PWSZ Tarnów

Wprowadzenie do Programowanie Logicznego z Ograniczeniami z wykorzystaniem ECLiPSe.

Autorzy:
Kamil Janczura
Tomasz Gabiga

Spis treści.

1. Wstęp
   1.1. Cel opracowania.
   1.2. Co jest potrzebne do zrozumienia opracowania.
   1.3. Wstęp do programowania logicznego z ograniczeniami.
   1.4. Co to jest ECLiPSe?
2. Praca z ECLiPSe.
   2.1. Instalacja oraz uruchomienie pakietu.
   2.2. Pierwsze kroki.
   2.3. Programowanie w ECLiPSe.
3. Programowanie logiczne z ograniczeniami.
   3.1. Programowanie przy biernych ograniczeniach.
   3.2. Biblioteka suspend.
   3.3. Core constraints.
4. Przykładowe zastosowania.
   4.1. SEND + MORE = MONEY.
   4.2. Kryptogramy.
   4.3. Sudoku.
   4.4. Aircrew.
5. Literatura.

1. Wstęp .

1.1. Cel opracowania.

Celem opracowania jest wprowadzenie czytelnika w tematykę rozwiązywania zadań z ograniczeniami przy użyciu pakietu ECLiPSe. Pisząc ten tekst nie mieliśmy na celu kompletnego omówienia problemu, co nie było by możliwe choćby ze względu na jego złożoność. Chcieliśmy raczej pokazać możliwości jakie tkwią w korzystaniu z CLP, szczególnie w porównaniu z innymi językami programowania. Dla zgłębienia tematu polecamy czytelnikowi literaturę, której spis znajduje się na końcu artykułu.

1.2. Co jest potrzebne do zrozumienia opracowania?

- podstawy programowania w Prologu,
- podstawy programowania w dowolnym jezyku strukturalnym/obiektowym (C, C++, Java),
- ogólne wiadomości o strukturach danych,
- podstawy z zakresu badań operacyjnych.

1.3. Wstęp do programowania logicznego z ograniczeniami.

W życiu codziennym bardzo często mamy do czynienia z problemami, które należy rozwiązać biorąc pod uwagę pewne ograniczenia. Przykładowo musimy ułożyć sobie grafik dnia, aby pozostało nam jak najwiecej wolnego, czasu mając na uwadze czas dojazdu na miejsce, to że dwa nasze spotkania nie mogą odbywać się jednocześnie, czy to, że chcemy jak najmniej "okienek" pomiędzy poszczególnymi zadaniami. Podobne zagadnienia pojawiają się także w innych dziedzinach – problem ułożenia trasy podróży, by koszt lub czas dotarcia był jak najmniejszy, problem dystrybucji surowców, zagadnienie sterowania ruchem lotniczym. We wszystkich tych problemach występują pewnego rodzaju ograniczenia (samolot nie może przebywać w nieskończoność w powietrzu, dany zakład nie przyjmie więcej niż X sztuk produktu) które możemy sformalizować. O ile dobrze radzimy sobie z mniejszymi problemami znanymi z życia codziennego to bardziej złożone zagadnienia zaczynają sprawiać nam problemy. Właśnie w celu rozwiązywania takiego typu problemów opracowano takie techniki jak programowanie z ograniczeniami (ang. *constraint programming, CP)* oraz programowanie logiczne z ograniczeniami (ang. *constraint logic programming, CLP).* Oba te pojęcia są ze sobą blisko związane.

W CP relacje między poszczególnymi zmiennymi są przedstawiane w formie ograniczeń pomiędzy nimi. W odróżnieniu od znanych nam języków programowania (np. C, Java) nie opisujemy tutaj poszczególnych kroków algorytmu, lecz to co ma zostać uzyskane. Języki CP są więc językami deklaratywnymi, podobnie jak np. SQL. Programowanie logiczne z ograniczeniami (*CLP)* jest swoistego rodzaju połączeniem języków programowania logicznego (np. Prolog) z programowaniem z ograniczenaimi. Pierwszymi implementacjami były stworzone na początku lat dziewięćdziesiątych w Europie PROLOG III, CLP(R), CHIP. W odróżnienieu od języków programowania logicznego w językach CLP mamy elementy programowania funkcyjnego czy imperatywnego.

Jak zostało wspomniane wcześniej CP jest zwieszone w języku nadrzędnym (języku-hoście, ang. *host language).* Pierwszymi językami-hostami były właśnie języki programowania logicznego, pozwalające na takie techniki jak nawracanie (*backtracking).* Połączenie tych dwóch paradygmatów programowania pozwaliłona efektywne modelowanie świata – niektóre ograniczenia mogą być bardzo łatwo reprezentowane przez CP, inne mogą zostać świetnie przedstawione za pomocą programowania

logicznego.

W CLP opisujemy dany problem przez zbiór ograniczeń, natomiast sam program poszukuje rozwiązania przez przyrównywanie różnych wartości do niewiadomych i sprawdzanie, czy nasze ograniczenia zostały spełnione. Możemy wyróżnić następujące domeny ograniczeń CLP:

- logiczne – gdzie ograniczenia przyjmują tylko wartości prawda/fałsz,
- całkowite oraz rzeczywiste – ograniczeniami są liczby całkowite lub rzeczywiste,
- liniowe – wszystkie ograniczenia są przedstawione w formie funkcji liniowych,
- skończone (ang. *finite*) – gdzie ograniczenia są zdefiniowane jako zbiory o skończonej mocy,
- domeny mieszane – połaczenie dwóch lub więcej powyższych.

Najszersze zastosowanie znalazła domena skończona, niekiedy zagadnienia z zakresu badań operacyjnych są identyfikowane z problemem CLP właśnie w tej domenie. Generalnie jednak o ile dla części problemów, którymi zajmują się Badania Operacyjne znamy dokładne algorytmy rozwiązywania, to problemy, którymi zajmuje się CLP są bardzo trudne do algorytmizacji, są to poprostu problemy tak nietypowe, że bardzo trudno jest znaleść jakąkolwiek "ścieżkę" rozwiązywania. Przykładem zagadnienia w domenie skończonej może być logiczna zagadka SEND+MORE=MONEY.

Do najpopularniejszych języków programowania z ograniczeniami należą:

- B-Prolog - jak nazwa wskazuje oparty o Prolog,
- CHIP v5 - oparty na Prologu, zwiera biblioteki C++ i C,
- ECLiPSE – to nim zajmiemy się w tym opracowaniu.

1.4. Co to jest ECLiPSe?

Jednym z popularniejszych implementacji CLP jest pakiet ECLiPSe. Na stronie głównej projektu (http://www.eclipse-clp.org/) znajdziemy informację, że ECLiPSe to

> *"[...] open-sourcowy projekt służący do projektowania i wdrażania aplikacji programowania z ograniczeniami, przykładowo programów wspomagających planowanie, dystrybucję surowców, zagadnienia transportowe itp. [...] Zawiera zbiór bibliotek rozwiązywania problemów CLP, wysokopoziomowy język modelowania zawierający instrukcje warunkowe itp, zintegrowane środowisko programistyczne oraz interfejsy pozwalające na zawieszenie programu napisanego w ECLiPSe w innej aplikacji."*

Projekt powstał w 1991 roku z połączenia trzech innych projektów, początkowo najwięcej czerpał z systemu CHIP. Przez kolejne lata ECLiPSe był sukcesywnie rozwijany wychodząc naprzeciw oczekiwaniom użytkowników.

Z praktycznego punktu widzenia ECLiPSe jest rozszerzeniem PROLOGu mającemu na celu ułatwienie modelowanie problemów z zakresu CLP. Programistom, którzy wcześniej pisali w językach obiektowych i strukturalnych pakiet oferuje szereg konstrukcji znanych z ich ulubionych języków, przykładowo tablice, pętle iteracyjne itd.

2. Praca z ECLiPSe.

2.2 Instalacja oraz uruchomienie pakietu.

Pakiet ECLiPSe mozna pobrac ze strony głównej projektu http://www.eclipse-clp.org/ z sekcji download. W momencie pisania tego opracowania (styczeń 2009) najnowszą wersją pakietu jest wersja 6.0_61. Istnieje możliwość pobrania dystrybucji dla kilku architektur, m.in. dla komputerów zgodnych z i386(wersje 32 i 64 bitowe) oraz dla komputerów opartych o architekturę SPARC. My dla zilustrowania procesu instalacji posłużymy się systemem Ubuntu Linux działającym na 64 bitowym procesorze.

Zajmijmy się teraz samym procesem instalacji. Najpierw należy pobrać ze wcześniej wspomnianej strony odpowiednie archiwa, w naszym przypadku (działamy na Ubuntu) przechodzimy w sekcji Download do katalogu /Distribution/6.0_61/x86_64_linux. Dla nas ważne będą następujące pliki:

- eclipse_basic.tgz - archiwum zawierające jądro systemu oraz podstawowe biblioteki
- eclipse_doc.tgz - paczka z dokumentacją pakietu
- eclipse_misc.tgz - archiwum z zależnościami ("3rd party libraries")
- tcltk.tgz - odpowiednia wersja biblioteki Tcl/Tk potrzebnej do uruchomienia GUI

Wszystkie wyżej wymienione pliki zapisujemy w jednym katalogu, który stanie sie katalogiem roboczym ECLiPSe. Po ściągnięciu plików z sieci należy je rozpakować do aktualnego katalogu (np. korzystając z polecenia tar -xvf [nazwa_pliku] dla każdego z nich). Strukturę katalogów po rozpakowaniu archiwów ilustruje rysunek poniżej.

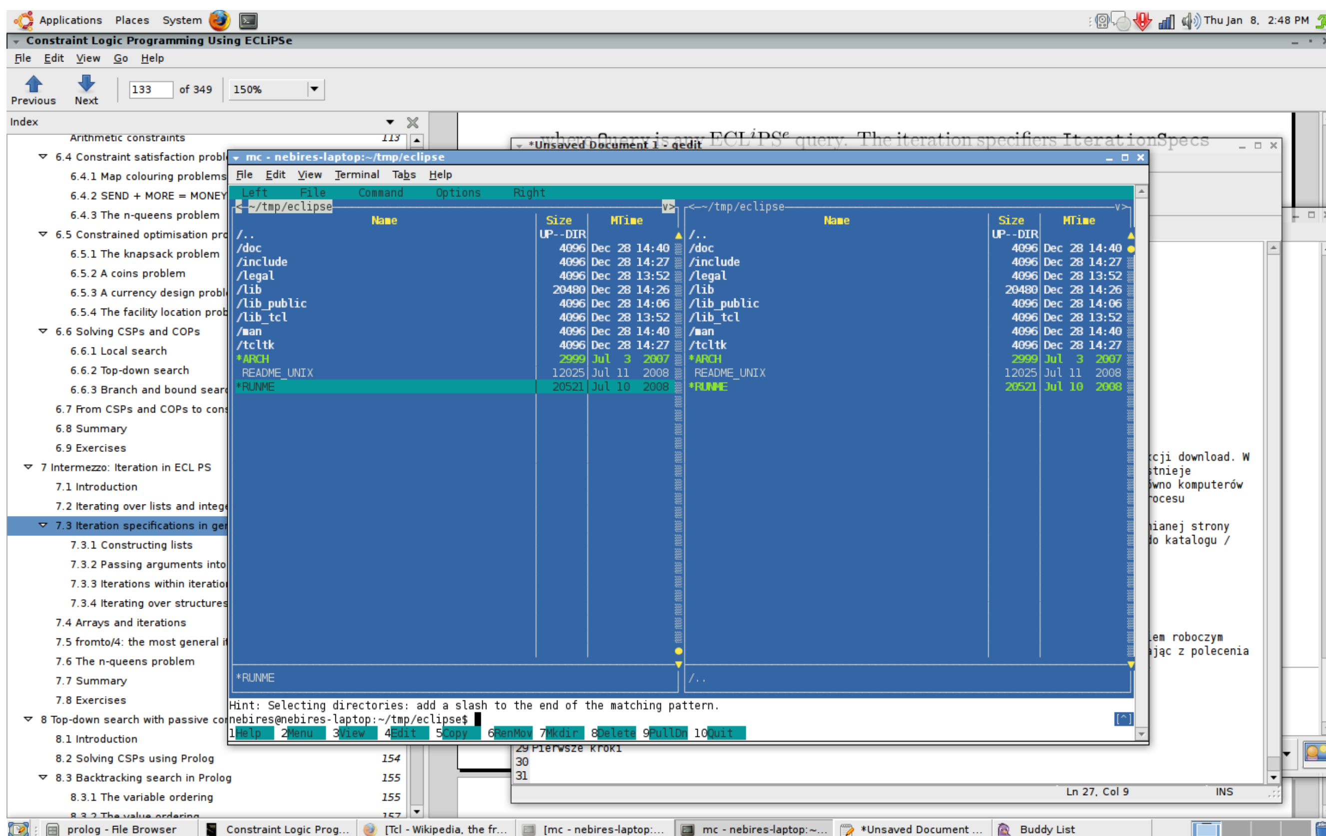


Illustration 1: Struktura katalogów po rozpakowaniu

Teraz, dla poprawnego zainstalowania pakietu, uruchamiamy skrypt RUNME, który przeprowadzi nas przez proces konfiguracji i instalacji. Skryp najpierw spyta o architekturę maszyny (tutaj x86 64 Linux), zatwierdzamy domyslny wybór klawisze ENTER, potem zostaniemy poproszeni o aktualny katalog, czyli lokalizację w której rozpakowaliśmy wcześniej pliki (current directory), znowu zatwierdzamy domyślny wybór Enterem. Kolejne pytania to pytania o ścieżke do instalacji plików wykonywalnych (domyślnie $ECLIPSEDIR/bin/x86_64_linux), wersję biblioteki Tcl/Tk, lokalizację pliku wish 8.5, kolejne szczegóły odnośnie biblioteki Tcl/Tk. Jeżeli instalator poprawnie wykrył wszystkie scieżki (nie powinien mieć z tym problemów, jeśli rozpakowaliśmy podane wcześniej archiwa w jednym katalogu) zatwierdzamy jego decyzje klawiszem ENTER. Następnie zostaniemy poproszeni o podanie scieżki do JRE, podajemy ją, jeśli chcemy uruchomić tryb interaktywny ECLiPSe korzystający z maszyny wirtualnej Javy. Na kolejne pytanie o odnowienie ("regenerate") dokumentacji odpowiadamy Enterem.

Po skończonej instalacji na ekranie konsoli powinien się pojawić napis: "*ECLiPSe installation done*". Tera wystarczy tylko dodac do zmiennej PATH katalog z plikami wykonywalnymi, tutaj $ECLIPSEDIR/bin/x86_64_linux/, aby można było uruchomić ECLiPSe z każdej lokacji w systemie.

2.3. Pierwsze kroki

ECLiPSe można uruchomić na kilka sposobów: w trybie tekstowym, w trybie tekstowym korzystającym ze środowiska Javy oraz z GUI zbudowanym z wykorzystaniem biblioteki Tcl/Tk. My zajmiemy się tutaj tym ostatnim przypadkiem, jako że jest najbardziej wygodny dla użytkownika. Zakładając, że instalacja zakończyła się pomyślnie, zewnętrzne biblioteki graficzne zostały skonfigurowane prawidłowo i do zmiennej PATH została dodana ścieżka z konsoli wydajemy polecenie tkeclipse. Na ekranie powinino pojawić się okno pokazane na rysunku 2.

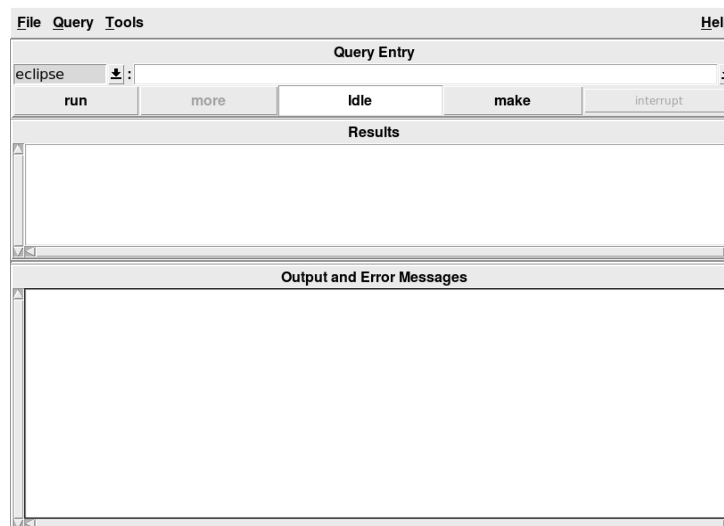


Illustration 2: Działające GUI ECLiPSe

Pakiet nie jest dostarczany z edytorem kodu, tak więc jesteśmy zmuszeni do wykorzystania zewnętrznego narzędzia. Warto zauważyć, że dla najpopularniejszych edytorów można znaleść roszerzenia umożliwiające kolorowanie składni obsługę wcięć itd. Ogólnie przyjętą konwencją jest nadawanie rozszerzenia **.pl** plikom zawierającym źródła w PROLOGU oraz rozszerzenia **.ecl** z źródłami ECLiPSe.

By **skompilować** wcześniej zapisany program należy posłużyć się poleceniem *Open* z menu

File. Polecenie to skompiluje wskazany plik wraz ze wszystkimi jego zależnościami.

2.4 Programowanie w ECLiPSe.

Pętle iteracyjne.

Pętle w ECLiPSe konstruuje się według składni: ***(IterationSpecs do Query)***. Gdzie *IterationSpecs* to jedna lub wiele instrukcji iteracyjnych rozdzielonych przecinkiem

Najprostrzym przykładem wykorzystania pętli będzie przejście przez wszystkie elementy listy w celu ich wyświetlenia. Wykorzystamy do tego instruckje **foreach**. Jeżeli w pole *Query Entry* wpiszemy:

```
( foreach(El,[a,b,c]) do writeln(El) ).
```

w kontrolce *Results* otrzymamy:

```
a
b
c
El = El
Yes (0.00s cpu)
```

Widać tutaj, że zmienna E1w każdym wykonaniu pętli przyjmuje inna wartość z listy podanej w drugim parametrze. Jak wiemy ECLiPSe jest tylko nakładką na PROLOG, w tym przypadku instrukcja *foreach* zostanie zamianiona na rekursywną procedurę przeglądania listy.

Inną, przydatną instrukcją jest instrukcja **count** która przechodzi przez podany zakres liczb całkowitych wykonując podaną instrukcję, przykładowo:

```
[eclipse 2]: ( count(I,1,4) do writeln(I) ).
1
2
3
4
I = I
Yes (0.00s cpu)
```

Możemy łączyć kilka instrukcji iteracyjnych w jednej instrukcji:

```
[eclipse 3]: ( foreach(El,[a,b,c]), count(I,1,_)
              do
                 writeln(I-El)
              ).
```

Jeśli kilka instrukcji iteracyjnych zostało wywołanych w ten sposób wszystkie instrukcje są wywoływane w jednym kroku (ang. *In step)*. Przykładowo dla pierwszej iteracji *E1* będzie wynosiło *a*, natomiast zmienna *I* będzie wynosiła *1*, w drugiej iteracji E1 będzie wynosiło b, zmienna I będzie równa 2. Po trzeciej iteracji wykonywanie instrukcji zostanie przerwane, gdyż pętla foreach przejdzie przez całą listę.

Tworzenie list za pomocą pętli iteracyjnych.

Załóżmy, że chcemy stworzyć listę składającą się z liczb całkowitych. Rozważmy następującą instrukcję oraz jej wynik:

```
[eclipse 6]: ( foreach(El,List), count(I,1,4) do El = I ).
El = El
List = [1, 2, 3, 4]
I = I
Yes (0.00s cpu)
```

Pętle iteracyjne i struktury.

Wykorzystanie złożonych termów zamiast list zwiększa efektywność programu, tak więc przykładowo zamiast listy [X, Y, Z] można z powodzeniem wykorzystać konstrukcję p(X,Y,Z). By iterować po argumentach złożonego termu wykorzystujemy konstrukcję **foreacharg/2**. Załóżmy, że chcemy wyświetlić argumenty. W tym celu posłużmy się instrukcją:

```
[eclipse 22]: ( foreacharg(Arg,p(a,b,c)) do writeln(Arg) ).
```

Pierszym argumentem instrukcji jest zmienna, która zostanie ukonkretniona w czasie iteracji argumentem termu podanego w drugim parametrze, tutaj termu p(a,b,c).

Tablice

Do tworzenia tablic, również wielowymiarowych, stosuje się wbudowany predykat **dim/2** wraz z funktorem **[]**. Przykładowo dla utworzenia tablicy jednowymiarowej o długości 3 posłużmy się konstrukcją:

```
[eclipse 24]: dim(Array,[3]).
Array = [](_162, _163, _164)
Yes (0.00s cpu)
```

W celu zapisania wartości do tablicy wykorzystuje się predykat **subscript/3**, przykładowo utwórzmy dwuwymiarową tablice Array i zapiszmy do elementu [1,2] wartość 5.

```
[eclipse 26]: dim(Array,[3,2]), subscript(Array,[1,2],5).
Array = []([](_263, 5), [](_260, _261), [](_257, _258))
Yes (0.00s cpu)
```

Predykat subscript/3 możemy również wykorzystać do pobrania wartości z tablicy, np.

```
[eclipse 27]: dim(Array,[3]), subscript(Array,[2],a),
              subscript(Array,[2], X).
Array = [](_287, a, _289)
X = a
Yes (0.00s cpu)
```

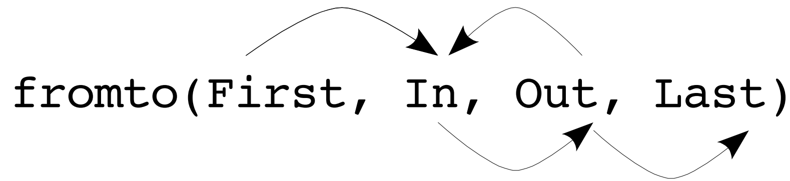
Najpierw tworzymy jednowymiarową tablicę o nazwie Array, później korzystając z predykatu subscript/3 zapisujemy do jej drugiego elementu wartość 'a', po czym w kolejnej instrukcji ukonkrentniamy zmienną X wartością drugiego elementu tablicy Array.

Instrukcja fromto/4.

W celu całkowitego weliminowania użycia rekursji z naszych programów (niektórzy programiści mogą się oburzyć) możemy wykorzystać najbardziej ogólny iterator oferowany przez ECLiPSe – **fromto/4**. Pozwala on na stworzenie bardziej złożonych form iteracji, w odróżnieniu od

poprzednich instrukcji, które zajmowały się pojedynczym elementem struktury lub listy. Składnia iteratora wygląda następująco:

```
fromto(First, In, Out, Last).
```



Wykonanie przebiega następująco: w pierwszej iteracji program przyjmuje **First = In.** Ta sama iteracja oblicza wynik **Out**, ten wynik jest traktowany jako wartość **In** kolejnej iteracji. Warunkiem zatrzymania pętli jest **Out = Last**. Dla rozjaśnienia sprawy posłużmy się przykładem:

```
[eclipse 34]: ( fromto([a,b,c],[Head | Tail],Tail,[])
              do
                writeln(Head)
              ).
a
b
c
Head = Head
Tail = Tail
Yes (0.00s cpu)
```

Jako parametr First podajemy listę [a,b,c], która przyjmie wartość In w pierwszej iteracji. Następnie dzielimy listę na głowę i ogon znaną z Prologu instrukcją [Head | Tail]. Na "wejście" kolejnej iteracji jest podawany sam ogon. Przed wykonaniem kolejnej iteracji głowa listy jest wyświetlana predykatem writeln(Head). Teraz pętla się zamyka – na wejście pętli zostaję podany sam ogon. Pętla zostaje przerwana gdy nie bedzie już nic do wyświetlenia, czyli lista będzie pusta – warunek [].

3. Programowanie logiczne z ograniczeniami.

3.1. Programowanie przy bierny ograniczeniach (ang. *Passive constraints).*

Zajmiemy się teraz już konkretnymi zastosowaniami CLP. Na początek rozpatrzymy najprostrzy przypadek – programowanie przy biernych ograniczeniach. Programiści Prologu zdają sobie sprawę, że konstrukcje typu 3*X < Y+2 nie znajdują zastosowania w Prologu, by dokonać jakiekokolwiek testu należy ukonkrentić wszystkie zmienne znajdujące się w zapytaniu. Pisząc programy w czystym Prologu możemy korzystać z ograniczeń jako testów.

W tej części rozważymy podejcie **top-down** do rozwiązywania zadań programowania z ograniczeniami. W podejściu tym podstawiamy coraz to inne wartości do zmiennych decyzyjnych w każdym "ruchu", podczas poszukiwania rozwiązania same zmienne i ich wartości mogą zostać zamienione statycznie lub dynamicznie. Pozatym już raz przyjęta wartość zmiennej decyzyjnej może zostać zmieniona dzięki nawracaniu (backtracking) znanemu z logicznych języków programowania.

Najbardziej ogólnym podejciem do rozwiązywania zadań CP w Prologu jest postępowanie według następującego schematu:

```
solve(List):-
    declareDomains(List),
    search(List),
    testConstraints(List).
```

Gdzie List to list zmiennych, pierwszy cel declaereDomains(List) tworzy domenę w której będziemy poszukiwać rozwiązania, natomiast search(List) poszukuje rozwiązań. Ostatni cel testConstraints(List) sprawdza, czy wygenerowane rozwiązania spełniają warunki ograniczeń.

Podejcie takie jest wysoce nieefektywne, przykładowo dla wspomnianego wcześniej problemu SEND+MORE=MONEY otrzymamy 10^8 potencjalnych rozwiązań, których należy jeszcze wybrać te, które spełniają ograniczenia. Oczywiste jest to, że należy sprawdzać wygenerowanie rozwiązanie z ograniczeniami tak szybko jak to tylko możliwe. Przykładowo dla poprzedniego zadania sprawdzamy warunek S=/=E zaraz po wygenerowaniu wartości S i E. Jeśli wartości są równe nie generujemy dalszej części drzewa.

Podczas oceniania efektywności algorytmu przydatne jest zliczanie nawrotów jakie zostały wykonanie podczas szukania rozwiązań. Im mniej nawróceń tym lepiej.

W ECLiPSe jest możliwe zliczanie ilości nawrotów, ale najpierw musimy zapoznać się ze zmiennymi nielogicznymi. Zmienne nielogiczne są atomami i zapisujemy jako zmienne zaczynające się od małej litery. Mamy cztery predykaty pozwalające na operacje na zmiennych nielogicznych:

- setval/2 – przypisuje wartość do zmiennej nielogicznej,
- incval/1 – zwiększa wartość zmiennej o jeden,
- decval/1 – zmniejsza wartość o jeden,
- getval/2 – przypisuje wartość zmiennej nielogicznej do zmiennej logicznej i służy do testowania, czy zmienna nielogiczna ma przypisaną jakąś wartość.

Weźmy pod uwagę przykład:

```
[eclipse 7]: N is 3, setval(count,N),
             incval(count), getval(count, M).
N = 3
M = 4
Yes (0.00s cpu)
```

Najpierw ukonkrentniamy zmienną logiczną N wartością 3, później do zmiennej nielogicznej count przypisujemy wartość N, zwiększamy ją o jeden i ukonkretniamy zmienną logiczną M wartością zmiennej count. Dla sprawdzenia, czy dana zmienna nielogiczna ma już przypisaną wartość rozważmy przykłady:

```
[eclipse 8]: setval(step,a), getval(step,a).
Yes (0.00s cpu)
[eclipse 9]: setval(step,a), getval(step,b).
No (0.00s cpu)
```

Rezultaty są oczywiste i nie wymagają większego komentarza.

3.2. Biblioteka suspend.

Zajmijmy się teraz biblioteką suspend, która pozwala na łatwe modelowanie ograniczeń. Biblioteka świetnie nadaje się do implementacji bardziej zaawansowanych *solverów* jak i nauczania podstaw programowania z ograniczeniami. Istotnym minusem biblioteki suspend jest brak propagacji ograniczeń (*suspend propagation).* Wspierane jest tylko programowanie z ograniczeniami pasywnymi, ale i tak znacząco upraszcza programowanie w porównaniu do kodowania w czystym Prologu.

Ogólny, wcześniej już wspomniany schemat programów ECLIPSE wygląda następująco:

```
:- lib(my_library).
solve(List):-
    declareDomains(List),
    generateConstraints(List),
    search(List).
```

Pierwsza linijka wczytuje wybraną przez nas bibliotekę. Następnie rozwiązujemy zadanie biorąc pod uwagę List, który jest niczym innym jak listą zmiennych. Cel declareDomans(List) generuje domenę poszukiwań, cel generateConstraints(List) generuje ograniczenia, na końcu posługujemy się celem search(List) dla znalezienia rozwiązań.

Teraz kilka słów na temat samych bibliotek. Każda biblioteka w ECLiPSe może zostać wykorzystana na trzy sposoby:

- jako prefiks w zapytaniu, np. Suspend:(2<Y+1),
- jako wczytanie w kontekście zapytania lib(my_library);
- jako wczytanie jej na początku programu, np. :-(my_library).

Biblioteka suspend umożliwia obejść ograniczenia stawiane przez składnie Prologu przez "opóźnienie" niektórych celów aż wszystkie potrzebne zmienne zostaną ukonkrentnione, przykłądowo następująca instrukcja zakończy się niepowodzeniem:

```
[eclipse 1]: 2 < Y+1, Y = 3.
instantiation fault in +(Y, 1, _173)
Abort
```

Znacznie utrudnia to CSP w Prologu. Natomiast wykorzystując bibliotekę suspend otrzymamy:

```
[eclipse 2]: suspend:(2 < Y+1), Y = 3.
Y = 3
Yes (0.00s cpu)
```

Implementuje ona dużo bardziej wyszukany sposób obliczeń od Prologu, gdzie do spełnienia brany jest

pierwszy cel z lewej strony.

3.3. Core constraints.

Biblioteka suspend oddaje następujące narzędzia wspomagające w modelowaniu ograniczeń:

- ograniczenia logiczne (ang. *Boolean constraints)*, które zawierają predykaty neg/1, and/2, or/2, =>/2 ("wynika, że"),
- ograniczenia arytmetyczne obsługiwane za pomocą wbudowanych operatorów
- deklaracje zmiennych,
- tzw. Reified constraints.

Najprostrzym narzędziem są ograniczenia logiczne, rozważmy przykładowy kod:

```
[eclipse 6]: suspend:(X or Y), X = 0, Y = 1.
X = 0
Y = 1
Yes (0.00s cpu)
```

Ciekawe możliwości oferują narzędzia do obsługi ograniczeń arytmetycznych biblioteki suspend. Zakładając, że mamy biblioteka została już wczytana możemy posłużyć się następującymi operatorami:

- mniejsze od - $<,
- mniejsze lub równe $=<,
- równe $=,
- nierówne $\=,
- większe lub równe $>=,
- większe $>,

Przykładowo:

```
[eclipse 12]: 1 + 2 $= Y.
Y = Y
Delayed goals:
        suspend : (1 + 2 =:= Y)
        Yes (0.00s cpu)
```

ECLiPSe umożliwia deklarację zmiennych, co będzie bardzo przydatne przy wykorzystaniu metod, które pozwalają na redukcję dziedziny poszukiwań. Biblioteka suspend pozwala na dwa rodzaje deklaracji zmiennych:

- range – dzięki temu narzędziowi zmienna przyjmie wartości liczb całkowitych z podanego zakresu, np.

```
[eclipse 17]: suspend:(X :: 1..9).
X = X
Delayed goals:
        suspend : (X :: 1 .. 9)
Yes (0.00s cpu)
```

Możemy również wykorzystać konstrukcję X :: 1..9 przy załadowanej bibliotece suspend. By

umożliwić wygodniejsze zapisywanie ograniczeń wprowadzono konstrukcję pozwalającą na zapisanie kilku ograniczeń w jednej linii, np. [S,E,N,D,M,O,R,Y] :: 0..9. .

- przy wykorzystaniu predykatów integers/1 oraz reals/1. By zdeklarować fakt, że X należy do liczb całkowitych posłużymy się instrukją integers(X). Analogiczna sytuacja jest w przypadku predykatu reals/1, który oznacza, że zmienna lub lista zmiennych podane w parametrze należa do liczb rzeczywistych.

4. Przykładowe zastosowania.

4.1 SEND + MORE = MONEY

Podczas analizowania możliwości ECLiPSe często sięgaliśmy przykładem do słynnej kryptograficznej zagadki SEND + MORE = MONEY. Przyszedł czas na prezentacje jej rozwiązania w ECLIPSe.

Zagadka polega na przypisaniu cyfr z zakresu od 0 do 9 do liter znajdujących się w równaniu:

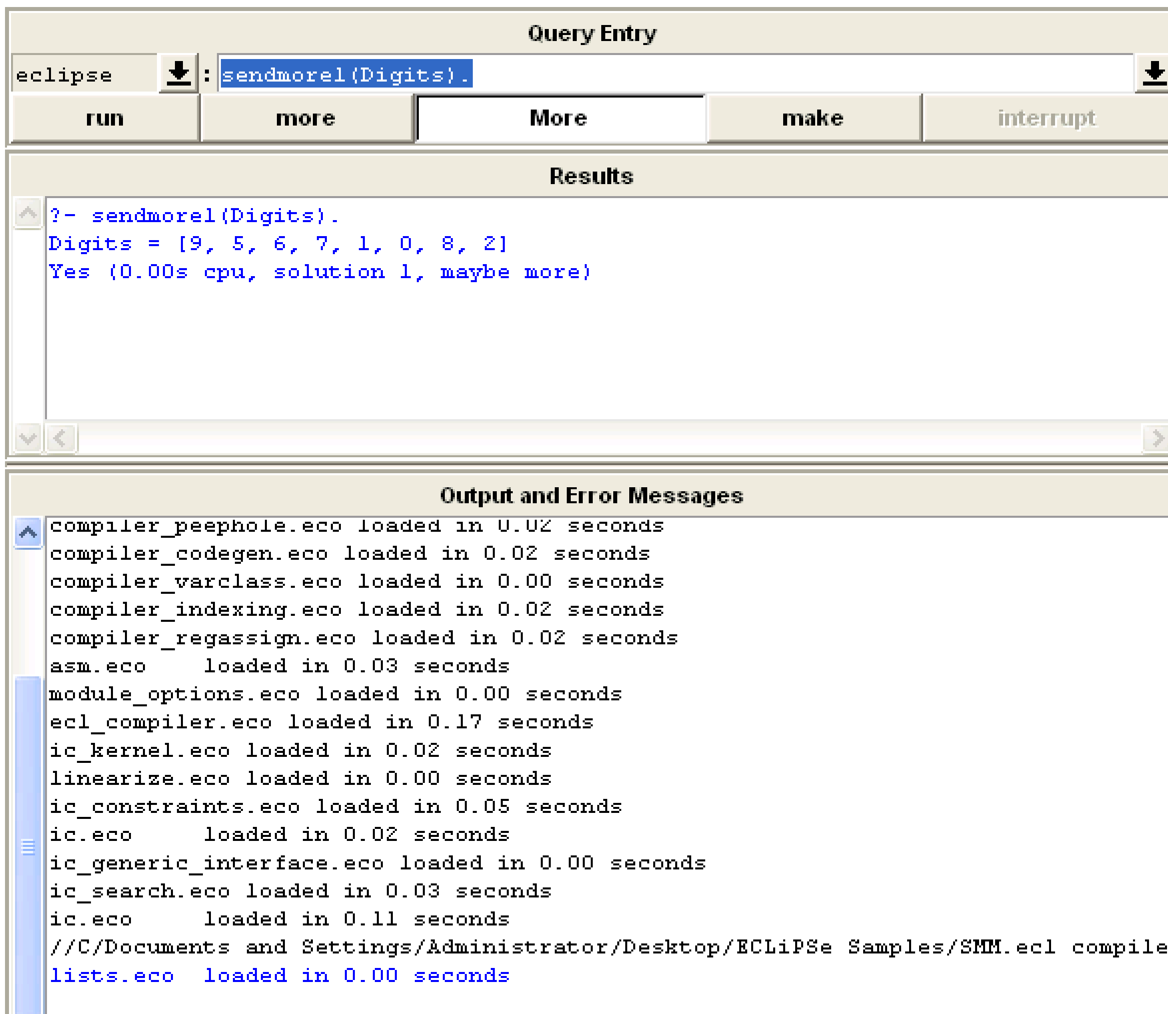
```
  SEND
+ MORE
------------
 MONEY
```

tak, aby po podstawieniu cyfr zamiast liter ów dodawanie było prawdziwe. Warunkiem rozwiązania tego zadanie jest niepowtarzalność przypisań (każda litera musi mieć swoją unikatową wartość cyfrową) oraz z założeń praw algebraicznych literka S oraz M muszą mieć wartość rożną od 0.

Kod programu:

:- lib(ic).	//wczytanie biblioteki IC(Interval Constant) umożliwiające arytmetyczne programowanie hybrydowe pomiędzy wartościami Integer/Real.
sendmore1(Digits) :-	//predyktat rozwiązania zadania z uwzględnieniem listy zmiennych Digits.
Digits = [S,E,N,D,M,O,R,Y],	-deklaracja listy zmiennych Digits zawierającej wszystkie litery zadania.
Digits :: [0..9],	//ograniczenia zadania: -wszystkie cyfry muszą być z zakresu od 0 do 9.
alldifferent(Digits),	-wszystkie litery muszą mieć różne wartości. Funkcja alldiffrent jest tutaj świetnym przykładem przewagi ECLiPSe nad Prologiem. Coś co musielibysmy uwarunkowywać przed długie linie kodu rozwiązujemy tutaj w jednej linijce.
S #\= 0, *M #\= 0,*	//ograniczenia na literkę S oraz M.
*1000*S + 100*E + 10*N + D* *+ 1000*M + 100*O + 10*R + E* *#= 10000*M + 1000*O + 100*N + 10*E + Y,*	//główny blok rozwiązujący zadanie. Zapis ów równania w taki sposób jest możliwy dzięki bibliotece IC pozwalając na najbardziej intuicyjne podejście do problemu.
labeling(Digits).	//funkcja przypisująca obliczone wartości do poszczególnych literek listy Digits.

W ten oto łatwy sposób, poprzez implementacje kilku linii kodu otrzymujemy łatwe i szybkie rozwiązanie zagadki w ECLiPSe:



Przykład ten pokazuje doskonale przewagę ECLiPSe nad Prologiem. Dzięki łatwiejszym operacjom arytmetycznym i możliwości implementacji funkcji wysokiego poziomu rozwiązanie ów zadanie otrzymujemy nie tylko w krótszym czasie ale i także z mniejszym nakładem pracy.

4.2. Kryptogramy.

Po prezentacji przykładu send+more=money ludzie często zadają sobie pytanie czy jest w ogóle możliwość rozwiązywania dowolnego kryptogramu. Otóż jest, w dodatku stosunkowo łatwa w realizacji w języku ECLiPSe.

Prezentowany poniżej program działa podobnie jak poprzedni program jednakże wykorzystuje dwa dodatkowe bloki: blok uniwersalnego wyszukiwania ograniczeń oraz blok uniwersalnego rozwiązania arytmetycznego zadanego równania.

Kod programu:

```
:- lib(ic).
:- lib(ic_search).
```

//wczytanie bibliotek.

```
cryptarith(Equations, Base) :-
        term_variables(Equations, Digits),
```

//predyktat funkcji rozwiązującej zadanie przyjmującej dwa argumenty:
-Equations – treść kryptogramu w postaci równania algebraicznego gdzie słowa są zapisane w tablicach.
-Base – tablica mówiąca nam o zakresie cyfr/liczb wykorzystywanych do rozwiązania zadanego kryptogtamu.

```
        Digits :: 0..Base-1,
        alldifferent(Digits),
        constraint(Equations, Base),
```

//wywołanie bloku wyszukującego konkretne ograniczenia równania.

```
    search(Digits, 0, first_fail, indomain,
complete, []),
        writeln(Equations).
```

//funkcja wyszukująca rozwiązanie zadania na podstawie cyfr z ograniczenia Digits, wyszukuje od początku do pierwszego błędu. Kiedy znajdzie rozwiązanie przypisuje je do odpowiednich liter za pomocą funkcji indomain a następnie wypisuje wynik na ekranie.

```
constraint([], _).
constraint([C|Cs], Base) :-
        constraint(C, Base),
        constraint(Cs, Base).
constraint(E1=E2, Base) :-
        expression(E1, CE1, Base),
        expression(E2, CE2, Base),
        eval(CE1) #= eval(CE2).
```

//blok analizujący postać zadanego polecenia. Sprawdza na początku ile równań znajduje się w poleceniu za pomocą constaint. Następnie analizowane są operacje dodawania, odejmowania oraz mnożenia w znalezionych równaniach. Dzięki tej analizie program dokładnie wie jaką postać algebraiczną ma treść programu. Dzięki temu może precyzyjnie ustalić w którym momencie mozliwe są przejścia wartości na następną literę, czy też która literka nie moze mieć wartości 0 gdyż występuje na początku wyrażenia. Do analizy wykorzystuje dodatkowe elementy tak zwane Carries (w tym zadaniu wartości z literką C w nazwie), których zadaniem jest przenoszenie wartości z jednej na drugą (np w dodawaniu 9 +10 przechowuje przejściową cyfrę 1).Wynik tego bloku przekazujemy do funkcji search aby ona już ostatecznie przypisała cyfry do liter.

```
expression(E1+E2, CE1+CE2, Base) :-
        expression(E1, CE1, Base),
        expression(E2, CE2, Base).
expression(E1-E2, CE1-CE2, Base) :-
        expression(E1, CE1, Base),
        expression(E2, CE2, Base).
expression(E1*E2, CE1*CE2, Base) :-
        expression(E1, CE1, Base),
        expression(E2, CE2, Base).
```

```
expression(Digits, sum(WeightedDigits), Base) :-
        Digits = [First|_],
        First #\= 0,
        (
           for(I,length(Digits)-1,0,-1),
           foreach(D,Digits),
           foreach(P*D,WeightedDigits),
           param(Base)
        do
           P is Base^I
        ).
```

Bardzo interesującym przykładem na działanie tego programu jest kryptogram:
KKK=666
KKK= GEORGE – WALKER + BUSH
zakładając, że liczby znajdujące się w kryptogramie są z zakresu 0-11.

Kryptogram wywołujemy za pomocą komendy:
cryptarith([[K,K,K] = [6,6,6], [K,K,K] = [G,E,O,R,G,E] - [W,A,L,K,E,R] + [B,U,S,H]], 12).

Rozwiązanie:
lists.eco loaded in 0.02 seconds
[[6, 6, 6] = [6, 6, 6], [6, 6, 6] = [1, 11, 10, 8, 1, 11] - [2, 0, 7, 6, 11, 8] + [9, 5, 4, 3]]

?- cryptarith([[K, K, K] = [6, 6, 6], [K, K, K] = [G, E, O, R, G, E] - [W, A, L, K, E, R] + [B, U, S, H]], 12).
K = 6
G = 1
E = 11
O = 10
R = 8
W = 2
A = 0
L = 7
B = 9
U = 5
S = 4
H = 3
Yes (0.03s cpu, solution 1, maybe more)

4.3. Sudoku.

Sudoku (jap. sūdoku; od sūji wa dokushin ni kagiru, czyli cyfry muszą być pojedyncze) – łamigłówka, której celem jest wypełnienie diagramu 9x9 w taki sposób, aby w każdym wierszu, w każdej kolumnie i w każdym dziewięciopolowym kwadracie 3x3 znalazło się po jednej cyfrze od 1 do 9.

2			6		7	5		
							9	6
6		7			1	3		
	5		7	3	2			
	7						2	
			1	8	9		7	
		3	5			6		4
8	4							
		5	2		6			8

Zasady przypominają trochę kwadrat łaciński, wymyślony i badany przez średniowiecznych matematyków z terenów Arabii (XIII wiek). W Sudoku, w przeciwieństwie do kwadratu łacińskiego, cyfry nie mogą się powtarzać nie tylko w żadnym wierszu i kolumnie, ale także w małym kwadracie 3x3.

W obecnej postaci Sudoku stało się popularne na świecie dzięki dołączaniu go do wielu gazet.

Sudoku jest łamigłówką czysto logiczną, nie wymaga żadnych operacji matematycznych. Wydaje się pozornie prosta jednakże bez cierpliwości oraz umiejętności logicznego myślenia rozwiązanie diagramu nie jest możliwe.

Z faktu iż jest to problem czysto logiczny ECLiPSe radzi sobie z tą łamigłówką wręcz wyśmienicie.

Kod programu:

```
:- lib(ic).                                   //wczytanie biblioteki IC oraz jednej dodatkowej funkcji
:- import alldifferent/1 from ic_global.      alldiffrent z biblioteki ic_global.

solve(ProblemName) :-                         //blok główny programu. Przyjmuje numer problemu,
        problem(ProblemName, Board),          którego deklaracje przykładową znajdziemy na końcu
                                              programu.

        print_board(Board),                   //wyrysowanie diagramu przed rozwiązaniem.

        sudoku(3, Board),                     //rozwiązanie diagramu. Pierwszy argument okresla wymiar
                                              macieży. Standardowa postać sudoku to 3 podbloki na 3
                                              podbloki, jednakże można spotkać się także z wersją 9x9.

      print_board(Board).                     //wytrukowanie diagramu po rozwiązaniu.


sudoku(N, Board) :-                           //blok rozwiązujący diagram.
        N2 is N*N,                            //ustalenie rozmiaru macierzy.
        dim(Board, [N2,N2]),
        Board[1..N2,1..N2] :: 1..N2,          //ustalenie dziedziny cyfr występujących w diagramie.

      ( for(I,1,N2), param(Board,N2) do       //pętla, której zadaniem jest sprawdzenie niepowtarzalności
          Row is Board[I,1..N2],              cyfr w każdym wierszu oraz kolumnie.
          alldifferent(Row),
          Col is Board[1..N2,I],
          alldifferent(Col)
        ),
```

(multifor([I,J],1,N2,N), param(Board,N) do *(multifor([K,L],0,N-1), param(Board,I,J), foreach(X,SubSquare) do* *X is Board[I+K,J+L]* *),*	//podzielenie macierzy na podbloki o wymiarach zadanych w problemie (standardowo 3x3).
alldifferent(SubSquare) *),*	//ustalenie niepowtarzalności cyfr w podblokach.
term_variables(Board, Vars),	//przepisanie macierzy początkowej do macierzy Vars w taki sposób, aby elementy już znajdujące się wcześniej w macierzy Board nie zostały nadpisane.
labeling(Vars).	//usupełnienie brakujących pól diagramu.

```
print_board(Board) :-
        dim(Board, [N,N]),
        ( for(I,1,N), param(Board,N) do
          ( for(J,1,N), param(Board,I) do
                X is Board[I,J],
                ( var(X) -> write("  _") ; printf(" %2d", [X]) )
          ), nl
        ), nl.
```

//funkcja służąca do wyrysowywania diagramu sudoku.
//najpierw sprawdza wymiar macierzy a nastepnie element po elemencie wypisuje go do konsoli.

```
%--------------------------------------------------------------
problem(1, [](
    [](_, _, 2, _, _, 5, _, 7, 9),
    [](1, _, 5, _, _, 3, _, _, _),
    [](_, _, _, _, _, _, 6, _, _),
    [](_, 1, _, 4, _, _, 9, _, _),
    [](_, 9, _, _, _, _, _, 8, _),
    [](_, _, 4, _, _, 9, _, 1, _),
    [](_, _, 9, _, _, _, _, _, _),
    [](_, _, _, 1, _, _, 3, _, 6),
    [](6, 8, _, 3, _, _, 4, _, _))).
```

//deklaracja problemu do rozwiązania. Pierwszy argument jest unikatowym numerem identyfikacyjnym (jeśli wpisalibysmy do programu z 10 przykładów każdy musi posiadać unikatowy identyfikator.
Drugi argument to tablica, której przykładową postać możemy zobaczyć po lewej stronie.

Rozwiązanie:

Przed:	Po:
_ _ 2 _ _ 5 _ 7 9 1 _ 5 _ _ 3 _ _ _ _ _ _ _ _ _ 6 _ _ _ 1 _ 4 _ _ 9 _ _ _ 9 _ _ _ _ _ 8 _ _ _ 4 _ _ 9 _ 1 _ _ _ 9 _ _ _ _ _ _ _ _ _ 1 _ _ 3 _ 6 6 8 _ 3 _ _ 4 _ _ *lists.eco loaded in 0.00 seconds*	3 6 2 8 4 5 1 7 9 1 7 5 9 6 3 2 4 8 9 4 8 2 1 7 6 3 5 7 1 3 4 5 8 9 6 2 2 9 6 7 3 1 5 8 4 8 5 4 6 2 9 7 1 3 4 3 9 5 7 6 8 2 1 5 2 7 1 8 4 3 9 6 6 8 1 3 9 2 4 5 7 *Yes (0.08s cpu, solution 1, maybe more)*

Przykład pokazuje iż problemy wysoce skomplikowane, w ECLiPSe mogą być swobodnie rozwiązywane w dodatku błyskawicznie (czas tego przykładu to 0.08 sekundy !).

4.4. Przykład praktycznego zastosowania – Aircrew

ECLiPSe, a dokładniej CLP ma zastowanie nie tylko do rozwiązywania zagadek. Często pytają ludzie „po co to wszystko jeżeli nie można tego praktycznie wykorzystać?”. Ten przykład wyprowadzi was z błędu pokazując jedno z najpopularniejszych zastosowań CLP w ECLiPSe: planowanie działań.

Przydział załogi.

Mała sieć lotnicza musi przydzielić 20 uczestników lotu do 10 lotów. Każdy lot musi być wykonany z określoną liczbą ludzi w załodze, z których każdy ma pewne warunki i pewne zadania. Po pierwsze, do obsługi klijentów międzynarodowych załoga musi znać języki Niemiecki, Hiszpaski oraz Francuski.

Dalej, minimalna liczba stewardess (badź stewardów) jest przypisana do każdego lotu. W końcu, każdy z członków załogi po odbytym locie ma wolne przez okres kolejnych dwóch lotów.

Konkretne wartości znajdziecie w kodzie programu poniżej:

```
:- lib(ic).
:- lib(ic_sets).
:- import subset/2 from ic_sets.
```

//wczytanie bibliotek

```
flights(
  [flight( 1,crew:4,stewards:1,stewardesses:1,french:1,
spanish:1,german:1),
   flight( 2,crew:5,stewards:1,stewardesses:1,french:1,
spanish:1,german:1),
   flight( 3,crew:5,stewards:1,stewardesses:1,french:1,
spanish:1,german:1),
   flight( 4,crew:6,stewards:2,stewardesses:2,french:1,
spanish:1,german:1),
   flight( 5,crew:7,stewards:3,stewardesses:3,french:1,
spanish:1,german:1),
   flight( 6,crew:4,stewards:1,stewardesses:1,french:1,
spanish:1,german:1),
   flight( 7,crew:5,stewards:1,stewardesses:1,french:1,
spanish:1,german:1),
   flight( 8,crew:6,stewards:1,stewardesses:1,french:1,
spanish:1,german:1),
   flight( 9,crew:6,stewards:2,stewardesses:2,french:1,
spanish:1,german:1),
   flight(10,crew:7,stewards:3,stewardesses:3,french:1,
spanish:1,german:1)]
).
```

//wymogi co do załogi dla każdego z lotów w postaci:
- numer lotu
-ilość potrzebnych stewardów
-ilość potrzebnych stewardess
-ilość osób ze znajomością języka francuskiego
-ilość osób ze znajomością języka hiszpańskiego
-ilość osób ze znajomością języka niemieckiego

//imiona uczestników załogi podzielonych wg. umiejętności
//stewardzi

```
attendants(
    stewards:
[tom,david,jeremy,ron,joe,bill,fred,bob,mario,ed],
    stewardesses:
[carol,janet,tracy,marilyn,carolyn,cathy,inez,jean,
heather,juliet],
    french:[inez,bill,jean,juliet],
```

//stewardessy

```
    german:[tom,jeremy,mario,cathy,juliet],
    spanish:[bill,fred,joe,mario,marilyn,inez,heather]
).
```

//osoby znające język francuski
//osoby znające język niemiecki
//osoby znające język hiszpański

```
crew :-
  attendants( stewards:Stewards,
            stewardesses:Stewardesses,
            french:French,
            german:German,
            spanish:Spanish),
```

//główny blok programu przypisujący konkretne osoby do konkretnego lotu.

```
  append(Stewards,Stewardesses,Attendants),
  length(Stewards,Nmales),
  length(Stewardesses,Nfemales),
  Nattendants is Nmales + Nfemales,
```

//określenie liczby ludzi do dyspozycji.

```
  ( foreach(A,Attendants), count(I,1,Nattendants),
    foreach(I,SetAttendants),
    fromto(FrenchSet,FrIn,FrOut,[]),
    fromto(GermanSet,GrIn,GrOut,[]),
    fromto(SpanishSet,SpIn,SpOut,[]),
    param(French,German,Spanish)
  do
    (member(A,French) -> FrIn = [I|FrOut] ; FrOut=FrIn ),
    (member(A,German) -> GrIn = [I|GrOut] ; GrOut=GrIn
),
    (member(A,Spanish) -> SpIn = [I|SpOut] ; SpOut=SpIn )
  ),
```

//przepisanie wartości symbolicznych na wartości liczbowe.

```
  StartFemales is Nmales + 1,
  ( for(I,1,Nmales), foreach(I,SetMales) do true ),
  ( for(I,StartFemales,Nattendants), foreach(I,SetFemales) do
true ),

  flights(Flights),

  ( foreach(F,Flights),
    foreach(Crew,Crews),
    param(SetAttendants,SetMales,SetFemales,FrenchSet,Ger
manSet,SpanishSet)
  do
    F=flight(_,crew:C,stewards:Nwards,stewardesses:Ndesse
s,french:NFr,spanish:NSp,german:NGr),
    Crew subset SetAttendants,
    #(Crew,C),
    #(Crew ∧ SetMales,Cmales), Cmales #>= Nwards,
    #(Crew ∧ SetFemales,Cfemales), Cfemales #>= Ndesses,
    #(Crew ∧ FrenchSet,CFr), CFr #>= NFr,
    #(Crew ∧ GermanSet,CGr), CGr #>= NGr,
    #(Crew ∧ SpanishSet,CSp), CSp #>= NSp
  ),
```

//blok przydzielający odpowiednie osoby do wymogów danego lotu wg tablic juz nie symbolicznych, ale liczbowych.

```
Crews = [Crew1,Crew2|_RestCrews],                    //zabezpieczenie przed przydziałem rotacyjnym

append(Crews,[Crew1,Crew2],AppCrews),                //wywołanie funkcji uwzgledniającej urlopy dla członków
two_days_off(AppCrews),                              załogi po odbytych lotach.

( foreach(Cr,Crews) do insetdomain(Cr,_,_,_) ),      //zamiana wartości liczbowych na symboliczne oraz
                                                     wypisanie imion członków załogi przypisanych do
Att =.. [names|Attendants],                          konkretnych lotów.

( foreach(Cr,Crews), param(Att)
do
  ( foreach(X,Cr), param(Att)
  do
     arg(X,Att,Name),
     write(Name), write(', ')
  ),
  nl
).

two_days_off([X,Y,Z|Rest]) :-                        //ciało funkcji rozpatrującej urlopy dla pracowników.
  all_disjoint([X,Y,Z]),
  two_days_off([Y,Z|Rest]).
two_days_off(_Rest).
```

Rozwiązanie:

?- crew.
Yes (0.89s cpu, solution 1, maybe more)
tom, david, jeremy, inez,
ron, joe, bill, fred, cathy,
bob, mario, ed, carol, jean,
tom, david, jeremy, janet, tracy, inez,
ron, joe, bill, fred, marilyn, carolyn, cathy,
bob, mario, ed, jean,
tom, carol, janet, heather, juliet,
david, jeremy, ron, joe, tracy, inez,
bill, fred, marilyn, carolyn, cathy, jean,
bob, mario, ed, carol, janet, heather, juliet,

Program pokazuje, iż przy odpowiednim podaniu wymogów lotu i członków lotu, nie musimy się zastanawiać tygodniami jak dopasować każdą osobę do konkretnego lotu lecz wynik otrzymujemy dużo szybciej, łatwiej i przejrzyściej.

5. Literatura.

Opracowanie to bazuje w głównej mierze na świetnej książce panów Krzysztofa R. Apt'a oraz Marka Wallace'a „*Constraint Logic programming using ECLiPSe*”. Wszystkie poruszone tutaj tematy są znacznie bardziej rozwinięte w tej pozycji, do tego opisano szereg nowych, jeszcze bardziej interesujących zagadnień.

Oprócz książki kopalnią wiedzy na temat ECLiPSe jest jej oficjalna dokumentacja odostępna na stronie głownej projektu http://www.eclipse-clp.org/. Oprócz świetnie wykonanej dokumentacji znajduje się tam również spis linków do tematów poświęconych CLP.

Osobom które nie mają podstaw z Prologu polecamy książkę autorstwa panów W.F. Clocksin'a oraz C.S. Mellish'a pt. „*Prolog. Programowanie*”.