – CWA (ang. closed world assumption). W Prologu przyjmuje się, że baza danych jest przedstawiona przy założeniu CWA.

Relację opisaną predykatem p(t1, ..., tn) można rozpatrywać rozpoczynając od predykatu p(X1, ..., Xn), w którym wszystkie termy są zmiennymi. W miejsce zmiennych można podstawić inne termy, będące stałymi lub termami złożonymi. Termy te są nazwami obiektów prostych lub złożonych występujących w rozpatrywanej dziedzinie. Tylko część podstawień opisuje rzeczywiste relacje zachodzące w rozpatrywanej dziedzinie i jedynie w tych przypadkach mówi się, że rozpatrywany predykat dla konkretnej interpretacji i dla określonych wartości zmiennych jest prawdziwy.

W logice klasycznej rozpatruje się wszystkie tego rodzaju podstawienia, tworząc tzw. *uniwersum Herbranda*. Liczba rozpatrywanych podstawień może być znacznie zmniejszona, dzięki uwzględnieniu typów obiektów związanych z poszczególnymi argumentami predykatów. Rozwiązanie takie stosuje się w Turbo Prologu.

Bezpośrednie podanie wszystkich wcieleń predykatów w postaci klauzul-faktów odpowiadających rozpatrywanym relacjom nie jest jedyną możliwością definiowania predykatów. Zamiast wymieniania wszystkich wcieleń predykatu w postaci klauzul-faktów stosuje się często pośredni sposób definiowania relacji, za pomocą reguł zawierających zmienne.

Klauzula *B1, B2, ..., Bm* **:–** *A1, A2, ..., An* jest prawdziwa, jeżeli co najmniej jeden z następników klauzuli (*B1, B2, ..., Bm*) jest prawdziwy lub co najmniej jeden z poprzedników (*A1, A2, ..., An*) jest fałszywy.

Klauzula jest fałszywa, jeżeli wszystkie następniki są fałszywe oraz równocześnie wszystkie poprzedniki są prawdziwe.

Sposób odczytywania klauzul zawierających zmienne podamy na podstawie wyrażeń zawierających zmienne X i Y oraz ograniczoną liczbę formuł atomowych B1, A1 i A2:

B1(X) **:–** *A1(X), A2(X)*

- dla każdego X, dla którego zachodzi *A1(X)* i *A2(X)*, zachodzi również *B(X)*.

B1(X) **:–** *A1(X,Y), A2(X,Y)*

- istnieje takie Y, że dla każdego X, dla którego zachodzi *A1(X,Y)* i *A2(X,Y)*, zachodzi również *B1(X)*.

:– *A1(Y)*

- nie istnieje takie Y, że zachodzi *A1(Y)*; inaczej: dla każdego Y nie zachodzi *A1(Y)*.

W Turbo Prologu tę klauzulę stosuje się wyłącznie do zadawania pytań; w tym przypadku ma ona postać *Goal A1(Y).*

B1(X) :–

- dla każdego X zachodzi B1(X).

Przykład 2.3.2

Dla dziedziny rozpatrywanej w przykładzie 2.4.1 wszystkie możliwe wcielenia predykatów, uwzględniające sensowne dla przyjętej interpretacji, dopuszczalne podstawienia termów (wartościowania), przedstawiono za pomocą następujących klauzul:

jest_członkiem_rodziny(piotr)
jest_członkiem_rodziny(agnieszka)
jest_członkiem_rodziny(ewa)
jest_członkiem_rodziny(ojciec(piotr))
jest_siostrą(piotr, agnieszka)
jest_matką(piotr, ewa)
jest_matką(agnieszka, ewa)
jest_ojcem(piotr, ojciec(piotr))
jest_ojcem(agnieszka, ojciec(piotr))
jest_psem(kalif)
są_małżeństwem(ojciec(piotr), ewa)

Spośród wielu wcieleń predykatów, przedstawiających relacje nie występujące dla rozpatrywanej interpretacji, można wymienić kilka przykładowych:

jest_psem(piotr)
jest_psem(ojciec(piotr))
jest_matką(piotr, agnieszka)

2.4. Wnioskowanie

Logika zajmuje się relacjami między założeniami a wnioskami. Przykładowo, jeśli założy się, że Marian jest ojcem Piotra oraz że ojcowie są rodzicami, to można wyciągnąć wniosek, że Marian jest rodzicem Piotra. Dwa pierwsze stwierdzenia pociągają za sobą (implikują formalnie) trzecie stwierdzenie; inaczej mówiąc, trzecie stwierdzenie jest konsekwencją dwu pierwszych.

W logice formalnej wnioskowanie odbywa się na podstawie określonych zasad, umożliwiających wyprowadzenie pewnych wniosków ze znanych przesłanek.

Wykonywanie programu w Prologu opiera się na jednej, bardzo uniwersalnej zasadzie wnioskowania, zwanej **zasadą rezolucji**. W sposób formalny zasadę tę zapisuje się (z pewnym uproszczeniem) w postaci reguły cięcia:

$$\frac{\mathrm{Al}, \mathrm{c} \ :\!-\ \mathrm{Bl}; \quad \mathrm{A2} \ :\!-\ \mathrm{B2}, \mathrm{c}}{\mathrm{Al}, \mathrm{A2} \ :\!-\ \mathrm{Bl}, \mathrm{B2}}$$

Dane są dwie klauzule, jedna z formułą atomową *c* po jej lewej stronie, a druga z tą samą formułą po jej prawej stronie. A1 i A2 oraz B1 i B2 oznaczają zbiory dowolnych formuł atomowych. Za pomocą zasady rezolucji tworzy się nową klauzulę (**rezolwentę**), której lewa strona jest zestawieniem lewych stron odpowiednich

klauzul, z pominięciem wycinanej formuły atomowej *c*, a prawa strona zestawieniem prawych stron tych klauzul, również z pominięciem wycinanej formuły *c*. Klauzule-przesłanki zapisano nad kreską, a klauzulę-wniosek – pod kreską.

Termy występujące w formule *c* w obu łączonych klauzulach muszą spełniać warunki umożliwiające ich **uzgodnienie**. Uzgodnienie i cięcie jest wykonywane łącznie.

Sposób uzgadniania (czyli unifikacji) wymaga wielu istotnych wyjaśnień. Podstawą uzgadniania jest twierdzenie, że każde wystąpienie zmiennej w klauzuli może być zastąpione inną zmienną lub odpowiednim termem. Na przykład dla klauzuli:

1iczba_parzysta(X) **:–** *dzie1i_się_przez(2, X),*

po uzgodnieniu jej zjedna z formuł:

1iczba_parzysta(2)
liczba_parzysta(4)
liczba_parzysta(Y)

otrzymuje się odpowiednio:

1iczba_parzysta(2) **:–** *dzie1i_się_przez(2,2)*
1iczba_parzysta(4) **:–** *dzie1i_się przez(2,4)*
1iczba_parzysta(Y) **:–** *dzie1i_się_przez(2,Y)*

Należy zwrócić uwagę, że każde wystąpienie określonej zmiennej musi być zastąpione tym samym termem. Z tego względu, dla klauzuli:

niższy(X,Y) **:–** *wyższy(Y,X)*

w wyniku uzgodnienia można uzyskać na przykład klauzule:

niższy(piotr,Y) **:–** *wyższy(Y, piotr)*
niższy(Y,Y) **:–** *wyższy(Y,Y)*
niższy(Z,W) **:–** *wyższy(W,Z)*

nie można natomiast uzyskać klauzul:

niższy(X,Y) **:–** *wyższy(Y,piotr)*
niższy(X,piotr) **:–** *wyższy(Y,piotr)*

Stosując zasadę rezolucji, należy pamiętać o lokalnym charakterze zmiennych w poszczególnych klauzulach (zmienne o tej samej nazwie, występujące w różnych klauzulach są uważane za różne). Z tego względu przed wykonaniem cięcia należy ewentualnie przemianować zmienne w łączonych klauzulach. Na przykład, jeżeli łączymy dwie następujące klauzule:

b1(X), a **:–** *c1(X)*
b2(X) **:–** *a, c2(X)*

to należy przemianować np. zmienną X w drugiej klauzuli. W wyniku zastosowania zasady rezolucji otrzymujemy

b1(X), b2(Y) **:–** *c1(X), c2(Y)* .

Przykład 2.4.1

jest_ojcem(marian, piotr) **:–**
jest_rodzicem(Osoba1) **:–** *jest ojcem(Osoba1,Osoba2)*

---

jest_rodzicem(marian) **:–**

Uzgadnianie polega tu na tym, że za zmienne *Osoba1* i *Osoba2* podstawia się w drugiej klauzuli stałe *marian* i *piotr*, dzięki czemu wycinana formuła *jest_ojcem(marian, piotr)* jest identyczna w obydwu klauzulach. Stwierdzenie zapisane pod kreską (rezolwenta) jest konsekwencją stwierdzeń zapisanych nad nią. W tym przypadku rezolwenta jest klauzulą-faktem.

Przykład 2.4.2

Rozważmy dwie następujące klauzule:

1) *jest_dziadkiem(X,Z)* **:–** *ma_wnuka(X,Z),*
jest_mężczyzną(X).

2) *ma_wnuka(X,Z)* **:–** *jest_rodzicem(X,Y),*
jest_rodzicem(Y,Z).

Po przemianowaniu w drugiej klauzuli zmiennej X na R, klauzula ta przyjmuje postać:

3) *ma_wnuka(R,Z)* **:–** *jest_rodzicem(R,Y),*
jest_rodzicem(Y,Z).

Stosując teraz regułę cięcia otrzymuje się:

jest_dziadkiem(R,Z) **:–** *jest_mężczyzną(R),*
jest_rodzicem(R,Y),
jest_rodzicem(Y,Z).

Należy zwrócić uwagę, że w wycinanych formułach *ma_wnuka(X,Z)* i *ma_wnuka(R,Z)* nastąpiło uzgodnienie zmiennych R i X.

W przypadku klauzul Horna zasada rezolucji przyjmuje prostszą postać:

$$\frac{a\ :-\ C1; \quad b\ :-\ C2, a}{b\ :-\ C1, C2}$$

Formuły a, b są formułami atomowymi (predykatami).

Opisana zasada rezolucji jest podstawą działania aparatu wnioskowania Prologu. Kolejne kroki przekształcania bazy danych wynikające ze stosowania reguły cięcia nie są jednak widoczne dla użytkownika. Na przykład podczas śledzenia[9] wykonywania programu w Turbo Prologu można obserwować jedynie nagłówki realizowanych kolejno klauzul, a nie całe klauzule (rezolwenty), uzyskiwane kolejno w wyniku zastosowania zasady rezolucji.

2.5. Rezolucyjny system zaprzeczania

W Prologu stosuje się dowodzenie twierdzeń z wykorzystaniem **rezolucyjnego systemu zaprzeczania** (łac. reductio ad absurdum). Baza danych zawiera zbiór klauzul, będących przesłankami. W celu udowodnienia pewnej klauzuli dokonuje się najpierw jej zaprzeczenia, a następnie dodaje się ją do bazy danych. Do otrzymanej w ten sposób rozszerzonej bazy stosuje się zasadę rezolucji, starając się wyprowadzić na podstawie tej zaprzeczonej klauzuli klauzulę pustą (**:-**). Programistę interesuje przede wszystkim wartościowanie predykatu występującego w zaprzeczonej klauzuli po uzyskaniu klauzuli pustej (tzn. przypisanie wartości poszczególnym zmiennym) albo potwierdzenie, że rozpatrywany predykat jest prawdziwy.

[9] J. Szajna, M. Adamski, T. Kozłowski, *Turbo Prolog. Programowanie...*, str.159

Przykład 2.5.1

Jako ilustrację rozpatrzmy ponownie bazę danych przedstawioną w przykładzie 2.5.1:

jest_ojcem(marian, piotr) **:–**
jest_rodzicem(Osoba1) **:–** *jest ojcem(Osoba1,Osoba2)*

Aby znaleźć odpowiedź, kto jest rodzicem, do bazy danych dodaje się stwierdzenie, że X nie jest rodzicem:

:– *jest_rodzicem(X)*

Jest ono zaprzeczeniem stwierdzenia:

jest_rodzicem(X) **:–**

Odpowiedni zapis w Turbo Prologu ma następującą postać:

GOAL: *jest_rodzicem(X)*

Uzgodnienie formuł *jest_rodzicem(X)* oraz *jest_rodzicem(Osoba1)* i zastosowanie cięcia prowadzi do uzyskania klauzul:

jest_ojcem(marian, piotr) **:–**

:– *jest_ojcem(X,Osoba2)*

Po kolejnym uzgodnieniu, polegającym na podstawieniu *marian* za *X* oraz *piotr* za *Osoba2* i zastosowaniu cięcia, otrzymuje się klauzulę pustą. Szukanym rozwiązaniem jest więc

X = marian

2.6. Wprowadzenie do języka Turbo Prolog

W celu przedstawienia podstawowej struktury programu napisanego w Turbo Prologu posłużymy się prostym przykładem określającym pewne związki rodzinne[10]. Aby ułatwić późniejsze omówienie programu, numeruje się jego wiersze (z prawej strony). Numery te nie są częścią programu. W Turbo Prologu nie używamy polskich liter.

Przykład 2.6.1

```
1   DOMAINS
2   imie=symbol
3
4   PREDICATES
5   ojciec (imie,imie)
6   matka (imie,imie)
7   dziadek (imie,imie)
8
9   CLAUSES
10  ojciec (stanislaw,janusz).    /* Stanisław jest ojcem Janusza */
11  ojciec (edmund,ewa).
12  ojciec (janusz,agnieszka).
13  ojciec (janusz,andrzej).
14
15  matka (cecy1ia,janusz).    /* Cecylia jest matką Janusza   */
16  matka (jozefa,ewa).
17  matka (ewa,agnieszka).
18  matka (ewa,andrzej).
```

[10] J. Szajna, M. Adamski, T. Kozłowski, *Turbo Prolog. Programowanie w języku logiki*, WNT, Warszawa 1991.

```
19
20  dziadek (A,B)   if              /*  A jest dziadkiem B  */
21        ojciec (A,X) and
22        ojciec (X,B).
23  dziadek (A,B)  if
24        ojciec (A,X) and
25        matka (X,B).
```

W przytoczonym programie można wyróżnić trzy sekcje: *DOMAINS, PREDICATES i CLAUSES.*

Sekcja *DOMAINS* (dziedziny) odpowiada definicjom typów w Pascalu i służy do określenia typów obiektów, którymi będzie operował program. W podanym przykładzie w wierszu 2 użytkownik deklaruje własną dziedzinę danych o nazwie *imię*, która odpowiada standardowej dla Turbo Prologu dziedzinie *symbol* (ciąg znaków). Jeśli nie deklarujemy własnych dziedzin, lecz korzystamy wyłącznie z dziedzin standardowych, to sekcję *DOMAINS* można pominąć.

Sekcja *PREDICATES* (predykaty) stanowi zapowiedź relacji zachodzących między określonymi obiektami (same zaś relacje są zdefiniowane w sekcji *CLAUSES*). Każdy z elementów sekcji *PREDICATES* odpowiada w przybliżeniu nagłówkowi procedury w Pascalu, określając nazwę relacji oraz liczbę i dziedziny obiektów, między którymi relacja ta zachodzi. Na przykład w wierszu 5 zdefiniowano relację o nazwie *ojciec*, wiążącą ze sobą pary obiektów należących do dziedziny *imię*.

Sekcja *CLAUSES* (klauzule) jest najważniejszą częścią programu. Poszczególne klauzule służą do zdefiniowania relacji między poszczególnymi obiektami. Na przykład klauzula podana w wierszu 10 oznacza, że Stanisław jest ojcem Janusza, lub inaczej, że obiekt *stanislaw* jest w relacji *ojciec* z obiektem *janusz* (słowa *stanislaw* i *janusz* pisane małymi literami są symbolami konkretnych osób).

Zbiór klauzul definiujących określoną relację tworzy **procedurę**[11]. W omawianym programie występują trzy procedury: *ojciec*, *matka* i *dziadek*.

Uwagi:

- W Turbo Prologu nazwy wszystkich zmiennych muszą zaczynać się od dużej litery, a nazwy stałych należących do dziedziny symbol – od małej litery lub muszą być ujęte w cudzysłów. Nazwy *janusz* lub "*Janusz*" oznaczają stałe, a nazwa *Janusz* – zmienną.
- Wszystkie klauzule tworzące daną procedurę muszą być zapisane obok siebie.
- Każda klauzula jest zakończona kropką.
- Tekst komentarza w Turbo Prologu rozpoczyna się znakami: /* (ukośnik, gwiazdka), a kończy znakami: */ (gwiazdka, ukośnik). Inny dogodny sposób podawania komentarzy polega na poprzedzeniu treści komentarza znakiem % (procent). Jako komentarz traktuje się wtedy cały tekst od znaku % do końca danego wiersza. Dobrym zwyczajem jest komentowanie znaczenia poszczególnych argumentów w sekcji *PREDICATES*. W omawianym przykładzie sekcja ta po dopisaniu komentarzy mogłaby np. wyglądać następująco:

```
PREDICATES
% ojciec (Ojciec, Dziecko)
   ojciec (imie, imie)
% matka (Matka, Dziecko)
   matka (imie, imie)
% dziadek (Dziadek, Wnuk)
```

[11] Niekiedy używa się zamiennie określeń procedura i predykat. Aby uniknąć nieporozumień, przez procedurę będziemy rozumieli zawsze zespół klauzul o tej samej nazwie, a przez predykat – jeden z elementów sekcji PREDICATES lub nazwę relacji między określonymi obiektami.

dziadek (imie, imie)

- Sekcje *DOMAINS*, *PREDICATES* i *CLAUSES* można zapisywać oddzielnie dla wybranych grup procedur, dzięki czemu poprawia się czytelność programu, np.

```
DOMAINS
imie=symbol

PREDICATES
% ojciec (Ojciec, Dziecko)
   ojciec (imie, imie)
% matka (Matka, Dziecko)
   matka (imie, imie)

CLAUSES
ojciec (edmund,ewa).
ojciec (stanisław,janusz).
ojciec (janusz,agnieszka).
ojciec (janusz,andrzej).
matka (jozefa,ewa).
matka (cecylia,janusz).
matka (ewa,agnieszka).
matka (ewa.andrzej).

PREDICATES
% dziadek (Dziadek, Wnuk)
   dziadek (imie, imię)

CLAUSES
dziadek(A,B) if
    ojciec(A,X) and
    ojciec(X,B).
dziadek(A,B) if
   ojciec(A,X) and
   matka(X,B).
```

Każda z klauzul może mieć postać **faktu** lub **reguły**. W przykładzie 2.6.1 faktami są wszystkie klauzule tworzące procedury *ojciec* i *matka*, regułami natomiast – klauzule tworzące procedurę *dziadek*.

Fakty są stwierdzeniami, że zachodzą określone relacje między obiektami; mają charakter bezwarunkowy i są zawsze prawdziwe. W ich zapisie występuje jedynie nazwa relacji oraz zestaw argumentów (w nawiasach).

Reguły mają charakter warunkowy – relacja zapisana po lewej stronie stówa *if* jest prawdziwa jedynie wówczas, gdy są prawdziwe wszystkie warunki zapisane po prawej strome tego słowa. Na przykład klauzulę:

dziadek(A.B) if ojciec(A,X) and ojciec(X,B)

w której użyto zmiennych A, B, X, można odczytać: dla każdej osoby A i osoby B: A jest dziadkiem B, jeśli istnieje taka osoba X, że A jest ojcem X i X jest ojcem B.

Każda reguła składa się z dwóch części: nagłówka i treści, rozdzielonych słowem *if*. Nagłówek jest utworzony przez nazwę relacji (predykatu) oraz listę argumentów, a treść – przez warunki oddzielone spójnikami *and*.[12]

[12] Stosując terminologię z dziedziny logiki, można powiedzieć, że reguła jest stwierdzeniem warunkowym (implikacją), w którym konkluzja odpowiada nagłówkowi reguły, a koniunkcja warunków - treści reguły.
Słowo *if* stosowane w zapisach reguł Turbo Prologu można zastąpić znakiem **:-** utworzonym z dwukropka i myślnika, a słowo *and* znakiem przecinka.

Fakty oraz reguły tworzą **bazę danych** programu. Część bazy danych, utworzona przez fakty, jest zdefiniowana w sposób bezpośredni, a część utworzona przez reguły – w sposób pośredni.

Należy zwrócić uwagę, że zmienne A, B oraz X zostały użyte w procedurze *dziadek* bez wcześniejszej deklaracji. W programach napisanych w Turbo Prologu nie deklaruje się bowiem zmiennych. Typ zmiennej wynika z pozycji, na której występuje ona na liście argumentów danej relacji (typy lub inaczej mówiąc dziedziny odpowiednich argumentów danej relacji są zdefiniowane w sekcji *PREDICATES*). Na przykład zmienna A występująca w klauzuli zapisanej w wierszach 20 – 22 należy do dziedziny *imię*, ponieważ jest ona użyta jako pierwszy argument relacji *dziadek* (wiersz 20) i jako pierwszy argument relacji *ojciec* (wiersz 21). Pozycje te, zgodnie z sekcją *PREDICATES*, odpowiadają dziedzinie *imię*.

Zmienne mają charakter lokalny, tzn. że są dostępne tylko w ramach jednej klauzuli. Oznacza to, że na przykład zmienna A użyta w pierwszej klauzuli procedury *dziadek* i zmienna A użyta w drugiej klauzuli tej procedury, to dwie zupełnie różne zmienne.

Zauważmy, że program przedstawiony w przykładzie 2.6.1 zawiera jedynie definicje relacji, nie ma natomiast sekcji, która odpowiadałaby programowi głównemu w innych językach programowania. Rolę tę odgrywają w *Prologu* pytania na temat bazy danych, które może zadawać użytkownik. Nazwa **pytanie** jest używana wymiennie z nazwą **cel**, gdyż można powiedzieć także, że użytkownik określa cel, jaki w danej chwili ma zrealizować program.

Pytania można zadawać od chwili, gdy po skompilowaniu i uruchomieniu programu, na ekranie ukaże się napis *GOAL***:** .

W najprostszym przypadku pytanie ma postać nazwy jednego z predykatów, po której wymienia się odpowiednie argumenty. Używając odpowiednio stałych i zmiennych można zadać wiele różnych pytań dotyczących bazy danych programu. Na przykład dla programu 2.6.1 mogą to być między innymi następujące pytania:

ojciec (janusz,andrzej)	Pytanie: Czy Janusz jest ojcem Andrzeja? Odpowiedź:
YES.	
ojciec(janusz,X)	Pytanie: Czyim ojcem jest Janusz? Odpowiedź:
X=agnieszka *X=andrzej*	
2 solutions	(dwa rozwiązania)
dziadek (edmund,ewa)	Pytanie: Czy Edmund jest dziadkiem Ewy? Odpowiedź:
NO.	
dziadek (X,ewa)	Pytanie: Kto jest dziadkiem Ewy? Odpowiedź:
No solutions	(nie ma rozwiązania, gdyż na podstawie podanej bazy danych nie można wskazać żadnego dziadka Ewy).
dziadek (Dziadek, Wnuk)	Pytanie o wszystkich dziadków i wnuków. Odpowiedzi:
Dziadek=stanislaw	

Wnuk=agnieszka

Dziadek=stanislaw
Wnuk=andrzej

Dziadek=edmund
Wnuk=agnieszka

Dziadek=edmund
Wnuk=andrzej

4 solutions

Jeżeli pewien argument predykatu aktualnie nas nie interesuje, to w jego miejsce można użyć w pytaniu tzw. **zmiennej anonimowej**. Ma ona postać znaku podkreślenia, a jej wartość nie jest wyznaczona. Na przykład gdybyśmy chcieli zapytać: „Kto jest dziadkiem?", a imiona wnuków by nas nie interesowały, to odpowiednie pytanie miałoby postać:

dziadek(X,_)

W wyniku zadania pytania *dziadek(X,_)* imię tego samego dziadka będzie wymieniane wielokrotnie, jeżeli ma on wielu wnuków.

Jak widać z przytoczonych powyżej przykładów, użycie stałej jako argumentu pytania sprawia, że argument ten jest traktowany jako parametr wejściowy odpowiedniej procedury, a użycie zmiennej (nie licząc zmiennej anonimowej) – że jest on traktowany jako parametr wyjściowy, którego wartość należy wyznaczyć. Charakterystyczną cechą Prologu jest przy tym to, że dla większości predykatów nie występuje trwały podział parametrów na wejściowe i wyjściowe. Kierunek przepływu informacji zależy jedynie od postaci pytania. Na przykład, w pytaniu: *dziadek (stanislaw,Wnuk)* pierwszy argument jest parametrem wejściowym, a drugi wyjściowym; w pytaniu *dziadek (X,andrzej)* jest natomiast odwrotnie.

Pytania mogą mieć także charakter złożony, tzn. mogą składać się z kilku pojedynczych pytań, połączonych spójnikami *and* lub *or*. Ale Prolog znajdzie odpowiedź na takie pytanie tylko wówczas, gdy wszystkie podcele tego pytania są jednocześnie spełnione; w przeciwnym razie odpowie NO. Na przykład pytanie złożone:

ojciec(X,andrzej) and ojciec(X,agnieszka)

jest pytaniem o osobę, która byłaby jednocześnie ojcem Andrzeja i Agnieszki. Komputer odpowie *X=janusz* (zakresem zmiennej w pytaniu złożonym, w którym występują jedynie spójniki *and*, są wszystkie podcele – warunki tworzące treść tego pytania, dlatego też X oznacza tu tę samą zmienną). Podobnie, chcąc zapytać o dziadka Agnieszki ze strony matki, możemy użyć następującego pytania:

ojciec(D,M) and matka(M,agmeszka)

Komputer odpowie *D=edmund* oraz dodatkowo *M=ewa*. Z kolei w odpowiedzi na pytanie:

ojciec(Ojciec,Dz1) and ojciec(Ojciec,Dz2) and Dzl<>Dz2

zostaną wymienione imiona tych ojców (oraz ich dzieci), którzy mają przynajmniej dwoje dzieci. W pytaniu tym użyto standardowego operatora <> relacji różności.

Jeśli w pytaniu złożonym użyto spójnika *or*, to poszczególne części pytania połączone tym spójnikiem są traktowane jako oddzielne, niezależne od siebie pytania. Na przykład na pytanie złożone:

ojciec(janusz, andrzej) or dziadek(janusz, andrzej)

uzyskamy odpowiedź pozytywną (YES), jeśli Janusz jest ojcem Andrzeja lub jego dziadkiem. Podobnie, gdy interesuje nas jedna z dwóch osób: osoba, która jest jednocześnie ojcem Andrzeja i Agnieszki lub osoba będąca matką Janusza, to odpowiednie pytanie ma następującą postać:

ojciec(X,andrzej) and ojciec(X,agnieszka) or matka(X,janusz)

Zakres zmiennych w pytaniu złożonym, w którym występują spójniki *or*, jest ograniczony do poszczególnych części pytania połączonych tymi spójnikami. Oznacza to, że na przykład ostatnie z podanych pytań można zapisać także następująco:

*ojciec(X,andrzej) and ojciec(X,agnieszka) or matka(**Y**,janusz)*

(zmienne X występujące po lewej i prawej stronie spójnika *or* w poprzedniej wersji pytania są traktowane jako zupełnie inne zmienne).

W *Prologu* możliwe jest wykonywanie podstawowych operacji arytmetycznych na liczbach całkowitych (integer) i rzeczywistych (real), przy użyciu operatorów: +, – , *, /, *div* i *mod*. Można także porównywać liczby, wyrażenia arytmetyczne, znaki i ciągi znaków, używając operatorów relacji: = , > , < , >= , <= , <> . Na przykład:

*A+5<>2*B/10*

(wszystkie zmienne występujące w porównywanych wyrażeniach muszą mieć ustaloną wcześniej wartość)

janusz < jaroslaw

(porównanie kolejnych liter zgodnie z ich pozycją w tablicy ASCII).

Język Turbo Prolog, w odróżnieniu od czystego *Prologu*, ma bogaty zestaw wbudowanych funkcji matematycznych, na przykład *sin*, *log*, *exp*, *sqrt*.[13] Dopuszczalny jest także zapis złożonych wyrażeń arytmetycznych w rodzaju

Z = 1+sin(1og(X+Y))

Operator = może być zarówno operatorem arytmetycznym przypisania, jak i operatorem porównania. Pierwsza sytuacja występuje tylko wówczas, gdy po jego lewej stronie występuje pojedyncza zmienna, której nie nadano wcześniej żadnej wartości.

Sposoby zapisywania operacji arytmetycznych i operacji porównania w językach Turbo Prolog i Pascal są do pewnego stopnia analogiczne. Analogia z językiem Pascal przestaje jednak obowiązywać, gdy chcemy zapisać wyrażenie, które w Pascalu ma postać X **:=** X + 1 i oznacza „zmiennej X przypisz jej dotychczasową wartość zwiększoną o jeden".

W Prologu zmienna, której w danej klauzuli nadano już raz. pewną wartość, nie może ulec zmianie aż do końca realizacji tej klauzuli. Oznacza to, że pojęcie zmiennej jest w Prologu zupełnie inaczej rozumiane, niż w Pascalu, Basicu, języku C itp., *„nazwy tej nie należy ani przez chwilę kojarzyć ze zmienną znaną z klasycznych języków programowania*"[14].

Używając terminologii Prologu, o każdej zmiennej można powiedzieć, że w danej chwili może być **ukonkretniona** lub **nieukonkretniona** (swobodna). Zmienna nieukonkretniona jest to zmienna, której nie nadano jeszcze żadnej wartości. W trakcie realizacji danej klauzuli zmienna taka może zostać ukonkretniona, np.

[13] Pełny zestaw standardowych operatorów, funkcji oraz predykatów arytmetycznych – J. Szajna, M. Adamski, T. Kozłowski, *Turbo Prolog. Programowanie...*, p. D.5.

[14] F. Kluźniak, F. Szpakowicz, *Prolog*, WNT, Warszawa 1983.

w wyniku przypisania X=5, lub też może zostać powiązana z inną zmienną, np. w wyniku operacji X=Y. Dwie powiązane zmienne reprezentują w istocie tę samą zmienną identyfikowaną jedynie przez dwie różne nazwy. Oznacza to, że ukonkretnienie jednej z tych zmiennych pociąga za sobą również ukonkretnienie drugiej.

Ukonkretnienie lub powiązanie zmiennych może odbywać się w wyniku operacji przypisania lub w wyniku tzw. uzgadniania parametrów między podanym celem a odpowiednim nagłówkiem klauzuli[15]. Zmienna ukonkretniona nie może więc ulec zmianie aż do końca realizacji danej klauzuli. Oznacza to, że zmienna taka jest już ukonkretniona we wszystkich podcelach danej klauzuli. Stosując kryteria z klasycznych języków programowania, można powiedzieć, że zmienna po ukonkretnieniu przestaje być zmienną, gdyż nie można jej już zmieniać. Oczywiście po zakończeniu realizacji klauzuli ukonkretnienie i powiązanie nie ma znaczenia dla innych klauzul, ze względu na lokalny charakter zmiennych (identyczne nazwy zmiennych występujące w różnych klauzulach reprezentują sobą różne zmienne).[16]

Biorąc pod uwagę to, co powiedziano o charakterze zmiennych, wspomnianą wcześniej operację zapisaną w Pascalu za pomocą instrukcji: *X := X + 1* można zrealizować w Prologu używając dodatkowej zmiennej, np. *X1 = X + 1* .

Problem powstałby jednak np. wówczas, gdybyśmy chcieli zmieniać wartość zmiennej wielokrotnie – tak jak w przypadku obliczania sumy kolejnych liczb. Odpowiada temu następujący algorytm zapisany w Pascalu:

```
suma:=0;
for i:= 1 to N do
suma: = suma + i;
```

Problem ten można rozwiązać, wykorzystując lokalny charakter zmiennych prologowych i budując odpowiednią **procedurę rekurencyjną**.

Z rekurencją mamy do czynienia wówczas, gdy w definicji jakiegoś obiektu powołujemy się na definiowany właśnie obiekt. Powszechnie znana jest na przykład rekurencyjna definicja silni (por. 1.6) Rekurencja jest jedną z fundamentalnych technik programowania w Prologu. Procedury rekurencyjne odgrywają podobną rolę, jak instrukcje pętli w klasycznych językach programowania.

Najczęściej procedura rekurencyjna składa się z dwóch części: ze zbudowanej nierekurencyjnie klauzuli definiującej przypadek kończący rekurencję (dla silni odpowiada on stwierdzeniu, że 0! = l) oraz z klauzuli, która ma postać rekurencyjnie zbudowanej reguły; w treści tej reguły następuje ponowne wywołanie definiowanej procedury, lecz tym razem już z innymi argumentami.

Przykład 2.6.2

```
PREDICATES
  % silnia(N     , N!    )
  si1nia(integer, integer)
```

[15] J. Szajna, M. Adamski, T. Kozłowski, *Turbo Prolog. Programowanie...*

[16] Jeżeli wyrażenie występujące po prawej stronie znaku równości zawiera choćby jeden operator arytmetyczny, to wszystkie zmienne w tym wyrażeniu muszą być wcześniej ukonkretnione. Wynika stąd, że wszystkie predykaty zdefiniowane za pomocą procedur zawierających operacje arytmetyczne mają charakter jednokierunkowy, czyli że występuje w nich trwały podział na parametry wejściowe i wyjściowe.

```
CLAUSES
  si1nia(0,1).
  % klauzula konczaca rekurencje
  si1nia(N,Nsi1)  if                    % klauzula podstawowa
  N > 0   and
  M = N – 1   and
  si1nia(M,Msil)   and
  Nsil = N*Msil.
```

W pierwszej z podanych klauzul stwierdza się fakt, że silnią zera jest jeden. Drugą z klauzul możemy natomiast odczytać następująco: liczba oznaczona przez *Nsil* jest silnią liczby *N*, jeżeli są spełnione jednocześnie wszystkie poniższe warunki:

- liczba *N* jest większa niż zero,
- liczba oznaczona przez *M* jest równa *N – 1*,
- liczba oznaczona przez *Msil* jest silnią liczby *M*,
- liczba *Nsil* jest iloczynem liczb *N* oraz *Msil*.

Ze względu na to, ze w procedurze *silnia* używa się operacji arytmetycznych, ma ona charakter jednokierunkowy, tzn. nie pozwala obliczyć wartości liczby na podstawie wartości silni tej liczby.

W Prologu nie ma instrukcji pętli takich, jak *while*, *for* itd. Zamiast nich stosuje się rekurencję. W przykładzie 2.6.3 podano sposób obliczenia sumy kolejnych dodatnich liczb całkowitych:

Przykład 2.6.3

```
PREDICATES
  % suma(liczba ,sumaLiczb)
  suma (integer.integer )
CLAUSES
  suma(0,0).                        % klauzula konczaca rekurencje
  suma(N, N_suma)  if       % klauzula podstawowa
          N > 0  and
          M = N – 1  and
          suma(M, M_suma)  and
  N_suma = N + M_suma.
```

W Pascalu algorytmowi temu odpowiadałaby prosta pętla:

```
suma: = 0;
for i:= 1  to  N   do
  suma := suma + i
```

Prolog jest bowiem językiem deklaratywnym, tj. opisowym, a nie algorytmicznym. Zadanie programisty polega tu przede wszystkim na określeniu relacji logicznych, jakie zachodzą między poszczególnymi obiektami, zamiast, jak w klasycznych językach programowania, na wskazaniu ciągu operacji, które ma wykonać komputer, aby uzyskać pożądany wynik.

Programowanie logiczne w rozwiązywaniu zadań

4.1. Nauczanie ekspertowe w przykładowych zadaniach „znajdowania drogi”.

Jeżeli do formułowania zadania i opisu metod jego rozwiązywania używamy języka logiki, to zadanie to może być przedstawione jako zadanie „znajdowania drogi”: dany jest stan początkowy A, stan docelowy Z i operacje przeprowadzające jeden stan w następny, a zadanie polega na znalezieniu drogi od A do Z, rozumianej jako ciąg kolejno osiąganych stanów.

Tego typu zadania charakteryzuje bardzo wielka rozpiętość, jeśli chodzi o złożoność problemu, tematykę, wiek „eksperta” badającego zagadnienie, poziom jego umiejętności. Większość problemów jest wielopoziomowa – zaskakująco głęboko sięgają one w matematykę (logikę), a tylko od nas samych zależy, jak głęboko sięgniemy. Od pozornie prostych rzeczy można nieraz dojść do ogólnych i ważnych prawidłowości.

W wybranych przeze mnie zadaniach „znajdowania drogi”[17] niejednokrotnie pojawia się propozycja modyfikacji ich treści, rozszerzenia problemu, uogólnienia, równoległe wersje, pytania o to, jak zmieniłby się sposób rozwiązania po małej modyfikacji treści, po dodaniu jeszcze jednego warunku. Zadanie „rośnie”, rozwija się razem z jego wykonawcą.

Nauczyciel przedstawiający uczniowi problem do rozwiązania powinien dbać o to, by ów problem starannie sformułować i opisać, co już na starcie ukierunkuje poczynania ucznia i zmobilizuje go do samodzielnych poszukiwań. Jak daleko w nich się posunie, zależy od jego indywidualnego zaangażowania, wytrwałości i wiedzy. Ważne jest to, że stając się ekspertem danego problemu, uczeń odnosi swój mały własny „matematyczny sukces”. Po pierwszym może przyjść drugi, trzeci... Na pewno warto spróbować!

Zadanie 1. *WIELKI MARSZ CHIŃCZYKÓW* (łamigłówka)

Wielu Chińczyków udało się na Wielki Marsz. Niestety, na drodze Marszu znajduje się Głęboka Rzeka. Szczęśliwie na brzegu jest mała łódka, którą bawią się dwaj chłopcy. Łódka jest na tyle duża, że może bezpiecznie przewieść albo jednego Chińczyka, albo co najwyżej dwóch chłopców. Jak Chińczycy mają dostać się na drugi brzeg, jeśli mogą skorzystać tylko z pomocy chłopców i ich łódki (aspekt pragmatyczny), a po przepłynięciu rzeki zwrócić łódkę chłopcom (aspekt moralny)?

[17] Patrz: dodatek do niniejszej pracy – zbiór zadań.

(Tekst: E. Bryniarski, materiały do wykładu z *Informatyki szkolnej)*

ŚRODEK INFORMATYCZNY (TECHNOLOGIA INFORMACYJNA)

1. ***Algorytm*** – dokonanie symulacji transportu Chińczyków za pomocą łódki z lewego brzegu rzeki na prawy.
2. ***Kompilacja*** – symulowane musi być: położenie Chińczyków, chłopców, brzegów oraz łódki i przemieszczanie się Chińczyków oraz chłopców za pomocą łódki – użytkownik komputera, korzystając z myszki, steruje wyróżnionymi obiektami graficznymi („chwyta" je myszką i „przeciąga" w pożądane miejsce – rys. 4.1.1).
3. ***Procesor*** – kompilacja algorytmu dokonywana jest w edytorze grafiki, np. przy użyciu paska rysowania edytora Word 2000 (patrz rys.4.1.1).
4. ***Monitorowanie*** – sterowanie obiektami graficznymi obrazowane jest na ekranie monitora jako przemieszczanie się tych obiektów.
5. ***Monitor*** (***system multimedialny***) – ekran monitora komputera wraz z myszką.
6. ***Implementacja*** – uzyskanie stanu odpowiadającego (adekwatnego do) sytuacji przeprawienia się wszystkich Chińczyków na drugi brzeg rzeki.

Rys.4.1.1. Symulacja powrotu jednego z chłopców na lewy brzeg. Należy „chwytać" myszką poszczególne obiekty i przeciągać we właściwe miejsce.
Źródło: E. Bryniarski, materiały do wykładu z *Informatyki szkolnej*

REPREZENTACJE

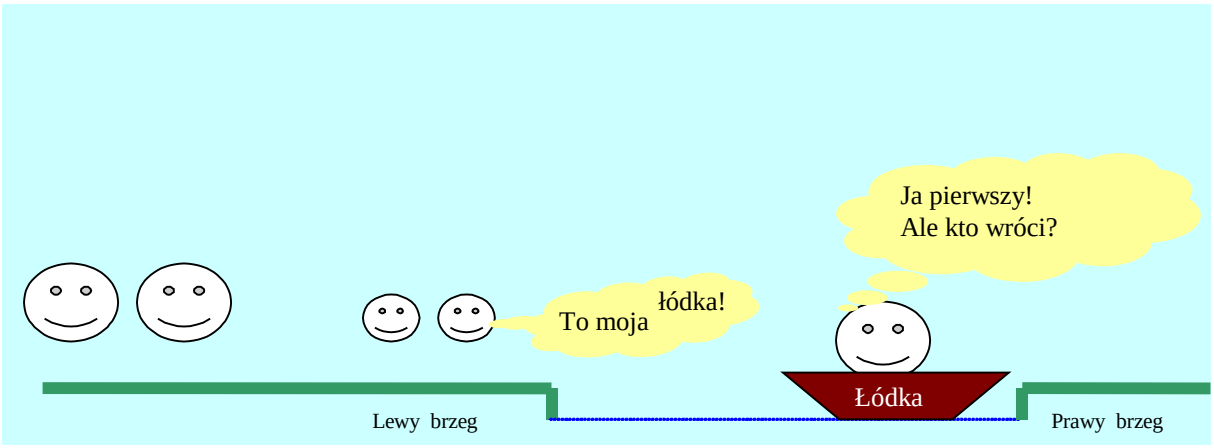
1. ***Reprezentacja ikoniczna*** – przedstawienie graficzne treści zadania na monitorze za pomocą edytora grafiki.

Wyobrażenie treści zadania możemy przedstawić graficznie następująco (rys. 4.1.2):

Rys.4.1.2. Graficzne przedstawienie treści zadania o Wielkim Marszu Chińczyków.
Źródło: E. Bryniarski, materiały do wykładu z *Informatyki szkolnej*

2. *Problem.*

Pojawia się sytuacja problemowa:



Rys.4.1.3. Ilustracja sytuacji problemowej w zadaniu o „Wielkim Marszu Chińczyków"
Źródło: E. Bryniarski, materiały do wykładu z *Informatyki szkolnej*

Czegoś nie wiemy! Nasze wyobrażenia o przeprawieniu się Chińczyków przez rzekę, a także wyniki symulacji dokonywanej w edytorze graficznym, tj. rzeczywistości wirtualne wytworzone w wyniku interakcji ze środkiem informatycznym, nie są adekwatne do rzeczywistości poznawczej, na którą wskazuje treść zadania. Ta nieadekwatność jest właściwym problemem informatycznym, który musimy pokonać. Najpierw jednak należy sformułować wnioski, wynikające z dotychczasowego doświadczenia w rozwiązywaniu zadania.

3. *Reprezentacja symboliczna.*

Wnioski:

- Na prawym brzegu powinien być co najmniej jeden z chłopców, ażeby łódka mogła powrócić na lewą stronę rzeki.
- Jeśli na prawym brzegu nie ma ani jednego chłopca, muszą się najpierw obaj przeprawić na prawy brzeg (jeden zostaje, drugi wraca).

Widzimy, że we wnioskach abstrahujemy od nieistotnych w rozwiązaniu cech i własności rzeczy: łódki, brzegów rzeki oraz grup osób składających się z Chińczyków i chłopców. Zauważmy, że nie jest istotny stan określający płynięcie łódki, gdyż o przeprawie przez rzekę decyduje wsiadanie do łódki i wysiadanie z łódki przepływających nią osób. Tak więc istotnym dla nas stanem rzeczy jest związek pomiędzy grupami osób na lewym i prawym brzegu rzeki z położeniem łódki (wsiadanie, wysiadanie na lewym lub prawym brzegu). W trakcie przeprawiania się przez rzekę zachodzą ściśle określone przejścia jednych stanów w drugie.

Na poziomie szkoły podstawowej i gimnazjum należy się ograniczyć do opisu słownego stanów i reguł postępowania w „Świecie Wielkiego Marszu", ale na poziomie liceum warto się pokusić o bardziej sformalizowany opis. Przedstawimy go poniżej.

Niech „a" oznacza Chińczyka, „b" – chłopca. Grupę osób składających się z Chińczyków i chłopców może reprezentować za pomocą wyrażenia algebraicznego:

$n*a + k*b$,

gdzie n oznacza liczbę Chińczyków, a k – liczbę chłopców. Brak osób jest szczególnym przypadkiem pojęcia grupy osób – „grupy pustej" – odpowiada jej wyrażenie „0". Położenie łódki na lewym brzegu, będziemy oznaczać przez „L", a na prawym – „P".

Stany S (x, y, z), które mają tu miejsce, opisane są przez zmienne x, y, z, reprezentujące odpowiednio grupę osób na lewym brzegu, grupę osób na prawym brzegu oraz położenie łódki (L lub P). Łączna grupa osób znajdujących się po lewej lub po prawej stronie rzeki nie ulega zmianie, jest więc reprezentowana przez wyrażenie: $3*a + 2*b$. Stan początkowy można opisać więc wyrażeniem: $S(3*a + 2*b, 0, L)$.

Uwaga! Nawet rozwiązując ten problem w wyobraźni, abstrahujemy od cech i własności poznawanych rzeczy (od rzeczywistości poznawczej) nadając strukturze wyobrażanej grupy osób postać wyrażenia algebraicznego (rzeczywistość wirtualna).

4. Reguły

Przyjmijmy w tej części notację zaczerpniętą z języka Turbo Prolog:

- „A :- B" oznacza, że A zachodzi, jeśli B zachodzi
- x= a; b; c; ..." oznacza, że x=a lub x=b lub x=c, itd.
- „A, B, ..." oznacza to samo, co „A i B i ...".
- „A; B; ..." oznacza to samo, co „A lub B lub ...".
- przez „not A" rozumiemy, że A nie zachodzi.

Zmienne: u, v, x, y, z.

Dziedzina zmiennych:

*D = {0, a, b, 2*a, 2*b, 3*a, a + b, a + 2*b, 2*a + b, 2*a + 2*b, 3*a + b, 3*a + 2*b}.*

Prawa:

P1. ***A(x) :- x = a; 2*a; 3*a; a + b; a + 2*b; 2*a + b; 2*a + 2*b; 3*a + b; 3*a + 2*b.***
„*A(x)*" czytamy: w grupie x znajduje się Chińczyk. W grupie x znajduje się Chińczyk, gdy znajduje się w niej co najmniej jeden Chińczyk.

P2. ***B(x) :- x = b; 2*b; a + b; a + 2*b; 2*a + b; 2*a + 2*b; 3*a + b; 3*a + 2*b.***
„*B(x)*" czytamy: w grupie x znajduje się chłopiec. W grupie x znajduje się chłopiec, gdy znajduje się w niej co najmniej jeden chłopiec.

P3. ***S(x, y, P) :- x + y = 3*a + 2*b.***
Gdy łódka znajduje się na prawym brzegu rzeki, na obu brzegach jest łącznie stała grupa osób: trzech Chińczyków i dwoje chłopców.

P4. ***S(x, y, L) :- x + y = 3*a + 2*b.***
Gdy łódka znajduje się na lewym brzegu rzeki, na obu brzegach jest łącznie stała grupa osób: trzech Chińczyków i dwoje chłopców.

P5. ***S(x, y, P) :- S(u, 0, L), x = u - 2*b, y = 2*b.***
Jeśli po prawej stronie rzeki nie ma nikogo, a łódka znajduje się po lewej stronie, to dwóch chłopców musi przepłynąć łódką na drugi brzeg rzeki.

P6. ***S(x, y, L) :- S(u, v, P), A(u), not B(u), x = u + b, y = v – b.***
Jeśli po lewej stronie rzeki jest co najmniej jeden Chińczyk i nie ma żadnego chłopca, a łódka znajduje się po prawej stronie, to jeden z chłopców z prawej strony rzeki musi przepłynąć na lewą stronę.

P7. ***S(x, y, L) :- S(u, v, P), A(u), B(u), B(v), x = u + b, y = v – b.***
Jeśli po lewej stronie rzeki jest co najmniej jeden Chińczyk i jeden z chłopców oraz po drugiej stronie rzeki jest drugi chłopiec, a łódka znajduje się po prawej stronie, to jeden z chłopców z prawej strony rzeki musi przepłynąć na lewą stronę.

P8. ***S(x, y, P) :- S(u, v, L), A(u), A(v), not B(v), x = u – 2*b, y = v + 2*b.***
Jeśli po lewej stronie rzeki jest co najmniej jeden Chińczyk i nie ma żadnego chłopca, a łódka znajduje się po lewej stronie, to dwóch chłopców z lewej strony rzeki musi przepłynąć na prawą stronę.

P9. ***S(x, y, P) :- S(u, v, L), A(u), A(v), B(u), B(v), x = u – a, y = v + a.***
Jeśli po lewej stronie rzeki jest co najmniej jeden Chińczyk i jeden z chłopców oraz po drugiej stronie rzeki jest drugi chłopiec, a łódka znajduje się po lewej stronie, to jeden Chińczyk z lewej strony rzeki musi przepłynąć na prawą stronę.

P10. ***S(x, y, L) :- S(u, v, P), not A(u), B(u), B(v), x = u + b, y = v – b.***
Jeśli po lewej stronie nie ma żadnego Chińczyka, a jest jeden z chłopców oraz po drugiej stronie rzeki jest drugi chłopiec, a łódka znajduje się po prawej stronie, to chłopiec z prawej strony rzeki musi przepłynąć na lewą stronę.

P11. ***S(x, y, L) :- S(u, v, P), A(v), not B(v), x = u + a, y = v - a.***
Jeśli po prawej stronie rzeki jest co najmniej jeden Chińczyk ale nie ma chłopców, a łódka także znajduje się po prawej stronie, to jeden Chińczyk z prawej strony rzeki musi wrócić na lewą stronę.

P12. ***S(x, y, P) :- S(u, v, L), A(u), not B(v), x = u - a, y = v + a.***
Jeśli po lewej stronie rzeki jest co najmniej jeden Chińczyk, po prawej nie ma chłopców, a łódka znajduje się po lewej stronie, to jeden Chińczyk z lewej strony rzeki może przeprawić się na prawą stronę.

Uwaga! Ze względu na aspekt moralny, prawo **P11** stosuje się nawet wtedy, gdy wszyscy Chińczycy przepłyną rzekę. W przeciwnym wypadku chłopcy nie odzyskają swojej łódki. Podobnie, ze względów moralnych, nie powinno się stosować prawa **P12**, jeśli na lewym brzegu jest jeden Chińczyk, chyba że zastosuje się następnie prawo **P11**, ale nie jest to efektywna metoda przeprawienia się Chińczyków przez rzekę.

Jak sprawdzić, że są to już wszystkie ważne prawa opisujące „świat przeprawy przez rzekę na drodze Wielkiego Marszu"? Takie sprawdzenie umożliwia reprezentacja enaktywna tego świata, wiążąca przyczynowo spełnianie lub nie spełnianie wszystkich cech i własności opisywanego świata z wykonywanymi operacjami, mogącymi doprowadzić do realizacji celu. Dogodną formą reprezentacji enaktywnej w tym przypadku, jak wiemy, są tablice decyzyjne, chociaż przy pewnych uproszczeniach, na tym etapie rozwiązywania problemu można się ograniczyć do opisów słownych (np. na poziomie gimnazjum).

5. Reprezentacja enaktywna

Tablica decyzyjna dotycząca „przeprawy przez rzekę" obejmuje pięć warunków i trzy operacje:

Warunki:

u + v = 3*a + 2*b - łączna grupa osób na lewym i prawym brzegu rzeki wynosi 3*a + 2*b,

A(u) – co najmniej jeden Chińczyk na lewym brzegu,

A(v) - co najmniej jeden Chińczyk na prawym brzegu,

B(u) - co najmniej jeden chłopak na lewym brzegu,

B(v) - co najmniej jeden chłopak na prawym brzegu,

s: = L – łódka znajduje się przy lewym brzegu.

Operacje:

x: = - nadanie nowej wartości zmiennej **x**: grupa osób na lewym brzegu,

y: = - nadanie nowej wartości zmiennej **y**: grupa osób na prawym brzegu,

z: = - nowa pozycja łódki (są ważne tylko dwie pozycje: łódka na lewym brzegu – L, łódka na prawym brzegu – P).

REGUŁY / WARUNKI	R1	R2	R3	R4	R5	R6	R7		
u + v = 3*a + 2*b	T	T	T	T	T	T	T	T	T
A(u}	T	N	T	T	T	N	T	N	T
A(v)	T	T	N	T	T	T	N	T	N
B(u)	T	T	T	N	T	T	T	N	N
B(v)	T	T	T	T	N	N	N	T	T
s: = L	T	T	T	T	T	T	T	T	T
OPERACJE									
x: =	u - a	u - b	u - a	u - a	u - 2*b	3*a	u - 2*b		
y: =	v + a	v + b	v + a	v + a	v + 2*b	2*b	v + 2*b		
z: =	P	P	P	P	P	L	P	sprz.	sprz.

REGUŁY / WARUNKI	R8	R9	R10	R11	R12	R13	R14		
u + v = 3*a + 2*b	T	T	T	T	T	T	T	T	T
A(u}	T	N	T	T	N	T	N	T	T
A(v)	T	T	N	T	T	N	T	N	T
B(u)	T	T	T	N	N	N	T	T	T
B(v)	T	T	T	T	T	T	N	N	N
s: = L	N	N	N	N	N	N	N	N	N
OPERACJE									
x: =	u + b	u + b	u + b	u + b	u+2*b	u + b	u + a		
y: =	v - b	v - b	v - b	v - b	v - 2*b	v - b	v - a		
z: =	L	L	L	L	L	L	L	sprz.	sprz.

Rys.4.1.4. Tablica decyzyjna dot. zadania o „Wielkim Marszu Chińczyków".
Źródło: E. Bryniarski, materiały do wykładu z Informatyki szkolnej.

Wyróżnienie pięciu warunków w tabeli decyzyjnej prowadzi do 32 (2^5) różnych wartościowań logicznych tych warunków – na 32 różne sposoby można nadać tym warunkom wartość prawdy (czy warunek jest prawdziwy – tak: T) i fałszu (czy warunek jest prawdziwy – nie: N). Niezwykle pouczające dla ucznia może być to, gdy zauważy, iż nie wszystkie wartościowania są adekwatne do poznawanej rzeczywistości. Można łatwo wskazać takie wartościowania, które odpowiadają sytuacjom nie mogącym zaistnieć w rzeczywistości. Na przykład: sumaryczna grupa osób po obu stronach rzeki jest większa niż zakłada zadanie, lub wszystkie osoby znajdują się po jednej stronie, a łódka po drugiej, choć nie miał jej kto tam przetransportować (patrz w tabeli kolumny, w których odnotowano sprzeczność: *sprz.*). Po znalezieniu wszystkich nieadekwatnych wartościowań, wykreślamy je z tabeli decyzyjnej. Pozostałe wartościowania określają przesłanki wszystkich możliwych reguł decyzyjnych. W tabeli oznaczone one zostały symbolami R1-R14. Nie wszystkie z tych przesłanek są zgodne z przesłankami praw, które wcześniej zostały sformułowane. Zauważmy, że regułom decyzyjnym R4, R6 i R12 nie odpowiadają żadne z wcześniej sformułowanych praw. Te dodatkowe prawa możemy wyrazić następująco:

P13. ***S(x, y, P) :- S(u, v, L), A(u), A(v), not B(u), x = u – a, y = v + a.***

Jeżeli na prawym i lewym brzegu są Chińczycy, a na lewym brzegu nie ma żadnego chłopca oraz łódka znajduje się przy lewym brzegu, to jeden z Chińczyków musi się przeprawić przez rzekę.

P14. ***S(x, y, L) :- S(u, v, L), not A(u), not B(v), x = 3* a, y = 2*b.***

Gdy nie ma Chińczyków na lewym brzegu rzeki oraz chłopców na prawym brzegu rzeki, a łódka znajduje się na lewym brzegu, to na lewym brzegu jest dwóch chłopców z łódką oraz wszyscy Chińczycy przeprawili się przez rzekę.

P15. ***S(x, y, L) :- S(u, v, P), not A(u), not B(u), x = u + 2*b, y = v – 2*b.***
Gdy wszystkie osoby znajdują się wraz z łódką po prawej stronie, to chłopcy wracają na lewy brzeg (a Chińczycy maszerują dalej).

Zauważmy, że podane prawa nie są równokształtne z wiernym tłumaczeniem reguł decyzyjnych, ale w prosty sposób z tych tłumaczeń wynikają. Na przykład reguła R1 ma tłumaczenie:

R1. ***S(x, y, P) :- A(u), A(v), B(u), B(v), s = L, u + v = 3*a + 2*b, x = u – a, y = v + a.***

Ponieważ z założeń zadania wynika, że wyrażenie

A(u), A(v), B(u), B(v), s = L, u + v = 3*a + 2*b

jest równoważne wyrażeniu

S(u, v, L), A(u), A(v), B(u), B(v),

więc **R1** możemy zapisać jako

P9. ***S(x, y, P) :- S(u, v, L), A(u), A(v), B(u), B(v), x = u – a, y = v + a.***
Z niektórych praw wynikają dwie reguły decyzyjne i na odwrót.

Czy tablice decyzyjne są kompletne? Niestety nie. Pouczające dla ucznia może być to, że przy niektórych przesłankach reguł decyzyjnych sensowne będzie także wykonywanie innych operacji, ale wykonanie tych operacji czyni przeprawę Chińczyków przez rzekę mało efektywną. Na przykład, gdy wszyscy są na lewym brzegu rzeki oraz łódka jest na lewym brzegu, to mało efektywne dla przeprawy przez rzekę jest przeprawienie się najpierw Chińczyka, albo jednego z chłopców, gdyż zarówno Chińczyk jak i chłopiec będą musieli wrócić na lewy brzeg, operacja ta więc nie doprowadzi do żadnej istotnej zmiany – przeprawa przez rzekę będzie znajdowała się dalej w tej samej fazie.

6. Pojęcia.

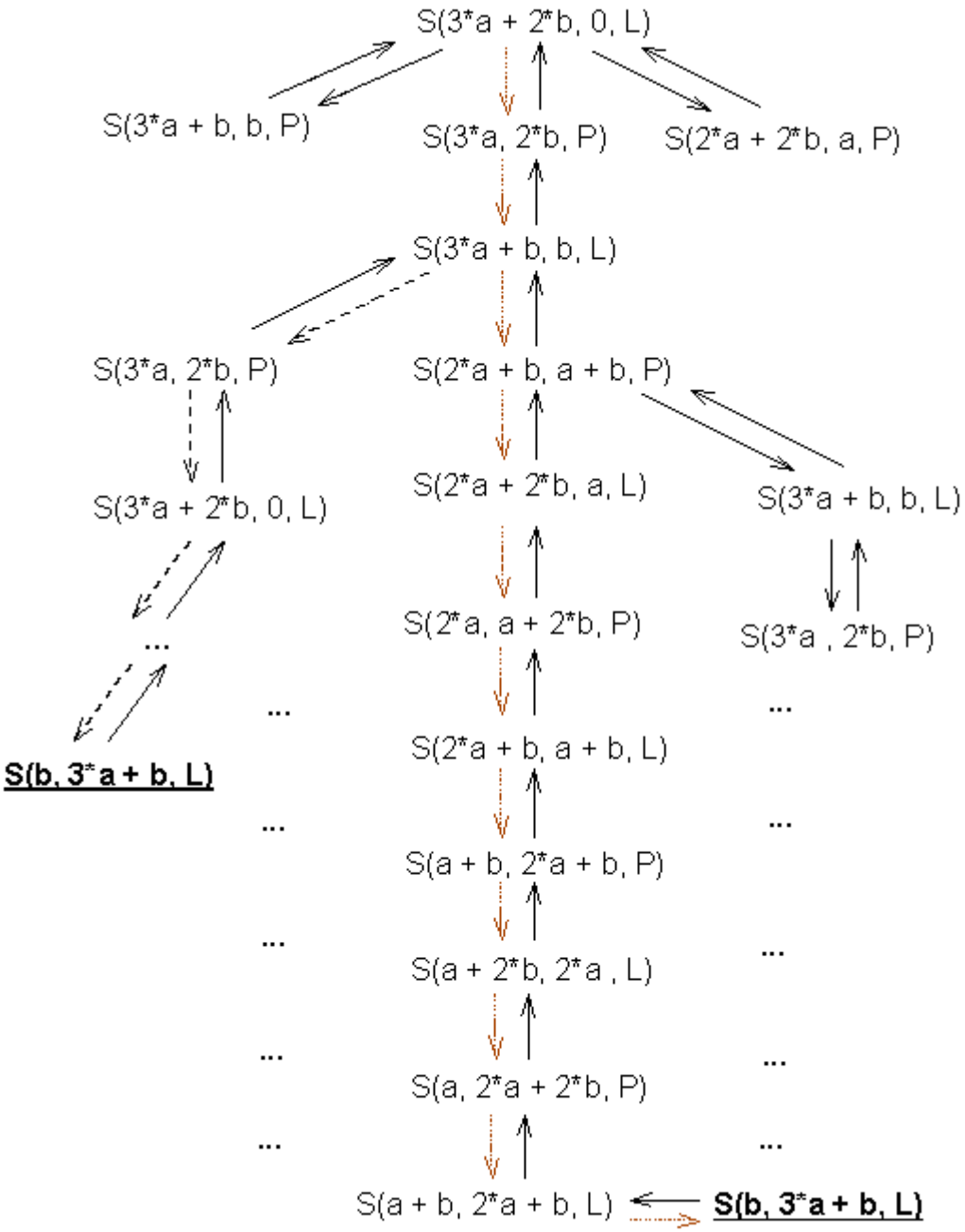
Podsumowując, opisane reprezentacje, dostrzeżenie możliwości pojawienia się sytuacji problemowej oraz wiedza o operacjach pozwalających na dokonywanie przeprawy „dają dopiero jakieś pojęcie” o przeprawieniu się Chińczyków przez rzekę. Innymi słowy, uczeń uczestnicząc w systemie reprezentowania obiektu jakim jest dla niego przepłynięcie przez rzekę łódką trzech osób przy podanych w treści zadania ograniczeniach, kształtuje pojęcie „takiej przeprawy przez rzekę”, pojęcie, które pozwoli mu zidentyfikować reprezentowany obiekt, tzn. rozwiązać zadanie. Ale ażeby tego dokonał musi najpierw uporządkować swą wiedzę na ten temat, musi stać się ekspertem w rozwiązywaniu danego zadania. System ekspertowy rozwiązywania danego zadania umożliwi zbudowanie

optymalnego algorytmu „przeprawy przez rzekę”. Na tym etapie rozwiązywania zadania uczeń ma szansę uświadomić sobie, że warto było ponieść trud lepszego poznania „świata przeprawy przez rzekę...”. Pozwala to jemu uniknąć wielu błędów, bylejakości i niechlujstwa przy szczegółowym rozpisaniu algorytmu na elementarne zadania: bloki decyzyjne wraz z blokami akcji (operacji). Nauczyciel powinien mieć na uwadze, że takie podejście do rozwiązywania problemów, zarówno dla prezentowanego tu rozwiązania zadania, jak i przy rozwiązywaniu innych zadań, ma niezaprzeczalne walory wychowawcze.

SYSTEM EKPERTOWY

1. ***Baza wiedzy*** – zbiór faktów i praw potrzebnych do rozwiązania zadania. Wystarczy ograniczyć się do faktu S(3*a + 2*b, 0, L) opisującego stan początkowy oraz tych praw, które nie prowadzą do faktów opisujących powrót Chińczyka z prawego brzegu na lewy brzeg. Nauczyciel powinien bazę wiedzy dostosować do poziomu nauczania (podstawowego, gimnazjalnego oraz licealnego i wyższego).
2. ***Reprezentacje*** – ikoniczna, symboliczna i enaktywna, opisane wyżej. Wykorzystanie którejś z reprezentacji zależy od fazy rozwiązywania problemu oraz poziomu nauczania.

3. ***Sieć semantyczna.***



Rys.4.1.5. Sieć semantyczna do zadania o „Wielkim Marszu Chińczyków”.
Źródło: opracowanie własne na podstawie: E. Bryniarski, wykłady z *Informatyki szkolnej*

4. Operacje

Analiza sieci semantycznej pozwala wyróżnić następujące operacje:

- S(x, y, L) |→ S(x – 2*b, y + 2*b, P),
- S(x, y, P) |→ S(x – 2*b, y + 2*b, L),
- S(x, y, L) |→ S(x – b, y + b, P),
- S(x, y, P) |→ S(x – b, y + b, L),
- S(x, y, L) |→ S(x + 2*b, y - 2*b, P),
- S(x, y, P) |→ S(x + 2*b, y - 2*b, L),

- S(x, y, L) |→ S(x + b, y - b, P),
- S(x, y, P) |→ S(x + b, y - b, L),
- S(x, y, L) |→ S(x – a, y + a, P),
- S(x, y, P) |→ S(x – a, y + a, L),
- S(x, y, L) |→ S(x + a, y - a, P),
- S(x, y, P) |→ S(x + a, y - a, L),

5. ***Rama*** – zbiór dróg rozwiązań prowadzących od stanu początkowego do stanu końcowego: wyniku rozwiązania zadania. Przykładowo na diagramie wyróżniono dwie takie drogi zaznaczone liniami: przerywaną i kropkowaną. Najkrótsza jest droga zaznaczona linią kropkowaną.

6. ***Realizacja rozwiązania problemu*** – możliwie najkrótsza droga rozwiązania problemu mieszcząca się w ramie rozwiązania. Umożliwia ona budowę optymalnego algorytmu i odpowiedni dobór kompilacji, procesora monitorowania i przedstawienia wyniku w monitorze, tj. umożliwia poprawną konstrukcję środka informatycznego prezentującego identyfikację obiektu „Przeprawa przez rzekę na drodze Wielkiego Marszu", zgodnie z zasadą adekwatności (patrz: 3.2).

Zadanie 2. *ODMIERZANIE WODY* *(łamigłówka)*

Marcysia, gosposia niezbyt obyta, grochówkę uwarzyć chciała.
W babcinym, starym jak świat, kajecie, przepis wnet wygrzebała.
Że akuratna być postanowiła, głowić się srodze poczęła,
Bo podług przepisu, wody ze źródła cztery litry jej trzeba.
Jakim sposobem odmierzy nieszczęsne CZTERY LITRY WODY,
gdy DZBANEK TRZY LITRY MIEŚCI, a GARNEK – PIĘCIOLITROWY?
Kto jej pomoże, a najmniej się strudzi owym wody przelewaniem,
od dzielnej Marcysi pełną miseczkę grochówki dymiącej dostanie.

(tekst: opracowanie własne)

ŚRODEK INFORMATYCZNY (TECHNOLOGIA INFORMACYJNA)

1. ***Algorytm*** – algorytm symulacji takiego przelewania wody, aby po którymś przelaniu uzyskać dokładnie 4 litry wody.
2. ***Kompilacja*** – symulowane muszą być: ilość wody w dzbanku, ilość wody w garnku, przelewanie wody z jednego naczynia do drugiego, wylewanie wody z któregoś naczynia i nalewanie wody do któregoś naczynia – użytkownik komputera „przelewa" wodę przy użyciu myszki, korzystając z programu „*Symulacja2*".[18]
3. ***Procesor*** – kompilacja algorytmu dokonywana w programie symulującym przelewanie wody.[19]

[18] Patrz: załączniki do niniejszej pracy: program „Symulacja2" na płycie CD
[19] Patrz: „Symulacja2"...

4. ***Monitorowanie*** – sterowanie obiektami graficznymi obrazowane jest za pomocą ikon na ekranie monitora jako napełnianie się lub opróżnianie naczyń.
5. ***Monitor*** (***system multimedialny***) – ekran monitora komputera wraz z myszką.
6. ***Implementacja*** – uzyskanie stanu odpowiadającego sytuacji uzyskania 4 litrów wody w jednym z naczyń.

REPREZENTACJE

1. ***Reprezentacja ikoniczna*** – przedstawienie graficzne treści zadania – rysunek wykonany przy pomocy edytora grafiki *Paint.*

Wyobrażenie treści zadania możemy przedstawić graficznie następująco:

Jakie czynności może wykonać Marcysia?

1. Napełnić dzbanek wodą ze źródła.
2. Napełnić garnek wodą ze źródła.
3. Przelać wodę z garnka do dzbanka.
4. Przelać wodę ze dzbanka do garnka.
5. Wylać całą zawartość dzbanka.
6. Wylać całą zawartość garnka.

Rys. 4.1.6. Graficzne wyobrażenie treści zadania o odmierzaniu wody.
Źródło: opracowanie własne

2. ***Problem.***

Nasze wyobrażenia o sposobie odmierzenia żądanej ilości wody nie są adekwatne do rzeczywistości poznawczej, na którą wskazuje treść zadania – reprezentacja ikoniczna. Nie można bowiem uzyskać jednego litra wody (czyli także żądanej ilości wody) bezpośrednio poprzez jedno przelanie. Ta nieadekwatność jest tutaj problemem informatycznym, który musimy pokonać. Sformułujmy więc wnioski, wynikające z dotychczasowego doświadczenia w rozwiązywaniu zadania w wyniku przeprowadzenia symulacji.

3. ***Reprezentacja symboliczna.***

Wnioski:

- Naczynie może być napełnione.
- Naczynie może być opróżnione.
- Wodę można przelewać z jednego naczynia do drugiego, do opróżnienia się pierwszego z nich lub wypełnienia drugiego.

Istotnym dla nas stanem rzeczy jest związek pomiędzy zawartościami obu naczyń. W trakcie nalewania, przelewania, wylewania wody zachodzą ściśle określone przejścia jednych stanów w drugie. Nietrudno więc sformułować treść zadania (prawa rządzące „Światem Odmierzania Wody") za pomocą klauzul hornowskich (patrz: 1.10), przyjmując notację klauzulową lub języka Turbo Prolog.

Interpretujemy wyrażenie

Stan (u, v)

jako wyrażające sytuację, w której pięciolitrowy garnek zawiera ***u*** litrów wody, a trzylitrowy dzbanek – ***v*** litrów. Zakładamy, że relacje $x + y = z$ oraz $x \leq y$ są już zdefiniowane. Mamy wtedy **klauzule**:

(NW1) *Stan (0, 0)* /*stan początkowy*/

(NW2) *Stan (4, y)* */* stan docelowy*/*

(NW3) *Stan (5, y)* *Stan (x, y)* */*akcja napełniania garnka*/*

(NW4) *Stan (x, 3)* *Stan (x, y)* */*akcja napełniania dzbanka*/*

(NW5) *Stan (0, y)* *Stan (x, y)* */*akcja opróżniania garnka*/*

(NW6) *Stan (x, 0)* *Stan (x, y)* */*akcja opróżniania dzbanka*/*

(NW7) *Stan (0, y)* *Stan (u, v) , u + v = y , y ≤ 3*
*/*przelewanie z garnka do dzbanka do chwili opróżnienia się garnka */*

(NW8) *Stan (x, 0)* *Stan (u, v) , u + v = x , x ≤ 5*
*/*przelewanie z dzbanka do garnka do chwili opróżnienia się dzbanka */*

(NW9) *Stan (5, y)* *Stan (u, v) , u + v = w , 5 + y = w*
*/*przelewanie z dzbanka do garnka do chwili napełnienia się garnka*/*

(NW10) *Stan (x, 3)* *Stan (u, v) , u + v = w , 3 + x = w*
*/*przelewanie z garnka do dzbanka do chwili napełnienia się dzbanka*/*

4. Reguły (w notacji *Turbo Prologa*)**:**

- „ *A :- B* " oznacza, że A zachodzi, jeśli B zachodzi
- „ *x = a; b; c; ...*" oznacza, że $x = a$ lub $x = b$ lub $x = c$, itd.
- „ *A, B, ...* " oznacza to samo, co „ A i B i ... ".
- „ *A; B; ...* " oznacza to samo, co „ A lub B lub ... ".

Zmienne: u, v, x, y, w.

Dziedzina zmiennych:

$D = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$.

Prawa: jw. – patrz: *NW1 – NW10.*

5. Reprezentacja enaktywna

Tablica decyzyjna dotycząca odmierzania wody obejmuje następujące warunki i operacje:

Warunki:

$u + v \leq 8$ – łączna ilość wody w naczyniach nie przekracza 8 l.

$u + v = y,\ y \leq 3$ – łączna ilość wody w naczyniach nie przekracza pojemności dzbanka
(warunek przelania z garnka do dzbanka do chwili opróżnienia się garnka)

u + v = x, $x \leq 5$ – łączna ilość wody w naczyniach nie przekracza pojemności garnka
(warunek przelania z dzbanka do garnka do chwili opróżnienia się dzbanka)

u + v = w, 5 + y = w – warunek przelania z dzbanka do garnka do chwili napełnienia się garnka

u + v = w, 3 + x = w – warunek przelania z garnka do dzbanka do chwili napełnienia się dzbanka

Operacje:

x: = - nadanie nowej wartości zmiennej **x**: ilość wody w garnku,

y: = - nadanie nowej wartości zmiennej **y**: ilość wody w dzbanku.

Reguły decyzyjne[20] (w notacji Turbo Prologa) :

(R1) *S(5, y)* **:-** *S(x, y)* */*akcja napełniania garnka*/*

(R2) *S(x, 3)* **:-** *S(x, y)* */*akcja napełniania dzbanka*/*

(R3) *S(0, y)* **:-** *S(x, y)* */*akcja opróżniania garnka*/*

(R4) *S(x, 0)* **:-** *S(x, y)* */*akcja opróżniania dzbanka*/*

(R5) *S(0, y)* **:-** *S(u, v) , u + v = y , y ≤ 3*
*/*przelewanie z garnka do dzbanka do chwili opróżnienia się garnka */*

(R6) *S(x, 0)* **:-** *S(u, v) , u + v = x , x ≤ 5*
*/*przelewanie z dzbanka do garnka do chwili opróżnienia się dzbanka */*

(R7) *S(5, y)* **:-** *S(u, v) , u + v = w , 5 + y = w*
*/*przelewanie z dzbanka do garnka do chwili napełnienia się garnka*/*

(R8) *S(x, 3)* **:-** *S(u, v) , u + v = w , 3 + x = w*
*/*przelewanie z garnka do dzbanka do chwili napełnienia się dzbanka*/*

REGUŁY **WARUNKI**	**R1**	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**
u + v ≤ 8	T	T	T	T	T	T	T	T
u + v = y, y ≤ 3	N	N	N	N	T	N	N	N
u + v = x, x ≤ 5	N	N	N	N	N	T	N	N
U + v = w , 5 + y = w	N	N	N	N	N	N	T	N
u + v = w , 3 + x = w	N	N	N	N	N	N	N	T
OPERACJE:								
x : =	**5**	**x**	**0**	**x**	**0**	**x**	**5**	**x**
y : =	**y**	**3**	**y**	**0**	**y**	**0**	**y**	**3**

Rys. 4.1.7. Tablica decyzyjna do zadania z odmierzaniem wody.
Źródło: opracowanie własne

6. Pojęcia – wszystkie możliwe stany:

S(0,0), S(0,1), S(0,2), S(0,3), S(1,0), S(1,1), S(1,2), S(1,3), S(2,0), S(2,1), S(2,2), S(2,3), S(3,0), S(3,1), S(3,2), S(3,3), S(4,0), S(4,1), S(4,2), S(4,3), S(5,0), S(5,1), S(5,2), S(5,3).

[20] Dla skrócenia zapisu dalej będziemy używać tylko pierwszej literki słowa *Stan*, czyli *S(x,y)* zamiast: *Stan(x,y)*.

Uczniowie (szkoły średniej) mogą obliczyć ilość wszystkich możliwych stanów, stosując wzory kombinatoryczne[21]: $C_6^1 \cdot C_4^1 = 6 \cdot 4 = 24$, albo po prostu mnożąc ilość wszystkich możliwych poziomów wody w garnku przez ilość takowych poziomów w dzbanku.

SYSTEM EKPERTOWY

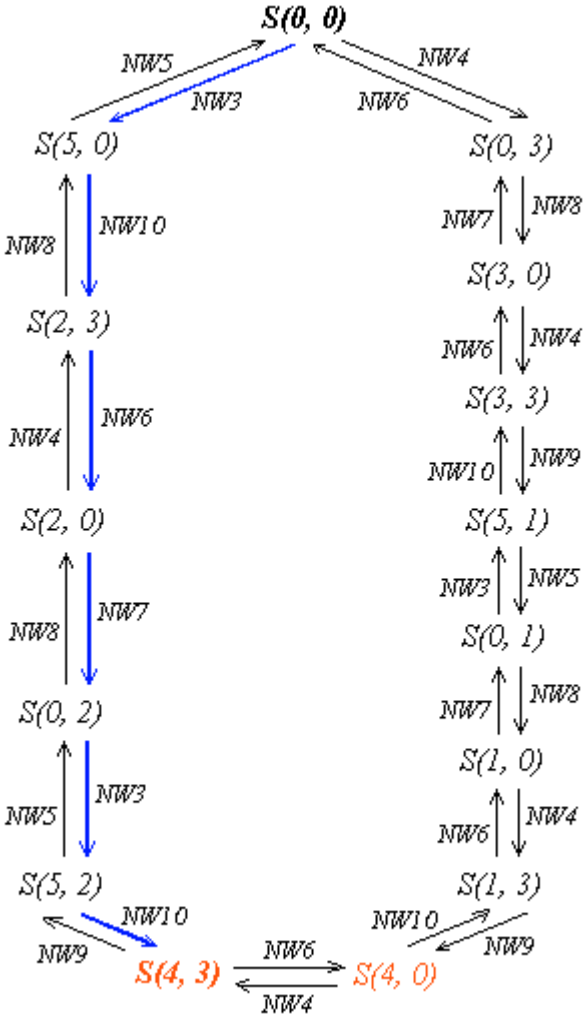
1. ***Baza wiedzy*** – zbiór faktów i praw potrzebnych do rozwiązania zadania. Nauczyciel powinien bazę wiedzy dostosować do poziomu nauczania (podstawowego, gimnazjalnego oraz licealnego i wyższego). Możliwości przedstawienia bazy wiedzy jest wiele: prace z wykorzystaniem edytorów grafiki (schematy, rysunki), własne programy[22], prezentacje, inscenizacje z wykorzystaniem rekwizytów, nagrania dźwiękowe, animacje (filmy), gry interaktywne itp.

2. ***Reprezentacje*** – ikoniczna, symboliczna i enaktywna, opisane wcześniej. Wykorzystanie którejś z reprezentacji zależy od fazy rozwiązywania problemu oraz poziomu nauczania.

3. ***Sieć semantyczna.***

[21] Symbolika standardowa dla kombinatoryki: C_n^k – kombinacja k - elementowa zbioru n - elementowego.

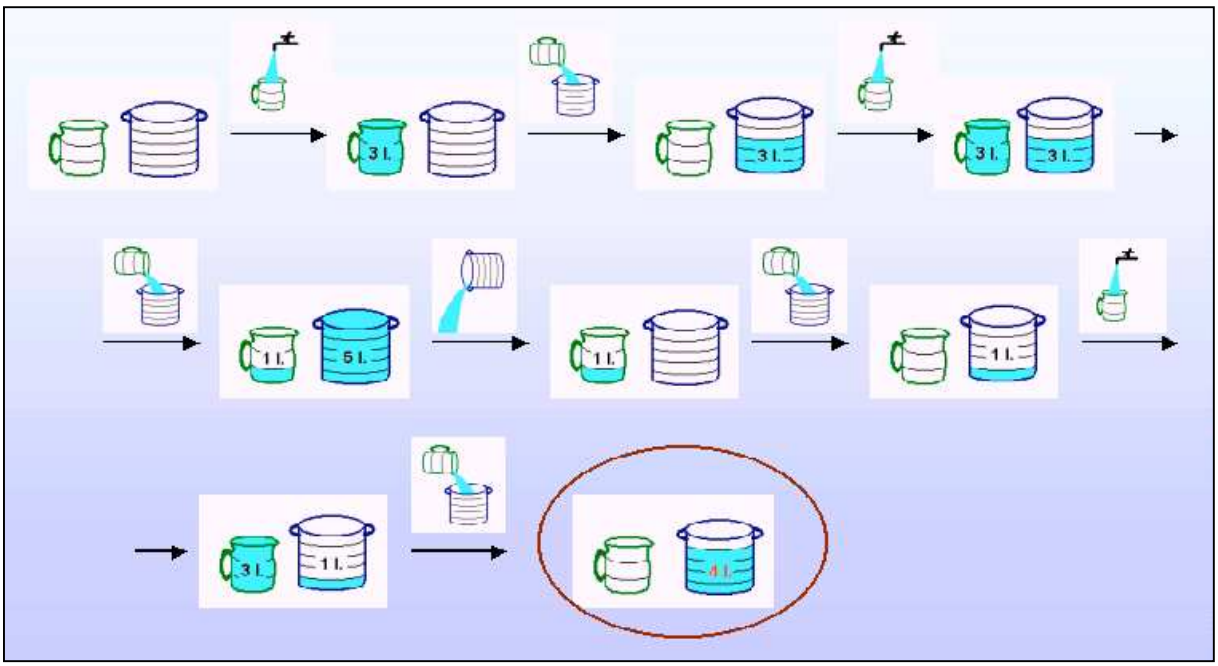
[22] Patrz: załączniki do niniejszej pracy: programy „*Symulacja2*" , „*Symulacja3*", „*Pokaz 2*" na płycie CD



Rys. 4.1.8. Sieć semantyczna do zadania o odmierzaniu wody.
Źródło: opracowanie własne

Na podstawowym poziomie nauczania zalecane jest jedynie wizualne przedstawianie sieci semantycznej, np. w formie prezentacji multimedialnej[23], pokazu, bądź w formie graficznej (rys. 4.1.9)

[23] Patrz: załączniki do niniejszej pracy: „Pokaz2" – prezentacja rozwiązania łamigłówki za pomocą programu *Power Point* - na płycie CD.



Rys. 4.1.9. Wizualne przedstawienie sieci semantycznej w zadaniu o odmierzaniu wody.
Źródło: opracowanie własne

2. ***Operacje –*** wykonywane według reguł przedstawionych wizualnie lub symbolicznie (patrz reguły).

3. ***Rama*** – Zbiór dróg rozwiązań prowadzących od stanu początkowego do stanu końcowego: wyniku rozwiązania zadania. Na diagramie sieci semantycznej widzimy dwie takie drogi prowadzące od stanu początkowego *S(0, 0)* do stanu końcowego *S(4, 3)*. Krótszą drogę wyznaczają strzałki niebieskie.

4. ***Realizacja rozwiązania problemu*** – możliwie najkrótsza droga rozwiązania problemu mieszcząca się w ramie rozwiązania[24] (krótsza droga w sieci semantycznej oznaczona niebieskimi strzałkami). Umożliwia ona budowę optymalnego algorytmu, odpowiedni dobór kompilacji, procesora, monitorowania i przedstawienia wyniku w monitorze.

Zadanie 3. *MAŁA PRZEPRAWA PRZEZ RZEKĘ* *(łamigłówka)*

MĘŻCZYZNA, KOZA, WILK I SAŁATA znajdują się na lewym brzegu rzeki. Na brzegu jest ponadto łódź na tyle pojemna, by pomieścić jednorazowo mężczyznę i tylko jedno z pozostałych stworzeń lub rzeczy. Mężczyzna wraz z anturażem pragnie przeprawić się na prawy brzeg rzeki. Jeżeli wilk i koza pozostaną razem bez dozoru mężczyzny, wilk z pewnością zje kozę. Podobnie, jeżeli bez jego dozoru koza pozostanie na brzegu razem z sałatą, koza sałatę niechybnie skonsumuje.

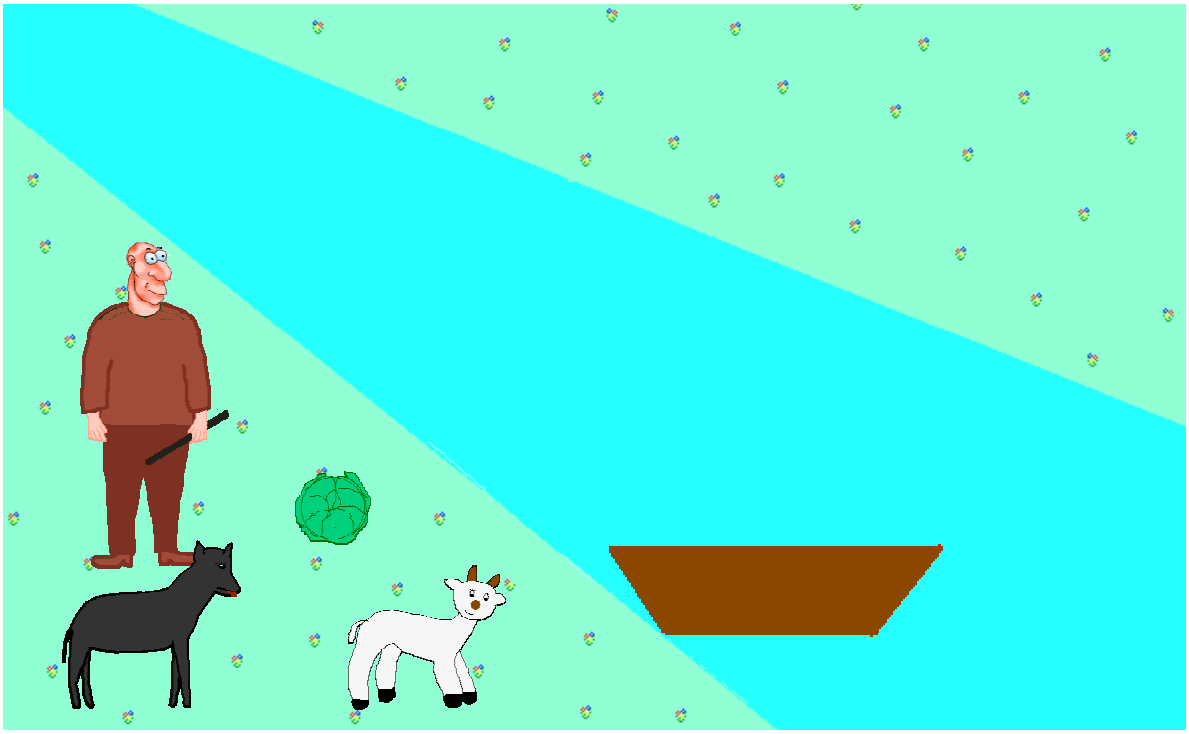
Czy możliwa jest przeprawa przez rzekę tak, by koza i sałata nie zostały zjedzone?

[24] Patrz: „Pokaz2” – j.w.

Źródło: J. Czelakowski, ćwiczenia do wykładu fakultatywnego z *Teorii modeli*

ŚRODEK INFORMATYCZNY (TECHNOLOGIA INFORMACYJNA)

1. ***Algorytm*** – algorytm symulacji takiej przeprawy przez rzekę, aby po którymś przepłynięciu wszystkie cztery stworzenia znalazły się na prawym brzegu.
2. ***Kompilacja*** – symulowane musi być: położenie mężczyzny, wilka, kozy i sałaty oraz łódki i przemieszczanie się ich wszystkich za pomocą łódki – użytkownik komputera, korzystając z myszki, steruje wyróżnionymi obiektami graficznymi („chwyta" je myszką i „przeciąga" w pożądane miejsce – rys. 4.1.10).



Rys.4.1.10. Symulacja przeprawy mężczyzny, wilka, kozy i sałaty przez rzekę. Należy „chwytać" myszką poszczególne obiekty i przeciągać we właściwe miejsce.
Źródło: opracowanie własne

3. ***Procesor*** – kompilacja algorytmu dokonywana jest w edytorze grafiki, np. za pomocą rysunków wykonanych w programie *Paint*. (patrz rys.4.1.10).
4. ***Monitorowanie*** – sterowanie obiektami graficznymi obrazowane jest za pomocą obrazków (ikon) na ekranie monitora jako przemieszczanie się tych obiektów.
5. ***Monitor*** (***system multimedialny***) – ekran monitora komputera wraz z myszką.
6. ***Implementacja*** – uzyskanie stanu odpowiadającego sytuacji przeprawienia się wszystkich stworzeń na drugi brzeg rzeki.

REPREZENTACJE

1. ***Reprezentacja ikoniczna*** – przedstawienie graficzne treści zadania – rysunek wykonany przy pomocy edytora grafiki *Paint.*

Wyobrażenie treści zadania możemy przedstawić graficznie następująco:

Rys. 4.1.11. Graficzne wyobrażenie treści zadania o „Małej Przeprawie Przez Rzekę”.
Źródło: opracowanie własne

2. Problem.

Rzeczywistości wirtualne wytworzone w wyniku interakcji ze środkiem informatycznym nie są adekwatne do rzeczywistości poznawczej, na którą wskazuje treść zadania – reprezentacja ikoniczna. Sformułujmy odpowiednie wnioski.

3. Reprezentacja symboliczna.

Wnioski:

- Kozy nie można pozostawić bez dozoru na żadnym brzegu z sałatą. Mężczyzna nie może więc zabrać do łódki wilka.
- Mężczyzna nie może również odpłynąć z sałatą (bo marny los kozy w wilczej paszczy).
- Z dwóch poprzednich wniosków wynika następny: mężczyzna powinien zabrać do łódki kozę, wtedy pozostawiony bez dozoru wilk sałaty nie tknie.
- Łódka nie wróci sama; mężczyzna musi wrócić po zwierzęta. Ba! Mężczyzna uczestniczy w <u>każdej</u> przeprawie łódki, więc w formalnym opisie stanów możemy równie dobrze przyjąć, że z niej nie wysiada.

W rozwiązaniu nieistotne są cechy i własności: łódki, brzegów, wilka, kozy, sałaty i mężczyzny. Nie jest istotny również stan określający płynięcie łódki. Co jest istotne? Interesującym nas stanem rzeczy jest związek pomiędzy istotami na lewym i prawym brzegu rzeki oraz położenie łódki. W trakcie przeprawiania się przez rzekę zachodzą ściśle określone przejścia jednych stanów w drugie.

Niech „w” oznacza wilka, „k” – kozę, „s” – sałatę. Brak stworzeń na brzegu oznaczymy przez „0” (por. zadanie 1). Grupę istot na danym brzegu rzeki niech reprezentuje suma algebraiczna, której składnikami byłyby odpowiednie litery. Istnieje 8 możliwości:

$w + k + s$	$k + s$	$w + s$	$w + k$
w	k	s	0

Położenie łódki na lewym brzegu, będziemy oznaczać przez „L”, a na prawym – „P”. Stany S (x, y, z), które mają tu miejsce, opisane są przez zmienne x, y, z, reprezentujące odpowiednio grupę istot na lewym brzegu, grupę istot na prawym brzegu oraz położenie łódki (L lub P). Łączna grupa osób znajdujących się po lewej lub po prawej stronie rzeki nie ulega zmianie, jest więc reprezentowana przez wyrażenie: $w + k + s$.

Stan początkowy: $S\ (w + k + s, 0, L)$.

4. ***Reguły*** (w notacji *Turbo Prologa*)**:**

- „ *A :- B* ” oznacza, że A zachodzi, jeśli B zachodzi
- „ $x = a; b; c;$ ...” oznacza, że $x = a$ lub $x = b$ lub $x = c$, itd.
- „ *A, B, ...* ” oznacza to samo, co „ A i B i ... ”.
- „ *A; B; ...* ” oznacza to samo, co „ A lub B lub ... ”.
- przez „not A” rozumiemy, że A nie zachodzi.

Zmienne: u, v, x, y, z.

Dziedzina zmiennych:

$D = \{ 0, w, k, s, w + s, w + k, s + k, w + k + s \}$.

Prawa: (w notacji języka Turbo Prolog, patrz: 2.3)

P1. ***S(x, y, L) :- x + y = w + k + s .***
Gdy łódka znajduje się na lewym brzegu rzeki, na obu brzegach jest łącznie stała grupa istot: wilk, koza i sałata.

P2. ***S(x, y, P) :- x + y = w + k + s .***
Gdy łódka znajduje się na prawym brzegu rzeki, na obu brzegach jest łącznie stała grupa istot: wilk, koza i sałata.

P3. ***S(x, y, P) :- not (x = w +k) .***
Gdy łódka znajduje się na prawym brzegu rzeki, na lewym brzegu koza nie może zostać z wilkiem (bo wtedy brak dozoru mężczyzny, który, jak założyliśmy, jest w łódce na przeciwnym brzegu).

P4. ***S(x, y, P) :- not (x = k +s) .***
Gdy łódka znajduje się na prawym brzegu rzeki, na lewym brzegu koza nie może zostać z sałatą (z podobnych powodów, co w P3).

P5. ***S(x, y, P) :- not (x = w +k + s) .***
Gdy łódka znajduje się na prawym brzegu rzeki, na lewym brzegu nie mogą zostać wszystkie trzy istoty razem (z podobnych powodów, jak wyżej).

P6. ***S(x, y, L) :- not (y = w +k) .***
Sytuacja analogiczna do P3: łódka z mężczyzną na brzegu lewym, więc „sąsiadujące ogniwa łańcucha pokarmowego" nie mogą zostać na brzegu prawym.

P7. ***S(x, y, L) :- not (y = k +s) .***
Sytuacja analogiczna do P4.

P8. ***S(x, y, L) :- not (y = w +k + s) .***
Sytuacja analogiczna do P5.

P9. ***S(x, y, L); S(x, y, P) :- (W(u), not W(v)); (not W(u), W(v)) .***
W(u) – na lewym brzegu jest wilk. *W(v)* – na prawym brzegu jest wilk.
Jeśli wilk jest na lewym brzegu, to nie ma go na prawym lub odwrotnie.

P10. ***S(x, y, L); S(x, y, P) :- (K(u), not K(v)); (not K(u), K(v)) .***
K(u) – na lewym brzegu jest koza. *K(v)* – na prawym brzegu jest koza.
Jeśli koza jest na lewym brzegu, to nie ma jej na prawym lub odwrotnie.

P11. ***S(x, y, L); S(x, y, P) :- (S(u), not S(v)); (not S(u), S(v)) .***
S(u) – na lewym brzegu jest sałata. *S(v)* – na prawym brzegu jest sałata.
Jeśli sałata jest na lewym brzegu, to nie ma jej na prawym lub odwrotnie.

5. Reprezentacja enaktywna

Tablica decyzyjna dotycząca „Małej Przeprawy Przez Rzekę" obejmuje następujące warunki i operacje:

Warunki:

u + v = w + k + s - łączna grupa istot na lewym i prawym brzegu rzeki wynosi ***w + k + s***,

W(u) – na lewym brzegu jest wilk,

W(v) – na prawym brzegu jest wilk,

K(u) – na lewym brzegu jest koza,

K(v) – na prawym brzegu jest koza,

S(u) – na lewym brzegu jest sałata,

S(v) – na prawym brzegu jest sałata,

ł: = L – łódka znajduje się przy lewym brzegu.

Operacje*:*

x: = – nadanie nowej wartości zmiennej **x**: grupa stworzeń na lewym brzegu,

y: = – nadanie nowej wartości zmiennej **y**: grupa stworzeń na prawym brzegu,

z: = – nowa pozycja łódki (są ważne tylko dwie pozycje: łódka na lewym brzegu – L, łódka na prawym brzegu – P).

REGUŁY / WARUNKI	R1	R2	R3	R4	R5	R6	R7	R8
u+v = w+k+s	T	T	T	T	T	T	T	T
W(u)	T	T	T	N	N	N	N	N
K(u)	T	N	N	N	T	T	T	N
S(u)	T	T	T	T	T	N	N	N
W(v)	N	N	N	T	T	T	T	T
K(v)	N	T	T	T	N	N	N	T
S(v)	N	N	N	N	N	T	T	T
ł: = L	T	N	T	N	T	N	T	N
OPERACJE:								
x : =	u - k	u	u - w	u + k	u - s	u	u - k	(stan końcowy)
y : =	v + k	v	v + w	v - k	v + s	v	v + k	
z : =	P	L	P	L	P	L	P	

Rys. 4.1.12. Tablica decyzyjna do zadania o „Małej Przeprawie Przez Rzekę”.
Źródło: opracowanie własne

W tablicy decyzyjnej mamy osiem warunków. Jest więc aż 256 (2^8) różnych wartościowań logicznych tych warunków. Nie wszystkie wartościowania są adekwatne do poznawanej rzeczywistości. Pominęłam takie wartościowania, które odpowiadają sytuacjom nie mogącym zajść. Pierwszy warunek (o łącznej liczbie osób na obu brzegach) musi być spełniony zawsze, bo nie bierzemy pod uwagę możliwości, że np. wilk przepłynie rzekę wpław, albo mężczyzna przerzuci sałatę na drugi brzeg. Niemożliwa jest też sytuacja, w której jedno ze zwierząt, sałata czy łódka znajduje się po obydwu stronach rzeki w tym samym czasie. Istnieją też wartościowania odpowiadające sytuacjom mogącym się zdarzyć, ale prowadzącym do „konfliktu” z którymś z praw „bezpiecznej przeprawy”, np. do zjedzenia kozy przez wilka, bądź sałaty przez kozę. Przy niektórych przesłankach reguł decyzyjnych sensowne będzie także wykonywanie innych operacji, ale ich wykonanie czyni przeprawę przez rzekę mało efektywną (np. przy stanie *S(k, w+s, P)* możliwy jest powrót wilka lub sałaty na lewy brzeg, ale to jedynie wydłuża czas przeprawy, która wciąż znajduje się w tej samej fazie. Dlatego takich operacji także nie uwzględniłam w tablicy decyzyjnej.

6. ***Pojęcia –*** wszystkie możliwe stany:

S(w+k+s, 0, L)
S(0, w+k+s, P)
S(w+k, s, L)
S(w+s, k, L)
S(k, w+s, L)
S(k+s, w, L)
S(w, k+s, P)
S(w+s, k, P)
S(k, w+s, P)
S(s, w+k, P)

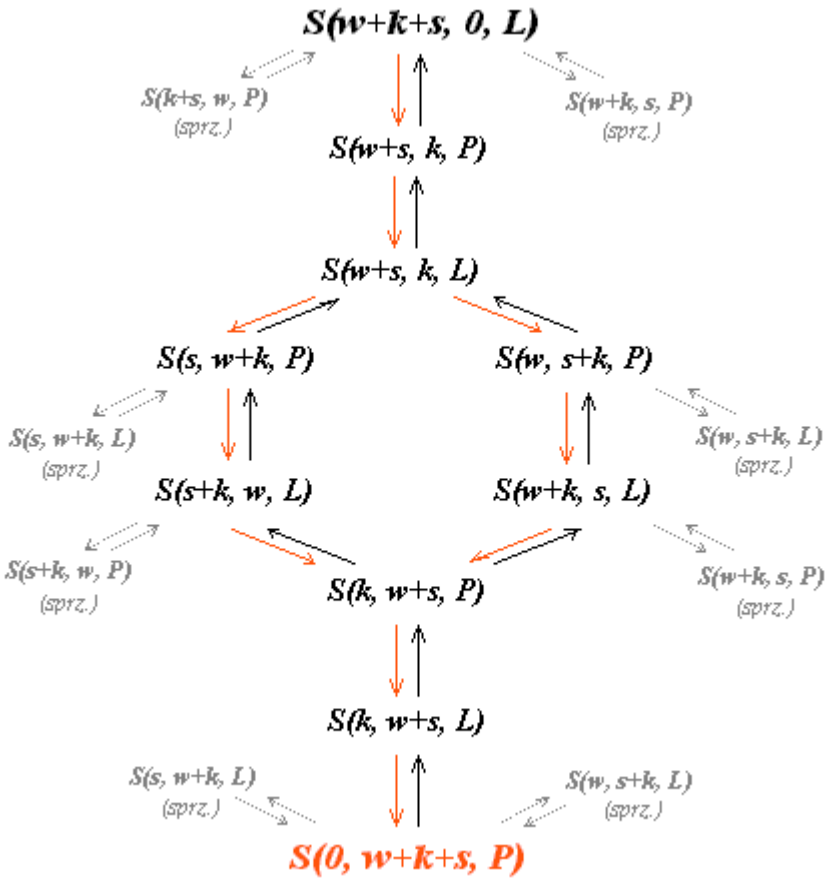
S(w+k, s, P) – *sprzeczny z prawem P3.*
S(k+s, w, P) – *sprzeczny z prawem P4.*
S(w+k+s,0, P) – *sprzeczny z prawem P5.*
S(s, w+k, L) – *sprzeczny z prawem P6.*
S(w, k+s, L) – *sprzeczny z prawem P7.*
S(0, w+k+s, L) – *sprzeczny z prawem P8.*

SYSTEM EKPERTOWY

1. ***Baza wiedzy*** – zbiór faktów i praw potrzebnych do rozwiązania zadania. Im wyższy poziom nauczania, tym więcej formalizmu można użyć w opisywaniu owych faktów i tym bardziej schematyczna może być reprezentacja ikoniczna. Na poziomie podstawowym wystarczą tylko (a może odwrotnie: są konieczne) kolorowe, piękne rysunki ilustrujące treść zadania i przyciągające zainteresowanie małego ucznia zadaniem (według zasady interaktywności, patrz: 3.3).

2. ***Reprezentacje*** – ikoniczna, symboliczna i enaktywna, opisane wyżej. Wykorzystanie którejś z reprezentacji zależy od fazy rozwiązywania problemu oraz poziomu nauczania.

3. ***Sieć semantyczna.***



Rys.4.1.13. Sieć semantyczna do zadania o „Małej Przeprawie Przez Rzekę”.
Źródło: opracowanie własne

4. ***Operacje.***

Analiza sieci semantycznej pozwala wyróżnić następujące operacje:

- S(x, y, L) $\vdash\!\rightarrow$ S(x – w, y + w, P),

- S(x, y, P) |→ S(x – w, y + w, L),
- S(x, y, L) |→ S(x – k, y + k, P),
- S(x, y, P) |→ S(x – k, y + k, L),
- S(x, y, L) |→ S(x – s, y + s, P),
- S(x, y, P) |→ S(x – s, y + s, L),
- S(x, y, L) |→ S(x + w, y – w, P),
- S(x, y, P) |→ S(x + w, y – w, L),
- S(x, y, L) |→ S(x + k, y – k, P),
- S(x, y, P) |→ S(x + k, y – k, L),
- S(x, y, L) |→ S(x + s, y – s, P),
- S(x, y, P) |→ S(x + s, y – s, L).

5. ***Rama*** – zbiór dróg rozwiązań prowadzących od stanu początkowego do stanu końcowego: wyniku rozwiązania zadania. Od stanu początkowego do końcowego można dojść wieloma drogami różnej długości (patrz: sieć semantyczna). Na przykład od stanu *S(s, w+ k, P)* można przejść do stanu *S(w+s, k, L)*, po czym i tak trzeba wrócić do *S(s, w+ k, P)*, albo pójść drugim rozgałęzieniem – *S(w, s+k, P)*.

6. ***Realizacja rozwiązania problemu.***

Na diagramie sieci semantycznej, w punkcie odpowiadającym stanowi *S(w+s, k, L)* napotykamy rozgałęzienie. Dzieje się tak, ponieważ po przeprawieniu kozy na prawy brzeg mężczyzna ma dwie możliwości: może zabrać do łódki albo wilka, albo sałatę. Po dwóch operacjach (po dwóch przepłynięciach łódki) drogi te znów się łączą. Możemy zatem wyróżnić dwie drogi zaznaczone czerwonymi strzałkami: obie są tej samej długości i są najkrótszymi drogami rozwiązania problemu, mieszczącymi się w ramie rozwiązania.

4.2. Programowanie logiczne w języku Turbo Prolog.

Programowanie w Turbo Prologu jest zbyt trudnym zagadnieniem dla uczniów szkoły podstawowej i gimnazjum, a nawet dla przeciętnego licealisty. Zrozumienie struktury takiego programu wymaga już pewnej znajomości teorii logiki, semantyki języka klauzul, semantyki i składni Turbo Prologa, a samodzielne napisanie programu w tym języku jest zadaniem jeszcze trudniejszym. Niniejszy podrozdział przeznaczony jest więc już „dla wtajemniczonych", co nie znaczy, że nie warto „wtajemniczać" w programowanie logiczne co zdolniejszych uczniów szkoły średniej. Omówimy i zanalizujemy trzy przykładowe zadania o coraz większym stopniu trudności i programy je wykonywujące[25].

[25] J. Szajna, M. Adamski. T. Kozłowski, *Turbo Prolog. Programowanie w języku logiki.* WNT, W-wa 1991.

Zadanie 1.

Należy napisać program umożliwiający znalezienie wszystkich możliwych układów oczek uzyskanych w trzech rzutach kostką przy założeniu, że suma oczek jest równa liczbie podanej w pytaniu.

Rozwiązanie zadania podano w przykładzie 4.2.1.

Przykład 4.2.1.

```
PREDICATES
% znajdzUk1adOczek (wynikRzutuA, wynikRzutuB, wynikRzutuC, suma)
   znajdzUk1adOczek (integer       , integer         , integer         , integer)
          wykonanieRzutu (integer)
CLAUSES
znajdzUk1adOczek (A,B,C,N)
    :- wykonanieRzutu(A),
          wykonanieRzutu(B),
          wykonanieRzutu(C),
          N=A+B+C.
wykonanieRzutu (l).
wykonanieRzutu (2).
wykonanieRzutu (3).
wykonanieRzutu (4).
wykonanieRzutu (5).
wykonanieRzutu (6).
```

Załóżmy, że podamy pytanie:

GOAL: znajdzUk1adOczek (A,B,C,9)

W odpowiedzi zostają podane wszystkie możliwe układy oczek, których suma jest równa 9.

Lp.	A	B	C	A+B+C=9
1	1			
2	1	1		
3	1	1	1	nie
4	1	1	2	nie
5	1	1	3	nie
6	1	1	4	nie
7	1	1	5	nie
8	1	1	6	nie
9	1	2		
10	1	2	1	nie
11	1	2	2	nie
12	1	2	3	nie
13	1	2	4	nie
14	1	2	5	nie
15	1	2	6	**tak**

Lp.	A	B	C	A+B+C=9
16	1	3		
17	1	3	1	nie
18	1	3	2	nie
19	1	3	3	nie
20	1	3	4	nie
21	1	3	5	**tak**
22	1	3	6	nie
23	1	4		
24	1	4	1	
25	1	4	2	nie
26	1	4	3	nie
27	1	4	4	**tak**
. . .				
257	6	6	5	nie
258	6	6	6	nie

Rys. 4.2.1. Proces konstruowania rozwiązania dla przykładu 4.2.1.
Źródło: J. Szajna, M. Adamski. T. Kozłowski, *Turbo Prolog. Programowanie w języku logiki.* WNT, Warszawa 1991.

Konstruowanie każdego z rozwiązań będzie przebiegało stopniowo, metodą prób i błędów, jak na rysunku 4.2.1, sporządzonym po napisaniu programu w celu zilustrowania procesu konstruowania rozwiązania. Na przykład w punktach 8 i 9 następuje wycofanie się z rozwiązania częściowego A = l, B = l i zastąpienie go wariantem A = l, B = 2.

Należy zaznaczyć, że program zaprojektowano bez szczegółowej analizy procesu nawracania, tylko na podstawie znajomości reguły, że gdy pewne rozwiązanie częściowe nie spełnia podanego warunku, to aparat wnioskowania wycofuje się z niego i przechodzi do konstruowania rozwiązania z innym wariantem rozwiązania częściowego.

Zadanie 2.

Należy napisać program wypisujący wszystkie możliwe układy oczek, które przy zadanej w pytaniu liczbie rzutów kostką dają sumę oczek podaną w tym pytaniu.[26]

(W zadaniu 1 liczba rzutów kostką była ustalona na stałe i wynosiła 3, tu można ją zmieniać.)

Rozwiązanie zadania podano w przykładzie 4.2.2.

Przykład 4.2.2.

```
DOMAINS
listaOczek = integer*

PREDICATES
        ZnajdzUkladOczek (integer,          % liczba rzutów do wykonania
                          integer,          % poszukiwana liczba oczek
                          listaOczek)       % poszukiwana lista oczek
        ukladOczek (integer,                % liczba rzutów
                    listaOczek)
        czySumaPoprawna (listaOczek,
                          integer)          % poszukiwana suma oczek
        wykonanieRzutu (integer)

CLAUSES
        ZnajdzUkladOczek (LiczbaRzutow, SumaOczek, UkladOczek)
                :- ukladOczek (LiczbaRzutow, UkladOczek)
                   czySumaPoprawna (UkladOczek, SumaOczek).
% ----------------------------------------------------------------------------
ukladOczek (0, [ ]).    % zakończenie rekurencji, gdy wykonano wszystkie rzuty

ukladOczek (LiczbaRzutowDoWykonania, [WynikRzutu|WynikiPozostalychRzutow])
        :- LiczbaRutowDoWykonania>0,
           WYKONANIE_RZUTU (WynikRzutu),
          PozostaloRzutow = LiczbaRutowDoWykonania – 1,
          UkladOczek (PozostaloRzutow, WynikiPozostalychRzutow).
% ----------------------------------------------------------------------------
czySumaPoprawna ([ ], 0).           % suma jest poprawna, jesli po odjeciu
                                    % od poszukiwanej sumy oczek wynikow wszystkich
                                    % rzutow reszta wynosi zero
```

[26] J. Szajna, M. Adamski. T. Kozłowski, *Turbo Prolog...*

```
czySumaPoprawna ([WynikRzutu|WynikiPozostalychRzutow], SumaOczek)
        :- Reszta = SumaOczek – WynikRzutu,
           czySumaPoprawna (WynikiPozostalychRzutow, Reszta).
% -----------------------------------------------------------------------------
wykonanieRzutu (1).
wykonanieRzutu (2).
wykonanieRzutu (3).
wykonanieRzutu (4).
wykonanieRzutu (5).
wykonanieRzutu (6).
```

W porównaniu z poprzednim, bardzo prostym przykładem, program ten używa **list** i jest zbudowany w sposób rekurencyjny[27]. Zastosowano w nim typową technikę używaną przy tworzeniu listy nieodwróconej[28]. Podobnie jak w przykładzie poprzednim, program znajduje wszystkie rozwiązania – w tym przypadku wszystkie listy spełniające podane warunki, dotyczące liczby rzutów i sumy oczek. Konstruowanie każdej z tych list odbywa się także w sposób stopniowy metodą prób i błędów. Jeżeli na przykład zadamy pytanie:

GOAL: ukladOczek(3,9,ListaOczek)

(zadanie podobne jak w przykładzie 4.2.1, tzn. uzyskanie sumy oczek równej 9 w trzech rzutach), to w pewnym momencie konstruowania rozwiązania nastąpi wycofanie się z częściowego rozwiązania [l, l, 6] i przejście do nowego wariantu rozwiązania częściowego: [l, 2 | **...**]. Jest to sytuacja podobna do przedstawionej na rys. 4.2.1.

Zadanie 3.

Należy napisać program, który pozwala tworzyć prosty harmonogram. Dane są:

- lista dni roboczych zawierająca liczby roboczogodzin podczas każdego dnia,
- lista zadań zawierająca numery zadań i liczby roboczogodzin potrzebnych do wykonania każdego zadania.

Program ma tak rozmieścić poszczególne zadania w czasie, aby:

- wszystkie zadania zostały wykonane,
- każde z zadań było wykonywane bez przerw w trakcie jednego dnia (bez rozdzielenia na dwa lub więcej dni).

Rozwiązanie zadania podano w przykładzie 4.2.3.

Przykład 4.2.3.

```
DOMAINS
    ListaDni=dzien*

    dzien=dzien (nrDnia,
                 godziny,       % dlugosc dnia
                 godziny,       % liczba zajetych godzin
```

[27] patrz: 2.6.
[28] J. Szajna, M. Adamski. T. Kozłowski, *Turbo Prolog...*

```
                listaZadan)   % harmonogram dnia
    listaZadan=zadanie*
    zadanie=zad(nrZad,godziny)
    nrZad,nrDnia,godziny=integer

PREDICATES
    rozdzielZadania (listaDni,      % ulozony harmonogram pracy
                     listaDni,    % wykaz dni przed ulozeniem harmonogramu
                     listaZadan)

    przydzielJednoZadanie (zadanie,
                           listaDni,    % przed przydzieleniem zadania
                           listaDni)    % po przydzieleniu zadania

    umiescZadanieWPodanymDniu (zadanie,
                               dzień,          % przed umieszczeniem zadania
                               dzień)          % po umieszczeniu zadania
CLAUSES
    rozdzielZadania (ListaDni, ListaDni, [ ]).      % komentarz l
    rozdzielZadania (KoncowaListaDni,
                     ListaDniPrzedPrzydziel,
                     [Zadanie|OgonZadan ])
    :-
    przydzielJednoZadanie (Zadanie,             % komentarz 2
                           ListaDniPrzedPrzydziel,
                           ListaDniPoPrzydziel),
    rozdzielZadania(KoncowaListaDni,
                    ListaDniPoPrzydziel,
                    OgonZadan).                 % komentarz 3
% ---------------
% Komentarz l: Gdy wyczerpie się lista zadan, to za koncowa liste dni (harmonogram)
%              przyjmuje się aktualna liste dni.
% Komentarz 2: Przydzielenie zadania stanowiacego głowe listy zadan.
% Komentarz 3: Rozdzielenie reszty zadan (ogona listy zadan).
% ---------------------------------------
    przydzielJednoZadanie (Zadanie,
                           [Dzień                |OgonDni],   % komentarz l
                           [DzienPoPrzydzieleniu |OgonDni])          % komentarz l
        :-  umiescZadanieWPodanymDniu (Zadanie,
                                       Dzien,
                                       DzienPoPrzydzieleniu).
    przydzielJednoZadanie (Zadanie,
                           [Dzien|OgonDni],                          % komentarz 2
                           [Dzien|OgonDniPoPrzydzieleniu])       % komentarz 2
        :-  przydzie1JednoZadanie(Zadanie,
                                  OgonDni,
                                  OgonDniPoPrzydzieleniu).
% ---------------
% Komentarz:   Procedura przeglada liste dni, starajac sie przydzielic zlecenie
%              do ktoregos z kolejnych dni.
% Komentarz l:  Pierwsza klauzula probuje umiescic zadanie w glowie listy dni
%               (ogon dni - bez zmian).
% Komentarz 2: Druga klauzula probuje umiescic zadanie w ogonie listy dni
%              (glowa tzn."Dzien" - bez zmian).

% ---------------------------------------

    umiescZadanieWPodanymDniu (zad(NrZad, CzasZad),
                               dzien (NrDnia,           % komentarz 1
                                      DlugoscDnia,
```

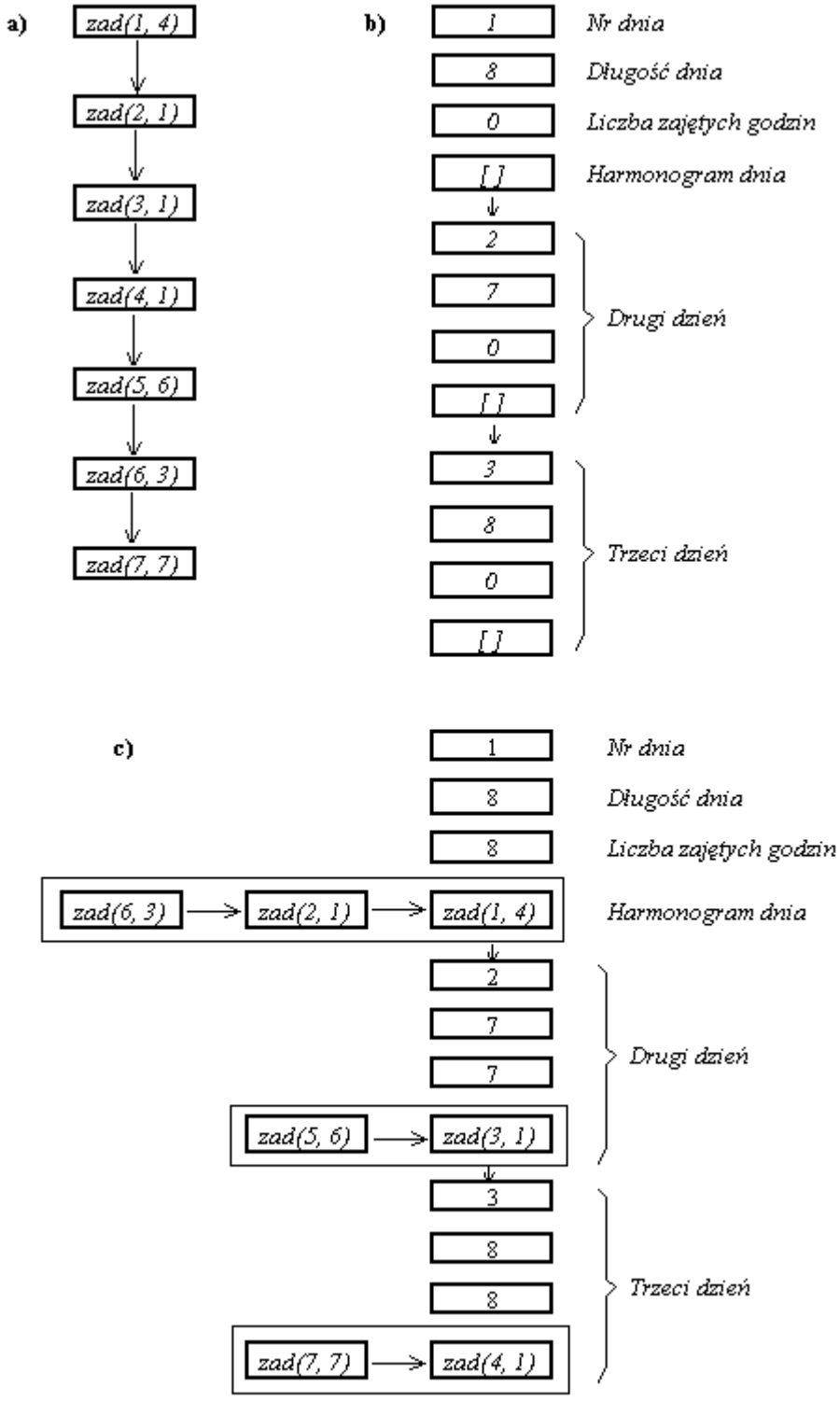
```
                                        LiczZajetychGodz,
                                        HarmDnia),
                                    dzien(NrDnia,                   % komentarz 2
                                        DlugoscDnia,
                                        LiczZajetychGodzPoPrzydz,
                                        [zad(NrZad, CzasZad)|HarmDnia]))
    :-  LiczZajetychGodzPoPrzydz = LiczZajGodz + CzasZad,
        LiczZajetychGodzPoPrzydz <= DlugoscDnia.

% ----------
% Komentarz:   Dolaczenie zadania do dotychczasowej listy zadan dnia. Po tej operacji
%              zmienia się tez liczba zajetych godzin w rozpatrywanym dniu.
% Komentarz l: Opis dnia przed umieszczeniem zadania.
% Komentarz 2: Opis dnia po umieszczeniu zadania.
```

Podstawową strukturą danych, którą operuje program, jest *listaDni* (rys. 4.2.2 b i c). Jest to lista, w której każdy z elementów składa się z czterech pól:

- numeru dnia,
- długości dnia, tzn. całkowitej liczby godzin, które są do dyspozycji w danym dniu,
- liczby godzin zajętych przez zadania przydzielone do realizacji w tym dniu,
- harmonogramu dnia, tzn. listy zadań przydzielonych do realizacji w tym dniu

(czwarte pole listy dni jest listą należącą do dziedziny *listaZlecen).*



Rys. 4.2.2. Struktura danych zastosowana w przykładzie 4.2.3.: a) lista zadań do rozdzielenia, b) lista dni na początku procesu, c) lista dni po rozdzieleniu zadań (jedno z rozwiązań).
Źródło: J. Szajna, M. Adamski. T. Kozłowski, *Turbo Prolog...*

Na początku działania programu nie ma jeszcze przydzielonych zadań na żaden dzień. Dlatego do pól określających liczbę zajętych godzin w poszczególnych dniach są wpisane zera, a do pól zawierających harmonogramy dni – listy puste (rys. 4.2.2 b).

Celem programu jest odpowiednie rozdzielenie zadań na poszczególne dni, tzn. odpowiednie utworzenie list – harmonogramów dla każdego z dni (listy tworzące czwarte pola listy dni). Rysunek 4.2.2 c przedstawia ostateczny stan listy dni dla jednego ze znalezionych rozwiązań, po zadaniu pytania:

GOAL: rozdzielZadania (Harmonogram,
[dzien (1,8,0,[ ]), dzien (2,7,0,[ ]), dzien (3,8,0,[ ])],
[zad (1,4), zad (2,1), zad (3,1), zad (4,1), zad (5,6),
zad (6,3), zad (7,7)])

Program znajduje wszystkie możliwe rozwiązania. Ogólną zasadę jego działania można opisać następująco:

1. Dla rozpatrywanego działania najpierw wykonuje się próbę umieszczenia go w pierwszym dniu, tzn. w głowie listy dni. Realizuje to pierwsza klauzula procedury *przydzielJednoZadanie.* Jeżeli nie jest to możliwe (liczba zajętych godzin w danym dniu nie może być większa niż długość tego dnia – kontroluje to procedura *umiescZadanieWPodanymDniu*), to następuje przejście do operacji opisanych w punkcie 2.
2. Wykonuje się próbę umieszczenia zadania w ogonie listy dni: wydziela się ten ogon (nagłówek drugiej klauzuli w procedurze *przydzielJednoZadanie*) i traktuje go jako rozpatrywaną listę dni, przechodząc ponownie do punktu l (rekurencyjne wywołanie procedury *przydzielJednoZadanie* w drugiej klauzuli tej procedury).

Po przydzieleniu zadania przechodzi się do rozdzielenia pozostałych zdań, czyli do rozdzielenia ogona zadań (wszystkie zadania, które należy rozdzielić, są zebrane w liście zadań (*listaZadan*), będącej daną wejściową do całego programu). Realizuje to procedura *rozdzielZadania*. Jeżeli kolejnego zadania nie można umieścić już w żadnym dniu, to znaczy, że dotychczasowe rozwiązanie częściowe nie może doprowadzić do znalezienia ostatecznego rozwiązania. W takiej sytuacji, podobnie jak w dwóch poprzednich przykładach, aparat wnioskowania wycofuje się o jeden krok z ustalonego już. rozwiązania częściowego, tzn. unieważnia ostatnio dokonane przydzielenie zadania i próbuje realizować inny wariant przydzielenia tego zadania (w którymś z następnych dni) itd. Cały proces kończy się po rozdzieleniu wszystkich zadań, tzn. gdy rozpatrywana lista zadań jest listą pustą (pierwsza klauzula procedury *rozdzielZadania*).

Każda z list wewnętrznych listy dni, tzn. list tworzących harmonogramy poszczególnych dni, jest budowana na zasadzie listy odwróconej[29]. Operację dołączenia nowego elementu (zadania) do istniejącej listy (*HarmDnia*) realizuje procedura *umiescZadanieWPodanymDniu*. Ponieważ w nagłówku procedury występuje wiele zmiennych pomocniczych, które utrudniają dostrzeżenie sposobu dołączania zadania do utworzonego wcześniej częściowego harmonogramu, więc poniżej podano ten nagłówek ponownie, wymieniając tylko te parametry, które wiążą się bezpośrednio z tą operacją:

umiescZadanieWPodanymDniu (zad(NrZad,CzasZad),
dzien(..., HarmDnia),
dzien(...,[zad(NrZad,CzasZad)|HarmDnia]))

Program z przykładu 4.2.3 znajduje wszystkie możliwe rozwiązania, tzn. wszystkie możliwe ukonkretnienia zmiennej *Harmonogram* podanej w pytaniu:

[29] patrz: J. Szajna, M. Adamski. T. Kozłowski, *Turbo Prolog...*

GOAL: rozdzie1Zadania(Harmnonogram, [...], [...]).