Madaras Szilárd

SZAKÉRTŐI RENDSZEREK

RISOPRINT
Cluj-Napoca • 2022

Dr. Madaras Szilárd

SZAKÉRTŐI RENDSZEREK

PADME | PALLAS ATHÉNÉ DOMUS MERITI ALAPÍTVÁNY

A könyv a Pallas Athéné Domus Meriti Alapítvány támogatásával valósult meg.

SAPIENTIA – ERDÉLYI MAGYAR TUDOMÁNYEGYETEM

Dr. Madaras Szilárd

SZAKÉRTŐI RENDSZEREK

egyetemi jegyzet

RISOPRINT KIADÓ

KOLOZSVÁR, 2022

© Editura RISOPRINT

Toate drepturile rezervate autorilor & Editurii Risoprint

*Editura **RISOPRINT** este recunoscută de **C.N.C.S.***
***(Consiliul Naţional al Cercetării Ştiinţifice)**.*
www.risoprint.ro ***www.cncs-uefiscdi.ro***

Opiniile exprimate în această carte aparţin autorilor şi nu reprezintă punctul de vedere al Editurii Risoprint. Autorii îşi asumă întreaga responsabilitate pentru forma şi conţinutul cărţii şi se obligă să respecte toate legile privind drepturile de autor.

Toate drepturile rezervate. Tipărit în România. Nicio parte din această lucrare nu poate fi reprodusă sub nicio formă, prin niciun mijloc mecanic sau electronic, sau stocată într-o bază de date fără acordul prealabil, în scris, al autorilor.
All rights reserved. Printed in Romania. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a data base or retrieval system, without the prior written permission of the author.

ISBN: 987-973-53-2896-2

SZAKÉRTŐI RENDSZEREK

Autor:
dr. Madaras Szilárd

Szerkesztette: dr. Madaras Szilárd
Lektorálta: dr. Bíró Piroska
Tördelő: Garda-Mátyás Zsolt

Director editură: GHEORGHE POP

PADME | PALLAS ATHÉNÉ DOMUS MERITI ALAPÍTVÁNY

A könyv a Pallas Athéné Domus Meriti Alapítvány támogatásával valósult meg.

TARTALOMJEGYZEK

INTRODUCTION 7
INTRODUCERE 10
BEVEZETÉS 13
1. SZAKÉRTŐI RENDSZEREK ELMÉLETI ALAPJAI 16
1.1. Bevezetés 16
1.2. Tudásbázisrendszerek szerkesztése és jellemzői 18
1.3. A szakértői rendszerek szerkezete és működése 23
1.4. Szakértői rendszerek gazdasági célú felhasználása 28
1.5 Szakértői rendszerek szerkesztésének logikai alapjai 29
2. A PROLOG PROGRAMOZÁSI NYELV 33
2.1 Bevezetés 33
2.2. Tények használata a Prologban 36
2.3. Hogyan működnek a kérdések a Prologban? 39
2.4. Szabályok a Prologban 44
2.5. Logikai szabályok a Prologban 53
2.6. Adatstruktúrák a Prologban 55
2.6.1. Fák 55
2.6.2. Listák 58
2.7. Backtracking algoritmus Prologban 60
2.8. Hanoi tornyai 63
3. SZAKÉRTŐI RENDSZEREK A CLIPS PROGRAMOZÁSI NYELVBEN 66
3.1. Bevezetés 66
3.2. Telepítés, első lépések 66
3.3. Tények a CLIPS-ben 67
3.4. Objektumok a CLIPS-ben 70
3.5. A CLIPS változók típusai: lokális vagy globális, egymezős vagy többmezős 71

3.6. Sztring műveletek a CLIPS-ben 77
3.7. Tudásreprezentáció a CLIPS-ben 79
3.7.1 Heurisztikus tudásreprezentáció, szabályokkal 79
3.7.2 Procedurális tudásreprezentáció 80
3.8. Műveletek nem rendezett tényekkel 80
3.9. Szabályok használata a CLIPS-ben 86
3.10. Logikai feltételek 97
3.11. Matematikai műveletek és függvények a CLIPS-ben 102
3.12. Gyakran használt matematikai függvények a CLIPS-ben 108
3.13. Ciklusok 111
3.14. A switch utasítás 114
3.15. A CLIPS szabályaira vonatkozó eljárások 115
3.16. Függvények 118
3.17. Példa egy CLIPS-ben szerkesztett szakértői rendszerre 121
4. SZAKÉRTŐIRENDSZER-MEGOLDÁSOK PYTHONBAN 131
4.1. Az Experta szakértői rendszer 131
4.2. Szakértői rendszerek összehasonlítása 143
5. FUZZY SZAKÉRTŐI RENDSZEREK 147
5.1. A fuzzy logika és a fuzzy halmazok meghatározása 147
5.2. Fuzzy szakértői rendszer 149
5.3. Pythonban szerkesztett fuzzy szakértői rendszer 151
FELHASZNÁLT IRODALOM 159
KÖSZÖNETNYILVÁNÍTÁS 163
MELLÉKLETEK 164

INTRODUCTION

The book is primarily aimed to help students of Economic Informatics, specialization, but all readers interested in computer science will find interesting things to learn about formal logic-based programming, rule-based expert systems, and fuzzy logic-based expert systems.

The book is structured in five chapters, each of which can be read as a stand-alone unit, but understanding the programming languages presented in practice is more effective if the reader starts with the theoretical foundations. For ease of understanding, instructions, code and example programs are presented, consistently in `Courier New, font size 11`, even within the text. For better understanding, we have mostly tried to show the results of running programs.

The first chapter presents the theory of expert systems. Starting with their historical development, we discuss in detail the expectations that led to their emergence and the evolution of knowledge representation on computers. In this chapter we discuss expert systems, their uses, limitations, creation and the conditions under which they operate. Storing, storing and "rendering" professional knowledge to the user in the form of a computer system is one of the most important modern socio-economic challenges. The development of expert systems therefore emerged at a relatively early stage in the development of artificial intelligence (AI), but their widespread adoption only became possible once the appropriate technological and computational conditions were in place. As a subsection of the theoretical part, we present the logical foundations of building expert systems. The logical operations and predicates presented here are complemented with examples in Prolog. Also here, through certain examples, we illustrate a number of economic applications of expert systems.

The second chapter introduces Prolog, the most widely used programming language for expert systems. Starting with logical statements, i.e. predicates, the many types of queries, continuing with the structure of rules and illustrated with a number of

economic examples, we reach a point where the reader can independently create functional applications in Prolog.

In the third chapter, the CLIPS language is introduced. In order, the use of variables, predicates and rules are discussed, followed by variables, knowledge representation, logical conditions, mathematical operations, functions and the most important instructions.

In chapter four, the expert system Experta, based on the Python language, is presented. Python is currently considered one of the most versatile and dynamic programming languages, so any Python instruction or function can be implemented in Experta. In this chapter, a comparative analysis of Prolog, CLIPS and Experta is presented, which is a brief summary of the three programming languages discussed before, analyzing their advantages and disadvantages.

The fifth chapter is devoted to the design of an expert system based on fuzzy logic. The limits of Boolean algebra are extended to the structure of fuzzy logic, which can be used to solve more complex problems than the "traditional" analysis of true and false statements and facts discussed so far. In the first part of this chapter, the theoretical foundations of fuzzy logic are reviewed. Because of our limited knowledge and understanding of the topics under investigation in the real world and in real economic activities, they can be better described by "degrees of belief" borrowed from probability calculus, i.e. by fuzzy variables. In the second part of this chapter, a fuzzy expert system is presented, which describes firms' attitudes towards outsourcing, according to two factors: corporate strategic planning and the number of business partners.

The examples discussed in the book, written in Prolog, CLIPS and Python, are mainly economic in nature and provide an opportunity to compare these programming languages and gain a deeper understanding of how rule-based expert systems work. The book also provides an overview of the evolution and directions of use of expert systems. For practical understanding, over 50 applications are presented, in the hope that the reader will gain a comprehensive understanding of expert systems programming.

We have also attempted to review the latest trends in the use of expert systems, which include a number of current development and research opportunities, such as the Experta expert system in Python and the fuzzy logic-based expert system written in `skfuzzy`.

Learning the specific logic used to build expert systems is useful for students and readers as it contributes to a deeper understanding of logic systems, the possibilities of knowledge representation, and ultimately the development of programming skills and abilities.

INTRODUCERE

Cartea se adresează în primul rând studenților de la Informatică economică, dar toți cititorii interesați de informatică vor găsi lucruri interesante de învățat despre programarea bazată pe logică formală, sisteme expert bazate pe reguli și sisteme expert bazate pe logică fuzzy.

Cartea este structurată în cinci capitole, fiecare dintre acestea putând fi citit ca o unitate de sine stătătoare, dar înțelegerea limbajelor de programare prezentate în practică este mai eficientă, dacă cititorul începe cu bazele teoretice. Pentru a facilita înțelegerea, instrucțiunile, codul și exemplele de programe sunt prezentate, în mod constant în `font Courier New, cu mărimea 11`, chiar și în interiorul textului. Pentru o mai bună înțelegere, am încercat în cea mai mare parte să arătăm rezultatele rulării programelor.

Primul capitol prezintă teoria sistemelor expert. Începând cu dezvoltarea lor istorică, vom discuta în detaliu așteptările care au determinat apariția lor și evoluția reprezentării cunoștințelor pe calculator. În acest capitol discutăm despre sistemele expert, despre utilizările, limitările, crearea lor și condițiile în care funcționează. Păstrarea, stocarea și „redarea” cunoștințelor profesionale către utilizator sub forma unui sistem informatic reprezintă una dintre cele mai importante provocări socio-economice moderne. Dezvoltarea sistemelor expert a apărut, prin urmare, într-o etapă relativ timpurie a dezvoltării inteligenței artificiale (IA), dar adoptarea lor pe scară largă a devenit posibilă doar odată cu apariția condițiilor tehnologice și computaționale adecvate. Ca subsecțiune a părții teoretice, prezentăm bazele logice ale construirii sistemelor expert. Operațiile logice și predicatele prezentate aici sunt completate cu exemple în Prolog. Tot aici, prin exemple concrete, ilustrăm și o serie de aplicații economice ale sistemelor expert.

Al doilea capitol prezintă limbajul Prolog, cel mai utilizat limbaj de programare pentru sistemele expert. Începând cu declarațiile logice, adică predicatele,

numeroasele tipuri de interogări, continuând cu structura regulilor și ilustrate cu o serie de exemple economice, vom ajunge la un punct în care cititorul poate crea în mod independent aplicații funcționale în Prolog.

În cel de-al treilea capitol, este prezentat limbajul CLIPS. În ordine, sunt discutate utilizarea variabilelor, predicatele și regulile, urmate de variabile, reprezentarea cunoștințelor, condițiile logice, operațiile matematice, funcțiile și instrucțiunile cele mai importante.

În capitolul patru, este prezentat sistemul expert Experta, bazat pe limbajul Python. Python este considerat în prezent unul dintre cele mai versatile și mai dinamice limbaje de programare, astfel încât orice instrucțiune sau funcție Python poate fi implementată în Experta. În acest capitol, este prezentată o analiză comparativă a limbajelor Prolog, CLIPS și Experta, care reprezintă o scurtă sinteză a celor trei limbaje de programare discutate mai înainte, analizând avantajele și dezavantajele acestora.

Al cincilea capitol este dedicat proiectării unui sistem expert bazat pe logica fuzzy. Limitele algebrei booleene sunt extinse la structura logicii fuzzy, care poate fi utilizată pentru a rezolva probleme mai complexe decât analiza „tradițională" a afirmațiilor și faptelor adevărate sau false discutate până acum. În prima parte a acestui capitol, sunt trecute în revistă bazele teoretice ale logicii fuzzy. Din cauza cunoștințelor și a cunoașterii noastre limitate legat de subiectele investigate din lumea reală și din activitățile economice reale, acestea pot fi descrise mai bine prin „grade de credință" împrumutate din calculul probabilităților, adică prin variabile fuzzy. În a doua parte a acestui capitol, este prezentat, un sistem expert fuzzy, care descrie atitudinea firmelor legat de outsourcing, în funcție de doi factori: planificarea strategică corporativă și numărul partenerilor de afaceri.

Exemplele discutate în carte, scrise în Prolog, CLIPS și Python, sunt în principal de natură economică și oferă o oportunitate de a compara aceste limbaje de programare și de a obține o înțelegere mai profundă a modului în care funcționează sistemele expert, bazate pe reguli. Cartea oferă, de asemenea, o prezentare generală a evoluției și a direcțiilor de utilizare a sistemelor expert. Pentru o înțelegere practică, sunt prezentate peste 50 de aplicații, în speranța că cititorul va dobândi o înțelegere cuprinzătoare a programării sistemelor expert.

Am încercat, de asemenea, să trecem în revistă cele mai recente tendințe în utilizarea sistemelor expert, care includ o serie de oportunități actuale de dezvoltare și

cercetare, cum ar fi sistemul expert Experta, în Python și sistemul expert bazat pe logica fuzzy, scris în `skfuzzy`.

Învățarea logicii specifice utilizate pentru a construi sisteme expert este utilă pentru studenți și cititori, deoarece contribuie la o înțelegere mai profundă a sistemelor logice, la posibilitățile de reprezentare a cunoștințelor și, în cele din urmă, la dezvoltarea competențelor și abilităților de programare.

BEVEZETÉS

A könyv elsősorban a gazdasági informatika szakos hallgatók számára készült, de minden olyan informatika iránt érdeklődő olvasó talál benne érdekes dolgokat, akit érdekel a formális logika alapján történő programozás, a szabályalapú szakértői rendszerek működése, illetve a fuzzy logika alapján megszerkesztett szakértői rendszerek.

Szerkezetét tekintve a könyv öt fejezetből áll, amelyek mindegyike akár különálló egységként is olvasható, viszont a gyakorlatban bemutatott programozási nyelvek megértését megkönnyíti, ha az olvasó első lépésként az elméleti alapokat tekinti át. A könnyebb megértést szolgálja, hogy az utasítások, kódok, példaprogramok következetesen a szövegen belül is, `Courier New betűtípussal, 11-es betűmérettel` vannak jelölve. A jobb megértés okán, többnyire törekedtem a programok lefuttatása eredményeinek a bemutatására is.

Az első fejezetben a szakértői rendszerek elméletét ismerhetjük meg. Kezdve a történeti fejlődésükkel, részletesen tárgyaltuk a kialakulásukat meghatározó elvárásokat, illetve a számítógépes tudásreprezentáció fejlődését. Ebben a fejezetben tárgyaljuk a szakértői rendszerek használati lehetőségeit, korlátjait, létrehozásukat és működésük feltételeit. A szakmai tudás megőrzése, tárolása és számítógépes rendszerként történő „visszaadása" a felhasználó számára a modern gazdasági-társadalmi kihívások egyik legfontosabbika. Ezért a szakértői rendszerek fejlődése már a mesterséges intelligencia (MI) viszonylag korai fejlődési szakaszában felmerült, de általános elterjedésük csak a megfelelő technológiai és számítástechnikai körülmények kialakulásával vált lehetővé. Az elméleti rész egyik alfejezeteként bemutatjuk a szakértői rendszerek szerkesztésének logikai alapjait. Az itt bemutatott logikai műveleteket és predikátumokat Prolog-példákkal egészítettük ki. Ugyanakkor, konkrét példákon keresztül, a szakértői rendszerek számos gazdasági felhasználási lehetőségét is bemutatjuk.

A második fejezetben a legáltalánosabban elterjedt, szakértői rendszerek céljából létrehozott programozási nyelv, a Prolog bemutatása történik. Kezdve a logikai

kijelentésekkel, vagyis a tényekkel, a lekérdezések számos típusával, folytatva a szabályok szerkezetével és számos gazdasági példával szemléltetve eljutunk oda, hogy az olvasó önállóan létre tudjon hozni működő alkalmazásokat a Prologban.

A harmadik fejezetben a CLIPS nyelvet mutatjuk be. Sorrendben tárgyaljuk a változók, tények és szabályok használatát, majd a változók, a tudásreprezentáció, a logikai feltételek, matematikai műveletek, függvények és fontosabb utasítások kerülnek bemutatásra.

A negyedik fejezetben a Python nyelvalapú, Experta szakértői rendszert mutatjuk be. A Python tekinthető jelenleg az egyik legsokoldalúbb és legdinamikusabban fejlődő programozási nyelvnek, így az Expertában bármilyen Python-utasítás vagy -függvény beilleszthetővé válik. Ebben a fejezetben található az a Prolog, CLIPS és Experta összehasonlító elemzés, amely az előnyöket és hátrányokat vizsgálva egy rövid összefoglaló szintézise az előzőekben tárgyalt három programozási nyelvnek.

Az ötödik fejezetnek a fuzzy szakértői rendszer megszerkesztése a célja. A Boole-algebra „éles határainak" a korlátait a fuzzy logika bővíti, amely az előzőekben tárgyalt, „hagyományos", igaz vagy hamis kijelentések és tények elemzésénél sokkal komplexebb problémák megoldására is alkalmas. A fejezet első részében a fuzzy logika elméleti alapjait tekintjük át. A vizsgálati tárgyakra vonatkozó korlátozott tudásunk és megismerési lehetőségeink miatt a valódi világ és a valódi gazdasági tevékenységek jobban leírhatók a valószínűségszámításból kölcsönzött „hiedelmi fokokkal", illetve a fuzzy halmazokkal. A fejezet második részében bemutatott, `skfuzzy`ban szerkesztett fuzzy szakértői rendszer a vállalatok outsourcing hajlandóságát írja le, két tényezőnek: a vállalati stratégiai tervezésnek és az üzleti partnerek számának függvényében.

A könyvben tárgyalt, Prolog, CLIPS és Python nyelvekben megszerkesztett, elsősorban gazdasági jellegű példák lehetőséget nyújtanak e programozási nyelvek összehasonlítására a szabályalapú rendszerek működésének mélyebb megértésére. A könyv egyben áttekintést nyújt a szakértői rendszerek fejlődéséről és felhasználási irányairól is. A gyakorlati megértést szolgálja, hogy több mint 50 alkalmazást mutatunk be, remélve ezáltal, hogy a tisztelt olvasó átfogó képet kap a szakértői rendszerek programozásáról.

Igyekeztünk a szakértői rendszerek használatának legújabb irányvonalait is áttekinteni, amelyekben számos aktuális fejlesztési és kutatási lehetőség is rejtőzik, ilyenek a Python szakértői rendszere, az Experta, illetve a fuzzy logikán alapuló, `skfuzzy`ban szerkesztett, szakértői rendszert is bemutattuk.

A szakértői rendszerek készítésénél használt sajátos logika elsajátítása hasznos a diákok és az olvasók számára, mert hozzájárul a logikai rendszerek mélyebb megértéséhez, a tudásreprezentáció lehetőségeinek megismeréséhez és végső soron a programozási képességek és készségek kifejlesztéséhez.

1. SZAKÉRTŐI RENDSZEREK ELMÉLETI ALAPJAI

1.1. Bevezetés

A szakértői rendszerek története szorosan kapcsolódik a mesterséges intelligencia (MI) fejlődéséhez. Az MI fejlődésének alapjai szerteágazóak: filozófiai alapjai a tudással, megismeréssel és logikai pozitivizmussal kapcsolatosak, pszichológiai alapjaihoz, többek között, a kognitív pszichológia tartozik, számos matematikai alapja közül a Boole-féle matematikát, az algoritmusokat és a valószínűségszámítást említjük, illetve biológiai alapjait a neuronokkal kapcsolatos kutatások és a neurális tudomány képezik. De a leglényegesebb alapot a számítástechnikai fejlődése jelentette, amely több fejlődési szakaszon keresztül jutott el a jelenlegi helyzetig. Az MI születését az 1956-os Dartmouth College-ban létrejött munkatalálkozó jelentette, ahol lefektették az új tudomány alapkoncepcióját: a korabeli technikai és tudományos (elsősorban matematikai eredményeket figyelembe véve) a gondolkodó gép vagy más szóval a mesterséges intelligencia létrehozása lehetséges, és mint ilyen különálló tudományterület.

Az első korszakot, amely az 1970-es évek elejéig tartott, a nagy fokú lelkesedés és merész álmok jellemezték. 1958-ban születik meg a Lisp nyelv, kifejezetten az MI programnyelveként. 1962-ban készül el a neuronháló egyik „őse", a Rosenblatt-féle perceptron.

A második fejlődési korszakban egyértelművé vált, hogy több korabeli fejlesztési elképzelést, a technológiai fejlődés lassúsága miatt, nagyon nehezen vagy egyáltalán nem lehet az elképzelt formában megvalósítani. Illetve az algoritmusok megoldási korlátaira az NP-teljesség elméletének kidolgozása világított rá. A számos csődbe jutott projekt között például egy idegen nyelvi gépi fordító algoritmus is található.

A formális logika, valamint a programozási nyelvek fejlődése az 1970-es években elvezetett a szimbolikus feldolgozórendszerek fejlődéséhez, amely nemcsak az MI fejlesztésében, hanem a szakértői rendszerek kialakulásában is meghatározó volt. A *szakértői rendszer* gondolata az MI második fejlődési időszakában született meg, a Standford Egyetemen, ahol heurisztikus programozási projekt keretében Feigenbaum és társai első alkalommal hoztak létre olyan szakértői rendszert (angolul: expert system), amely valamilyen szakterület tudását használta. Az általuk készített, MYCIN nevű orvosi diagnosztikai rendszer, amely 450 szabállyal

működött, teljesítményben elérte egy átlagos szakértőét. Így igazolást nyert az elképzelésük, hogy a tudásfeldolgozás ilyen formája eredményes, a szabályok tartalmazzák az emberi tudást, illetve hogy a szabályoknak a bizonytalanság kérdését is kezelniük kell.

Az MI egyik fejlődési irányvonalát az 1957-es évektől a számítógépes nyelvészet (computational linguistics) jelentette, amely első megközelítésben a nyelvtanulás behaviorista (viselkedésalapú – pszichológiai elmélet) jellegét vizsgálta. Majd a természetes nyelvfeldolgozással (natural language processing), amely a megértést, a nyelv újszerű elemzését szolgálja, és a mondatszerkezet, logikai kapcsolatok vizsgálatával történik, az újabb kutatások elvezettek a tudásreprezentáció (knowledge representation) témaköréhez, amely a tudás kifejezésére, tárolására, feldolgozására és elemzésére a korábbi nyelvészeti ismereteket is felhasználta.

1980 után az MI-kutatások új lendületet vesznek. Egyrészt az ötödik generációs számítógépekkel kapcsolatosan végzett kutatások, másrészt a neurális hálókkal kapcsolatos fejlesztések, majd az intelligens ágens koncepciója és alkalmazási területei jelentettek új dimenziókat az MI terén. A szakértői rendszerek fejlődésének meghatározó időszaka szintén az 1980-as évekre tehető.

Szakértői rendszer: tudásalapú számítógépes program, amely a tárgyakra, eseményekre, helyzetekre és cselekvési módokra vonatkozó szakértői tudást tartalmazza, és amely az adott terület emberi szakértőinek érvelési folyamatát utánozza. Fő részei: tudásbázis, keresőmotor és felhasználói interfész. Két fő típusa a szabályalapú és a modellalapú szakértői rendszerek. (Wiig, 1990:161-162)

A fő tevékenységei a következők:

- a tudás rögzítése;
- a tudás megőrzése;
- a tudás strukturálása;
- a tudás kiterjesztése vagy bővítése;
- a tudás terjesztése;
- kezdő alkalmazottak képzése. (Wiig, 1990:56)

Hayes–Roth (1988:4-5) megfogalmazásában a szakértői rendszer (Expert System – angolul) olyan tudásintenzív számítógépes program, amely olyan problémákat old meg, melyek hagyományosan emberi szaktudást igényelnek; számos funkciót lát el, melyek elsősorban „szakemberekre” jellemzőek, mint releváns kérdések

megválaszolása és a válasz indokainak a megmagyarázása. Közös jellemzői a szakértői rendszereknek az alábbi tulajdonságok:

– A szakértő személyeknél jobban vagy hozzájuk hasonló színvonalon tudják megoldani a legnehezebb problémákat.
– A szakértők által bevett gyakorlatok alapján heurisztikus magyarázatot szolgáltatnak.
– A felhasználókkal való interakció természetes nyelven és megfelelő módon történik.
– Szimbolikus jeleket használ működés közben és a magyarázatok kidolgozása során.
– Sikeresen kezeli a bizonytalanságot bizonyos szabályok esetében, ahol ezeknek megfelelő adatokat használ.
– Képes „mérlegelni" a nagyszámú, egymással versengő, szimultán hipotézisek között.
– Meg tudja magyarázni, hogy miért tett fel egy kérdést.
– Számos alkalmazási területei között megtalálható: diagnosztika, tervezés, szaktanácsadás, ütemezési feladatok megoldása és mások.
– Meg tudja indokolni a következtetéseit.

Mindezek fényében fontos megjegyezni, hogy a nagyszámú adatokat algoritmusokkal feldolgozó rendszerek, mint például amilyenek a DP (data processing) típusú rendszerek, nem tekinthetők szakértői rendszereknek, mert ezekből hiányzik az utóbbiakra jellemző számos funkció, mint a szaktanácsadás, diagnosztika és hasonlóak, melyek a tudás rendszerszintű feldolgozásából erednek. (Hayes–Roth, 1988:4-5)

1.2. Tudásbázisrendszerek szerkesztése és jellemzői

Tudás: igazságok, megközelítések, ítéletek és módszertanok, amelyek adott helyzetek kezeléséhez rendelkezésre állnak. A tudás egy adott körülményre vagy esetre vonatkozó információk „értelmezésére" szolgál. (Wiig, 1990:163)

Ugyanakkor a tudás egy szakterület ismereteit foglalja magában, heurisztikákat, elméleteket vagy rutinjellegű ismereteket. Tartalmukat tekintve a tények, alapelvek és módszertanok három szinten érhetők el:

(i) *nyilvánosan*, az oktatásban és könyvekben;

(ii) *szakértői tudásként* vagy „know-how" formájában, amelyet szakértők között osztanak meg, és nagy fokú tapasztalatot is tartalmaz;

(iii) *személyi tudásként,* amelyet az egyének „őriznek", viszont nem osztják meg, a legtöbbször azért, mert nincs explicit megfogalmazási mód vagy szótár ennek kifejezésére, ezért jellemzően a „megértés" tudása. (Wiig, 1990:148)

A tudás négy típusát különböztetjük meg, a növekvő absztrakció, aggregáció és komplexitás szerint:

- A tények, adatok és ismert oksági kapcsolatok (beleértve a matematikai modelleket is) egy adott tudásterülethez és az elemzendő konkrét összefüggésekhez tartoznak.
- A rendelkezésre álló tényekre és adatokra épülnek az összetett helyzetekről alkotott nézetek vagy fogalmak („Gestalt"-ok). Ez a fajta tudás olyan fogalmi képeket foglal magában, mint például hogy miként kell tekinteni egy gazdasági helyzetet, hogyan kell gondolkodni egy nehéz vegyi üzem viselkedéséről és működési állapotáról (például amikor az üzemeltető azt mondja: „instabil"), milyen vonatkoztatási keret vonatkozik egy adott versenyhelyzetre stb.
- A munkahipotézisek és ítéletek leírják az összetett helyzetek alakulására vonatkozó elvárásokat és nézeteket. Az elvárások a helyzetek működésére és az azokat befolyásoló tényezőkre vonatkozó munkahipotéziseken alapulnak. Az elvárások érvelési lépcsőfokokat is tartalmaznak a lehetséges következtetések és az összefüggések értelmezése felé.
- Az érvelési stratégiák arra vonatkozó „meta-tudások", hogy hogyan lehet egy adott kontextusban gondolkodni és érvelni bizonyos helyzetekről a helyzetekre vonatkozó információk és a tények, adatok, nézetek és ítéletek tekintetében meglévő háttérismeretek alapján. (Wiig, 1990:149)

Tudásbázis (Knowledge base – KB): a tudásalapú rendszer azon összetevője, amely tartalmazza a rendszer területi tudását valamilyen reprezentációban, amely alkalmas arra, hogy a rendszer érveljen vele. A tudásbázisokban lévő tudást jellemzően szabványos formátumban reprezentálják. (Wiig, 1990:163)

Tudásalapú rendszer (Knowledge-based system – KBS): olyan számítógép-alapú rendszer, amely az explicit szakterületi tudást tartalmazza, amelyet kifejezetten a következtetések levonására használnak, meghatározott helyzetek levonására. A szakértői rendszer a KBS egy speciális fajtája. (Wiig, 1990:163)

A tudásábrázolás jelenti a technikát, amellyel leírjuk a szaktudást a tudásbázisban. Ez szakterületenként különböző struktúrában és a leírási formában jelenik meg. A leggyakrabban használt technikák a következők (1. táblázat):

1. táblázat. Tudásábrázolási technikák

Tudástípus	Tudásábrázolási technika
Deklaratív	Logika Objektumok
Strukturált	Szemantikus háló Frame Forgatókönyv, táblázat Szabálycsoport
Procedurális	Szabály Eljárás, függvény Agenda Startégia

(Forrás: Borgulya, 1995:24)

A tudásábrázolás egyben a tudásnak a problémamegoldásban betöltött szerepét is tükrözi, abból a szempontból, hogy mennyire ad útmutatást:

- Deklaratív tudás esetében a problémára vonatkozó ismeretek között nincs összefüggés vagy olyan leírás, hogy miként kell őket alkalmazni. Például ilyen az egyszerű logikai kifejezések, tények, fogalmak vagy objektumok leírása.
- Strukturált tudás esetében létezik egy kapcsolati modell az adatstruktúrával jellemezhető fogalmak és objektumok között, amely általában grafikus formában is megjeleníthető. Ez bővíthető eljárásokkal és szabályokkal, az objektumok használatára, illetve szabálycsoportokkal a részproblémákra. A strukturált tudás az átmenetet jelenti a deklaratív tudás és a procedurális tudás között.
- A procedurális tudás konkrétan tartalmazza a problémák megoldásának lépéseit, szabályok, eljárások, függvények használatával. A megoldási folyamatokra létezhetnek stratégiák is, amelyek tartalmazzák a lehetséges megoldási célokat vagy célok rendszerét. (Borgulya, 1995:23-24)

A tudásbázisok egy szakterületen felhalmozott tudás strukturális formában történő tárolását, előhívását teszik lehetővé. A tudás a legtöbb esetben, például diagnosztikai problémák esetében, a „tünetek” vagy már mérhető jelzések (pl. rendszerüzenetek) alapján történő ajánlásokat fogalmaz meg. A tudásbázisok különböznek a környezet

(milyen programozási nyelvben készültek), a tudásmodell (következtetési motorjuk), illetve tárgyterületük szerint. A felhasználói oldalról képes kell hogy legyen a tudásbázis valamilyen platformon keresztül a „párbeszéd" folytatására, amely során a problémákat vagy kérdéseket könnyen tudja a felhasználó megfogalmazni, majd választ kap ezekere. Így használatuknak számos előnye van, míg hátrányait elsősorban az adatokra vonatkozó korlátok jelentik. (2. táblázat)

2. táblázat. A hagyományos KBS (Knowledge Base System) tudásbázisrendszerek (TBR) jellemzői

Előnyök	Hátrányok
• Egyszerű megvalósítani valamely standard TBR rendszerfejlesztési keretei között. • Tartalmaznak valamilyen formájú „aktív" tudásértelmező részt, amely segít a felhasználó kérdéseinek interpretálásában. • Nagyszámú esetet tartalmaznak. • Jelentős mélységig tudnak keresni a tárolt esetek között, amelyek esetenként adatbázisban vannak tárolva.	• TBR-szaktudást igényel a megvalósításuk. • Megfelelő szaktudást igényel a tudás előhívása és modellezése. • Részben passzív tudáselemeket is tartalmaz, amelyekhez szükséges a felhasználók másodlagos értelmezése. • Korlátozott és előrestrukturált érvelési képességgel rendelkezik. • Matematikai modellekkel való kapcsolódási lehetőségekkel nem rendelkezik.

(Forrás: Wiig, 1990:26)

A hagyományos tudásbázisrendszerekhez képest a természetes nyelvhez hasonló szabályalapú rendszerek, amelyeknek sokkal tágabb tudásanyag érhető el, és általában egy adott tudásmodellre készülnek. Ezeket nevezzük **széles körű tudásbázisrendszerek**nek, amelyek tágabb perspektívában történő értelmezésre, asszociációkra és érvelésre képesek. (3. táblázat)

3. táblázat. A széles körű KBS TBR jellemzői

Előnyök	Hátrányok
• Mélységi adatmodell, sokkal szélesebb körű funkcionális konceptualizálást és specifikációt tesz lehetővé. • A felhasználó tudását és fogalmait felhasználva sokkal hatékonyabb érvelést valósít meg. • Egyszerű megvalósítani, felhasználva az alapvető TBR fejlesztési technikákat. • Extenzív, aktív tudást tartalmaz, melynek értelmezését megkönnyíti a felhasználó számára. • Nagyszámú esetet tartalmaznak. • Jelentős mélységig tudnak keresni a tárolt esetek között, amelyek esetenként adatbázisban vannak tárolva.	• TBR-szaktudást igényel a megvalósításuk. • Megfelelő szaktudást igényel a tudás előhívása és modellezése. • Részben passzív tudáselemeket is tartalmaz, amelyekhez szükséges a felhasználók másodlagos értelmezése. • Korlátozott és előrestrukturált érvelési képességgel rendelkezik. • Matematikai modellekkel való kapcsolódási lehetőségekkel nem rendelkezik.

(Forrás: Wiig, 1990:27)

A TBR használatából számtalan vállalati előny származik. A tudás vállalati értéke ágazatonként eltér (tudásintenzívebb iparágak esetében felértékelődik), illetve vállalati tevékenységenként különbözik a szervezeti céloknak megfelelően. Az alábbi implicit, használati és végértékelőnyök a vállalati tudásnak a TBR-ben történő tárolásával és felhasználásával kapcsolatosak. (4. táblázat)

4. táblázat. A TBR vállalati használatának előnyei

Implicit előnyök	Használati előnyök
A tudás felhasználása olcsóbb. A tudás kevesebb tapasztalattal rendelkező egyének számára is elérhető. A kódolt tudással lehetővé válik a legjobb tudás kiválasztása. A TBR a tudás számára széles körű elérhetőséget és felhasználást jelent.	A termékek és szolgáltatások alacsonyabb költségei A termékek és szolgáltatások minőségének javulása Nagyobb flexibilitás a termékek szállításában és kézbesítésében Nagyobb hatékonyság a belső adminisztrációs és személyzeti tevékenységekben

<table>
<tr><th>Implicit előnyök</th><th>Használati előnyök</th></tr>
<tr><td rowspan="3">Az elosztott TBR alkalmazása javítja az üzemeltetési gyakorlatok szabványosítását.
A TBR-alkalmazás javítja a vállalati célok belső terjesztését.
A tudás kodifikációja csökkenti a személyzeti változásokból eredő tudásveszteséget.
A megfelelően bevezetett TBR-alkalmazások lehetővé teszik a szakértői tudás megváltoztatását, ha a körülmények megváltoznak
A TBR-ek használata megkönnyíti és pontosabbá teszi a tudás kialakítását.
A tudás tudásbázisokba történő kódolása gyorsítja a tudás felhalmozását.
Jobb munkavállalói morál
Nagyobb képzési hatékonyság és a képzési programok jobb minősége
Alacsonyabb képzési költségek
Biztonságosabb munkakörülmények</td><td>Alacsonyabb személyzeti fluktuáció
Magasabban képzett alkalmazottak
Magasabb bevételek
Csökkenő teljes költségek
Kedvezőbb befektetői megítélés
Jobb piaci megítélés és fogyasztói kapcsolatok</td></tr>
<tr><th>Végértékelőnyök</th></tr>
<tr><td>Növekvő profitabilitás
Növekvő cash-flow
Nagyobb szervezeti várható élettartam
Javított vagyonmaximálás
Jobb gazdasági, szociális és fizikai környezet
Jobb humánus feltételek</td></tr>
</table>

(Forrás: Wiig, 1990:27)

A TBR-ek legújabb fejlődési trendjei többek között a webalkalmazások, döntéstámogató rendszerek, illetve természetes nyelvfeldolgozó rendszerek irányába mutatnak (Alor-Hernández és Valencia-García, 2017).

1.3. A szakértői rendszerek szerkezete és működése

A számítógépes tudásreprezentáció folyamatában a tudást elsősorban racionális, logikai-szerkezeti, elvont, absztraktizált, szimbólumokkal jelölt objektumokként tudjuk kezelni. Ez egy komplex és összetett folyamat, amely számos sajátos módszertant tartalmaz. Első kérdés a szakmai tudás számítógépes „memorizálása”, második ennek megfelelő „előhívása”. A szakértői rendszerben tárolt tudás ugyanakkor a téma szakértőinek (az emberi szakértőinek) egy átfogó szintézisét kell hogy tartalmazza, oly módon, hogy a szakmai kérdések megválaszolása könnyű, felhasználóbarát és problémamegoldó módon történjen.

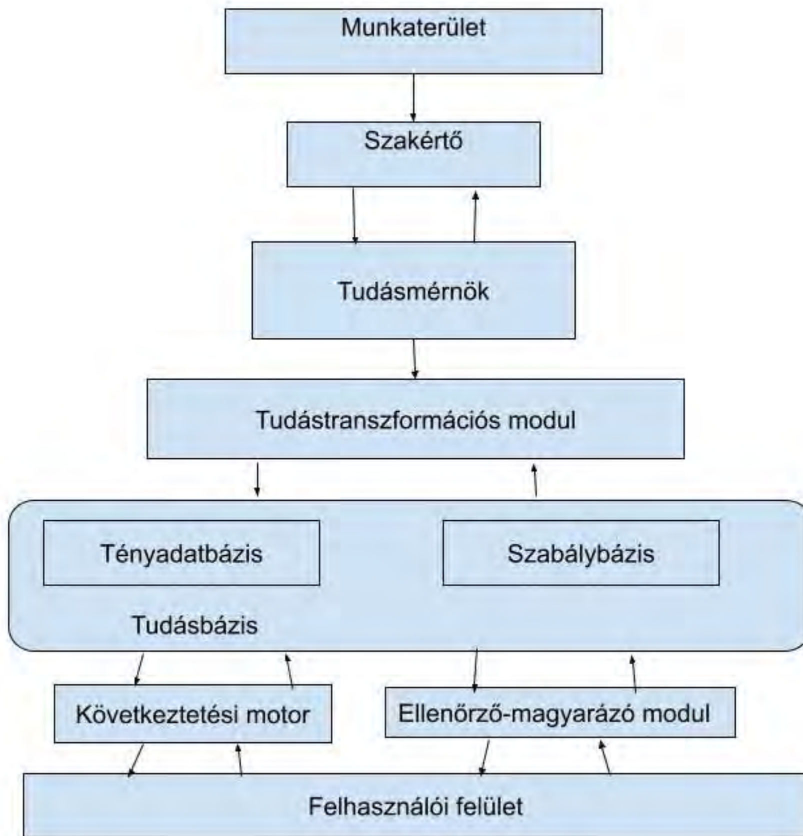
A szakértői rendszerrel történő „konzultálás", tehát egy szakterület specifikus problémáira kell választ adnia. De fontos jellemző, a problémamegoldás vagy „rezolúció" mellett, az új tudás felhalmozásának, a tudásbázis „bővítésének" a képessége. Ha meg van oldva a nagy mennyiségű tudás tárolásának, kezelésének a kérdése, a szakértői rendszer fejlesztése szinte automatikusan történik új tudással, a rendszer által biztosított tudás aktuális és naprakész.

A szakértői rendszeren belül az analitikus elemzési módszerek elengedhetetlenek. Számos ok közül az egyik, hogy a szakértői rendszer az emberi szakértőt helyettesítő együttműködő, konzultáns (és még ennél is több), emiatt nem korlátozódhat egyfajta gondolkodási sémára. Számos emberi tevékenység között a gazdasági tevékenységekre is például az jellemző, hogy kisebb vagy nagyobb bizonytalansággal működnek, így az itt használt tudásrendszereknek a bizonytalanságot, például a valószínűségszámítás elmélete alapján, kezelniük kell, ezáltal túllépve a csak analitikus elemzés korlátain, és tapasztalati tudást is megfelelően adminisztrálniuk kell.

A szakértői rendszernek, az emberhez hasonlóan, sok esetben a keresést „intuitívan", vagyis sokkal hatékonyabb heurisztikus folyamat eredményeként kell megtalálnia.

A szakértő személy számos probléma megoldása során tapasztalatot szerez, amely jótékonyan hat szakértői tevékenységére. Ez a tapasztalat gyakorlatias, új tudáshoz vezet vagy új tapasztalatok alapján bővült. Viszont a szakértőknek sok kérdésben eltérő véleményük lehet, akkor is, ha a kérdésről azonos ismereteik vannak, mivel eltérő a szemléletük, perspektívájuk egy-egy probléma kapcsán. Az emberi intelligencia azért is komplex, mert eltérő képzésekkel, tudásszervezéssel és tudásbemutatási lehetőségekkel rendelkezik. A képzési szint növelésével tökéletesíthetjük az emberi szakértőt, viszont egy szakterület tudása minden esetben, ahogy az előzőekben már megfogalmaztuk: átfogó szintézise kell hogy legyen az emberi tudásnak.

1. ábra. A szakértői rendszer szerkezete



(Forrás: Năstase és Zaharie, 1999; Orzan, 2007:48)

A szakértői rendszerek általános infrastruktúrája (1.3. ábra) a következő alapvető elemekből áll:

Munkaterület: az emberi szakértő által nyújtott szakértői tudás forrása, amely az a szakterület, ahonnan a szakértői rendszernek megoldásra benyújtott problémák származnak.

Az emberi szakértő: az a személy, személycsoport, aki, illetve amely képes a megoldandó problémát a szakterületéről az általános és a szakértői tudás területére transzformálni. Ez a szakértő formalizálja a tudást olyan formában, amely az ő szempontjából érthető és elfogadható.

Tudásmérnök: az információs rendszerek tervezése terén az elemző szerepét tölti be, ezért elsősorban azt a feladatot látja el, hogy az emberi szakértő által nyújtott tudást fogalmilag átvegye és modellezze a szakértői rendszer tudásbázisának tudásreprezentációs módszereivel kompatibilis módon.

Tudástranszformációs modul: ez a modul a tudást a kognitív tudás kifejezésformátumából a tudásbázis tárolására szolgáló technikai adathordozóra jellemző belső tárolási formátumba konvertálja. Ez a modul biztosítja a kommunikációs interfészt is az adatbázissal vagy más rendszerekkel.

Tudásbázis: tartalmazza a valós világ „objektumait" és a megválasztott közötti kapcsolatokat a megszólított és a szakértői-kognitív útvonalon továbbított tartományban – a tudás átalakítási módja. A tudásnak többféle ábrázolási módszere van: szabályok, szemantikus hálózatok és keretrendszerek. A tudásbázis szerkezetileg a ténybázisból és a szabálybázisból áll.

Tényadatbázis: tartalmazza a megoldandó konkrét probléma adatait (probléma megfogalmazása), valamint a következtetőmotor által a tudásbázison végzett következtetésből származó tényeket.

Szabálybázis: azokat a szabályokat tartalmazza, amelyek alkalmazásával az ismert tényekből kiindulva új, úgynevezett levezetett tényeknek nevezett információk kerülhetnek be a ténybázisba. A szabálybázisban újabb szabályok alkalmazhatók, amelyek újabb tényeket eredményeznek, amíg a végső következtetés vagy cél meg nem lesz fogalmazva.

A következtetési motor: a szakértői rendszer tényleges feldolgozóeleme, amely a tényekből (a probléma bemeneti adataiból) kiindulva aktiválja a tudásbázis megfelelő tudását, és így új tényekhez vezető következtetéseket állít fel. A probléma sajátosságainak megfelelően, a terület ismereteinek felhasználásával megoldási tervet készít. A következtetőmotor működése során a tudásbázis új elemek hozzáadásával vagy a meglévők módosításával gazdagodik. A következtetési motor egy olyan program, amely következtetési algoritmusokat (deduktív, induktív vagy vegyes) valósít meg, de független a tudásbázistól.

Ellenőrző-magyarázó modul: célja, hogy széles körben hozzáférhető formában (természetes nyelven) bemutassa a következtetőmotor által végzett következtetés indoklását és a felhasználó által megválaszolandó kérdéseket. Ez a modul az emberi szakértő számára is hasznos a tudásbázis konzisztenciájának ellenőrzéséhez.

Felhasználói felület: a felhasználó és a szakértői rendszer párbeszédét végzi a bemeneti adatok megadása és a megoldandó probléma eredményeinek megjelenítése tekintetében, ablakok, képek, menük rendszerén keresztül. Más megfogalmazásban a felhasználói felület (1. ábra), amely a modulok közötti adatátmenetet végzi, a *kommunikációs struktúra* nevet viseli. (Demolombe, 1988; Orzan, 2007:46-48)

5. táblázat. A szakértői rendszer funkciói a felhasználási területek szerint

Funkciók/Területek	Leírás
1. Kontroll és monitorozás	A rendszerek intelligens kontrollját valósítja meg.
2. Hibaelhárítás és javítás	A rendszerek működési hiányosságaira tesz korrekciós javaslatokat.
3. Tervezés	Termékek és rendszerek tervezése
4. Diagnosztika és karbantartás	A működési hibákat azonosítja és a szükséges korrekciós javaslatokat fogalmazza meg.
5. Oktatás	Computer Assisted Instruction
6. Tolmácsolás/értelmezés	Helyzetek osztályozása és új helyzetek értelmezése a helyzetelemzés jelzései alapján
7. Ütemtervek készítése	Célirányos tevékenységtervek szerkesztése
8. Előrejelzés	Az ismert adatok alapján történő előrejelzések készítése
9. Szimulációk készítése	A tevékenységeknek és eseményeknek rendszerből levezetett következményei
10. Osztályozás	Kategóriák vagy osztályok szerint történő osztályozás

(Zaharia és tsai., 2003:43)

A szakértői rendszerek által betöltött funkcionális feladatok felhasználási területük szerint (5. táblázat), a TBR-ek felhasználásával, a következő példákkal szemléltethetők, a kommunikáció, érvelés, problémamegoldás, tudásmenedzsment és képzés esetében:

- *Kommunikáció* a különböző feladatkörök, illetve felhasználók között. Például egy TBR-ben integrálják a gyártás információit, amely biztosítja a kommunikáció átvitelét műszakváltás esetén.
- *Érvelés*, az összetett emberi érvelés támogatása vagy jól meghatározott és kognitívan jól illeszkedő kiegészítő érvelési feladatok. Például a TBR a tünetekből meghatározza egy probléma okát (ok-okozati következtetéssel).

- *Problémamegoldó memória*, a rövid távú memória bővítésére, illetve problémák folyamatos „kezelése” (tanácsadással vagy diagnosztikai rendszerrel). Például TBR-alapú szakértői rendszerek használata hibaelhárításban.
- *Tudásszintézis*, vagyis a különböző forrásokból származó tudás összegyűjtése és koherens, releváns módon történő bemutatása. Például riasztáskezelő vagy üzemfelügyeleti rendszer.
- *Képzés*, új témakörök elsajátításának támogatása. Például olyan TBR használata, amely szimulálja az üzem működését az üzemvezető számára. (Wiig, 1990:56-57)

1.4. Szakértői rendszerek gazdasági célú felhasználása

A *szakértői rendszerek* gazdasági célú felhasználására számtalan példát találunk a szakirodalomban: könyvvizsgálatnál használt szakértői rendszert (Dillard és Mutchler, 1988), befektetői kockázatvállalás-értékelőt (Rozenholc, 1988), illetve termelés tervezésében és a menedzsment számos más területén (Liao, 2005).

Zaharia és társai (2003) VP-Expert típusú, szabályalapú szakértői rendszert használ, hitelállomány elemzésével kapcsolatos döntések modellezésére, állásinterjú pontozásának kiszámításához, gazdasági ügynök havi díjának kiszámításához, konkurenciaelemzéshez, cash flow számításhoz, hitelelbíráláshoz, költségelemzéshez, illetve piaci versenykörnyezet elemzéséhez.

Orzan (2007) VP-Expert típusú, szabályalapú szakértői rendszert használ az új termék piacra juttatásának döntéseit tartalmazó marketingtevékenységekre.

Továbbá banki és pénzügyi területeken mint személyi pénzügyi megtakarítási tanácsadó (Mersnissi 1988), illetve Jakab és társai (2006) szerint a Magyar Nemzeti Bank (MNB) egy időben előrejelzésre használta a szakértői rendszereket, melyeket idővel kiegészítettek, majd felváltottak a makrogazdasági modellek, illetve a közigazgatás számos területén, ahogyan Futó (2019) könyve bemutatja.

A *fuzzy szabályalapú szakértői rendszerek* számos gazdasági alkalmazása közül elég, ha csak a beszállítók kiválasztásában (Altinoz, 2008), a vállalatok pénzügyi értékelésében (Shue és társai, 2009), illetve a marketingstratégia kialakításában (Li és társai, 2011) által használt rendszereket említjük itt.

1.5 Szakértői rendszerek szerkesztésének logikai alapjai

A szakértői rendszerek a kijelentéseken alapuló, logikai algebra szerkesztési szabályaira épülnek. Alapelemei a kijelentések, melyek Igaz vagy Hamis értéket vehetnek fel, logikai összefüggéseit pedig a Boole-matematika határozza meg.
Az alapműveletek igazságtáblázatai, az És (konjunkció), Vagy (diszjunkció) és Nem (negáció) Prologban történő utasításokkal együtt vannak megszerkesztve az alábbi táblázatban.

6. táblázat. A logikai műveletek igazságtáblázatai Prolog-példákkal

Logikai kijelentés		Művelet		Prolog
		És (∧)	Eredmény	
Igaz	Igaz	Igaz ∧ Igaz	Igaz	`?- true,true.` `true.`
Igaz	Hamis	Igaz ∧ Hamis	Hamis	`?- true,false.` `false.`
Hamis	Igaz	Hamis ∧ Igaz	Hamis	`?- false,true.` `false.`
Hamis	Hamis	Hamis ∧ Hamis	Hamis	`?- false,false.` `false.`
		Vagy (∨)	Eredmény	
Igaz	Igaz	Igaz ∨ Igaz	Igaz	`?- true;true.` `true.`
Igaz	Hamis	Igaz ∨ Hamis	Igaz	`?- true;false.` `true.`
Hamis	Igaz	Hamis ∨ Igaz	Igaz	`?- false;true.` `true.`
Hamis	Hamis	Hamis ∨ Hamis	Hamis	`?- false;false.` `false.`
		Nem (¬)	Eredmény	
	Igaz	¬ Igaz	Hamis	`?- not(true).` `false.`
	Hamis	¬ Hamis	Igaz	`?- not(false).` `true.`

A logikai kijelentések (predikátumok) lehetnek egy, kettő vagy több argumentumúak (Pólos és Ruzsa, 1978:45-51). A könnyebb érthetőség kedvéért, az alábbi predikátumok mellett megszerkesztettük ezek Prolog-„tényeit" is (a Prolog-nyelvbe történő részletes bevezetés a következő fejezetben történik majd):

7. táblázat. Predikátumok típusai Prolog-példákkal

Predikátumok	Prolog
Egyargumentumú predikátumok	

A vásárló elégedett.
A vállalat kisvállalkozás.
Az ágazat IT.
Legó készleten.
Készleten 147 darab.
	`vasarlo(eledegett).`
	`vallalat(kisvallalkozas).`
	`agazat(iT).`
	`keszleten(lego_jatek).`
	`keszleten(147).`

Kétargumentumú predikátumok	

A vásárló helyi és elégedett.
 	
A vállalat kisvállalkozás, IT-ban működik.
 	
Készleten 147 darab van a legóból.
	`vasarlo(eledegett, helyi).`
	
	`vallalat(kisvallalkozas, iT).`
	
	`keszleten(lego_jatek,147).`

Háromargumentumú predikátumok	

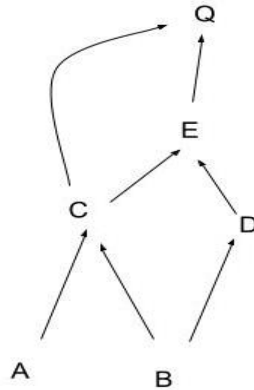
A vásárló helyi, fiatal korosztályú és elégedett.
 	
A vállalat kisvállalkozás, IT-ban működik, és a forgalma 128 000.
 	
Készleten 147 darab van a legóból, az L287907-es típusból.
	`vasarlo(eledegett, helyi, fiatal_korosztaly).`
	
	`vallalat(kisvallalkozas, iT, 128000).`
	
	`keszleten(lego_jatek, 147, L287907).`

A logikai összefüggések szerkesztése során figyelembe vesszük ezek felcserélhetőségét (kommutativitás) és csoportosíthatóságát (disztributivitás) mint alaptulajdonságokat. Például:
$A \wedge B = B \wedge A$, illetve $A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$.

Előreláncolás során abból indulunk ki, hogy az ismert predikátumok, vagyis a logikai előzmények igazak. A logikai feltételek közötti kapcsolatokat szabályok írják le, a következtetési lánc a cél irányában előremutató, vagyis ez az úgynevezett *előreláncolt következtetési eljárás*. A *hátraláncolás* következtetési módszer ennek pontosan az ellentéte, mert a célból kiindulva, hátrafelé visszafejtve az utolsó feltételig vizsgálja a predikátumok igazságtartalmát.

Előreláncolás során, az alábbi logikai feltételrendszerben a Q-val jelölt cél igazságtartalmára következtetünk, úgy, hogy a kapcsolatokat logikai És jelenti.

2. ábra. Példa előreláncolt logikai feltételrendszerre



Tehát:

$A \wedge B \rightarrow C,$
$B \rightarrow D,$
$C \wedge D \rightarrow E,$
$C \wedge E \rightarrow Q.$

Kérdések:

1. Mit nevezünk szakértői rendszernek?

2. Hogyan tudjuk meghatározni a tudásrendszerekben tárolt tudást, és milyen típusai vannak?

3. Mutassa be a tudásábrázolási technikákat!

4. Melyek a tudásbázisrendszerek (TBR) jellemzői?

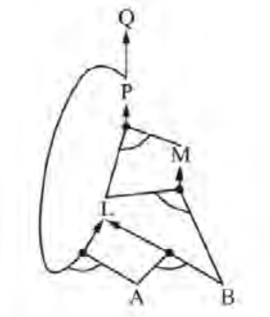
5. Melyek a tudásbázisrendszerek (TBR) használatának vállalati előnyei?

6. Mutassa be a szakértői rendszerek szerkezetét és működését!

7. Mutassa be a szakértői rendszerek funkcióit a felhasználási terület szerint!

8. Milyen gazdasági területeken használják a szakértői rendszert?

9. Szerkessze meg az alábbi predikátumok előreláncolt logikai szabályait:



2. A PROLOG PROGRAMOZÁSI NYELV

2.1 Bevezetés

A Prolog megnevezés az angol „PROgramming in LOGic” kifejezésből ered, ami a logikai programozásra utal, és eredetileg a természetes nyelv feldolgozásával kapcsolatos alkalmazásokat jelentette. Az MI azon alkalmazási területei, amelyekre használták a Prologot: a szakértői rendszerek, természetesnyelv-feldolgozás, intelligens adatbázisok, és ezeken kívül a hagyományos alkalmazások készítésére is alkalmas. Szemantikailag a Prolog programok nagyban hasonlítanak az eredeti logikai specifikációra, így könnyen fejleszthetők. Viszont a Prolog jelentősen eltér a többi programozási nyelvtől, mivel alapvetően *deklaratív* típusú nyelv: a logikai tényeket és kapcsolatokat „kijelentjük”, majd ezek alapján a Prolog képes meghatározni, „következtetni” bizonyos kijelentések igaz vagy hamis voltára. Ez a megközelítés elméleti és gyakorlati szempontból is érdekes kihívást jelent.

Jelen könyvben a SWI-Prolog 8.4.1-es verziójában készítettük el a példaprogramokat, amelyek ingyenesen elérhetők a swi-prolog.org oldalon[1]. (3. ábra)

3. ábra. A SWI-Prolog programkönyezete

```
SWI-Prolog (AMD64, Multi-threaded, version 8.4.1)
File  Edit  Settings  Run  Debug  Help
Welcome to SWI-Prolog (threaded, 64 bits, version 8.4.1)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?-
% SWI-Prolog version (threaded, 64 bits, version 8.4.1)
% SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
% Please run ?- license. for legal details.
```

A Prolog programok futtatása a File/Consult menü alatt vagy a `consult` utasítással történik:

[1] Telepítés: https://www.swi-prolog.org/Download.html
Dokumentációk: https://www.swi-prolog.org/pldoc/doc_for?object=manual

```
?- consult('vallalatok.pl').
```

A Prolog program tartalmilag elemi moduláris egységekből, úgynevezett predikátumokból („kijelentésekből”) áll, melyek legjobban egy hagyományos programozási nyelvben található szubrutinokhoz hasonlítanak. A predikátumok külön hozzáadhatók és tesztelhetők a Prolog programban, ami lehetővé teszi a lépésről lépésre történő fejlesztést.

A továbbiakban bemutatott Prolog-utasítások hatékony megértéséhez ajánlott a kipróbálásuk és kisebb változtatásokkal történő tesztelésük, majd egyéni bővítésük. A hagyományosan első program a Prologban így néz ki:

```
?- write('Hello World').
Hello World
True.
```

A tények és szabályok első rövid bemutatása után, ezt a következő fejezetekben részletesen tárgyaljuk majd. Nézzük tehát, hogyan jelenik meg egy logikailag igaz kijelentés, egy predikátum, más néven „tény” (angolul „facts”), a Prologban. A predikátumok kapcsolati rendszereket fejeznek ki, például a szülője Szilárd Zentének, illetve a testvére Albert Annának így néz ki:

```
szulo(Szilard, Zente).
testver(Albert, Anna).
```

A Prolog-kijelentések és -utasítások mindig pontban végződnek. Nézzük az alábbi kétsoros Prolog-utasítást, ami tulajdonképpen egy szabályt, „rule”-t és egy kijelentést, predikátumot tartalmaz. Ezek meghatározását az alábbiakban fogalmazzuk meg. A Prolog-fájlokat *.pl néven tudjuk elmenteni, a `File/Consult` menüpont alatt tudjuk megnyitni és a parancssorból futtatni.

```
fogyaszto(X):- szemely(X).
szemely(albert).
```

A szabályok, szerkezetüket tekintve, a logikai implikációnak felelnek meg, vagyis „Ha… Akkor…” („If… Then…”) utasításcsoportoknak. Az előbbi példában az első sor, a `fogyaszto(X)` szabály, azt jelenti, hogy „minden személyt” egyben „fogyasztónak” is tekintünk. Ezért a következő kérdezések működnek, ahol X a változó (Prologban a változókat nagybetűvel jelöljük):

```
?- fogyaszto(albert).
yes
?- fogyaszto(X).
X = albert.
```

Látható, hogy a Prolog deklaratív jellegében áll a nagy erőssége is. Olyan kódokat lehet írni, amelyek szerkezete könnyen átlátható. Bővítsük két új ténnyel:

```
szemely(bence).
szemely(erika).
```

Az alábbi `potencialis_fogyasztok` szabállyal fel tudjuk sorolni mindegyik potenciális fogyasztónkat:

```
potencialis_fogyasztok:-
write('Ismert fogyasztók:'),
nl,
fogyaszto(X),
write(X), nl,
fail.?- potencialis_fogyasztok.

Ismert fogyasztók:

albert
bence
erika
false.
```

Szabályra két példát láttunk az előbbiekben: `fogyaszto(X)` és `potencialis_fogyasztok`. Tekintsük át a „tényként" vagy logikai kijelentésként használt programelemeket még egyszer! A Prolog azért sajátos, mert a szerkezetét tekintve a program egyben az adatbázis is (tudásbázisnak is nevezzük), logikailag igaz kijelentésekből áll. Ezek a *predikátumok* vagy *állítások* (angolul: predicate), amelyek meg vannak határozva saját névvel, és az argumentumaik (jellemzők vagy mezők) számával (ez utóbbit angolul „arity"-nek is nevezik). Emiatt két azonos nevű, de eltérő argumentumszámmal meghatározott predikátumot a Prologban különbözőeknek tekintünk.

Egy predikátumhoz az adatbázisban egy vagy több klóz `(clause)` van hozzárendelve. A klóz lehet tény `(fact)` vagy szabály `(rule)`. A fenti példákban a `fogyaszto()` nevű predikátumhoz*,* három klóz tartozik: `albert`, `bence` és `erika`, amelyek tulajdonképpen tények.

A következő fejezetben részletesen tárgyaljuk a tények használatát a Prologban.

2.2. Tények használata a Prologban

A tények, *predikátumok* vagy *állítások* (angolul: predicate), a Prolog nyelv szerkezeti alapegységeit jelentik. Legjobban egy relációs adatbázis rekordjaihoz hasonlítanak. Szintaxisuk:

```
pred(arg1, arg2, argN).
ahol:
pred       A tény neve
arg1, arg2,     Az argumentumok
N          Aritás (angolul „arity”), az argumentumok
száma.
```

Példa nulla argumentummal rendelkező tényre:

```
pred.
```

Az argumentumok lehetnek szabványos Prolog-kifejezések (termek), az alábbi típusokkal:

egész szám (integer)	**Pozitív vagy negatív egész szám.**
tizedes számok	**Lebegőpontos számok.**
szövegek (strings)	**Dupla idézőjellel jelölt ASCII-szövegek, melyeket a program listaként kezel.** **Például: „Prémium kategóriás üzleti partner”, „Törzsvásárló”.**
állandó (atom)	**Állandó, amelyet kisbetűvel kezdődő szöveggel jelölünk.** **Például: hello, ketSzovegbolAll, a147, long_name, x_28**
változó	**Nagybetűvel jelölt állandó vagy alulvonással (_) kezdődő alfanumerikus szerkezet.** **Például: X, Y, List_of_employees, _elso_valtozo, MZperX2000**

Egyszerű idézőjellel bármilyen nem szabályos állandó szabályossá tehető. Például: nem szabályos állandóként: `Nagybetu, Kis István, Consumer,`

`Valtozo12`. Szabályos állandó: `'Nagybetu'`, `'Kis István'`, `'Consumer'`, `'Valtozo12'`…

A dupla idézőjel a szövegformátumnak van fenntartva.
Az atomok tartalmazhatják a következő karaktereket is: - - > ++
A változók nagybetűsek, például: X, Y1, Zs, Ab, Vallalat, Termek_tipus stb.

A Prolog-karakterek a következőkből állhatnak:

nagybetűk: A–Z
kisbetűk: a–z
számjegyek: 0–9
szimbólumok: + - * /\ ^ . , ~ : ? @ # $ &

A fenti egységekből alakíthatjuk ki a tényeket, ahol a tény nevét követően atomok, szövegek vagy számok lesznek a tény argumentumai. A tényeket egyben adatok tárolására is használjuk, például az üzleti partnerünk státusza, az alábbi példákban:

```
uzleti_partner('Kis István', seriuos_kft, vip_ugyfel).
uzleti_partner('Szabó Árpád', vandor_rt, standard_ugyfel).
```

Ebben a példában neveknél szükségesek az egyszerű idézőjelek, hogy megfeleljenek a szövegformátum előírásának.

Egy rendszer működési és hibajelző üzeneteinek az argumentumai lehetnek például:

```
rendszer(alapallapot, 15, 32, 45, 25, 7).
rendszer(rendszerhibak, 102, 250, 95, 136, 88).
rendszer(veszhelyzet, 122, 285, 125, 222, 444).
```

Egy HR (human resource) rendszer része lehet az alábbi példa:

```
szakismeret(idegen_nyelv1, angol).
szakismeret(idegen_nyelv2, nemet).
```

Most, hogy néhány tényt megszerkesztettünk, nézzük meg, hogyan működik ezekre a lekérdezés. A Prolog interpreter a tények rögzítése mellett a lekérdezésük (query – lekérdezdés, vagy call - hívás) eszközét is biztosítja.

Például, az alábbi utasítással az X változóhoz hozzárendeli az `angol`-t:

```
?- szakismeret(idegen_nyelv1, X).
```

```
X = angol.
```

Az alábbi példában az ügyfelekre vonatkozó tényekkel kezdjük, az első argumentum az ügyfél neve, a második a település, a harmadik a hitelminősítése.

```
ugyfel('Biro Arpad', kolozsvar, bbb).
ugyfel('Nagy Maria', marosvasarhely, ccc).
ugyfel('Kovacs Bela', csikszereda, aaa).
ugyfel('Kis Istvan', csikszereda, aaa).
```

Az alábbi utasítással a `Nev` változóhoz hozzárendeli a `bbb` hitelminősítésű személyeket. Figyeljük meg, hogy a helyszínt alulvonással jelöltük, így bármilyen lehet:

```
?- ugyfel(Nev,_, bbb).
Nev = 'Biro Arpad'.
```

Hasonló a helyzet az `aaa` hitelminősítésű személyek esetében is, azonban itt, ha az első találat után pontot nyomunk, nem keres tovább.

```
?- ugyfel(Nev,_, aaa).
Nev = 'Kovacs Bela'.
```

Helyesen az `aaa` hitelminősítésű személyt a logikai Vagy-ot jelentő pontosvessző lenyomása után fogja felsorolni:

```
?- ugyfel(Nev,_, aaa).
Nev = 'Kovacs Bela' ;
Nev = 'Kis Istvan'.
?-
```

A `termek` tények első argumentuma a termékazonosító ID-ja, a második a neve, a harmadik a raktár helyét mutatja.

```
termek(id_187, iroasztal_tipus_12, raktar1).
termek(id_193, ejjeliszekreny_tipus_5, raktar1).
termek(id_145, kanape_kek_tipus_6, raktar2).
termek(id_156, forgoszek_tipus_2, raktar2).
```

Hasonlóan, a következő lekérdezésnél a `Raktar_helyszin` változóhoz rendeljük hozzá a két keresett termék készletének helyszínét:

```
?- termek(_, forgoszek_tipus_2, Raktar_helyszin).
```

```
Raktar_helyszin = raktar2.
?- termek(_, kanape_kek_tipus_6, Raktar_helyszin).
Raktar_helyszin = raktar2.
```

És végül a készletnyilvántartó tények első argumentuma a termékazonosító ID, a második a `készleten` tárolt mennyiség.

```
keszleten(id_187, 21).
keszleten(id_193, 0).
keszleten(id_145, 2).
keszleten(id_156, 7).
```

A következő kérdésekkel tudjuk megvizsgálni, hogy az `id_187` és `id_193` termékekből hány darab van készleten:

```
?- keszleten(id_187, Darab).
Darab = 21.
?- keszleten(id_193, Darab).
Darab = 0.
```

2.3. Hogyan működnek a kérdések a Prologban?

Első lépésben az egyszerű kérdéseket (lekérdezéseket) tekintsük át, amelyeket olyan példákkal illusztrálunk, amelyek már megszerkesztett tényekre épülnek. A Prolog-kérdések (queries) mintaillesztéssel működnek. A lekérdezési mintát célnak nevezzük. Ha van a célnak megfelelő tény, akkor a lekérdezés sikeres és az interpreter igennel (yes) válaszol. Ha nincs egyező tény, akkor a lekérdezés sikertelen, és az értelmező nemmel (no) válaszol.

Ezt a Prolog mintaillesztését *egységesítésnek* (unification) nevezzük. Ha az adatbázisunk csak tényeket tartalmaz, az egységesítés akkor sikeres, ha a következő három feltétel fennáll:

- A célban és az adatbázisban megnevezett predikátum megegyezik.
- Mindkét predikátum azonos aritású (az argumentumok száma egyenlő).
- Minden argumentum egyenlő.

A célok általánosítása a Prologban a változókkal történik, amelyek a más programozási nyelvektől eltérően, elsősorban logikai változókként működnek. A logikai változók egy vagy több argumentumot helyettesítenek a céllekérdezésben.

Ezáltal az egyesítésben is új dimenziót adnak, mert a tények esetében, ahogy láttuk, az egyesítés csak úgy sikeres, ha az állítmánynevek és az aritások azonosak. Az argumentumok összehasonlításakor azonban egy változó sikeresen illeszkedik bármely kifejezéshez (termhez). Sikeres egységesítés után a logikai változó felveszi annak a kifejezésnek az értékét, amellyel talált. Ezt hívják a változó kötéseinek (binding angolul). Amikor egy változót tartalmazó cél sikeresen egyesül egy adatbázisban található ténnyel, a Prolog visszaadja az újonnan kötött logikai változó értékét.

Mivel előfordulhat, hogy egy változóhoz több érték is köthető, és mégis megfelel a célnak, a Prolog lehetőséget biztosít arra, hogy alternatív értékeket használjunk. Ha pontosvesszőt (;) írunk be a nyíl után, a Prolog alternatív kötéseket keres a változókhoz.

Például használhatunk egy logikai változót az összes személy megkeresésére:

```
szemely(bence).
szemely(erika).
?- szemely(X).
X = bence ;
X = erika.
```

Az alábbi példa szerint találhatunk meg minden csíkszeredai fogyasztót a ténylistából. Emlékezzünk vissza, hogy a változók nagybetűkkel kezdődnek!

```
fogyaszto(bence, csikszereda).
fogyaszto(erika, csikszereda).
fogyaszto(andras, kolozsvar).

?- fogyaszto(Szemely, csikszereda).
Szemely = bence ;
Szemely = erika.
```

Két változóval a személyeket és a helyszíneket is felsorolhatjuk:

```
?- fogyaszto(Szemely, Hely).
Szemely = bence,
Hely = csikszereda ;
Szemely = erika,
Hely = csikszereda ;
Szemely = andras,
```

```
Hely = kolozsvar.
```

Amikor a Prolog megpróbál teljesíteni egy predikátumcélt, mint például a `fogyaszto` helyét, (vagyis második) argumentumát, végighalad a kereséssel a klózokon. Amikor megtalálja az egyezéseket a változóiban, jelzi azt a konkrét klózt, amelyet a cél kielégítésére használtak. Ezután, ha a felhasználó további válaszokat kér, a keresés folytatódik az adott klóz tagmondatai között. Nézzük erre az alábbi példát:

```
Minta                fogyaszto(X, csikszereda).

Klóz                 fogyaszto(andras, kolozsvar).
```

Ez az egyesítés sikertelen. A tények nevei megegyeznek, az aritás (argumentumok száma) is talál, de a második argumentumok (csikszereda és kolozsvar) különböznek. Az alábbi példában viszont már mindhárom jellemző talál, ezért ez az egyesítés sikeres. Az első argumentumok akkor lehetnek azonosak, ha az X változó felveszi az erika értéket.

```
Minta                fogyaszto(X, csikszereda).

Klóz                 fogyaszto(erika, csikszereda).
```

Tehát a Prolog interpreter kiírja, hogy sikeres, és az X kötését rögzítette. Sikeréről számol be, és az X változó kötéseit kiírja:

```
?- fogyaszto(X, csikszereda).
X = erika.
```

Bővítsük ki `anna` és `lehel` fogyasztókkal, akiknek szintén `kolozsvar` hely van megadva:

```
fogyaszto(andras, kolozsvar).
fogyaszto(anna, kolozsvar).
fogyaszto(lehel, kolozsvar).
```

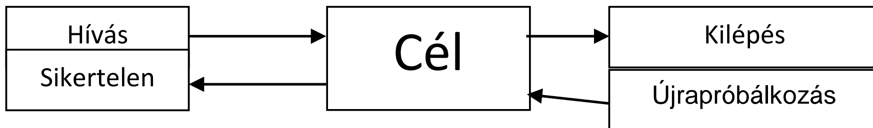
Mivel több azonos helyhez kötött személy is egyezhet a lekérdezési mintával, használjuk a pontosvessző (;) szimbólumot:

```
?- fogyaszto(X, kolozsvar).
X = andras ;
X = anna ;
X = lehel.
```

Ennek a lefuttatása során az X változó a pontosvessző után felveszi a második és harmadik értéket. Ezt hívjuk visszalépésnek (backtracking).

A Prolog végrehajtása megértésének legjobb módja a végrehajtás nyomon követése (trace bekapcsolása) lesz. De először a célok mélyebb megértésére van szükség. A Prolog-cél négy porttal rendelkezik, amelyek a célon keresztüli vezérlés áramlását jelzik: hívás, kilépés, újraindítás és sikertelenség. Először a cél meghívása történik meg. Sikeres esetben kilép. Ha nem, akkor nem sikerül. A pontosvessző (;) beírásával a célban újra próbálkozik, a nyílnál (=>), és akkor az újbóli portot írjuk be. (4. ábra)

4. ábra. A cél portjainak szerkezete



(Forrás: Adventure in Prolog, Dennis Merritt, 1990, Springer)

Elemei:

- Hívás (`Call`): elkezdi keresni a céllal egységesülő záradékokat.
- Kilépés (`Exit`): azt jelzi, hogy a cél teljesült, helyjelzőt állít be a klóznak, és a változóhoz hozzá van rendelve a megfelelő kötés.
- Újra próbálkozás (`Redo`): megpróbálja újra a célt, feloldja a változók kötését és folytatja keresést.
- Sikertelen (`Fail`): azt jelzi, hogy nem egyezik egyetlen klóz sem a céllal.

Nézzük meg, hogyan működik az előző utasítás esetében:

```
[trace] 22 ?- trace.
true.
[trace]  ?- fogyaszto(X, kolozsvar).
   Call: (10) fogyaszto(_8568, kolozsvar) ? creep
   Exit: (10) fogyaszto(andras, kolozsvar) ? creep
X = andras ;
   Redo: (10) fogyaszto(_8568, kolozsvar) ? creep
   Exit: (10) fogyaszto(anna, kolozsvar) ? creep
X = anna ;
   Redo: (10) fogyaszto(_8568, kolozsvar) ? creep
```

```
   Exit: (10) fogyaszto(lehel, kolozsvar) ? creep
X = lehel.
[trace]  ?-
```

A sikertelen keresésre példa:

```
[trace]  ?- fogyaszto(X, nagyvarad).
   Call: (10) fogyaszto(_17716, nagyvarad) ? creep
   Fail: (10) fogyaszto(_17716, nagyvarad) ? creep
false.
```

Hozzunk létre olyan tényt, amely minden klózra teljesül:

```
potencialis_vasarlo(Z).

?- potencialis_vasarlo(brigitta).
true.
?- potencialis_vasarlo(lehel).
true.
```

Figyeljük meg, hogy ebben az esetben nem történnek meg a `Z = brigitta` és `Z = lehel` hozzárendelések, bár így történik a kötésük, mivel az interpreter csak a lekérdezésben említett változókat sorolja fel, a programban használtakat nem.

További lekérdezéspéldákhoz, amelyekben a kérdések logikai És-sel vannak összekötve, tekintsük az alábbi ténylistát:

```
vasarol(vallalat1,vallalat5).
vasarol(vallalat1,vallalat2).
vasarol(vallalat5,vallalat7).
vasarol(vallalat2,vallalat7).
vasarol(vallalat5,vallalat1).
```

Megnézzük, hogy az 1-es és az 5-ös vállalat, illetve az 1-es és a 7-es kölcsönösen vásárol-e egymástól, úgy, hogy a vessző a logikai És-t jelenti az alábbi kérdésnél:

```
vasarol(vallalat1,vallalat5),
vasarol(vallalat5,vallalat1).
?- vasarol(vallalat1,vallalat5),
vasarol(vallalat5,vallalat1).
true.
?- vasarol(vallalat1,vallalat2),
vasarol(vallalat2,vallalat1).
false.
```

Látható, hogy csak az első válasz helyes.
Használjuk az X változót, hogy azonosítsuk azt a vállalatot, amelytől a 2-es és az 5-ös vállalat vásárol termékeket:

```
?- vasarol(vallalat2,X), vasarol(vallalat5,X).
X = vallalat7 .
```

Tekintsük az **1. mellékletben** alkalmazottakra vonatkozó tényeket. A következő programmal a ténylistából, ahol a tapasztalatot az évek számával jelöljük, megtaláljuk a legtapasztaltabb személyt:

```
tapasztalat(janos,4).
tapasztalat(istvan,7).
tapasztalat(arpad,12).
tapasztalat(huba,6).
tapasztalat(jolan,9).

leg_tapsztaltabb(Nev):-
tapasztalat(Nev,Y),
\+ (tapasztalat(_, Y1), Y1 > Y).
```

Az eredményben a `Nev` változóhoz rendeli hozzá a szabály a megfelelő személy nevét:

```
?- leg_tapsztaltabb(Evek).
Evek = arpad .
```

2.4. Szabályok a Prologban

A logikai programozásban a szabály az implikáció („=>") megfelelője. A Prologban a szabálynak két fő része van: cím (head) és törzs (body), amelyeket a „:-„ jel köt össze, amely a logikai „Ha" (if) feltételnek felel meg. A tényekhez és minden utasításhoz hasonlóan a szabályok is ponttal végződnek. A törzs azon célokat vagy tényeket fogalmazza meg, amelyeknek teljesülniük kell ahhoz, hogy a cím igaz legyen. Tehát a feltétel a szabály törzsében fogalmazódik meg.

```
fogyaszto(albert).
fogyaszto(bela).
preferencia(albert, alacsony_ar, minoseg).
preferencia(bela, barmely_ar, minoseg).
preferencia(sandor, alacsony_ar, minoseg).

ar_preferencia(Nev)  :- /* kisbetűvel jelöljük a szabály
nevét*/
     preferencia(Nev, alacsony_ar, _),
     fogyaszto(Nev), /* nagybetűvel jelöljük a változókat
*/
     write(Nev),write(' vasarlo preferenciaja az alacsony
ar').
```

A három fogyasztóra az `ar_preferencia` szabály eredményei:

```
?- ar_preferencia(bela).
false.

?- ar_preferencia(albert).
albert vasarlo preferenciaja az alacsony ar
true.

?- ar_preferencia(sandor).
false.
```

Miért nem teljesült `bela` fogyasztó esetében, hiszen a `preferencia` argumentuma nála is `barmely_ar`? A szabály harmadik sorában van a megoldás, `bela` esetében nem teljesül a `fogyaszto` tény. Figyeljük meg, hogy a vesszővel elválasztott feltételek a logikai „És"-nek felelnek meg.

A következő programban vállalkozások mutatóira szerkesztettünk szabályokat.

```
agazat(iTSoft, "IT").     /*kisbetűvel jelöljük a
tényeket*/
agazat(aBCcom, "kereskedelem").
agazat(vASMuvek, "ipar").
agazat(tECHDevelopment, "IT").
agazat(mADMAX, "ipar").
agazat(wEBgobe, "IT").

forgalom(iTSoft, 180000).
forgalom(aBCProd, 260000).
forgalom(vASMuvek, 480000).
forgalom(tECHDevelopment, 175000).
forgalom(mADMAX, 586000).
```

```
forgalom(wEBgobe, 398000).

alkalmazottak(iTSoft, 35).
alkalmazottak(aBCProd, 28).
alkalmazottak(vASMuvek, 51).
alkalmazottak(tECHDevelopment, 12).
alkalmazottak(mADMAX, 8).
alkalmazottak(wEBgobe, 7).

profit(iTSoft, 124000).
profit(vASMuvek, -36000).
profit(aBCProd, 25000).
profit(tECHDevelopment,32000).
profit(mADMAX, 147000).
profit(wEBgobe, 210000).
```

Az első két szabály a forgalom és alkalmazottak száma szerint összehasonlít vállalatokat:

```
tobb_forgalom(CegA, CegB) :-     /*kisbetűvel jelöljük a
szabály nevét*/
     forgalom(CegA, ForgA),     /*nagybetűvel jelöljük a
változókat a szabályon belül*/
     forgalom(CegB, ForgB),
     ForgA > ForgB.            /* a feltétel, a szabály
második felében található */
```

Eredmény:

```
?- tobb_forgalom(iTSoft, mADMAX).
false.
?- tobb_forgalom(tECHDevelopment, mADMAX).
false.
?- tobb_forgalom(vASMuvek,aBCProd).
true.
```

A `get_agazat` szabályban a `GetAgazat` változó értékét a billentyűzetről olvassuk be, a `forall` utasítással végighaladunk a `agazat` tényeken, ahol azt teszteljük, hogy a tényekben rögzített, `Agazat`-hoz rendelt változó és a `GetAgazat` azonosak-e.

```
get_agazat(GetAgazat) :-
```

```
    forall(agazat(VallakNev, Agazat),
      (Agazat == GetAgazat -> format('Vállalat neve: ~w
Ágazat: ~w ~n',
      [VallakNev, Agazat]);
      write(""))).
```

A beolvasás és futtatás eredménye az `"IT"` ágazat elemzésére három esetben teljesült:

```
?- get_agazat("IT").
Vállalat neve: iTSoft , Ágazat: IT
Vállalat neve: tECHDevelopment , Ágazat: IT
Vállalat neve: wEBgobe , Ágazat: IT
true.
```

A forgalomelemzési szabályban, a `GetForgalom`-ban olvassuk be a billentyűzetről a változó értékét, a `forall` utasítással végighaladunk a `forgalom` tényeken, leteszteljük a beolvasott értéknél magasabb eseteket, majd kiírjuk az eredményt. Öt vállalkozást talált meg a program, amelyek nevét a `VallakNev`, illetve forgalmukat a `Forgalom` változók használatával listáztuk ki:

```
min_forgalom(GetForgalom) :-
    write(GetForgalom), write("  - nal magasabb
forgalommal rendelkezo vallalatok: " ), nl,
      forall(forgalom(VallakNev, Forgalom),
      (Forgalom > GetForgalom -> format('Vallalat neve: ~w
forgalma: ~w ~n', [VallakNev, Forgalom]);
      write(""))).
```

Eredmény:

```
?- min_forgalom(200000).
200000  - nal magasabb forgalommal rendelkezo vallalatok:
Vallalat neve: aBCProd   forgalma: 260000
Vallalat neve: vASMuvek   forgalma: 480000
Vallalat neve: mADMAX   forgalma: 586000
Vallalat neve: wEBgobe   forgalma: 398000
true.
```

A `kis_vallalkozasok` szabályban a `forall` utasítással végighaladunk a tényeken, leteszteljük, hogy kisvállalkozások-e, vagyis 10 személynél kevesebb alkalmazottal működnek-e, majd a `write` utasítással kiírjuk az eredményt. Két

változót használunk: a `VallakNev`, illetve `Alkalmazottak` nevűeket, a listázás során. Ahogy látható, két ilyen vállalkozás van.

```
kis_vallalkozasok :-
    write(" 10 - nel kevesebb alkalmazottal rendelkezo
vallalatok: " ), nl,
      forall(alkalmazottak(VallakNev, Alkalmazottak),
      (Alkalmazottak < 10  -> format('Vallalat neve: ~w
Alkalmazottak: ~w ~n', [VallakNev, Alkalmazottak]);
      write(""))).
```

Eredmény:

```
?- kis_vallalkozasok .
 10 - nel kevesebb alkalmazottal rendelkezo vallalatok:
Vallalat neve: mADMAX   Alkalmazottak: 8
Vallalat neve: wEBgobe   Alkalmazottak: 7
true.
```

Hasonló logikával a profit alapján a veszteséges vállalatok tesztelésénél, a `profit` tények között, megtaláljuk a negatív profittal rendelkezőket, majd a `VallakNev` , illetve `Alkalmazottak` nevű változók használatával, a `write` utasítással kiírjuk az eredményt.

```
veszteseges_vallalkozasok :-
    write("Veszteseges vallalatok: " ), nl,
      forall(profit(VallakNev, Profit),
      (Profit < 0  -> format('Vallalat neve: ~w   Profit:
~w ~n', [VallakNev, Profit]);
      write(""))).
```

Eredmény:

```
?- veszteseges_vallalkozasok.
Veszteseges vallalatok:
Vallalat neve: vASMuvek   Profit: -36000
true.
```

A következő példában a tények laptopok jellemzőit tartalmazzák, a szabályok pedig a megadott feltételek alapján következtetnek a legmegfelelőbb termékekre.

```
memoria(laptop1,16).
memoria(laptop2,16).
```

```
memoria(laptop3,16).
memoria(laptop4,64).
memoria(laptop5,16).
memoria(laptop6,64).
memoria(laptop7,64).

tipus(laptop1,home).
tipus(laptop2,home).
tipus(laptop3,home).
tipus(laptop4,business).
tipus(laptop5,business).
tipus(laptop6,gaming).
tipus(laptop7,ultraport).

brand(laptop1,lenovo).
brand(laptop2,hp).
brand(laptop3,hp).
brand(laptop4,asus).
brand(laptop5,acer).
brand(laptop6,dell).
brand(laptop7,apple).

kepernyo(laptop1,15.6).
kepernyo(laptop2,14.9).
kepernyo(laptop3,14.9).
kepernyo(laptop4,15.6).
kepernyo(laptop5,14.9).
kepernyo(laptop6,16).
kepernyo(laptop7,15.6).

ar(laptop1,2500).
ar(laptop2,2450).
ar(laptop3,1999).
ar(laptop4,3200).
ar(laptop5,3199).
ar(laptop6,5499).
ar(laptop7,5999).
```

A következő szabályban a billentyűzetről beolvasott és a `GetMemoria` változóhoz rendelt értékkel egyenlő vagy nagyobb memóriájú laptopokat keressük meg.

```
min_memoria(GetMemoria) :-
    write(GetMemoria),write(" minimum memoriaval
rendelkezo laptopok: " ), nl,
      forall(memoria(LaptopNev, Memoria),
      (GetMemoria =< Memoria -> format('Laptop neve: ~w,
memoria: ~w ~n', [LaptopNev, Memoria]);
```

```
      write(""))).
```

Eredmény, 64 MB-re:

```
?- min_memoria(64).
64 minimum memoriaval rendelkezo laptopok:
Laptop neve: laptop4,  memoria: 64
Laptop neve: laptop6,  memoria: 64
Laptop neve: laptop7,  memoria: 64
true.
```

Nézzünk olyan szabályt, amely a típus és brand szerinti laptopokat listázza ki:

```
tipus_brand(GetTipus,GetBrand) :-
    write(GetTipus),write(" tipusu laptopok: " ), nl,
      forall(tipus(LaptopNev1, Tipus),
      (GetTipus == Tipus -> format('Laptop neve: ~w,
tipus: ~w ~n', [LaptopNev1, Tipus]);
      write(""))),
    write(GetBrand),write(" gyartoju laptopok: " ), nl,
      forall(brand(LaptopNev2, Brand),
      (GetBrand == Brand -> format('Laptop neve: ~w,
brand: ~w ~n', [LaptopNev2, Brand]);
      write(""))).
```

Eredménye a `home` típus és hp brandre:

```
?- tipus_brand(home,hp).
home tipusu laptopok:
Laptop neve: laptop1,  tipus: home
Laptop neve: laptop2,  tipus: home
Laptop neve: laptop3,  tipus: home
hp gyartoju laptopok:
Laptop neve: laptop2,  brand: hp
Laptop neve: laptop3,  brand: hp
true.
```

Az alábbi `ar_min_max` szabály a beolvasott minimum- és maximumértékek közötti árnak megfelelőeket listázza ki.

```
ar_min_max(GetArMin,GetArMax) :-
    write(GetArMin),write(" minimum ar es "
),write(GetArMax),write("maximum ar kozotti laptopok: " ),
nl,
      forall(ar(LaptopNev, Ar),
```

```
      (GetArMin < Ar, Ar < GetArMax  -> format('Laptop
neve: ~w   Ar: ~w ~n', [LaptopNev, Ar]);
      write(""))).
```

Például 3000 és 6000 ár közötti laptopokra az eredmény így néz ki:

```
?- ar_min_max(3000,6000).
3000 minimum ar es 6000maximum ar kozotti laptopok:
Laptop neve: laptop4   Ar: 3200
Laptop neve: laptop5   Ar: 3199
Laptop neve: laptop6   Ar: 5499
Laptop neve: laptop7   Ar: 5999
true.
```

Prologban az aritmetikai műveleteket integer vagy float típusú számokkal lehet elvégezni. Nézzük elsőként a `between(`*`+AlsóH, +FelsőH, ?Érték`*`)` utasítást, amely az *`?Érték`*-ként beolvasott utasítás esetében azt ellenőrzi, hogy az első és felső határok között van-e. A `kkv_k` szabályban az alkalmazottak száma és a forgalom (millió euróban) értékei alapján történik a besorolás[2], úgy, hogy egyszerre mindkét feltétel kell teljesüljön egy kategóriánál.

```
kkv_k(Alkalmazottak,Forgalom) :-
      between(0,10,Alkalmazottak),between(0,2,Forgalom),wr
ite('mikro vallalkozas');
      between(10,50,Alkalmazottak),between(2,10,Forgalom),
write('kis vallalkozas');
      between(50,250,Alkalmazottak),between(10,50,Forgalom
),write('kozepes vallalkozas').

?- kkv_k(15,7).
kis vallalkozas
true .
?- kkv_k(2,1).
mikro vallalkozas
true .
?- kkv_k(75,38).
kozepes vallalkozas
true.
```

Prologban a `max_list` és `min_list` függvények keresik meg a listák maximumát és minimumát. Amikor meghívjuk, specifikálnunk kell a lista elemeinek típusát, ahogyan `num` (numerikus) az alábbi példában.

[2] A 2014. évi 346-os törvény, illetve az Európai Bizottság ajánlása alapján történik a besorolás. Forrás: http://publications.europa.eu/resource/cellar/1bd0c013-0ba3-4549-b879-0ed797389fa1.0014.02/DOC_2

```
lista_maximum(num,List,Max) :-
      max_list(List,Max).

lista_minimum(num,List,Min) :-
      min_list(List,Min).
```

Eredménye:

```
?- lista_maximum(num,[147,1526,1848,1705,2000,2022],Max).
Max = 2022.
?- lista_minimum(num,[147,1526,1848,1705,2000,2022],Min).
Min = 147.
```

Más megoldás a lista minimum- és maximumértékeire, ha az alábbi, `lista_minimum`[3] és `lista_maximum` szabályokkal számoljuk ki. A lista törzsét képező B és V abban az esetben cserél helyet, ha a B kisebb, így a lista fej – törzs felbontásával, sorrendben minden elemen végighaladva, összehasonlítja a minimumértékkel. A `lista_maximum` szabályban, mint a nagyobb érték feltételének teljesülése esetében, cserélnek helyet a B és V változók.

```
lista_minimum([Lista_min],Lista_min).
lista_minimum([H,B|V],Min) :-
     H =< B,
     lista_minimum([H|V],Min).
lista_minimum([H,B|V],Min) :-
     H > B,
     lista_minimum([B|V],Min).

lista_maximum([Lista_max],Lista_max).
lista_maximum([H,B|V],Max) :-
     H  >= B,
     lista_maximum([H|V],Max).
lista_maximum([H,B|V],Max) :-
     H < B,
     lista_maximum([B|V],Max).
```

Például, a `[147, 1526, 1848, 1705, 2000, 2012]` lista minimuma:

```
?- lista_minimum([147, 1526, 1848, 1705, 2000, 2012],
Minimum).
Minimum = 147
```

Illetve a `[147, 1526, 1848, 1705, 2000, 2022]` lista maximuma:

[3] Forrás: a http://andrewprice.me.uk/geek/prolog2/ alapján.

```
lista_maximum([147, 1526, 1848, 1705, 2000, 2022],
Maximum).
Maximum = 2022
```

2.5. Logikai szabályok a Prologban

A logikai ítéletkalkulus, vagyis a kijelentések és a köztük levő kapcsolatok igazságtartalmának a vizsgálata a Prologban a következőképpen valósítható meg. Ha a világról kijelentéseket szeretnénk megfogalmazni, terminusokat (term – angolul) használunk, amelyek lehetnek: állandó szimbólumok (más néven atomok), változók, illetve összetett kifejezések. Az összetett kifejezésekben az objektum és argumentum elemei, ahogy az előzőekben már tárgyaltuk, valamilyen kapcsolatot fejez ki. Például `vallalat(ipar, nev(N), hely(L))` jelenthet egy iparban tevékenykedő vállalatot, melynek a nevét és a működési helyét az N és L változókkal tudjuk majd megadni.

Fontos tehát, hogy az objektumok közötti kapcsolatokat megfelelően ki tudjuk fejezni. Ezt elsősorban a kijelentésekkel tesszük meg, vagyis az atomi tényekkel, melyek argumentumokat tartalmaznak. Például: `tulajdonos(W, vallalat(W))`, `preferencia(nok, termek_tipus_2)`.

Nézzük most a logikai kapcsolatok típusait, amelyek az ítéletkalkulus során felhasználhatók! (8. táblázat)

8. táblázat. A logikai kapcsolatok típusai Prolog-példákkal

Kapcsolat	*Ítélet-kalkulus*	*Prolog*	*Jelentése*
Negáció	¬α	~α	nem α
Konjunkció	α ∧ β	α , β	α és β
Diszjunkció	α ∨ β	α ; β	α vagy β
Implikáció	α => β	α :- β α -> β	α implikálja β-t
Ekvivalencia	α ≡ β	α <-> β	α ekvikalens β-val

Az alábbi első példában egy személy fogyasztó, amennyiben vásárló és személy.

```
szemely(janos).
szemely(monika).
vasarlo(janos).
```

```
fogyaszto(Sz) :-
      szemely(Sz),
      vasarlo(Sz).
```

A két feltétel csak `janos` esetében teljesül:

```
?- fogyaszto(janos).
true.
?- fogyaszto(monika).
false.
```

A `vasarlas` szabályban a logikai „Vagy"-gyal vannak összekötve a feltételek:

```
termek_preferencia(janos,laptop).
termek_preferencia(monika,camera).
termek_preferencia(istvan,tablet).

vasarlas(V) :-
      termek_preferencia(V,laptop);
      termek_preferencia(V,camera).
```

A két feltétel közül az egyik csak az első két tény esetében teljesül:

```
?- vasarlas(janos).
true .
?- vasarlas(monika).
true.
?- vasarlas(istvan).
false.
```

Az implikációra legfontosabb példaként a szabály működését kell tekintenünk, ahogy külön fejezetben tárgyaltuk. Ezenkívül a negáció mint kudarc példában a `->` típusú implikációt használjuk az alábbi, `nem` szabályban[4], az előbbi `vasarlas` harmadik esetére, melynek eredeti válasza `false` volt.

```
nem(Valasz) :-
      (call(Valasz) -> fail ; true).
```

Eredménye:

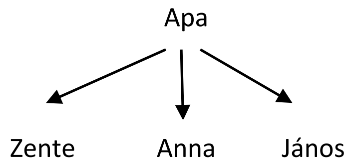
```
?- nem(vasarlas(istvan)).
true.
```

[4] Forrás: a https://www.cpp.edu/~jrfisher/www/prolog_tutorial alapján.

2.6. Adatstruktúrák a Prologban

2.6.1. Fák

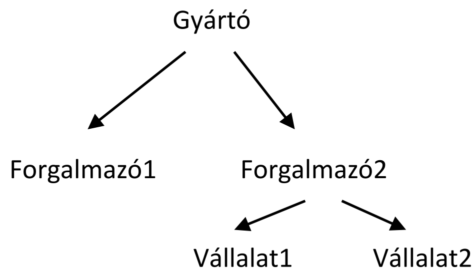
Az első struktúra, amit Prologban tárgyalunk, a fa-adatstruktúra, amelynek szerkezete csomópontokkal és ágakkal írható le. Minden ág egy új leágazásban folytatódhat, tehát ezek egymásba illeszkedve ismétlődnek. A fastruktúra legegyszerűbb ábrázolási módja a diagram, amelyben a kapcsolat típusát is feltüntetjük. Például szülők, üzleti partnerek, de matematikai műveletek is lehetnek:

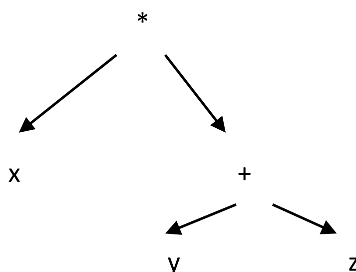


A fenti példákban Apának három gyereke van: Zente, Anna és János. Ez a szerkezet hármas fastruktúraként írható le:

```
fa(Elso_ag, Masodik_ag, Harmadik_ag).
```

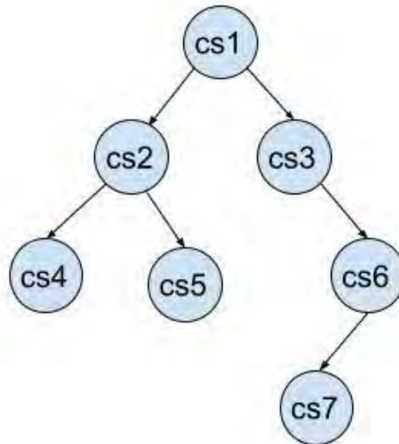
Ennél sokkal egyszerűbb a bináris fastruktúra, amelyben csak két ág van, például az első ábrán két forgalmazó vagy két vállalat, a másodikon két művelet.





Tehát a bináris fastruktúrákban, a Gyártónak két forgalmazón keresztül jutnak el a termékei a piacra, a Forgalmazó2 beszállítója a Vállalat1-nek és Vállalat2-nek, illetve a harmadiknál az x*(y+z) művelet kapcsolati rendszere van fastruktúrában ábrázolva. A bináris fa leírását úgyis tekinthetjük, mint „lista a listában", hogy üres listával jelöljük, ha nincs a fának elágazása.

A következő listának hét csomópontja van: 1 csomópont a 0. mélységben (a gyökér csomópont), 2 csomópont 1-es mélységben, 3 csomópont 2 mélységben és 1 csomópont hármas mélységben.



Szerkezete Prologban:

```
fa(cs1,fa(cs2,fa(cs4,nil,nil),fa(cs5,nil,nil)),fa(cs3,nil,
fa(cs6,fa(cs7,nil,nil),nil))).
```

Első lépésben leteszteljük, hogy bináris faszerkezetű-e, vagyis bal és jobb oldali ágakból áll-e:

```
bfa_teszt(nil).
bfa_teszt(fa(_,Bal,Jobb)) :-
      bfa_teszt(Bal),
      bfa_teszt(Jobb).
```

```
?-
bfa_teszt(fa(cs1,fa(cs2,fa(cs4,nil,nil),fa(cs5,nil,nil)),f
a(cs3,nil,fa(cs6,fa(cs7,nil,nil),nil)))).
true.
```

A `bfa_teszt` szabály hasonlóan helyes szerkezetűnek találja az alábbi egy csomópontos bináris fát:

```
?- bfa_teszt(fa(cs0,nil,nil)).
true.
```

A második program a bináris fa egyik elemének a tesztelésére vonatkozik. Megnézzük, hogy a `cs1, cs3,` majd `cs0` elemei-e a fának:

```
binarisfa_resze(fa(Csomopont,_,_),Csomopont).
binarisfa_resze(fa(_Csomopont,Bal,Jobb),N) :-
      binarisfa_resze(Bal,N);
    binarisfa_resze(Jobb,N).
```

Eredmény:

```
?-
binarisfa_resze(fa(cs1,fa(cs2,fa(cs4,nil,nil),fa(cs5,nil,n
il)),fa(cs3,nil,fa(cs6,fa(cs7,nil,nil),nil))),cs1).
true .
?-
binarisfa_resze(fa(cs1,fa(cs2,fa(cs4,nil,nil),fa(cs5,nil,n
il)),fa(cs3,nil,fa(cs6,fa(cs7,nil,nil),nil))),cs3).
true .
?-
binarisfa_resze(fa(cs1,fa(cs2,fa(cs4,nil,nil),fa(cs5,nil,n
il)),fa(cs3,nil,fa(cs6,fa(cs7,nil,nil),nil))),cs0).
false.
```

Harmadik programmal egy listába tesszük át a bináris fa elemeit.

```
binfa_listaba(T, Bal) :-
    binfa_listaba(T, [], Bal).

binfa_listaba(nil, Bal, Bal).
binfa_listaba(fa(Jobb, Balb, Jobb1), Bal0, [Jobb|Bal2]) :-
    binfa_listaba(Jobb1, Bal0, Bal1),
    binfa_listaba(Balb, Bal1, Bal2).
```

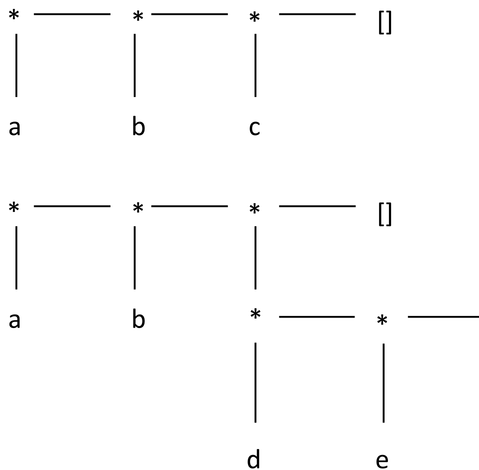
Eredménye:

```
?-
binfa_listaba(fa(cs1,fa(cs2,fa(cs4,nil,nil),fa(cs5,nil,nil
)),fa(cs3,nil,fa(cs6,fa(cs7,nil,nil),nil))),Csomopontok).
Csomopontok = [cs1, cs2, cs4, cs5, cs3, cs6, cs7].
```

2.6.2. Listák

A listák szerkezete a fákhoz képest elsősorban lineáris. Tehát a csomópontok egy vonal mentén vannak összekötve, és az allisták innen kiinduló elágazások, mint az alábbi második példában látható. (5. ábra)

5. ábra. Listaszerkezetek Prologban



A fenti két lista adatszerkezete a következőképpen néz ki, ahol a második listában [d, e] az allista:

```
[a, b, c]
[a, b, [d, e]]
```

A következő módon jelöljük az egy-, két- és háromelemű listákat:

```
[x].
[x,y].
[x,y,z].
```

Üres lista jelölése, amelynek nincs sem feje, sem teste:

```
[]
```

Az általánosan használt, lista adatszerkezet a Prologban: [H|B], ahol a H jelöli a lista fejét (Head), B a lista testét (Body). Nézzük erre a p és lista nevű listák szerkezetét:

```
p([m,n,p]).
lista([gitar]).
?- p([X|Y]).
X = m,
Y = [n,p].
lista([gitar]).
?- lista([X|Y]).
X = gitar,
Y = [].
```

A listákra vonatkozó rekurzív keresés azon alapul, hogy ellenőrzi, hogy egy elem megtalálható-e a lista fejében, illetve ha nem található meg, felbontja a lista testét további fej-test szerkezetekre, amelyeket sorra megvizsgál. Az alábbi szabállyal tudjuk ellenőrizni, hogy egy változó a lista elemét képezi-e[5]:

```
member(X,[X|_]).
member(X,[_|L]):-
      member(X,L).
```

Az `[m,n,p,q]` négyelemű lista esetében így működik:

```
?- member(n, [m,n,p,q]).
true
?- member(o, [m,n,p,q]).
false
```

Azt, hogy egy adatszerkezet lista-e, a következő szabállyal tudjuk ellenőrizni[6]:

```
islist([A|B]) :- islist(B).
islist([]).
```

Ebben a lista elemeinek vizsgálatánál, a legutolsó elemig felbontva, az üres lista ([]) kell legyen, hogy a szabály feltétele teljesüljön.

Listák összefüzése során az L1 első és az L2 második lista egyesítésénének eredményét az `egyesit` szabály az E listához rendeli, úgy, hogy felhasználja a H lista fejét az egyesítés során[7]:

```
egyesit([],L2,L2).
```

---

[5] Forrás: Clocksin és Mellish, 2003:54

[6] Forrás: uo.

[7] Aszalós (2014) alapján. Forrás: https://gyires.inf.unideb.hu/GyBITT/18/ch03s02.html

```
egyesit([H|L1],L2,[H|E]):-
  egyesit(L1,L2,E).

?- egyesit([a,b,c],[y,z,w],E).
E = [a, b, c, y, z, w].
```

A listák sorrendjének megfordítása, a `fordit` szabállyal, hasonló logika alapján történik: felhasználja a H lista fejét és az egyesítésre használt `egyesit` szabályt[8].

```
fordit([],[]).
fordit([H|L1],E):-
  fordit(L1,L2),
  egyesit(L2,[H],E).

?- fordit([a, b, c, y, z, w],Eredmeny).
Eredmeny = [w, z, y, c, b, a].
```

A listák utolsó elemét az alábbi `utolso_elem` szabály úgy találja meg, hogy a lista testét jelölő B-t felbontja, majd megfordítja egészen a lista végéig haladva[9]:

```
utolso_elem([X|B],E):-
      utolso_elem(B,X,E).
utolso_elem([],E,E).
utolso_elem([X|B],_,E):-
      utolso_elem(B,X,E).

?- utolso_elem([a, b, c, y, z, w],U).
U = w.
```

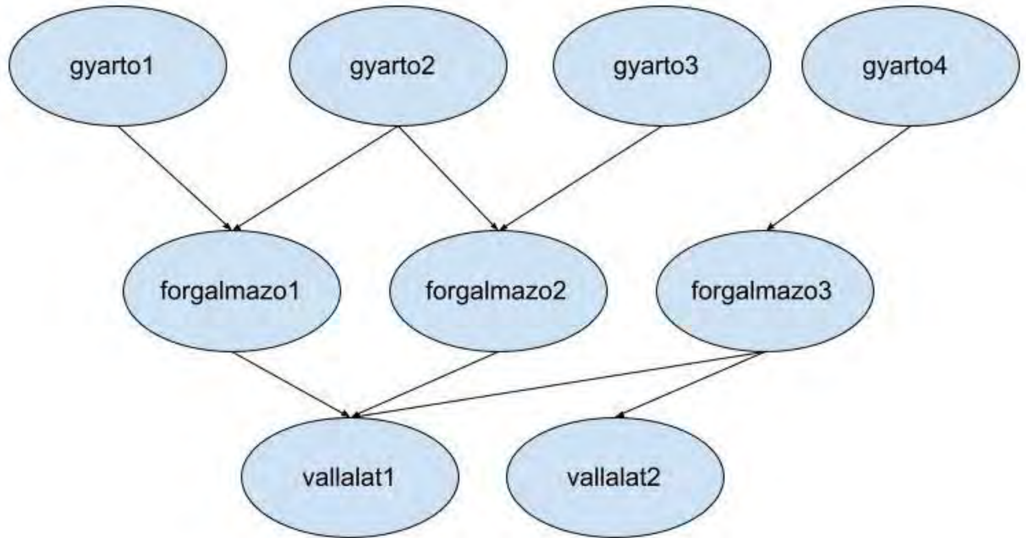
2.7. Backtracking algoritmus Prologban

A backtracking algoritmus olyan folyamatnak tekinthető, amely során a program visszalép egy korábbi célhoz, és ezt megpróbálja újra teljesíteni, vagyis egy másik módot találni a kielégítésére. Ezeket általában családi kapcsolatok feltérképezésére, felmenők visszakeresésére (Bramer, 2005:39), illetve listákon való keresésre (Blackburn és társai, 2006: 53, 76) használják a Prologban.

Az alábbi példában két vállalkozás beszállítói láncát szerkesztettük meg, amelyeket a `beszallit` és `disztributor` tényekkel írtunk le.

[8] Forrás: uo.

[9] Forrás: uo.

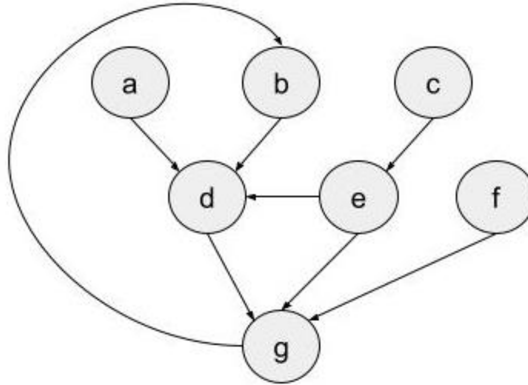



```
beszallit(gyarto1,forgalmazo1).
beszallit(gyarto2,forgalmazo1).
beszallit(gyarto3,forgalmazo2).
beszallit(gyarto4,forgalmazo3).
disztributor(forgalmazo1,vallalat1).
disztributor(forgalmazo2,vallalat1).
disztributor(forgalmazo3,vallalat1).
disztributor(forgalmazo3,vallalat2).
```

A `vallalat1` üzleti partnereit az alábbi utasítással tudjuk felsorolni:

```
?-disztributor(Partner,vallalat1),write('A forgalmazok:
'), write(Partner), nl.
```

Természetesen a vállalkozások üzleti kapcsolati hálója számtalan formát ölthet. A következő példában tíz vállalkozás esetében (melyeket az egyszerűség kedvéért az ábécé kisbetűivel jelöltünk, a-tól g-ig) mutatjuk be a beszállítói kapcsolatok szerkezetét. A gráfok „éleinek” megfelelő kapcsolatot tehát beszállító kapcsolatként írjuk le, majd a keresést a kapcsolatokon keresztül szerkesszük meg.

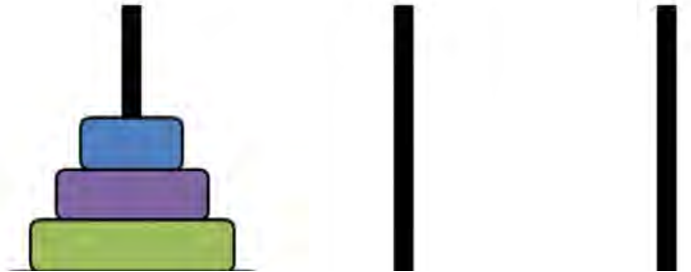



```
beszallit(a,d).
beszallit(b,d).
beszallit(e,d).
beszallit(c,e).
beszallit(d,g).
beszallit(e,g).
beszallit(f,g).
beszallit(g,b).


kapcsolat(X,Y) :-
beszallit(X,Y).

besz_kapcsolat(X,Y,_) :-
beszallit(X,Y).
besz_kapcsolat(X,Z,L) :-
beszallit(X,Y), not(member(Y,L)),
besz_kapcsolat(Y,Z,[X|L]).

?- kapcsolat(a,d).
true
?- kapcsolat(e,g).
true
kapcsolat(a,e).
false.

?- besz_kapcsolat(a,g,[a]).
true
?- besz_kapcsolat(a,f,[a]).
false
?- besz_kapcsolat(c,g,[c]).
true
```

2.8. Hanoi tornyai

A Hanoi tornyai játék háromkorongos változatában (6. ábra) ahogy az ábrán látható, a bal oldali oszlopról kell átvinnünk a három korongot, úgy, hogy közben a kisebb csak a nagyobbra tehető, illetve egyszerre csak egy korong mozdítható.

6. ábra. Hanoi tornyai



Prologban ezt a feladatot a `mozgat`[10] nevű szabállyal oldhatjuk meg. Itt a szabály lényegi eleme a megfelelően elvégzett csere, amelyet lépésenként kiírunk.

```
mozgat(1,X,Y,_) :-
    write('Athelyezi a korongot '),
    write(X), write('-rol, '),
    write(Y), write('-ra. '), nl.

mozgat(N,X,Y,Z) :-
    N>1, M is N-1,
    mozgat(M,X,Z,Y),
    mozgat(1,X,Y,_),
    mozgat(M,Z,Y,X).
```

A Hanoi tornyaira a `mozgat` szabály futtatásának eredménye:

```
?- mozgat(3,bal,jobb,kozep).
Athelyezi a korongot bal-rol, jobb-ra.
Athelyezi a korongot bal-rol, kozep-ra.
Athelyezi a korongot jobb-rol, kozep-ra.
Athelyezi a korongot bal-rol, jobb-ra.
Athelyezi a korongot kozep-rol, bal-ra.
Athelyezi a korongot kozep-rol, jobb-ra.
Athelyezi a korongot bal-rol, jobb-ra.
```

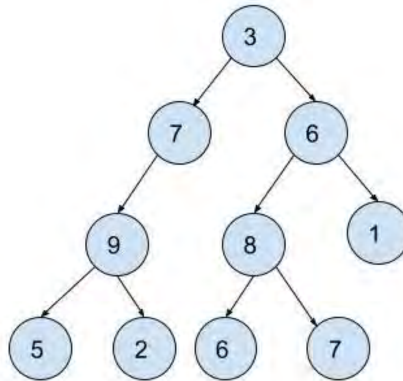
10 Forrás: https://www.cpp.edu/~jrfisher/www/prolog_tutorial

```
True .
```

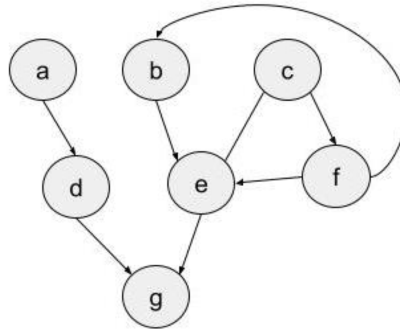
Kérdések

1. Szerkesszen egy három argumentummal rendelkező ténylistát egy termékcsoportra (például: típus, gyártó, ár)!
2. Szerkesszen egy szabályt, amely kiírja az előző alpontban létrehozott termékcsoportra a 150-nél drágább termékeket, egy másik szabályt, amely adott gyártó szerint listázza a termékeket!
3. Szerkesszen egy listát a termékek áraiból, és keresse meg a lista maximumértékét!

4. Szerkessze meg Prologban az alábbi bináris fát, majd tesztelje, hogy megfelel-e a bináris fa feltételének:



4. Szerkessze meg az alábbi gráf kapcsolatait, és tesztelje szabállyal ezeket a kapcsolatokat:


a
b
c
d
e
f
g

3. SZAKÉRTŐI RENDSZEREK A CLIPS PROGRAMOZÁSI NYELVBEN

3.1. Bevezetés

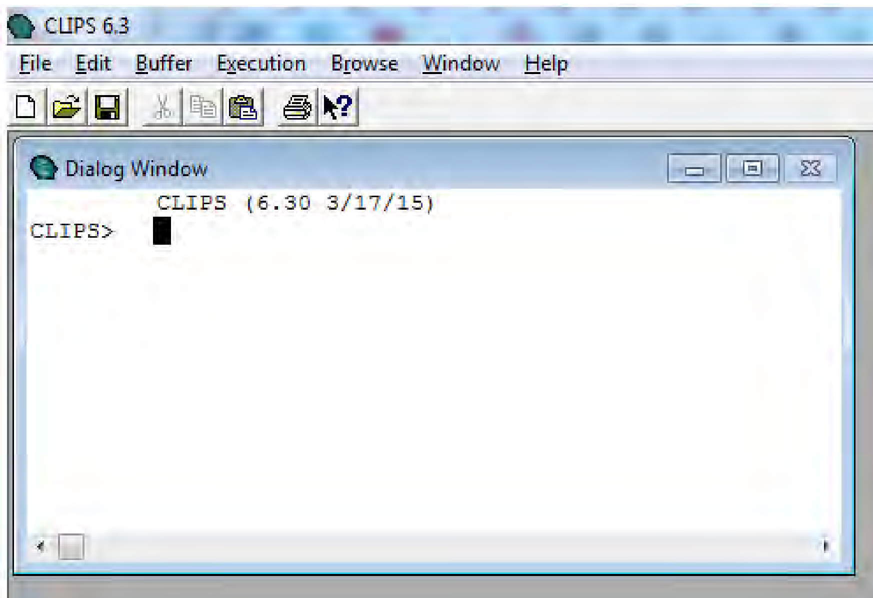
A CLIPS (C Language Integrated Production System) szakertői rendszerek készítésére kifejlesztett programozási nyelv. Eredetét tekintve 1984-ben a NASA's Johnson Space Centerben kezdődött el a fejlesztése. Első kiadása 1986-ban történt, a gyártó: Software Technology Branch (STB), NASA/Lyndon B. Johnson Space Center. Az ekkori időkben a NASA mesterséges intelligenciával foglalkozó központja, tucatnyi szakértői rendszer prototípust fejlesztett, amelyek közül csak néhány érte el a rendszeres felhasználási szintet. Ezen prototípusok rendszerszoftvereszköz alapnyelveként a LISP-et használták, amelynek alacsony elérhetősége volt a hagyományos számítógépek széles körében, magas volt a hardverköltsége, és gyenge integrációval rendelkezett más nyelvekkel. A fejlesztési csoport meglátása, hogy a C programozási nyelvben történő fejlesztés ezeket a problémákat kiküszöböli, idővel igazolást nyert. Így idővel a CLIPS széles körben elfogadottá vált a kormányzati, ipari és akadémiai szervezeteknél.

Forráskódja C nyelven teljes mértékben hozzáférhető, a 6.3 verziótól integrálható a JAVA és a C++ nyelvekkel. A szabályalapú programozás mellett támogatja az objektumorientált programozást is.

3.2. Telepítés, első lépések

A jelen könyvben tárgyalt példák és bemutatott programok a CLIPS 6.3 verziójára készültek. Telepítése a http://www.clipsrules.net/ weboldalról történhet. Elindítása után, az alább látható ablakban, a CLIPS parancssorában (command prompt) fogjuk a programokat betölteni és futtatni (7. ábra). Innen, a parancssorból tudunk elindítani függvényhívásokat, összetett szerkezeteket, lokális vagy globális változókat vagy konstansokat.

7. ábra. A CLIPS 6.3 verziójának parancssora



3.3. Tények a CLIPS-ben

Fontos megjegyezni már az elején, hogy a CLIPS-ben a logikai kijelentések a konjunktív normál forma, KNF (CNF – conjunctive normal form) szerinti atomok, amelyeknek a jelölésére zárójeleket használunk.

Kétfajta tény ismert a CLIPS-ben: a rendezett tények és rendezetlen tények. A *rendezett tények* egy szimbólumból állnak, amelyet egy üres vagy több mezőből álló sorozat követ, egymástól szóközökkel elválasztva, bal oldalon nyitó, jobb oldalon záró zárójellel határolva. A rendezett tény első mezője a „relációt" határozza meg, amely a fennmaradó mezőkre vonatkozik. Például `(alkalmazott janos universal_kft)` azt jelenti, hogy János az Universal Kft. alkalmazottja. Számos más kapcsolatot is hasonlóan megjeleníthetünk:

```
(apja joe jim) % Jim a Joe apja
(vasarlo anna hutoszekreny) % Anna hűtőszekrény-vásárló
(rendszer-jel on) % a rendszer jelzése "on", hasonlóan
lehetne „off" vagy hibajelzés
(termek ar 259) % a termék ára 259
```

A fenti első két példánál két-két mezős rendezett tényeket látunk, a reláció után, a harmadik-negyediknél pedig egy-egy mezős tényeket. A nem rendezett tény mezői a

primitív adattípusok bármelyikébe tartozhatnak (kivéve az első mezőt, amelynek szimbólumnak kell lennie), és a mezők sorrendje nincs korlátozva. Mivel utasításnevek, szimbólumként fenntartottak a következő kifejezéssel: `test, and, or, not, declare, logical, object, exists,` és `forall`, ezért semmilyen rendezett vagy nem rendezett tényben nem használhatók első mezőként (a reláció jelölésére).

A rendezett tények hely szerint kódolják az információkat. Az információk eléréséhez a felhasználónak nem csak azt kell tudnia, hogy egy tényben milyen adatokat tárolnak, hanem azt is, hogy melyik mező tartalmazza az adatokat.

A *nem rendezett* (vagy `deftemplate` szerkezetű) tények lehetőséget adnak a felhasználónak, hogy szabadon kialakítsa egy tény szerkezetét, úgy, hogy a tény minden egyes mezőjéhez nevet rendel hozzá. Így a mezők név szerint elérhetőek, mivel egy sablon (`template`) alapján vannak létrehozva.

A *nem rendezett* tény első mezője a sablon neve, amelyet követhet üres mezőnév (slotok) vagy több mezőnév, amelyek szerint a tények mezői létrehozhatók és elérhetők lesznek majd.

```
(vasarlo (nev "Nagy Janos") (id 14790728))

(vallalkozas (nev "Hovirag RT") (profit 24000)
(alkalmazottak 18))
```

A tények, vagyis logikailag igaz kijelentések létrehozása a CLIPS nyelvben kétféle módon történhet: az `assert` utasítással hozzuk létre az egyedi (atomi) tényeket, illetve a deffacts utasítással a rendezett szerkezetű ténylistákat, több argumentummal. További tényekre vonatkozó műveletek: törlés vagy visszavonás (`retract`), módosítás (`modify`) és másolás (`duplicate`).

Rendezett tények létrehozása:

```
(assert (szemely Albert))
(assert (szemely Timea))
(assert (vasarlas igen))
(assert (auto zold))
```

Alapértelmezetten egy tény egyszer hozható létre, és nem írható felül. Ha másodszorra deklaráljuk, `FALSE`-t ír ki:

```
CLIPS> (assert (szemely Albert))
<Fact-1>
CLIPS> (assert (szemely Albert))
FALSE
CLIPS> (facts)
f-0     (initial-fact)
f-1     (szemely Albert)
For a total of 2 facts.
CLIPS>
```

A fenti példában `(facts)` utasítással írattuk ki a tényeket. Az f-0-val jelölt `(initial-fact)`, minden indításnál betöltött „alap-tény”, amelynek használatát a továbbiakban részletesen tárgyaljuk. A tények létrejöttének és a program futtatásának, vagyis minden lépésnek a kiírását a `(watch all)` utasítással tudjuk elvégezni. A tények duplázására a `set-fact-duplication` utasítással van lehetőség, ahogy az alábbi példában láthatjuk, amikor ennek feltétele igaz (`TRUE`), többször létrehozhatjuk ugyanazt a tényt, viszont a hamis (`FALSE`) feltételre visszatér az alapértelmezett módban:

```
CLIPS> (set-fact-duplication TRUE)
TRUE
CLIPS> (watch all)
CLIPS> (assert (elso))
==> f-1     (elso)
<Fact-1>
CLIPS> (assert (elso))
==> f-2     (elso)
<Fact-2>
CLIPS> (facts)
f-0     (initial-fact)
f-1     (elso)
f-2     (elso)
For a total of 3 facts.
CLIPS> (clear)
CLIPS> (set-fact-duplication FALSE)
TRUE
CLIPS> (assert (elso))
==> f-1     (elso)
<Fact-1>
CLIPS> (assert (elso))
FALSE
CLIPS> (facts)
f-0     (initial-fact)
f-1     (elso)
For a total of 2 facts.
```

```
CLIPS>
```

Nem rendezett (vagy `deftemplate` szerkezetű) tények létrehozása:

```
(deftemplate vallalkozas (slot nev) (slot profit) (slot
alkalmazottak))
(deffacts vallalkozas-tenyek
(vallalkozas (nev "Hovirag RT") (profit 24000)
(alkalmazottak 18))
(vallalkozas (nev "Fenyo Kft") (profit 56000)
(alkalmazottak 7))
)
```

A nem rendezett (vagyis `deftemplate` szerkezetű) tények konstrukciója lehetővé teszi az előzetes vagy kezdeti ismeretek halmazának megadását tények gyűjteményeként, amelyre a továbbiakban szabályokat szerkeszthetünk, bővíthetjük vagy keresést végezhetünk ezeken.
Az „a priori" vagy kezdeti tudáshalmaz, amelyet a felhasználó az osztályokkal együtt létrehoz, a CLIPS környezet alaphelyzetbe állításával (a `reset` paranccsal), a CLIPS tudásbázis `definstances` konstrukciójában teljes mértékben hozzáadódik a tény-listához.

3.4. Objektumok a CLIPS-ben

Az objektumok olyan meghatározott elemek, melyek lehetnek szimbólumok, karakterláncok, lebegőpontos vagy egész számok, többmezős tények vagy felhasználó által meghatározott osztályok. Az osztály, ebben az értelemben, a példányait képező objektumok közös tulajdonságainak és viselkedésének sablonja.

9. táblázat. Osztályok a CLIPS-ben

Osztály	**Objektum**
szimbólum	`vip-vasarlo`
karakterlánc (string)	`"Nagy Pál, törzsvásárló"`
lebegőpontos szám (float)	`14.07`
egész szám (integer)	`28`
többmezős tény	`(vasarlo laptop 4500)`
vásárlócsoport – felhasználó által meghatározott osztály	`[vasarlo csoport]`

Az objektumok két fontos kategóriába sorolhatók: primitív típusúak és felhasználó által definiált osztályokhoz tartozók. Ez a kéttípusú objektum különbözik, ahogyan hivatkoznak rájuk, ahogy létrehozzák őket és törölik, valamint tulajdonságaik megadásának módjában.

A primitív típusú objektumokra egyszerűen az érték megadásával hivatkozunk, létrejönnek és szükség esetén törlődnek. A primitív típusú objektumoknak nincs nevük vagy helyük, az osztályaikat a CLIPS előre definiálja.

A felhasználó által definiált osztályra név vagy cím hivatkozik, és kifejezetten üzenetek és speciális funkciók útján jönnek létre és törlődnek. Egy felhasználó által definiált osztály példányának tulajdonságait mezőneve (`slotok`) halmaza fejezi ki, amelyet az objektum az osztályából nyer. A korábban meghatározottak szerint a helyek neve egymezős vagy többmezős érték lehet.

3.5. A CLIPS-változók típusai: lokális vagy globális, egymezős vagy többmezős

A `defglobal` utasítással olyan változókat hozhatunk létre, amelyek globális hatókörűek a CLIPS-környezetben, ezek a *globális változók*. Vagyis egy globális változó bárhol elérhető a CLIPS-környezetben, és más konstrukcióktól függetlenül megtartja értékét. Például:

```
(defglobal
?*s* = 28
?*z* = 14
?*szoveg* = "Az ÁFA 19%"
?*afa1* = 0.19
?*afa2* = 0.5
?*elso* = "19%"
?*masodik* = "5%"
)
```

A *lokális változókat* olyan konstrukciókkal hozzuk létre, mint például defrule és deffunction, és csak a konstrukción belül meghatározottak, itt hivatkozhatunk rájuk, de nincs jelentésük a konstrukción kívül.

A továbbiakban a változók adattípusait tárgyaljuk. A CLIPS-ben használt változók: szimbólum, karakterlánc (`string`), lebegőpontos szám (`float`) és egész szám (`integer`). A szimbólum változót, amely bármilyen betűvel kezdődő, betűkből álló vagy betű-szám kombinációból lehet szerkesztve, általában a tényeken belül használjuk. Például:

```
CLIPS> (assert (albert vip-client-251))
<Fact-1>
CLIPS> (facts)
f-0     (initial-fact)
f-1     (albert vip-client-251)
For a total of 2 facts.
CLIPS>
```

A `symbolp` függvény a `TRUE` szimbólumot adja vissza, ha argumentuma szimbólum, ellenkező esetben a `FALSE` szimbólumot adja vissza. Az ötödik példában a „`b`” nem szimbólum, hanem sztring típusú változó, amelyet az alábbiakban tárgyalunk majd.

```
CLIPS> (symbolp 7)
FALSE
CLIPS> (symbolp 5.2)
FALSE
CLIPS> (symbolp a)
TRUE
CLIPS> (symbolp a147)
TRUE
CLIPS> (symbolp "b")
FALSE
```

A szimbólum változók egyesítése a `sym-cat` utasítással történik:

```
CLIPS> (sym-cat a 1)
a1
CLIPS> (sym-cat 1 14.7)
114.7
CLIPS> (sym-cat "a" b)
ab
CLIPS> (sym-cat "a" "b")
ab
```

A CLIPS-ben lebegőpontos szám (`float`) és egész szám (`integer`) típusokat tudunk használni. Nézzük meg az alábbi példákat három termék árának meghatározására:

```
CLIPS> (assert (monitor-price 1250))
<Fact-1>
CLIPS> (assert (keyboard-price 60))
<Fact-2>
CLIPS> (assert (laptop-price 3250.50))
<Fact-3>
CLIPS> (facts)
f-0     (initial-fact)
f-1     (monitor-price 1250)
f-2     (keyboard-price 60)
f-3     (laptop-price 3250.5)
For a total of 4 facts.
CLIPS>
```

Az `integerp` függvény a `TRUE` szimbólumot adja vissza, ha argumentuma egész szám, ellenkező esetben a FALSE szimbólumot adja vissza.

```
CLIPS> (integerp 7)
TRUE
CLIPS> (integerp 5.2)
FALSE
CLIPS> (integerp a)
FALSE
CLIPS>
```

A `numberp` függvény a `TRUE` szimbólumot adja vissza, ha argumentuma `float` (lebegőpontos szám) vagy `integer` (egész szám), ellenkező esetben a `FALSE` szimbólumot adja vissza.

```
CLIPS> (numberp 7)
TRUE
CLIPS> (numberp 5.2)
TRUE
CLIPS> (numberp a)
FALSE
CLIPS>
```

A floatp függvény a `TRUE` szimbólumot adja vissza, ha argumentuma `float`, ellenkező esetben a `FALSE` szimbólumot adja vissza.

```
CLIPS> (floatp 7)
FALSE
CLIPS> (floatp 5.2)
TRUE
CLIPS> (floatp a)
FALSE
CLIPS>
```

A szöveg (`string`) változó létrehozására a `" "` utasítást használjuk. Például:

```
CLIPS> (assert (elso-szoveg "Ez az első szöveg"))
<Fact-1>
CLIPS> (assert (masodik-szoveg "Ez is egy szöveg formátum.
A második."))
<Fact-2>
CLIPS> (facts)
f-0     (initial-fact)
f-1     (elso-szoveg "Ez az első szöveg")
f-2     (masodik-szoveg "Ez is egy szöveg formátum. A
második.")
For a total of 3 facts.
CLIPS>
```

A `stringp` függvény a `TRUE` szimbólumot adja vissza, ha argumentuma egy karakterlánc, ellenkező esetben a `FALSE` szimbólumot adja vissza.

```
CLIPS> (stringp 7)
FALSE
CLIPS> (stringp 5.2)
FALSE
CLIPS> (stringp a)
FALSE
CLIPS> (stringp ’b’)
FALSE
CLIPS> (stringp "b")
TRUE
```

A `lexemep` függvény a `TRUE` szimbólumot adja vissza, ha argumentuma karakterlánc vagy szimbólum, ellenkező esetben a `FALSE` szimbólumot adja vissza.

```
CLIPS> (lexemep 7)
```

```
FALSE
CLIPS> (lexemep 5.2)
FALSE
CLIPS> (lexemep a)
TRUE
CLIPS> (lexemep "b")
TRUE
CLIPS>
```

A többmezős változók létrehozása a `create$` utasítással történik, amely tetszőleges számú mezőt összefűz egy többmezős változóvá. Vagy `explode$` `utasítással, stringek` esetében.

```
CLIPS> (create$ a b c d e)
(a b c d e)
CLIPS> (explode$ "a b c d e")
(a b c d e)
```

Viszont az `implode$` utasítással többmezős változóból tudunk létrehozni stringet:

```
CLIPS> (implode$ (create$ a b c d e) )
"a b c d e"
CLIPS>
```

Az `nth$` utasítás egy többmezős értékből egy megadott mezőt ad vissza. Ehhez hasonlóan a `first$` az első mezőt, míg a `rest$`, az első mező kivételével a többi mezőt adja vissza.

```
CLIPS> (nth$ 2 (create$ a b c d e) )
b
CLIPS> (first$  (create$ a b c d e))
(a)
CLIPS> (rest$ (create$ a b c d e))
(b c d e)
CLIPS>
```

A `member$` utasítással azt tudjuk megvizsgálni, hogy egy elem tagja-e vagy nem egy többmezős változónak. Ha tagja, válaszként megkapjuk, hogy hányadik elem, ha viszont nem tagja, kiírja: `FALSE`.

```
CLIPS> (member$ c (create$ a b c d e) )
3
CLIPS> (member$ f (create$ a b c d e) )
FALSE
CLIPS>
```

A `subsetp` utasítással azt tudjuk megvizsgálni, hogy egy többmezős változó, egy másiknak részhalmaza-e vagy nem:

```
CLIPS> (subsetp (create$ a b) (create$ a b c d e) )
TRUE
CLIPS>
```

A `replace$` utasítással egy többmezős változó sorrendi helyzete szerinti számmal megjelölt elemeit tudjuk cserélni, míg az `insert$`-el beilleszteni.

```
CLIPS> (replace$ (create$ a b c d e) 1 1 z)
(z b c d e)
CLIPS> (replace$ (create$ a b c d e) 1 4 z)
(z e)
CLIPS> (replace$ (create$ a b c d e) 3 4 x z)
(a b x z e)
CLIPS> (insert$ (create$ a b c d e) 1 z)
(z a b c d e)
CLIPS> (insert$ (create$ a b c d e) 4 z)
(a b c z d e)
CLIPS> (insert$ (create$ a b c d e) 5 (create$ mz per x))
(a b c d mz per x e)
CLIPS>
```

A `delete$` utasítással egy többmezős változó sorrendi helyzete szerinti számmal megjelölt elemeit tudjuk törölni. Például az alábbi második esetben az 5 mezőnek 1-től 4-ig mindegyik tagját töröltük. A `subseq$` utasítás egy többmezős változó sorrendi helyzete szerinti számmal megjelölt elemeit adja vissza.

```
CLIPS> (delete$ (create$ a b c d e) 1 1)
(b c d e)
CLIPS> (delete$ (create$ a b c d e) 1 4)
(e)
CLIPS> (delete$ (create$ a b c d e) 3 4)
(a b e)
CLIPS> (subseq$ (create$ a b c d e) 1 1)
(a)
CLIPS> (subseq$ (create$ a b c d e) 1 4)
```

```
(a b c d)
CLIPS> (subseq$ (create$ a b c d e) 3 4)
(c d)
CLIPS>
```

3.6. Sztring műveletek a CLIPS-ben

A `str-length` függvény egy karakterlánc hosszát egész számként adja vissza.

```
CLIPS> (str-length "qwertzuiop")
10
CLIPS> (str-length "a b c")
5
CLIPS>
```

A szöveg (string) formátum a CLIPS-ben idézőjellel van létrehozva. Egyesítésre a `str-cat` utasítást (angolul string concatenation – sztring egyesítés, lefordítva) használjuk:

```
CLIPS>
(str-cat "első " termék)
"első termék"
CLIPS> (str-cat "második " termék)
"második termék"
CLIPS> (str-cat a b c d e)
"abcde"
CLIPS> (str-cat "a " "b " "c " "d " "e ")
"a b c d e "
CLIPS>
```

Egy sztringből a két szám közötti pozícióban lévő elemeket a `sub-string` (angolul substracting string– sztring kiszűrés vagy kiválasztás, lefordítva) utasítással tudjuk kiemelni.

```
CLIPS> (sub-string 1 3 "qwertzuiop")
"qwe"
CLIPS> (sub-string 1 3 "a b c d e")
"a b"
CLIPS> (sub-string 5 5 "qwertzuiop")
"t"
CLIPS> (sub-string 1 6 "qwertzuiop")
```

```
"qwertz"
```

Az `str-index` utasítás az első argumentumként megadott szöveg kezdőpozícióját adja vissza, egytől kezdve, a második argumentumként megadott szövegben. Illetve a `FALSE` szimbólumot adja vissza, ha nem található.

```
CLIPS> (str-index "r" "qwertzuiop")
4
CLIPS> (str-index "tz" "qwertzuiop")
5
```

Szövegként megadott CLIPS-utasításokat az `eval` utasítással tudjuk kiértékelni, mintha nem sztring lenne, hanem egyenesen a parancssorba lenne beírva.

```
CLIPS> (eval "(+ 14.7 14) ")
28.7
CLIPS> (eval "(create$ mz per x)")
(mz per x)
CLIPS>
```

Ehhez hasonlóan működik a `build` utasítás, amely a parancssorban, CLIPS-utasításként, megszerkeszti a megadott stringet. Az alábbi példában egy `elso` nevű szabályt szerkesztünk meg a `build`del:

```
CLIPS> (build "(defrule elso (initial-facts)  => (assert
(lepes igen)))" )
TRUE
CLIPS> (rules)
elso
For a total of 1 defrule.
CLIPS>
```

Az `upcase` utasítással tudjuk egy szöveg minden elemét nagybetűvé alakítani és `lowcase` utasítással kisbetűsíteni (kivéve a magyar karaktereket, ahogy alább látszik).

```
CLIPS> (upcase "A termék megtalálható a készleten. ")
"A TERMéK MEGTALáLHATó A KéSZLETEN. "
CLIPS> (upcase "Product is in stock. ")
```

```
"PRODUCT IS IN STOCK. "
CLIPS> (lowcase "A TERMÉK MEGTALÁLHATÓ A KÉSZLETEN. ")
"a termÉk megtalÁlhatÓ a kÉszleten. "
CLIPS> (lowcase "Product is in stock. ")
"product is in stock. "
CLIPS> (lowcase "PRODUCT IS IN STOCK. ")
"product is in stock. "
CLIPS>
```

Az `str-compare` függvény két karakterláncot úgy hasonlít össze, hogy meghatározza logikai kapcsolatukat (azaz egyenlő, kisebb, nagyobb). Az összehasonlítást karakterenként hajtja végre, amíg a karakterláncok ki nem merülnek (ez egyenlő karakterláncokat jelent), vagy nem talál egyenlőtlen karaktereket. Az egyenlőtlen karakterek pozíciója az ASCII karakterkészletben az egyenlőtlen st logikai kapcsolatának meghatározására szolgál. Az alábbi harmadik esetben azért teljesül, mert 1 < 2.

```
CLIPS> (< (str-compare "szöveg1" "szöveg1") 0)
FALSE
CLIPS> (= (str-compare "szöveg1" "szöveg1") 0)
TRUE
CLIPS> (< (str-compare "szöveg1" "szöveg2") 0)
TRUE
CLIPS>
```

3.7. Tudásreprezentáció a CLIPS-ben

3.7.1 Heurisztikus tudásreprezentáció, szabályokkal

A CLIPS *heurisztikus* és *procedurális (eljárási)* paradigmákat biztosít a tudás reprezentálásához.

A *heurisztikus tudásreprezentáció szabályokkal történik*. A tudás CLIPS-ben való megjelenítésének egyik elsődleges módszere a szabály. A szabályok a heurisztikák vagy „ökölszabályok" ábrázolására szolgálnak, amelyek meghatározzák az adott helyzetben végrehajtandó műveletek halmazát. A szakértői rendszer fejlesztője szabályokat határoz meg, amelyek együttesen dolgoznak a probléma megoldásán. A szabály egy feltételelőzményből és egy következményből áll. A szabály előzményét a HA-résznek (If-résznek) vagy a szabály bal oldalának (LHS – Left-Hand Side) is nevezik, a szabály következményét a szabály akkori részének vagy jobb oldalának (RHS – right-hand side) is nevezik.

A feltételrendszer a tények legösszetettebb logikai kapcsolatrendszerét tartalmazhatja, amelynek teljesülnie kell, hogy a szabály alkalmazható legyen. Ezeket a megadható feltételtípus, vagyis a minta (pattern) megfelelő kialakításával érhetjük el. A minták korlátozások halmazából állnak, amelyek annak meghatározására szolgálnak, hogy mely tények vagy objektumok felelnek meg a minta által meghatározott feltételnek. A tények és tárgyak mintákhoz való illesztésének folyamatát mintaillesztésnek (pattern-matching) nevezzük.

Abban az esetben, ha a szabály alkalmazható, egy sor művelet elvégzésére kerül sor. Ha több szabály alkalmazható, egy konfliktusmegoldó stratégia (conflict resolution strategy) alapján lesz kiválasztva, hogy melyik szabály teljesül, vagy több szabályt milyen sorrendben végzünk el.

Nézzük meg ezt a HA-AKKOR (`IF-THEN`) feltételrendszert. Hasonló a C nyelvben is megtalálható, azzal a különbséggel, hogy a CLIPS-ben a szabályok esetében, jobban hasonlít egy „*BÁRMIKOR*-AKKOR" feltételrendszerhez, mivel a következtetési motor mindig nyomon követi azokat a szabályokat, amelyek feltételei teljesülnek, így ezeket *azonnal* végrehajtjuk, amikor alkalmazhatók.

3.7.2 Procedurális tudásreprezentáció

A CLIPS *procedurális (eljárási) tudásreprezentációja* olyan eljárási paradigmát is támogat a tudás megjelenítésére, mint a hagyományosabb programozási nyelvek. A függvények meghatározásával (`deffunction` utasítással) lehetővé válik a felhasználó számára, hogy új végrehajtható elemeket definiáljon a CLIPS-hez, amelyek hasznos közvetett hatást fejtenek ki, vagy hasznos értéket adnak vissza. Ezek az új függvények ugyanúgy hívhatók, mint a CLIPS beépített függvényei. Lehetővé válik a felhasználó számára, hogy meghatározza az objektumok viselkedését az üzenetekre adott válaszuk megadásával. A `deffunction`, az általános függvények és az üzenetkezelők mind a felhasználó által meghatározott eljárási kódrészletek, amelyeket a CLIPS a megfelelő időpontokban értelmező módon hajt végre. A `defmodulok` pedig lehetővé teszik a tudásbázis particionálását.

3.8. Műveletek nemrendezett tényekkel

A nemrendezett (vagy `deftemplate`) tények, előre megszerkesztett keretben, sablon szerint, a megfelelő mezőkhöz rendelhetők, majd a ténymezők név szerint elérhetők. Maga az utasításnév is ennek a rövidítése, angolul: definition template, vagyis *deftemplate*, ami magyarul sablonmeghatározást jelent. Ez a szerkezeti forma hasonlít a CLIPS, a Pascal és a C programozási nyelvekben megtalálhatókéhoz.

```
A deftemplate szintaxisa:

(deftemplate <sablonnév meghatározása> [<megjegyzés>]
<mező-meghatározás> ::= < egy-mező-meghatározása > | <
több-mező-meghatározása >
< egy-mező-meghatározása > ::=
(mező < mező -név> <sablon-attribútum>*)
< több -mező-meghatározása > ::=
(több-mező < mező -név> <sablon-attribútum>*)
<alapértelmezett attribútum meghatározása> ::=
(alapértelmezett ?DERIVE | ?NONE | <kifejezés>*) |
(alapértelmezett-dinamikus <kifejezés>*)
```

Egy `deftemplate` szerkezet tetszőleges számú, egy vagy több mezővel (slottal) rendelkezhet, de hiba több értéket tárolni (vagy egyeztetni) egy egymezős slotban. A deftemplate újradefiniálása az előző definíció elvetését eredményezi. Egy `deftemplate` nem definiálható újra használat közben (például egy szabályban lévő tény vagy minta alapján). Az alábbi példában (bútortermékek) a név, sku azonosító szám, tömeg, hossz és szélesség mezőket definiáltunk. A `deffacts` szerkezetben tetszőleges számú terméket hozzáadhatunk, minden mezőhöz értéket rendelve.

```
(deftemplate termek
(slot nev)
(slot sku_id)
(slot tomeg)
(slot hossz)
(slot szelesseg)
)

(deffacts butorok
(termek (nev iroasztal_cs_1) (sku_id 723) (tomeg 17.5)
(hossz 145) (szelesseg 75))
(termek (nev iroasztal_tl_7) (sku_id 458) (tomeg 22)
(hossz 152) (szelesseg 64))
```

```
(termek (nev szekreny_cs_4) (sku_id 224) (tomeg 35) (hossz
175) (szelesseg 64))
(termek (nev szekreny_cs_9) (sku_id 112) (tomeg 42) (hossz
160) (szelesseg 80))
)
```

A programot elmentjük (.CLP formátumban), beolvassuk a parancssorban, majd a `(reset)` utasítással „visszaállítjuk” vagy helyreállítjuk a megszerkesztett ténylistát. Ahogy alább látható, a (facts) utasítással, a felsorolt tények az `(initial-fact)`-en kívül tartalmazzák a négy terméket.

```
CLIPS> (load "E:/egyetem/szakertoi
rendszerek/CLIPS/butorok.CLP")
Defining deftemplate: termek
Defining deffacts: butorok
TRUE
CLIPS> (reset)
CLIPS> (facts)
f-0     (initial-fact)
f-1     (termek (nev iroasztal_cs_1) (sku_id 723) (tomeg
17.5) (hossz 145) (szelesseg 75))
f-2     (termek (nev iroasztal_tl_7) (sku_id 458) (tomeg
22) (hossz 152) (szelesseg 64))
f-3     (termek (nev szekreny_cs_4) (sku_id 224) (tomeg
35) (hossz 175) (szelesseg 64))
f-4     (termek (nev szekreny_cs_9) (sku_id 112) (tomeg
42) (hossz 160) (szelesseg 80))
For a total of 5 facts.
```

Mi történik, ha beolvasás után nem adjuk ki a `reset` utasítást? Nem történik meg a beolvasott tények inicializálása, tehát csak az f-0 indexszel jelölt (initial-fact) „érvényes”. Az `(initial-fact)` az első ténynek vagy alapfeltételének felel meg, amely minden programban működik, ezért amint azt a továbbiakban a szabályoknál látni fogjuk, az első szabály feltétele is lehet.

```
CLIPS> (load "E:/egyetem/szakertoi
rendszerek/CLIPS/butorok.CLP")
Defining deftemplate: termek
Defining deffacts: butorok
TRUE
CLIPS> (facts)
```

```
f-0     (initial-fact)
For a total of 1 fact.
```

Hogyan tudunk új tényt hozzáadni a ténylistához? Az `assert` utasítással, minden mezőt kitöltve megtehetjük, ahogy alább látható:

```
CLIPS> (assert (termek (nev ejjeliszekreny_cs_2) (sku_id
474) (tomeg 16.8) (hossz 120) (szelesseg 65)))
<Fact-5>
CLIPS> (facts)
f-0     (initial-fact)
f-1     (termek (nev iroasztal_cs_1) (sku_id 723) (tomeg
17.5) (hossz 145) (szelesseg 75))
f-2     (termek (nev iroasztal_tl_7) (sku_id 458) (tomeg
22) (hossz 152) (szelesseg 64))
f-3     (termek (nev szekreny_cs_4) (sku_id 224) (tomeg
35) (hossz 175) (szelesseg 64))
f-4     (termek (nev szekreny_cs_9) (sku_id 112) (tomeg
42) (hossz 160) (szelesseg 80))
f-5     (termek (nev ejjeliszekreny_cs_2) (sku_id 474)
(tomeg 16.8) (hossz 120) (szelesseg 65))
For a total of 6 facts.
```

Ha olyan új tényt adunk hozzá, amely nem tartalmaz egy vagy több mezőnél értéket, a CLIPS a szerkezetnek megfelelően tárolja, viszont a hiányos tényt a listában üresnek jelöli (`nil`-lel):

```
CLIPS> (assert (termek (nev ejjeliszekreny_tl_3) (sku_id
474) (hossz 120))
f-6     (termek (nev ejjeliszekreny_tl_3) (sku_id 474)
(tomeg nil) (hossz 120))
```

A `(get-deffacts-list)` utasítással ki tudjuk íratni a deffacts szerkezet nevét:

```
CLIPS> (get-deffacts-list)
(initial-fact butorok)
```

A `deftemplate` - `deffacts` szerkezet mezőinek attribútumai alapértelmezetten statikus értéket tárolnak. Amikor tényeket rendelünk ehhez a szerkezethez (például assert utasítással), és a mező a `?NONE` kulcsszóval van alapértelmezve, akkor szükségszerűen hozzá kell rendelni egy értéket. Azon mezőknél, ahol nem szerepel a `?NONE` kulcsszó, nem szükséges mindegyik tény esetében értéket definiálni. Ha a

?DERIVE kulcsszó az alapértelmezett érték, a ténylistát kötelező módon kitölti, vagyis minden esetben értéket rendel hozzá. Pontosabban üreshalmaz – nil – lesz a mező értéke, ha az asserttel nem adunk meg értéket, ahogy látni fogjuk. A dinamikus mezőérték hozzárendelését a gensym* utasítással tudjuk elvégezni. A default-dynamic a dinamikus alapértelmezett típust jelöli, amelyet minden alkalommal kiértékel a tény hozzárendelésekor, és a gensym sorrendjében megadott értéket követi. Az alábbi példában ezért veszi fel a gen2 értéket.

```
CLIPS> (deftemplate attributumok
(slot kotelezo (default ?NONE))
(slot alapert (default ?DERIVE))
(slot dinam1 (default (gensym*)))
(slot dinam2 (default-dynamic (gensym*))))
CLIPS> (assert (attributumok))

[TMPLTRHS1] Slot kotelezo requires a value because of its
(default ?NONE) attribute.
CLIPS> (assert (attributumok (kotelezo 28)))
<Fact-1>
CLIPS> (assert (attributumok (kotelezo masodik)))
<Fact-2>
CLIPS> (facts)
f-0     (initial-fact)
f-1     (attributumok (kotelezo 28) (alapert nil) (dinam1
gen1) (dinam2 gen2))
f-2     (attributumok (kotelezo masodik) (alapert nil)
(dinam1 gen1) (dinam2 gen3))
For a total of 3 facts.
CLIPS>
```

A setgen utasítás lehetővé teszi a felhasználó számára a gensym és a gensym* által használt szám beállítását. Például a (setgen 147) utasítás után, ha meghívjuk a gensymet, akkor gen147, gen148 stb. értékeket vesz fel:

```
CLIPS> (setgen 147)
147
CLIPS> (assert (attributumok (kotelezo harmadik)))
<Fact-3>
CLIPS> (facts)
f-0     (initial-fact)
```

```
f-1     (attributumok (kotelezo 28) (alapert nil) (dinam1
gen1) (dinam2 gen2))
f-2     (attributumok (kotelezo masodik) (alapert nil)
(dinam1 gen1) (dinam2 gen3))
f-3     (attributumok (kotelezo harmadik) (alapert nil)
(dinam1 gen1) (dinam2 gen147))
For a total of 4 facts.
CLIPS>
```

A `progn` függvény kiértékeli az összes argumentumot, és visszaadja az utolsó argumentum értékét. Használjuk ezt a `(setgen 7)` után, ha meghívjuk a `gensym`re:

```
CLIPS> (progn (setgen 7) (gensym))
gen7
CLIPS>
```

Illetve ha csak az értéket kérjük vissza:

```
CLIPS> (setgen 7) (progn (gensym))
7
CLIPS>
```

A `progn$` függvény egy sor műveletet hajt végre egy többmezős érték minden egyes mezőjében. Az aktuális iteráció mezője a `<mező-változó>` paraméterrel vizsgálható, ha meg van adva. Az alábbi példában a mező változója: `?mezo` és az indexváltozó: `?mezo-index`.

```
CLIPS> (progn$ (?mezo (create$ egy ketto harom negy))
(printout t "Mező értéke: " ?mezo " és indexe: " ?mezo-
index crlf))
Mező értéke: egy és indexe: 1
Mező értéke: ketto és indexe: 2
Mező értéke: harom és indexe: 3
Mező értéke: negy és indexe: 4
CLIPS>
```

A `(multifieldp)` utasítással tudjuk ellenőrizni, hogy egy tény többmezős-e vagy nem:

```
CLIPS> (multifieldp (create$ q w e r t y u i o p))
TRUE
```

A `length` utasítás egy egész számként adja meg, a többmezős tény esetében, a mezők számát vagy egy karakterlánc vagy szimbólum hosszát.

```
CLIPS> (length "szoveg")
6
CLIPS> (length (create$ q w e r t y u i o p))
10
CLIPS>
```

3.9. Szabályok használata a CLIPS-ben

A szabály általános szerkezete a CLIPS-ben a következő:

```
(defrule szabaly_neve
<logikai-feltételek>
=>
<utasítás/utasítások>)
```

A `<logikai-feltételek>` előzetesen megszerkesztett tényekből álló logikai struktúra, amelyben a logikai ÉS, VAGY stb. utasításokat használtuk fel. Nézzünk a `termekek.CLP`-re példát:

```
(deffacts elso (termek laptop) (termek smartphone) (termek
memory_drive))
(defrule termek_listazas
(termek ?)
  =>
(printout t "termék a listában" crlf))
```

A `deffacts` utasítással létrehoztuk a három tényt, amelyeket a `termek_listazas` nevű szabály „megtalált", vagyis ahogy látható, három esetben teljesült a szabály feltétele.

```
CLIPS> (load "E:/egyetem/szakertoi
  rendszerek/CLIPS/termekek.CLP")
Defining deffacts: elso
```

```
Defining defrule: termek_listazas +j+j
TRUE
CLIPS> (reset)
CLIPS> (run)
termék a listában
termék a listában
termék a listában
```

A `rules` utasítással tudjuk az érvényes szabályokat felsorolni:

```
CLIPS> (rules)
termek_listazas
```

A változók jelölése a szabályon belül: `?változónév` olyan esetekben használható, amikor a szabály bal oldalán kell teljesülnie egy feltételnek, és ezzel a jobb oldalán valamilyen műveletet végzünk el. Figyeljük meg, hogy a `?valtozo` nevű változó használatával alább kiírjuk a termékeket.

```
(defrule termek_listazas
(termek ?valtozo)
  =>
(printout t ?valtozo "  termék van a listában" crlf))

CLIPS> (reset)
CLIPS> (run)
memory_drive  termék van a listában
smartphone  termék van a listában
laptop  termék van a listában
```

A változókkal összetett logikai feltételek szerkesztésére is van mód, ahogy az alábbi példában látjuk.

```
(deffacts elso (termek laptop) (termek smartphone) (termek
memory_drive) (termek camera) (keszleten smartphone)
(keszleten camera)  (tartozek camera allvany) (tartozek
smartphone selfie_stick) )
(defrule termek_kiegeszitok
(termek ?valtozo1)
(tartozek ?valtozo1 ?valtozo2)
  =>
(assert (termek ?valtozo2))
(printout t ?valtozo2 "  termékkiegészítő is bekerült a
terméklistába" crlf))
```

A szóközzel elválasztott tények a feltételben a logikai „És”-sel vannak összekapcsolva, vagyis mindegyiknek teljesülnie kell. Tehát az alábbi két feltételnek azonos a jelentése:

```
(termek ?valtozo1) (tartozek ?valtozo1 ?valtozo2)
(and (termek ?valtozo1) (tartozek ?valtozo1 ?valtozo2))
```

A `?valtozo1` a `(termek)` tényekhez van hozzárendelve, a `?valtozo2` a `(tartozek)` tényekhez. Két esetben teljesül, azoknál a termékeknél, amelyeknek van tartozékuk. A szabály jobb oldali részében, mindkét esetben létrejön a `(termek ?valtozo2)` tény, ahol a `?valtozo2`-t a feltételből visszük át. A futtatás eredményeként két új tény jött létre (a 9-es és a 10-es), vagyis a kiegészítők is bekerültek a termékek közé, ahogyan a tényeket listázva látjuk. Tetszőleges mezővel rendelkező ténylista esetén hasonlóan megszerkeszthető a feltétele bármelyik szabálynak.

```
CLIPS> (reset)
CLIPS> (run)
selfie_stick termékkiegészítő is bekerült a terméklistába
allvany  termékkiegészítő is bekerült a terméklistába
CLIPS> (facts)
f-0     (initial-fact)
f-1     (termek laptop)
f-2     (termek smartphone)
f-3     (termek memory_drive)
f-4     (termek camera)
f-5     (keszleten smartphone)
f-6     (keszleten camera)
f-7     (tartozek camera allvany)
f-8     (tartozek smartphone selfie_stick)
f-9     (termek selfie_stick)
f-10    (termek allvany)
For a total of 11 facts.
CLIPS>
```

Az alábbi szabállyal tényt törlünk a listából. A `(keszleten)` tényeket töröltük ki, vagyis sorrendben az 5-ös és a 6-os tényt, ahogy a szabály második részében kiírtuk:

```
(defrule teny_torlese
?fact <- (keszleten ?)
```

```
=>
(printout t " torolve a: " ?fact crlf)
(retract ?fact))
```

Eredménye:

```
CLIPS> (run)
torolve a: <Fact-6>
torolve a: <Fact-5>
CLIPS>
```

Az alábbi példában az `or` (logikai „Vagy") utasítással ellenőrizzük, hogy készleten van-e egy adott termék.

```
(deffacts elso (termek laptop) (termek smartphone) (termek
memory_drive) (termek camera) (raktar1 laptop) (raktar1
smartphone) (raktar2 camera)  (tartozek camera allvany)
(tartozek smartphone selfie_stick) )
(defrule keszlet_vagy
(or (raktar1 laptop)
(raktar2 laptop))
=>
(printout t " a laptop készleten van" crlf)
(assert (keszleten laptop))
)
```

Beolvasás a billentyűzetről. A CLIPS információkat kap a felhasználótól a `(read)` utasítással. Bárhol, ahol a `(read)`-et meghívják, a program megvárja, amíg a felhasználó beír valamit, majd helyettesíti a választ. Például az `(assert (felhasznalotol-beolvas (read)))` paranccsal a CLIPS-parancssorból beolvassuk az adatot, majd hozzárendeljük a tényhez:

```
CLIPS> (assert (felhasznalotol-beolvas (read)))
1234567
<Fact-1>
CLIPS> (facts)
f-0     (initial-fact)
f-1     (felhasznalotol-beolvas 1234567)
For a total of 2 facts.
CLIPS>
```

Az alábbi termékek esetében vannak olyanok, amelyek készleten vannak, és vannak, amelyek nincsenek. Az `adat-beolvasas` nevű szabállyal ezt tudjuk beolvasni.

```
(deffacts elso (termek laptop) (termek smartphone) (termek
memory_drive) (termek camera) (keszleten laptop igen)
(keszleten smartphone igen) (keszleten camera igen))
(defrule adat-beolvasas
(termek ?nev)
(not (keszleten ?nev igen))
=>
(printout t "Készlet kibővült a: " ?nev " termékkel?
(igen/nem) ")
(assert (keszleten ?nev (read)))
)
```

A futtatás eredményeként a `?nev` változóval (amely a `memory_drive` lesz, mert ez nincs készleten) és a `(read)` utasítással beolvasott „igen" szóval a szabály második részében létrehozunk egy új tényt, amelyet tények kiírásával ellenőriztünk, hogy sikeresen létrejött (a 8-as tény a listában).

```
CLIPS> (load "E:/egyetem/szakertoi
rendszerek/CLIPS/beolvasas.CLP")
Defining deffacts: elso
Defining defrule: adat-beolvasas +j+j+j
TRUE
CLIPS> (reset)
CLIPS> (run)
Készlet kibővült a: memory_drive termékkel? (igen/nem)
igen
CLIPS> (facts)
f-0     (initial-fact)
f-1     (termek laptop)
f-2     (termek smartphone)
f-3     (termek memory_drive)
f-4     (termek camera)
f-5     (keszleten laptop igen)
f-6     (keszleten smartphone igen)
f-7     (keszleten camera igen)
f-8     (keszleten memory_drive igen)
For a total of 9 facts.
CLIPS>
```

Nézzük olyan esetet is, amikor egyetlen termék sincs készleten, és ez a szabály feltétele:

```
(deffacts elso (termek laptop) (termek smartphone) (termek
memory_drive) (termek camera))
(defrule keszlet-ures
(termek ?nev)
(not (keszleten ?nev))
=>
(printout t "Készlet kibővült a: " ?nev " termékkel?
(igen/nem) ")
(assert (keszleten ?nev (read)))
)
```

Ekkor minden ténylistában rögzített termék esetén teljesül a szabály feltétele és a program, minden esetben hozzárendeli hozzá az új tényt, („igen”-t vagy „nem”-et, annak függvényében, hogy milyen választ adtunk meg a `(read)` beolvasásánál).

```
CLIPS> (load "E:/egyetem/szakertoi
rendszerek/CLIPS/beolvasas02.CLP")
Defining deffacts: elso
Defining defrule: keszlet-ures +j+j+j
TRUE
CLIPS> (reset)
CLIPS> (run)
Készlet kibővült a: camera termékkel? (igen/nem) igen
Készlet kibővült a: memory_drive termékkel? (igen/nem) nem
Készlet kibővült a: smartphone termékkel? (igen/nem) igen
Készlet kibővült a: laptop termékkel? (igen/nem) nem
CLIPS> (facts)
f-0     (initial-fact)
f-1     (termek laptop)
f-2     (termek smartphone)
f-3     (termek memory_drive)
f-4     (termek camera)
f-5     (keszleten camera igen)
f-6     (keszleten memory_drive nem)
f-7     (keszleten smartphone igen)
f-8     (keszleten laptop nem)
For a total of 9 facts.
CLIPS>
```

A helyettesítő karakterminta (pattern – angolul) illesztése tehát a `"?"` szimbólummal úgy működik, hogy a szabály bal oldalán lévő szimbólum helyéről átvezet a jobb

oldali részre, egy szimbólummal egyezve. Ennek bővítésére szolgál a többmezős helyettesítő karakter, a "$?", amely nulla vagy több szimbólumnak felelhet meg. Nézzük a használatát az alábbiakban:

```
(deffacts masodik (keszleten laptop asus) (keszleten
smartphone samsung) (keszleten memory_drive hp) (keszleten
laptop dell) (keszleten laptop hp))
(defrule keszleten-lista-kiiras
(keszleten ?termek $?)
=>
(printout t "Készleten van a:  " ?termek crlf))

CLIPS> (load "E:/egyetem/szakertoi
rendszerek/CLIPS/keszleten.CLP")
Defining deffacts: masodik
Defining defrule: keszleten-lista-kiiras +j+j
TRUE
CLIPS> (reset)
CLIPS> (run)
Készleten van a:  memory_drive
Készleten van a:  smartphone
Készleten van a:  laptop
```

Látható, hogy a laptop esetében egyszer írja ki, hogy készleten van, viszont a ténylistában három laptop van felsorolva. Ilyen esetben használható a többmezős helyettesítő karakter (a `$?`), ahogyan a tobbmezos-kiiras szabályban van, sorrendben felvéve az `asus, dell` és `hp` értékeket.

```
(deffacts masodik (keszleten laptop asus dell hp)
(keszleten smartphone samsung) (keszleten memory_drive
hp))
(defrule tobbmezos-kiiras
(keszleten ?termek $? ?tipus $?)
=>
(printout t "Készleten van:  " ?termek "-nek a:  "
?tipus" típusa" crlf))
```

Eredménye:

```
CLIPS> (load "E:/egyetem/szakertoi
rendszerek/CLIPS/keszleten.CLP")
Defining deffacts: masodik
Defining defrule: tobbmezos-kiiras +j+j
```

```
TRUE
CLIPS> (reset)
CLIPS> (run)
Készleten van:  memory_drive-nek a:  hp típusa
Készleten van:  smartphone-nek a:  samsung típusa
Készleten van:  laptop-nek a:  asus típusa
Készleten van:  laptop-nek a:  dell típusa
Készleten van:  laptop-nek a:  hp
```

Kis változtatással is működik:

```
(defrule tobbmezos-kiiras
(keszleten ?termek $?tipus )
=>
(printout t "Készleten van:  " ?termek "-nek a:  "
?tipus" típusa" crlf))
```

Ekkor a futtatás eredménye a tények „zárójeles” kiírását adja, vagyis nem veszi fel mindegyik értéket sorrendben, hanem egyben:

```
CLIPS> (load "E:/egyetem/szakertoi
rendszerek/CLIPS/keszleten.CLP")
Defining deffacts: masodik
Defining defrule: tobbmezos-kiiras +j+j
TRUE
CLIPS> (reset)
CLIPS> (run)
Készleten van:  memory_drive-nek a:  (hp) típusa
Készleten van:  smartphone-nek a:  (samsung) típusa
Készleten van:  laptop-nek a:  (asus dell hp)
```

A további példákban a ténylistát (deftemplateắ (deffactsắ szerkezettel hozzuk létre, mert a feltételek, logikai kifejezések jobban szemléltethetőek. Elsőként nézzük meg, hogyan működik egy kiírásszabály ilyen ténylista esetében.

```
(deftemplate termek
(slot nev) (slot sku_id) (slot tomeg) (slot hossz) (slot
szelesseg) (slot keszleten)
)
```

```
(deffacts butorok
(termek (nev iroasztal_1) (sku_id 723) (tomeg 17.5) (hossz
145) (szelesseg 75) (keszleten 24))
(termek (nev iroasztal_7) (sku_id 458) (tomeg 22) (hossz
152) (szelesseg 64)(keszleten 12))
(termek (nev szekreny_4) (sku_id 224) (tomeg 35) (hossz
175) (szelesseg 64)(keszleten 32))
(termek (nev szekreny_9) (sku_id 112) (tomeg 42) (hossz
160) (szelesseg 80)(keszleten 17))
)

(defrule kiiras
(termek (nev ?nev) (hossz ?hossz)  (szelesseg ?szelesseg)
(keszleten ?keszleten))
=>
(printout t ?nev "  hossza: " ?hossz "  szélessége: "
?szelesseg  "  készleten van: " ?keszleten crlf)
)

CLIPS> (load "E:/egyetem/szakertoi
rendszerek/CLIPS/kiiras.CLP")
Defining deftemplate: termek
Defining deffacts: butorok
Defining defrule: kiiras +j+j
TRUE
CLIPS> (reset)
CLIPS> (run)
szekreny_9  hossza: 160  szélessége: 80  készleten van: 17
szekreny_4  hossza: 175  szélessége: 64  készleten van: 32
iroasztal_7  hossza: 152  szélessége: 64  készleten van:
12
iroasztal_1  hossza: 145  szélessége: 75  készleten van:
24
```

A `deftemplate` szerkezetében megadhatunk előre engedélyezett értékeket, amelyeket később a feltételében felhasználhatunk. Az alábbi példában két vállalkozást keresünk meg, amelyeknél az ügyfélszolgálat és a PR is jó:

```
(deftemplate szolgaltatas
(slot nev)
(slot ugyfelkezeles (allowed-symbols kozepes jo rossz))
(slot pr (allowed-symbols jo kozepes rossz))
)
(deftemplate valasztas
```

```
(slot vallalat)
)

(deffacts vallalatok
(szolgaltatas (nev abc_kft) (ugyfelkezeles jo) (pr
kozepes))
(szolgaltatas (nev xyz_kft) (ugyfelkezeles jo) (pr jo))
(szolgaltatas (nev uvt_kft) (ugyfelkezeles jo) (pr jo))
)

(defrule feltetelek
        (szolgaltatas (nev ?vallalat) (ugyfelkezeles
?ugyfelkezeles&:(eq ?ugyfelkezeles jo))
        (pr ?pr&:(eq ?pr jo)))
        =>
        (assert (dont igen))
        (assert (valasztas (vallalat ?vallalat)))
)

(defrule kiiras
       (dont igen)
       (valasztas (vallalat ?vallalat))
        =>
      (printout t "Jó ügyfélekezelés és PR az "?vallalat "
vállalatnál " crlf)
)
```

A kiírással meg is találja a program a ténylistában azt a két vállalkozást, amelynél egyszerre teljesülnek a feltételek:

```
CLIPS> (load "E:/egyetem/Tantargyfejlesztes/szakertoi
rendszerek/CLIPS/szolgaltatas04.CLP")
Defining deftemplate: szolgaltatas
Defining deftemplate: valasztas
Defining deffacts: vallalatok
Defining defrule: feltetelek +j+j
Defining defrule: kiiras +j+j+j
TRUE
CLIPS> (reset)
CLIPS> (run)
Jó ügyfélkezelés és PR az uvt_kft vállalatnál
Jó ügyfélkezelés és PR az xyz_kft vállalatnál
CLIPS>
```

Szerkesszünk olyan szabályt, amely egy ténylistában egy feltétel alapján az egyik `deftemplate` szerkezetből egy másikat szerkeszt meg. A vállalatok példánál

maradva, a `deftemplate innov-vallalat`-ban azokat az eseteket fogjuk csak átmásolni, melyeknél teljesül az innováció feltétele.

```
(deftemplate vallalkozas
(slot nev)
(slot agazat)
(slot forgalom)
(slot alkalmazottak)
(slot innovacio)
(slot profit)
)
(deffacts vallalkozasok
(vallalkozas (nev Progress) (agazat ipar) (forgalom
200000)(alkalmazottak 18)(innovacio van)(profit 30000))
(vallalkozas (nev IT2000) (agazat IT)(forgalom
15000)(alkalmazottak 50)(innovacio van)(profit 8000))
(vallalkozas (nev IMPEX)(agazat mezogazdasag)(forgalom
12000)(alkalmazottak 100)(innovacio nincs)(profit 20000))
(vallalkozas (nev RETRO)(agazat szolgaltatas)(forgalom
18000)(alkalmazottak 8)(innovacio van)(profit 60000))
(vallalkozas (nev Max)(agazat kereskedelem)(forgalom
20000)(alkalmazottak 15)(innovacio nincs)(profit 40000))
)
(deftemplate innov-vallalat
(slot nevek)
(slot agazat)
)
(defrule feltetel
   (vallalkozas (nev ?nev) (agazat ?agazat) (innovacio
van))
   =>
   (assert (innov-vallalat (nevek ?nev) (agazat ?agazat)))
   (assert (innovacio igen))
)

(defrule kiir
   (innovacio igen)
   (innov-vallalat (nevek ?nev) (agazat ?agazat))
   =>
   (printout t ?nev " egy " ?agazat " - ban tevékenykedő
vállalat, amelynél van innováció. " crlf)
)
```

Amikor futtatjuk a programot, kiírja ezt a három vállalatot, illetve a ténylistában látjuk a `deftemplate innov-vallalat` szerkezetben elhelyezett új tényeket:

```
CLIPS> (load "E:/egyetem/Tantargyfejlesztes/szakertoi
rendszerek/CLIPS/innovalo-vallalatok02.CLP")
Defining deftemplate: vallalkozas
Defining deffacts: vallalkozasok
Defining deftemplate: innov-vallalat
Defining defrule: feltetel +j+j
Defining defrule: kiir +j+j+j
TRUE
CLIPS> (reset))
CLIPS> (run)
RETRO egy szolgaltatas - ban tevékenykedő vállalat,
amelynél van innováció.
IT2000 egy IT - ban tevékenykedő vállalat, amelynél van
innováció.
Progress egy ipar - ban tevékenykedő vállalat, amelynél
van innováció.
CLIPS> (facts)
f-0     (initial-fact)
f-1     (vallalkozas (nev Progress) (agazat ipar)
(forgalom 200000) (alkalmazottak 18) (innovacio van)
(profit 30000))
f-2     (vallalkozas (nev IT2000) (agazat IT) (forgalom
15000) (alkalmazottak 50) (innovacio van) (profit 8000))
f-3     (vallalkozas (nev IMPEX) (agazat mezogazdasag)
(forgalom 12000) (alkalmazottak 100) (innovacio nincs)
(profit 20000))
f-4     (vallalkozas (nev RETRO) (agazat szolgaltatas)
(forgalom 18000) (alkalmazottak 8) (innovacio van) (profit
60000))
f-5     (vallalkozas (nev Max) (agazat kereskedelem)
(forgalom 20000) (alkalmazottak 15) (innovacio nincs)
(profit 40000))
f-6     (innov-vallalat (nevek RETRO) (agazat
szolgaltatas))
f-7     (innovacio igen)
f-8     (innov-vallalat (nevek IT2000) (agazat IT))
f-9     (innov-vallalat (nevek Progress) (agazat ipar))
For a total of 10 facts.
CLIPS>
```

3.10. Logikai feltételek

Kifejezések és szimbólumok esetében az egyenlőség tesztelését az eq utasítással végezzük el, a nem egyenlőségét pedig a neq utasítással:

```
CLIPS> (eq szam1 szam1 szam1)
TRUE
CLIPS> (eq 5 7)
FALSE
CLIPS> (neq szam1 szam1 szam1)
FALSE
CLIPS> (neq 5 7)
TRUE
CLIPS>
```

Számok esetében az egyenlőség tesztelését az `'='` (egyenlőség) utasítással végezzük el. Ez nem használható kifejezésekre és szimbólumokra, ahogy a harmadik példában látható:

```
CLIPS> (= 5 7)
FALSE
CLIPS> (= 14.7 14.7)
TRUE
CLIPS> (= 28 28)
TRUE
CLIPS> (= szam1 szam1 szam1)
[ARGACCES5] Function = expected argument #1 to be of type
integer or float
```

Számok esetében a nem egyenlőség tesztelését az `'<>'` utasítással végezzük el:

```
CLIPS> (<> 5 7)
TRUE
CLIPS> (<> 14.7 14.7)
FALSE
CLIPS> (<> 28 28)
FALSE
CLIPS>
```

Kisebb-nagyobb összehasonlítás, mint logikai feltétel (< és >), számok esetében az alábbi példákon keresztül van bemutatva. Ahogy a második és negyedik utasításnál látható, akár több szám sorrendjét is vizsgálhatjuk egyszerre:

```
CLIPS> (> 9 7)
TRUE
CLIPS> (> 12 9 7)
TRUE
```

```
CLIPS> (> 6 14)
FALSE
CLIPS> (< 7 8 9)
TRUE
CLIPS> (< 9 7)
```

Ez előbbiekhez teljesen hasonló módon működnek a nagyobb vagy egyenlő és kisebb vagy egyenlő `(>= és <=)` logikai feltételek:

```
CLIPS> (>= 7 7 6)
TRUE
CLIPS> (>= 5 7)
FALSE
CLIPS> (<= 4 6 6)
TRUE
CLIPS> (<= 7 5)
FALSE
CLIPS>
```

A szabály HA részének logikai feltételei számos lehetőséget tartogatnak, ilyenek a logikai és, vagy, létezik, nem létezik, mindegyik eset, érték tesztelése stb. Tulajdonképpen abban áll a CLIPS erőssége, hogy a De Morgan-szabályok bármelyike vagy bármilyen összetett logikai kifejezés megszerkeszthető.

A tények közötti feltételes keresés a `(test)` utasítással oldható meg. Például alább a 30 kg-nál nagyobb tömegű termékeket listázzuk ki:

```
(deftemplate termek
(slot nev) (slot sku_id) (slot tomeg) (slot hossz) (slot
szelesseg) (slot keszleten)
)
(deffacts butorok
(termek (nev iroasztal_1) (sku_id 723) (tomeg 17.5) (hossz
145) (szelesseg 75) (keszleten 24))
(termek (nev iroasztal_7) (sku_id 458) (tomeg 22) (hossz
152) (szelesseg 64)(keszleten 12))
(termek (nev szekreny_4) (sku_id 224) (tomeg 35) (hossz
175) (szelesseg 64)(keszleten 32))
(termek (nev szekreny_9) (sku_id 112) (tomeg 42) (hossz
160) (szelesseg 80)(keszleten 17))
)

(defrule nagyobbmint
```

```
(termek (nev ?nev) (tomeg ?tomeg))
(test (> ?tomeg 30))
=>
(printout t ?nev " Tömege 30 kg fölötti: " ?tomeg " kg-os"
crlf)
)
```

A ténylistából látható, hogy két esetben teljesül a teszt feltétele:

```
CLIPS> (load "E:/egyetem/szakertoi
rendszerek/CLIPS/nagyobbmint.CLP")
Defining deftemplate: termek
Defining deffacts: butorok
Defining defrule: nagyobbmint +j+j
TRUE
CLIPS> (reset)
CLIPS> (run)
szekreny_9 Tömege 30 kg fölötti: 42 kg-os
szekreny_4 Tömege 30 kg fölötti: 35 kg-os
```

A logikai ÉS és VAGY `(and és or)` utasítások a szabályok bal oldali részében leggyakrabban vannak használva. Nézzük meg még egyszer a fenti példában a 'nagyobb mint' szabály bal oldali feltételét. Itt tulajdonképpen egyszerre teljesül a két feltétel, tehát köztük ÉS van. Az alábbi két utasítás tehát azonos:

```
(termek (nev ?nev) (tomeg ?tomeg))
(test (> ?tomeg 30))

(and
      (termek (nev ?nev) (tomeg ?tomeg))
(test (> ?tomeg 30))
)
```

Ha azokat a termékeket szeretnénk kilistázni, amelyek 40 kg-nál nagyobbak, VAGY 25 kg alattiak, az `'or'` utasítást használjuk:

```
(defrule kisebb-nagyobb
(termek (nev ?nev) (tomeg ?tomeg))
(or (test (> ?tomeg 40))
(test (< ?tomeg 25))
      )
=>
```

```
(printout t ?nev " Tömege 40 kg fölötti vagy 25 kg alati:
" ?tomeg " kg-os" crlf)
)

CLIPS> (load "E:/egyetem/szakertoi
rendszerek/CLIPS/kisebb-nagyobb.CLP")
Defining deftemplate: termek
Defining deffacts: butorok
Defining defrule: kisebb-nagyobb +j+j
+j+j
TRUE
CLIPS> (reset)
CLIPS> (run)
szekreny_9 Tömege 40 kg fölötti vagy 25 kg alatti: 42 kg-
os
iroasztal_7 Tömege 40 kg fölötti vagy 25 kg alatti: 22 kg-
os
iroasztal_1 Tömege 40 kg fölötti vagy 25 kg alatti: 17.5
kg-os
CLIPS>
```

A „létezik" univerzális logikai kvantornak a CLIPS-ben a megfelelő utasítása az `(exists…` szerkezet.

```
(defrule letezik-teny
(exists (termek (nev ?nev) (keszleten ?keszleten) ))
=>
(printout t "Létezik legalább egy termék a készleten"
crlf)
)
```

Amely abban az esetben teljesül, ha legalább egy tény kielégíti az exists-feltételt.

```
CLIPS> (load "E:/egyetem/Tantargyfejlesztes/szakertoi
rendszerek/CLIPS/letezik.CLP")
Defining deftemplate: termek
Defining deffacts: butorok
Defining defrule: letezik-teny +j+j
TRUE
CLIPS> (reset)
CLIPS> (run)
Létezik legalább egy termék a készleten
```

Az univerzális logikai kvantor használatához a `(forall…` utasítást használjuk, ahogy alább látható:

```
(defrule mindegyik-teny
(forall (nev ?nev) (sku_id ?sku_id))
=>
(printout t "Mindegyik terméknek van sku_id rendelve"
crlf)
)
```

Amely akkor teljesül, ha mindegyik terméknek a listában van `(sku_id` kód hozzárendelve.

```
Defining deftemplate: termek
Defining deffacts: butorok
Defining defrule: mindegyik-teny +j+j+j+j
TRUE
CLIPS> (reset)
CLIPS> (run)
Mindegyik terméknek van sku_id rend
```

3.11. Matematikai műveletek és függvények a CLIPS-ben

Az `integer` (egész) számtípusról `float` típusra történő alakítást a róla elnevezett átalakító függvénnyel tudjuk elvégezni:

```
CLIPS> (float 225.0)
225.0
CLIPS> (float -5)
5.0
CLIPS>
```

Hasonlóan, az `integer` utasítás egész számot alakít minden számtípusból:

```
CLIPS> (integer 225.0)
225
CLIPS> (integer -5)
5
CLIPS> (integer 225.55)
225
CLIPS>
```

A CLIPS-ben a matematikai műveletek elvégzésének sorrendje a következő: (operátor operandus1 operandus2 operandus3…), ahol az operátor lehet +*/-. Nézzünk erre néhány példát:

```
CLIPS> (+ 3 4)
7
CLIPS> (+ 10 7 4)
21
CLIPS> (/ 28 5)
5.6
CLIPS> (- 28 5)
23
CLIPS> (* 28 5)
140
```

Viszont az alábbi utasítás hibát ad ki:

```
CLIPS> (* 10 + 7 4)
[ARGACCES5] Function * expected argument #2 to be of type
integer
```

Helyesen, előbb az összeadást elvégezve:

```
CLIPS> (* 10 (+ 7 4))
110
```

A fentihez hasonlóan, a CLIPS-ben bármilyen összetett matematikai művelet megszerkeszthető, a zárójelek megfelelő használatával. Például, 147 + 3*8 - 37/7 ilyen formában számolható ki:

```
CLIPS> (+ 147 (- (* 3 8) (/ 37 7)))
165.7142857142
```

Szabályként az alábbi példa mutatja, hogyan használhatók műveletek:

```
(deffacts szamok (sz1 147) (sz2 35) (sz3 42))
(defrule muveletek
(sz1 ?a)
(sz2 ?b)
(sz3 ?c)
=>
(bind ?osszeg (+ ?a ?b ?c))
(bind ?szorzat (* ?a ?b ?c))
(printout t ?a " + " ?b " + " ?c "= " ?osszeg crlf)
(printout t ?a " * " ?b " * " ?c "= " ?szorzat crlf)
(assert (osszeg ?osszeg))
```

```
(assert (szorzat ?szorzat)))
```

A fenti szabályt futattva látható, hogy a két új tényt is létrehozta az összeg és szorzat értékekkel:

```
CLIPS> (load "E:/egyetem/szakertoi
rendszerek/CLIPS/osszeadas.CLP")
Defining deffacts: szamok
Defining defrule: muveletek +j+j+j+j
TRUE
CLIPS> (reset)
CLIPS> (run)
147 + 35 + 42= 224
147 * 35 * 42= 216090
CLIPS> (facts)
f-0     (initial-fact)
f-1     (sz1 147)
f-2     (sz2 35)
f-3     (sz3 42)
f-4     (osszeg 224)
f-5     (szorzat 216090)
For a total of 6 facts.
CLIPS>
```

A következőkben a számok elemzéséhez az `if… then…` utasítást használjuk. Első lépésben tekintsük meg az `if… then… else` általános szerkezetét:

```
(if <feltétel-kifejezés>
then
<utasítás>*
[else
< utasítás>*])
```

A következő programban egyszerű összehasonlítást végzünk az `if… then…` szerkezettel:

```
(deffacts szam (a 12))
(defrule feltetel-pelda
(a ?a)
 =>
(if (> ?a 10)
then
(printout t "Az " ?a" szám nagyobb mint 10. " crlf)
else
(printout t "Az " ?a" szám kisebb mint 10. " crlf)
    )
)
```

A fenti program futtatásának eredménye:

```
CLIPS> (load "E:/egyetem/Tantargyfejlesztes/szakertoi
rendszerek/CLIPS/if-then-pelda.CLP")
Defining deffacts: szam
Defining defrule: feltetel-pelda +j+j
TRUE
CLIPS> (reset)
CLIPS> (run)
Az 12 szám nagyobb mint 10.
CLIPS>
```

Természetesen a számolás során felhasználhatunk **globális változók**at is, amelyek minden szabályban elérhetők.

```
(deftemplate termek
(slot nev) (slot afa_tip) (slot sku_id) (slot tomeg) (slot
hossz) (slot szelesseg) (slot keszleten) (slot ar)
)
(deffacts butorok
(termek (nev iroasztal_1) (afa_tip 1) (sku_id 723) (tomeg
17.5) (hossz 145) (szelesseg 75) (keszleten 24) (ar 745))
(termek (nev iroasztal_7) (afa_tip 1) (sku_id 458) (tomeg
22) (hossz 152) (szelesseg 64) (keszleten 12) (ar 950))
(termek (nev szekreny_4) (afa_tip 1) (sku_id 224) (tomeg
35) (hossz 175) (szelesseg 64) (keszleten 32) (ar 2450))
(termek (nev szekreny_9) (afa_tip 1) (sku_id 112) (tomeg
42) (hossz 160) (szelesseg 80) (keszleten 17) (ar 3500))
(termek (nev konyv_12) (afa_tip 2) (sku_id 25) (tomeg
0.25) (hossz 20) (szelesseg 12) (keszleten 5) (ar 24))
(termek (nev konyv_37) (afa_tip 2) (sku_id 48) (tomeg 0.4)
(hossz 22) (szelesseg 10) (keszleten 8) (ar 42))
)

(defglobal
?*afa1* = 0.19
?*afa2* = 0.5
?*elso* = "19%"
?*masodik* = "5%"
)

(defrule afa-kiiras
(termek (nev ?nev) (afa_tip ?afa_tip) (ar ?ar))
=>
(if (eq ?afa_tip 1)
```

```
then (printout t ?nev "  ára: " ?ar "   " ?*elso*  "  ÁFÁ-
val számoljuk, végső ára: " (+ ?ar (* ?ar ?*afa1*)) crlf))
(if (eq ?afa_tip 2)
then (printout t ?nev "  ára: " ?ar "   " ?*masodik* "
ÁFÁ-val számoljuk, végső ára: " (+ ?ar (* ?ar ?*afa2*))
crlf))
)
```

A fenti programban a kétfajta áfa értéke és a hozzájuk rendelt szövegformátumok a globális változók, amelyeket az afa-kiiras nevű szabályban hívunk meg, aszerint, hogy termékenként milyen az `(afa_tip)` mező értéke.

```
CLIPS> (load "E:/egyetem/szakertoi
rendszerek/CLIPS/afa2021 11 1703.CLP")
Defining deftemplate: termek
Defining deffacts: butorok
Defining defglobal: afa1
Defining defglobal: afa2
Defining defglobal: elso
Defining defglobal: masodik
Defining defrule: afa-kiiras +j+j
TRUE
CLIPS> (reset)
CLIPS> (run)
konyv_37  ára: 42  5%  ÁFÁ-val számoljuk, végső ára: 63.0
konyv_12  ára: 24  5%  ÁFÁ-val számoljuk, végső ára: 36.0
szekreny_9  ára: 3500  19%  ÁFÁ-val számoljuk, végső ára:
4165.0
szekreny_4  ára: 2450  19%  ÁFÁ-val számoljuk, végső ára:
2915.5
iroasztal_7  ára: 950  19%  ÁFÁ-val számoljuk, végső ára:
1130.5
iroasztal_1  ára: 745  19%  ÁFÁ-val számoljuk, végső ára:
886.55
```

Egy szabályon belül a változókkal is végezhetők műveletek. Az alábbi példában növeljük a készletet:

```
(deftemplate termek
(slot nev)
(slot sku_id)
(slot tomeg)
(slot hossz)
(slot szelesseg)
(slot keszleten)
)
```

```
(deffacts butorok
(termek (nev iroasztal_1) (sku_id 723) (tomeg 17.5) (hossz
145) (szelesseg 75) (keszleten 24))
(termek (nev iroasztal_7) (sku_id 458) (tomeg 22) (hossz
152) (szelesseg 64)(keszleten 12))
(termek (nev szekreny_4) (sku_id 224) (tomeg 35) (hossz
175) (szelesseg 64)(keszleten 32))
(termek (nev szekreny_9) (sku_id 112) (tomeg 42) (hossz
160) (szelesseg 80)(keszleten 17))
)

(defrule elso
(initial-fact)
=>
(printout t
"Melyik termék készlete bővül? (iroasztal_1 / iroasztal_7
/ szekreny_4 / szekreny_7): " crlf)
(bind ?valasz (read))
(if (eq ?valasz iroasztal_1) then (assert (hozzaadas
iroasztal_1)))
(if (eq ?valasz iroasztal_7) then (assert (hozzaadas
iroasztal_7)))
(if (eq ?valasz szekreny_4) then (assert (hozzaadas
szekreny_4)))
(if (eq ?valasz szekreny_7) then (assert (hozzaadas
szekreny_7)))
)

(defrule keszletnoveles
?keszlet <- (hozzaadas ?nev)
?teny <- (termek (nev ?nev) (keszleten ?darab))
=>
(printout t "Készlet hány darab: " ?nev " termékkel bővül?
")
(bind ?valasz (read))
(modify ?teny (keszleten (+ ?darab ?valasz)))
(retract ?keszlet)
)
```

Ahogy látható, az `iroasztal_1` ténynél, a készlet értékét kettővel növeltük, a bővítés műveletét, a billentyűzetről beolvasott értékkel, a szabály `(+ ?darab ?valasz)` részében végezzük el.

```
CLIPS> (load "E:/egyetem/szakertoi
rendszerek/CLIPS/keszletnoveles.CLP")
Defining deftemplate: termek
```

```
Defining deffacts: butorok
Defining defrule: elso +j+j
Defining defrule: keszletnoveles +j+j+j
TRUE
CLIPS> (reset)
CLIPS> (run)
Melyik termék készlete bővül? (iroasztal_1 / iroasztal_7 /
szekreny_4 / szekreny_9):
iroasztal_1
Készlet hány darab: iroasztal_1 termékkel bővül? 2
CLIPS> (facts)
f-0     (initial-fact)
f-2     (termek (nev iroasztal_7) (sku_id 458) (tomeg 22)
(hossz 152) (szelesseg 64) (keszleten 12))
f-3     (termek (nev szekreny_4) (sku_id 224) (tomeg 35)
(hossz 175) (szelesseg 64) (keszleten 32))
f-4     (termek (nev szekreny_9) (sku_id 112) (tomeg 42)
(hossz 160) (szelesseg 80) (keszleten 17))
f-6     (termek (nev iroasztal_1) (sku_id 723) (tomeg
17.5) (hossz 145) (szelesseg 75) (keszleten 26))
For a total of 5 facts.
CLIPS>
```

3.12. Gyakran használt matematikai függvények a CLIPS-ben

Az alábbiakban néhány általánosan használt matematikai függvény működését tekintjük át. Kezdjük a kerekítéssel! A `round` tizedes számokat kerekít, a 0.5-ös határral, ahogyan alább látható.

```
CLIPS> (round 4.6)
5
CLIPS> (round 3.2)
3
CLIPS> (round 7.5)
7
CLIPS>
```

Az `abs` függvény egyetlen argumentumának abszolút értékét adja vissza.

```
CLIPS> (abs 14.7)
14.7
CLIPS> (abs -1526.0929)
1526.0929
CLIPS>
```

A `max` utasítás, egy számsorból, a legnagyobb szám argumentumának értékét adja vissza, a min hasonlóan a legkisebbet.

```
CLIPS> (max 14.7 11 42 28.7)
42
CLIPS> (min 14.7 11 42 28.7)
11
CLIPS>
```

A `div` utasítás hányadost, a `mod` utasítás az osztási maradékot számolja ki:

```
CLIPS> (div 12 6)
2
CLIPS> (div 12 5)
2
CLIPS> (mod 12 6)
0
CLIPS> (mod 12 5)
2
```

Egy szám négyzetgyökét az `sqrt` utasítással tudjuk kiszámítani, a hatványt pedig a `**` utasítással.

```
CLIPS> (sqrt 625)
25.0
CLIPS> (sqrt 81)
9.0
CLIPS> (sqrt 2)
1.4142135623731
CLIPS> (** 25 2)
625.0
CLIPS> (** 9 2)
81.0
CLIPS> (** 1.41 2)
1.9881
CLIPS>
```

Az `exp` függvény a természetes alapú exponenciális hatványát adja vissza, a megadott számnak, a `log` függvény a természetes (e alapú) logaritmusát, illetve a `log10` függvény a tízes alapú logaritmusát:

```
CLIPS> (exp 0)
1.0
CLIPS> (exp 1)
2.71828182845905
```

```
CLIPS> (exp 3)
20.0855369231877
CLIPS> (log 2.718281828459045)
1.0
CLIPS> (log 1)
0.0
CLIPS> (log 10)
2.30258509299405
CLIPS> (log10 1000)
3.0
CLIPS> (log10 1000000)
6.0
CLIPS>
```

Páros és páratlan számok tesztelésére az `evenp` és `oddp` függvényeket használjuk:

```
CLIPS> (oddp 7)
TRUE
CLIPS> (evenp 7)
FALSE
CLIPS> (evenp 12)
TRUE
CLIPS> (oddp 7)
TRUE
CLIPS> (oddp 12)
FALSE
CLIPS>
```

A `random` függvény egy „véletlenszerű" természetes szám értéket ad vissza, ahogy alább látható:

```
CLIPS> (random)
41
CLIPS> (random)
18467
CLIPS> (random)
6334
```

A `random` függvény által véletlenszerűen létrehozott természetes számot az előbbiekben bemutatott `mod` függvénnyel (mely az osztási maradékot számolja ki) vezetjük vissza 1 és 6 közé, mint a kockadobás esetében:

```
(deffacts indit (kockadobas igen))
(defrule kockadobas
(kockadobas igen)
```

```
=>
(bind ?dobas1 (+ (mod (random) 6) 1))
(bind ?dobas2 (+ (mod (random) 6) 1))
(printout t "A dobásod: " ?dobas1 " a gép dobása: "
?dobas2 crlf)
)
```

Amely futtatva minden esetben eltérő értékeket ad meg, például alább egy „játék menet”:

```
CLIPS> (load "E:/egyetem/szakertoi
rendszerek/CLIPS/kockadobas.CLP")
Defining deffacts: indit
Defining defrule: kockadobas +j+j
TRUE
CLIPS> (reset)
CLIPS> (run)
A dobásod:  5 a gép dobása: 3
CLIPS>
```

3.13. Ciklusok

A while ciklus szerkezete a CLIPS-ben:

```
(while <logikai-feltétel> [do]
<utasítás/utasítások>*
```

Az alábbi példában a `while-pelda` nevű szabály szerkezetében a `while` utasítással a kezdő értéknek megadott számig számoljuk ki a természetes számok összegét, ahogy csökkenő sorrendben összegzi, az `?ossz` változóban, majd kiírja.

```
(deffacts kezdo-ertekek (szam 7) (ossz 0))
(defrule while-pelda
(szam ?szam) (ossz ?ossz)
=>
(printout t "Számoljuk ki számok összegét, " ?szam " -ig:
" crlf)
(while (> ?szam 0)
(printout t "Összeg: " ?ossz crlf)
(bind ?ossz (+ ?ossz ?szam))
(bind ?szam (- ?szam 1))
      )
(printout t "A számok összege: "  ?ossz crlf)
```

```
)
```

A futtatás során minden lépésben kiírja az összeg aktuális értékét:

```
CLIPS> (load "E:/egyetem/szakertoi rendszerek/CLIPS/while-
pelda.CLP")
Defining deffacts: kezdo-ertekek
Defining defrule: while-pelda +j+j+j
TRUE
CLIPS> (reset)
CLIPS> (run)
Számoljuk ki számok összegét, 7 -ig:
Összeg: 0
Összeg: 7
Összeg: 13
Összeg: 18
Összeg: 22
Összeg: 25
Összeg: 27
A számok összege: 28
CLIPS>
```

A `break` utasítás azonnal leállítja az éppen iteráló while-ciklust. Hasonlóan működik a `progn` esetében vagy bizonyos példánykészlet-lekérdezési függvényeknél: `do-for-instance, do-for-all-instances` źs `delayed-do-for-all-instances`. A `break` nem használható argumentumként egy másik függvényhíváshoz, és más utasítások esetében nincs hatása.

```
(deffunction breake-pelda (?szam)
(bind ?i 0)
(while TRUE do
(if (>= ?i ?szam) then
(break))
(printout t ?i " " crlf)
(bind ?i (+ ?i 1))
  )
)
```

Beolvassuk és meghívjuk két értékre, amelyeknél az if feltételben a break megállítja a ciklust:

```
CLIPS> (load "E:/egyetem/szakertoi rendszerek/CLIPS/break-
pelda.CLP")
Defining deffunction: breake-pelda
```

```
TRUE
CLIPS> (breake-pelda 7)
0
1
2
3
4
5
6
CLIPS> (breake-pelda 1)
0
CLIPS>
```

A `loop-for-count` ciklus szerkezete a CLIPS-ben:

```
(loop-for-count <index-spec> [do] <action>*)
<index-spec> ::= <end-index> |
(<loop-variable> [<start-index> <end-index>])
<start-index> ::= <integer-expression>
<end-index>   ::= <integer-expression>
```

Például háromszor írjuk ki a "`Helló világ!`" szöveget, a for ciklussal:

```
(defrule ciklus
(initial-fact)
=>
(loop-for-count (?i 1 3) do
(printout t "Helló világ! " ?i crlf)
)
)

CLIPS>    (load    "E:/egyetem/Tantargyfejlesztes/szakertoi
rendszerek/CLIPS/ciklus2.CLP")
Defining defrule: ciklus +j+j
TRUE
CLIPS> (reset)
CLIPS> (run)
Helló világ! 1
Helló világ! 2
Helló világ! 3
CLIPS>
```

Szerkesszünk olyan programot, amely két ciklust egymásba ágyazva kiírja egy mátrix elemeinek sor-oszlop indexeit:

```
(defrule ciklus-pelda
(initial-fact)
=>
(loop-for-count (?i 1 5) do
(loop-for-count (?j 1 5) do
(printout t "(" ?i  ", " ?j    ") " )
      )
(printout  " " crlf)
)
)
```

Alább látható a futtatás eredménye:

```
CLIPS>              (load              "E:/egyetem/szakertoi
rendszerek/CLIPS/ciklus.CLP")
Defining defrule: ciklus-pelda +j+j
TRUE
CLIPS> (reset)
CLIPS> (run)
(1, 1) (1, 2) (1, 3) (1, 4) (1, 5)
(2, 1) (2, 2) (2, 3) (2, 4) (2, 5)
(3, 1) (3, 2) (3, 3) (3, 4) (3, 5)
(4, 1) (4, 2) (4, 3) (4, 4) (4, 5)
(5, 1) (5, 2) (5, 3) (5, 4) (5, 5)
```

3.14. A switch utasítás

A `switch` utasítás lehetővé teszi egy adott műveletcsoport (több műveletcsoport közül) végrehajtását egy meghatározott érték alapján.

```
(switch <teszt-kifejezés>
<eset - kifejezés >
< eset - kifejezés >+
[<alapértelmezett- kifejezés >])
< eset - kifejezés > ::=
(case < kifejezés -összehasonlítás> then <művelet>*)
< alapértelmezett- kifejezés > ::= (alapértelmezett
<művelet>*)
```

Például az alábbi függvény három esetre tartalmazza a `switch – case` szerkezet felosztását:

```
(defglobal ?*a* = 12)
(defglobal ?*b* = 14)
```

```
(defglobal ?*c* = 28)

(deffunction esetek (?ertek)
(switch ?ertek
(case ?*a* then a)
(case ?*b* then b)
(case ?*c* then c)
(default none)
  )
)
```

Amelyet ha meghívunk a parancssorból, az `a, b` és `c` globális változók értékeit adja vissza:

```
CLIPS>              (load              "E:/egyetem/szakertoi
rendszerek/CLIPS/esetek.CLP")
Defining defglobal: a
Defining defglobal: b
Defining defglobal: c
Defining deffunction: esetek
TRUE
CLIPS> (esetek 14)
b
CLIPS> (esetek 28)
c
CLIPS>
```

3.15. A CLIPS szabályaira vonatkozó eljárások

Több szabály esetében alapszabály, hogy ezek különböző nevűek kell hogy legyenek. A szabályok lefutási sorrendje független a CLIPS programon belüli sorrendtől.
A szabályok érvényességi sorrendjét változóval is tudjuk szabályozni. A (`salience <<szám>>`) olyan változót hoz létre, amelynél a magasabb értékkel rendelkező szabály fut le elsőként. Például nézzük az alábbi két szabályt:

```
(deftemplate termek
(slot nev) (slot sku_id) (slot tomeg) (slot hossz) (slot
szelesseg) (slot keszleten)
)
(deffacts butorok
```

```
(termek (nev iroasztal_1) (sku_id 723) (tomeg 17.5) (hossz
145) (szelesseg 75) (keszleten 24))
(termek (nev iroasztal_7) (sku_id 458) (tomeg 22) (hossz
152) (szelesseg 64) (keszleten 12))
(termek (nev szekreny_4) (sku_id 224) (tomeg 35) (hossz 175)
(szelesseg 64) (keszleten 32))
(termek (nev szekreny_9) (sku_id 112) (tomeg 42) (hossz 160)
(szelesseg 80) (keszleten 17))
)

(defrule szelesseg
(declare (salience 10))
(termek (nev ?nev) (szelesseg ?szelesseg) )
(test (> ?szelesseg 70))
=>
(printout t "Ez a termék szélessebb 70 cm-nél: " ?nev crlf)
)

(defrule tomeg
(declare (salience 20))
(termek (nev ?nev) (tomeg ?tomeg) )
(test (> ?tomeg 33) )
=>
(printout t "A termék súlyosabb 33 kg-nál: " ?nev crlf)
)
```

Ahogyan a `szabaly-sorrend.CLP` fájlban látható, a tomeg szabály sorrendben a második, viszont a 20-as `salience` érték miatt ez fog elsőnek lefutni:

```
CLIPS> (load "E:/egyetem/szakertoi
rendszerek/CLIPS/szabaly-sorrend.CLP")
Defining deftemplate: termek
Defining deffacts: butorok
Defining defrule: szelesseg +j+j
Defining defrule: tomeg +j+j
TRUE
CLIPS> (reset)
CLIPS> (run)
A termék súlyosabb 33 kg-nál: szekreny_9
A termék súlyosabb 33 kg-nál: szekreny_4
Ez a termék szélesebb 70 cm-nél: szekreny_9
Ez a termék szélesebb 70 cm-nél: iroasztal_1
CLIPS>
```

A következő szabályban a `?darab` nevű változó határozza meg, hogy hányszor ismétlődjön a `for` ciklus, illetve ezáltal azt is, hogy a `keszlet` nevű ténynek hány mezője legyen.

```
(defrule termekek
=>
 (printout t "Új termékek száma: ")
 (bind ?darab (read))
 (bind ?nev (create$))
 (loop-for-count ?darab
     (printout t "Termék neve: ")
     (bind ?nev (create$ ?nev (readline))))
 (assert (keszlet ?nev))
)

(defrule print-names
   (keszlet $?nev)
=>
  (printout t "A készleten levő termékek: " crlf)
  (foreach ?termek ?nev
    (printout t "   " ?termek crlf))
)
```

Futtatáskor, az alábbiakban, öt terméket viszünk be és rögzítünk a terméklistában:

```
CLIPS> (load "E:/egyetem/szakertoi
rendszerek/CLIPS/termekek-for-pelda.CLPCLP")
Defining defrule: termekek +j+j
Defining defrule: print-names +j+j
TRUE
CLIPS> (reset)
CLIPS> (run)
Új termékek száma: 5
Termék neve: laptop
Termék neve: smartphone
Termék neve: keyboard
Termék neve: camera
Termék neve: adahordozo
A készleten levő termékek:
   laptop
   smartphone
   keyboard
   camera
   adathordozo
CLIPS> (facts)
f-0     (initial-fact)
```

```
f-1   (keszlet  "laptop"  "smartphone"  "keyboard"  "camera"
"adathordozo")
For a total of 2 facts.
CLIPS>
```

3.16. Függvények

A `deffunction` szerkezete:

```
(deffunction <name> [<comment>]
    (<regular-parameter>* [<wildcard-parameter>])
    <action>*)
<regular-parameter>   ::= <single-field-variable>
<wildcard-parameter>  ::= <multifield-variable>
```

Két összehasonlító függvényt láthatunk alább, az első függvény akkor ad `TRUE` választ, ha a második szám a nagyobb. A `nagyobb-szoveg` függvény pedig hasonlóan a szövegek hosszát teszteli. A függvény meghívása a parancssorból vagy a programon belül a megfelelő számú argumentummal történhet. A harmadik példában ezért ad ki hibát, mert nem talál az argumentumok száma, és meg is nevezi, hogy a `nagyobb-szoveg` nevű függvénynek két argumentuma van.

```
CLIPS> (deffunction nagyobb-szam (?szam1 ?szam2)
   (< ?szam1 ?szam2 ))
CLIPS> (nagyobb-szam 14 28)
TRUE
CLIPS> (deffunction nagyobb-szoveg (?sz1 ?sz2)
   (< (str-compare ?sz1 ?sz2 ) 0))
CLIPS> (nagyobb-szoveg "elso" "masodik")
TRUE
CLIPS> (nagyobb-szoveg "elso" "masodik" "harmadik")
[ARGACCES4] Function nagyobb-szoveg expected exactly 2
argument(s)
CLIPS>
```

A `return` utasítás azonnal leállítja az éppen futó `deffunction`-t, általános függvénymódszert, üzenetkezelőt, a defrule jobb oldali részét vagy a következő lekérdezési függvényeket: `do-for-instance`, `do-for-all-instances` és `delayed-do-for-all-instances`. A `return` nemcsak a deffunctionsnál használható, hanem a metódusok és a defrules jobb oldali részén is. Ha az argumentum nincs meghatározva, nincs visszatérési értéke sem a `return`nek. Ha azonban szerepel egy argumentum, akkor annak kiértékelését a `deffunction`

metódus vagy üzenetkezelő visszatérési értékeként adjuk meg. A `return` nem használható argumentumként egy másik függvényhíváshoz.

```
(deffunction elojel (?szam)
(if (> ?szam 0) then
(return pozitiv))
(if (< ?szam 0) then
(return negativ))
  )
```

Amely így hívható meg a parancssorból:

```
CLIPS> (load "E:/egyetem/szakertoi
rendszerek/CLIPS/elojel.CLP")
Defining deffunction: elojel
TRUE
CLIPS> (elojel 14)
pozitiv
CLIPS> (elojel -2)
negativ
CLIPS> (elojel 28)
pozitiv
CLIPS>
```

A függvényeket a `get-deffunction-list` utasítással tudjuk kilistázni:

```
CLIPS> (deffunction elso ())
CLIPS> (deffunction masodik ())
CLIPS> (get-deffunction-list)
(elso masodik)
CLIPS>
```

Az alábbi függvény egy természetes szám faktoriálisát számolja ki. Figyeljük meg, hogy az ötödik sorban a függvény saját magát hívja meg:

```
(deffunction faktorialis (?n)
   (if (= ?n 0) then
       1
      else
    (* ?n (faktorialis (- ?n 1)) )
  )
)
```

Futtatása a parancssorból és a 8! értékének kiszámítása a következőképpen történik:

```
CLIPS>(load "E:/egyetem/szakertoi
rendszerek/CLIPS/faktorialis.CLP")
Defining deffunction: faktorialis
TRUE
CLIPS> (faktorialis 8)
40320
CLIPS>
```

Hasonlóan szerkesztettük meg *n* elem *k*-ad rendű kombinációjának a kiszámítását:

```
(deffunction faktorialis (?n)
   (if (= ?n 0) then
        1
       else
     (* ?n (faktorialis (- ?n 1)) )
  )
)

(deffunction kombinacio (?n ?k)
    (/ (faktorialis  ?n)
     (* (faktorialis (- ?n ?k)) (faktorialis ?k))
    )
)
```

Itt a `kombinacio` nevű függvény három esetben hívja meg a `faktorialis` függvényt, és az `?n` változó mindkét függvényen belül lokális. Figyeljük meg, hogy a `deffunction` szerkezetben, a második függvénynél, az érték, amit visszaad, a kombináció kiszámítására használt képlettel van kiszámolva. Mindkét függvény akkor működik helyesen, ha pozitív természetes számokat adunk meg:

```
CLIPS> (load "E:/egyetem/szakertoi
rendszerek/CLIPS/kombinacio.CLP")
Defining deffunction: faktorialis
Defining deffunction: kombinacio
TRUE
CLIPS> (kombinacio 5 2)
10.0
CLIPS> (kombinacio 12 3)
220.0
```

Az alábbi kérdező függvény `do-while` ciklusa addig teszi fel a kérdést, amíg a beírt válasz találni fog az `elfogadott-valaszok`ban meghatározottakéval:

```
(deffunction kerdezes (?kerdes $?elfogadott-valaszok)
(printout t ?kerdes crlf)
```

```
(bind ?valasz (read)) (if (lexemep ?valasz) then (bind
?valasz (lowcase ?valasz)))
(while (not (member ?valasz ?elfogadott-valaszok)) do
(printout t ?kerdes crlf)
(bind ?valasz (read)) (if (lexemep ?valasz) then (bind
?valasz (lowcase ?valasz))))
?valasz)

(defrule kerd
(not (kerd ?))
=>
(bind ?felelet (kerdezes  "Milyen turisztikai egység?
(kemmping/panzio/hotel)" kemmping panzio hotel))
(if (eq ?felelet kemmping ) then (assert (kerd kemmping
)))
(if (eq ?felelet panzio ) then (assert (kerd panzio )))
(if (eq ?felelet hotel) then (assert (kerd hotel)))
)
```

A `kerd` nevű szabály meghívja a függvényt, a válasz beolvasása után a nagybetűket kisbetűvé is alakítja. Ahogy látható, a harmadik válasz a helyes, amelynél leáll a kérdezőciklus.

```
CLIPS> (load "E:/egyetem/szakertoi
rendszerek/CLIPS/askq04.CLP")
Defining deffunction: kerdezes
Defining defrule: kerd +j+j
TRUE
CLIPS> (reset)
CLIPS> (run)
Milyen turisztikai egység? (kemmping/panzio/hotel)
agroturisztikai panzio
Milyen turisztikai egység? (kemmping/panzio/hotel)
hostel
Milyen turisztikai egység? (kemmping/panzio/hotel)
Hotel
CLIPS> (facts)
f-0     (initial-fact)
f-1     (kerd hotel)
For a total of 2 facts.
CLIPS>
```

3.17. Példa egy CLIPS-ben szerkesztett szakértői rendszerre

Az alábbi, 10 szabályt tartalmazó szakértői rendszer ténylistájában a program beolvasásakor 5 vállalkozás adatai vannak megadva. Az ágazat, forgalom, alkalmazottak száma, innováció jelenléte és profit külön mezőkben van tárolva, a `deftemplate`-szerkezetben, és mindegyikhez egy elemzési szabály tartozik. A `get-vallalkozas` nevű szabállyal egy új vállalkozást is beolvashatunk és rögzíthetünk a ténylistában. Az új vállalkozás hozzzáadása után, akárcsak egy-egy jellemző elemzése után, lehetőség van újra futtatni az `elemzes` nevű szabályt, amelynek így összesen 7 tény-feltétele van az `or` (vagy) szerkezettel megadva.

```
(deftemplate vallalkozas (slot nev) (slot agazat) (slot
forgalom) (slot alkalmazottak) (slot innovacio) (slot
profit)
)

(deffacts vallalkozasok
(vallalkozas (nev Progress) (agazat ipar) (forgalom
200000)(alkalmazottak 18)(innovacio van)(profit 30000))
(vallalkozas (nev IT2000) (agazat IT)(forgalom
15000)(alkalmazottak 50)(innovacio nincs)(profit 8000))
(vallalkozas (nev IMPEX)(agazat mezogazdasag)(forgalom
12000)(alkalmazottak 100)(innovacio nincs)(profit 20000))
(vallalkozas (nev RETRO)(agazat szolgaltatas)(forgalom
18000)(alkalmazottak 8)(innovacio nincs)(profit 60000))
(vallalkozas (nev Max)(agazat kereskedelem)(forgalom
20000)(alkalmazottak 15)(innovacio nincs)(profit 40000))
)

(defrule elso
(initial-fact)
=>
(printout t "Új vállalkozás hozzáadása? (igen/nem): ")
(bind ?valasz (read))
(if (eq ?valasz igen) then (assert (hozzaadas igen)))
(if (eq ?valasz nem) then (assert (elemzes  igen)))
)

(defrule get-vallalkozas
   (hozzaadas igen)
   =>
   (printout t "Milyen nevű vállalkozást szeretne
hozzáadni?: ")
   (bind ?valasz1 (read))
   (printout t "Milyen ágazatban működik a vállalkozás
(ipar/IT/mezogazdasag/szolgaltatas/kereskedelem)?: ")
   (bind ?valasz2 (read))
```

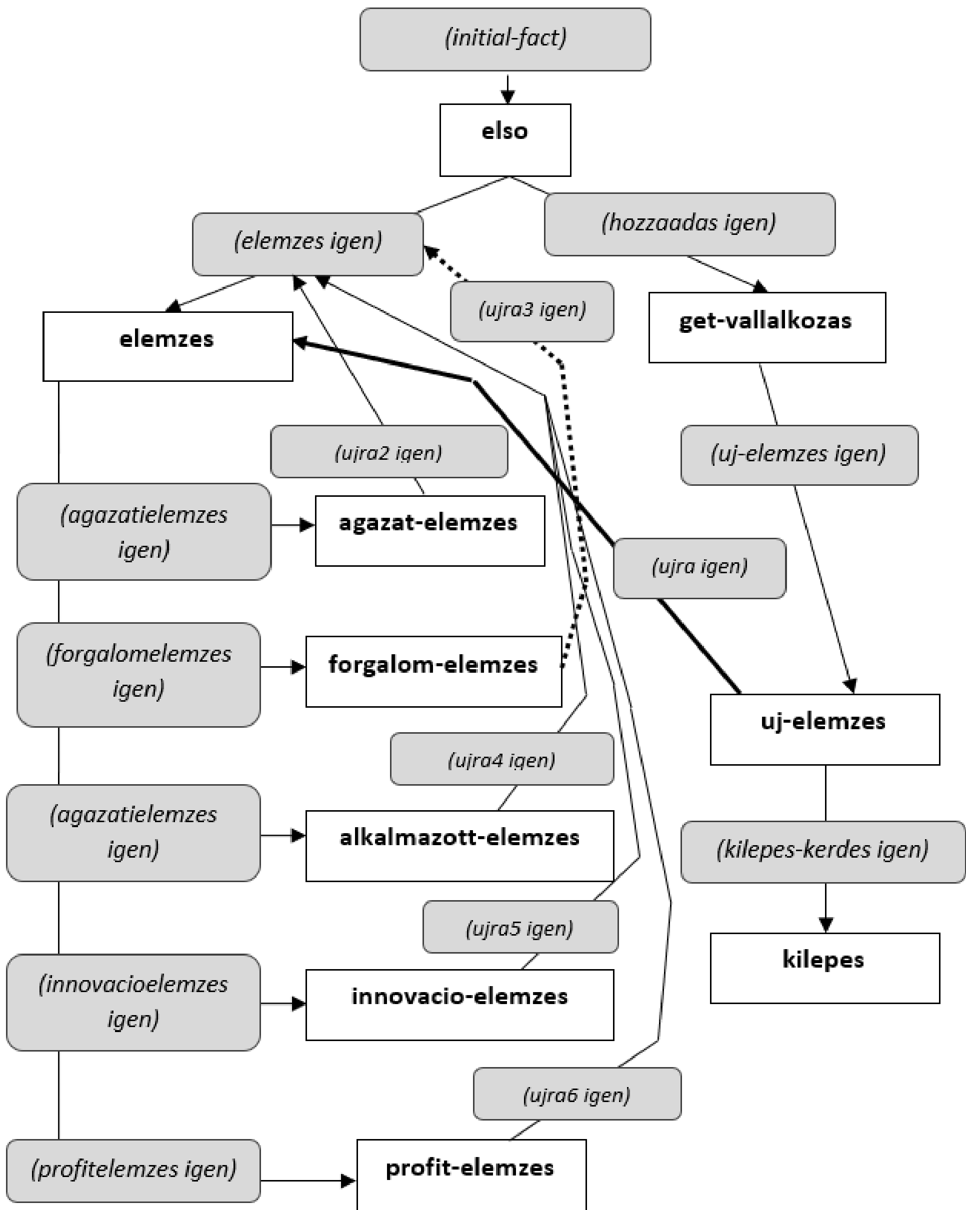
```
   (printout t "Mekkora a vállalkozás forgalma (lejben)?: 
")
   (bind ?valasz3 (read))
   (printout t "Hány alkalmazott van?: ")
   (bind ?valasz4 (read))
   (printout t "Van innováció a vállalkozásban 
(van/nincs)?: ")
   (bind ?valasz5 (read))
   (printout t "Mekkora a profit (lejben)?: ")
   (bind ?valasz6 (read))
   (assert (vallalkozas (nev ?valasz1)(agazat 
?valasz2)(forgalom ?valasz3)(alkalmazottak ?valasz4)
(innovacio ?valasz5) (profit ?valasz6)))

(assert (uj-elemzes igen))
)

(defrule uj-elemzes
   (uj-elemzes igen)
=>
   (printout t "Szeretne elemzést elvégezni (igen/nem)" 
crlf)
   (bind ?valasz (read))
   (if (eq ?valasz igen) then (assert (ujra igen)))
   (if (eq ?valasz nem) then (assert (kilepes-kerdes 
igen)))
)

(defrule elemzes
(or (elemzes  igen) (ujra igen) (ujra2 igen) (ujra3 igen)
(ujra4 igen) (ujra5 igen) (ujra6 igen))
=>
(printout t "Milyen jellemzők érdeklik? 
(agazat/forgalom/alkalmazottak/innovacio/profit/semmilyen)
: ")
(bind ?valasz (read))
(if (eq ?valasz agazat) then (assert (agazatielemzes 
igen)))
(if (eq ?valasz forgalom) then (assert (forgalomelemzes 
igen)))
(if (eq ?valasz alkalmazottak) then (assert 
(alkalmazottelemzes igen)))
(if (eq ?valasz innovacio) then (assert (innovacioelemzes 
igen)))
(if (eq ?valasz profit) then (assert (profitelemzes 
igen)))
```

```
(if (eq ?valasz semmilyen) then (assert (kilepes-kerdes
igen)))
)

(defrule agazat-elemzes
   (and (agazatielemzes igen)
      (vallalkozas (nev ?valtozo) (agazat ipar))
   )
=>
   (printout t "Iparban működő vállalat: " ?valtozo crlf)
 (assert (ujra2 igen))
)

(defrule forgalom-elemzes
   (and (forgalomelemzes igen)
   (vallalkozas (nev ?valtozo)  (forgalom ?f) )
   (test (>= ?f 100000))
   )
=>
  (printout t "100000 lej fölötti forgalommal rendelkezo
vállalat: " ?valtozo crlf)
 (assert (ujra3 igen))
)

(defrule alkalmazott-elemzes
   (and (alkalmazottelemzes igen)
   (vallalkozas (nev ?valtozo)  (alkalmazottak ?a) )
   (test (<= ?a 10))
   )
=>
  (printout t "10 alkalmazottnál kevesebbel működő,
kisvállalkozás: " ?valtozo crlf)
 (assert (ujra4 igen))
)

(defrule innovacio-elemzes
   (and (innovacioelemzes igen)
   (vallalkozas (nev ?valtozo)  (innovacio van) )
   )
=>
  (printout t "Van innováció, ennél a vállalkozásnál: "
?valtozo crlf)
 (assert (ujra5 igen))
)

(defrule profit-elemzes
   (and (profitelemzes igen)
```

```
    (vallalkozas (nev ?valtozo)  (profit ?p) )
    (test (>= ?p 50000))
    )
=>
   (printout t "50000 lej fölötti profittal rendelkezo
vállalat: "  ?valtozo crlf)
 (assert (ujra6 igen))
)

(defrule kilepes
    (kilepes-kerdes igen)
=>
   (printout t "Az elemzés véget ért. " crlf)
)
```

A 8. ábrán vastagított betűkkel vannak jelölve a szabályok, dőlt betűkkel láthatók a tények, melyek ezek feltételeiként működnek. Az `elemzes` nevű szabályból öt elágazással választhatunk a vállalkozások jellemzőire vonatkozó szabályok közül.

8. ábra. Egy vállalatelemző szakértői rendszer szerkezete a CLIPS-ben



Forrás: saját szerkesztés

A program futtatása az alábbi eredménnyel történik, ha egy vállalkozást is hozzáadunk:

```
CLIPS> (load "E:/egyetem/Tantargyfejlesztes/szakertoi
rendszerek/CLIPS/vallalkozas-elemezes.CLP")
Defining deftemplate: vallalkozas
Defining deffacts: vallalkozasok
Defining defrule: elso +j+j
Defining defrule: get-vallalkozas +j+j
Defining defrule: uj-elemzes +j+j
Defining defrule: elemzes +j+j
Defining defrule: agazat-elemzes +j+j+j
Defining defrule: forgalom-elemzes +j+j+j
Defining defrule: alkalmazott-elemzes +j+j+j
Defining defrule: innovacio-elemzes +j+j+j
Defining defrule: profit-elemzes +j+j+j
Defining defrule: kilepes +j+j
TRUE
CLIPS> (reset)
CLIPS> (run)
Új vállalkozás hozzáadása? (igen/nem): igen
Milyen nevű vállalkozást szeretne hozzáadni?: madmax
Milyen ágazatban működik a vállalkozás
(ipar/IT/mezogazdasag/szolgaltatas/kereskedelem)?: IT
Mekkora a vállalkozás forgalma (lejben)?: 80000
Hány alkalmazott van?: 25
Van innováció a vállalkozásban (igen/nem)?: nincs
Mekkora a profit (lejben)?: 30000
Szeretne elemzést elvégezni (igen/nem)
igen
Milyen jellemzők érdeklik?
(agazat/forgalom/alkalmazottak/innovacio/profit/semmilyen)
: agazat
Iparban működő vállalat: Progress
Milyen jellemzők érdeklik?
(agazat/forgalom/alkalmazottak/innovacio/profit/semmilyen)
: forgalom
100000 lej fölötti forgalommal rendelkezo vállalat:
Progress
Milyen jellemzők érdeklik?
(agazat/forgalom/alkalmazottak/innovacio/profit/semmilyen)
: alkalmazottak
10 alkalmazottnál kevesebbel működő, kisvállalkozás: RETRO
Milyen jellemzők érdeklik?
(agazat/forgalom/alkalmazottak/innovacio/profit/semmilyen)
: innovacio
Van innováció ennél a vállalkozásnál: Progress
```

```
Milyen jellemzők érdeklik?
(agazat/forgalom/alkalmazottak/innovacio/profit/semmilyen)
: profit
50000 lej fölötti profittal rendelkezo vállalat: RETRO
Milyen jellemzők érdeklik?
(agazat/forgalom/alkalmazottak/innovacio/profit/semmilyen)
: semmilyen
Az elemzés véget ért.
CLIPS>
```

A tényeket kilistázva látható egyrészt a 7-es számmal a ténylistához hozzáadott új vállalkozás, másrészt a szerkezeten belül az egyes szabályokat elindító tények, ahogyan a 3.17 ábrán is láthattuk.

```
CLIPS> (facts)
f-0     (initial-fact)
f-1     (vallalkozas (nev Progress) (agazat ipar)
(forgalom 200000) (alkalmazottak 18) (innovacio van)
(profit 30000))
f-2     (vallalkozas (nev IT2000) (agazat IT) (forgalom
15000) (alkalmazottak 50) (innovacio nincs) (profit 8000))
f-3     (vallalkozas (nev IMPEX) (agazat mezogazdasag)
(forgalom 12000) (alkalmazottak 100) (innovacio nincs)
(profit 20000))
f-4     (vallalkozas (nev RETRO) (agazat szolgaltatas)
(forgalom 18000) (alkalmazottak 8) (innovacio nincs)
(profit 60000))
f-5     (vallalkozas (nev Max) (agazat kereskedelem)
(forgalom 20000) (alkalmazottak 15) (innovacio nincs)
(profit 40000))
f-6     (hozzaadas igen)
f-7     (vallalkozas (nev madmax) (agazat IT) (forgalom
80000) (alkalmazottak 25) (innovacio nincs) (profit
30000))
f-8     (uj-elemzes igen)
f-9     (ujra igen)
f-10    (agazatielemzes igen)
f-11    (ujra2 igen)
f-12    (forgalomelemzes igen)
f-13    (ujra3 igen)
f-14    (alkalmazottelemzes igen)
f-15    (ujra4 igen)
f-16    (innovacioelemzes igen)
f-17    (ujra5 igen)
```

```
f-18    (profitelemzes igen)
f-19    (ujra6 igen)
f-20    (kilepes-kerdes igen)
For a total of 21 facts.
CLIPS>
```

Kérdések

1. Hozzon létre CLIPS-tényeket a következő kijelentésekre: termék megfelelő; vásárló elégedett; kifizetés teljesítve; rendelés kiszállítva; igen.
2. Hozzon létre a CLIPS-ben `deffacts` ténylistát könyvtermékekre, amely a következő mezőket tartalmazza: szerző, cím, ár, kiadási év, kategória, olvasói értékelés, és töltse fel hét könyvvel. Példa egy könyv adataira: Kodolányi János, A vas fiai, 2013, történelmi regény, 0.93.
3. Hozzon létre CLIPS-tényeket a következő kijelentésekre: termék megfelelő; vásárló elégedett; kifizetés teljesítve; rendelés kiszállítva; igen.
4. Hozzon létre a CLIPS-ben `deffacts` ténylistát könyvtermékekre, amely a következő mezőket tartalmazza: szerző, cím, ár, kiadási év, kategória, olvasói értékelés, és töltse fel hét könyvvel. Példa egy könyv adataira: Kodolányi János, A vas fiai, 2013, történelmi regény, 0.93.
5. Hozzon létre egy hét mezőből álló ténylistát, a1, … a7 adatokkal, majd törölje a harmadik, ötödik elemet, írja ki a tartalmát, és cserélje a negyedik elemet b1-re.
6. A megfelelő függvények használatával írassa ki a „VIP üzleti partner." szöveg elemeinek számát, a karaktereket 3-tól 7-ig, alakítsa kisbetűvé, illetve nagybetűvé.
7. Hozzon létre egy `deffacts` ténylistát egy terméktípusra, például személygépkocsik, fényképezőgépek, videókártyák vagy számítógép-tartozékok, 6-8 jellemzőnek megfelelő mezővel, 8-10 termékkel.
8. Az előző kérdésnél létrehozott ténylistát bővítse három új termékkel, az `(assert)` utasítást felhasználva, majd a `(facts)` utasítással ellenőrizze, hogy a tények létrejöttek.
9. Szerkesszen olyan szabályt, amely a fogyasztói értékelések alapján rangsorol termékeket, 0.90 fölött „prémium kategória", 0.75 – 0.90 között „jó értékelés", 0.5 – 0.75 között „közepes", 0.5 alatt „alacsony" kategóriákkal.
10. Szerkesszen olyan szabályokat, amelyek termékek ténylistáiból megkeresi a legolcsóbb, legdrágább terméket.

11. Szerkesszen egy olyan szabályt, amely a fejezetben tárgyalt (`vallalkozas...`) ténylistában, megkeresi a kisvállalkozásokat (a 10 alkalmazottal kevesebbel működő vállalatokat).
12. A fejezetben tárgyalt (`deffacts butorok...`) ténylistára szerkesszen egy olyan szabályt, amely megkeresi a készleten lévő összes típust.
13. Felhasználva az összehasonlítás lehetőségeit, a (`deffacts butorok...`) ténylistára szerkesszen olyan szabályokat, amely páronként összehasonlítja a termékeket tömeg, hossz és szélesség alapján.
14. Szerkesszen olyan szabályt, amely az előző pontban létrehozott, könyvek ténylistára kategóriánként, majd szerzőnként összegzi a készleten levő darabszámot.
15. Globális 9%-os áfával szerkessze meg a könyvek ténylistaárainak szabállyal történő módosítását.
16. Szerkesszen olyan szabályokat, amelyek random függvénnyel a pénzfeldobást, illetve a „kő-papír-olló" játékot szimulálja.
17. Szerkesszen olyan szabályokat, amelyek egy 12 havi idősorban megkeresik a minimum, maximum, illetve valamilyen feltétel szerinti értékeket (például havi foglalkoztatási ráta értéke 65% fölött).
18. Szerkesszen olyan szabályokat, amelyek a mátrix formában írják ki egy ténylista termékjellemzőit.
19. Használja a `switch – brake` utasításokat egy ténylista termékjellemzőinek felhasználó elvárásai szerinti kiválasztására.

4. SZAKÉRTŐIRENDSZER-MEGOLDÁSOK PYTHONBAN

4.1. Az Experta szakértői rendszer

A jelenleg egyik legsokoldalúbb és legdinamikusabban fejlődő programozási nyelv, a Python is teret nyújtott a szakértői rendszerek készítésének, több fejlesztési projektben, mint a Pyknow[11], Experta[12], PyKE[13] és mások[14]. Ezek közül az Experta szakértői rendszert tárgyaljuk az alábbiakban, amely egyúttal betekintést enged a szakértői rendszerek jelenlegi helyzetére és felhasználási lehetőségeire.
Erősségei közé tartozik, hogy teljes mértékben Python-alapú, logikai programozást lehetővé tevő, tudásalapú, következtető rendszer. A CLIPS-szel és Prologgal ellentétben, bármilyen Python-környezetben elérhető, mivel az Experta tulajdonképpen egy Python-modul. Ezért bármilyen Python-utasítás vagy függvény beilleszthető az Expertában.
A Python 3.8 környezetben mutatjuk be a továbbiakban az Experta használatát, vagy használhatjuk a PyCharm IDE-et[15]. Telepítése a parancssorból az alábbi utasítással történik:

```
python -m pip install experta
```

A logikai kijelentések és kapcsolatok az Experta esetében is tényekkel és szabályokkal vannak leírva. A tények létrehozására a `declare` vagy `Fact` utasítást használjuk, mivel a Pythonban az `assert`-et kulcsszóként használjuk. Hozzuk létre az első tényeket az Experta modulból importált `Fact`tel:

```
from experta import Fact
tenyek1 = Fact('laptop', 'monitor', 'keyboard')
tenyek2 = Fact(laptop = 3850, monitor = 2450, keyboard =
350)
>>> tenyek1[1]
'monitor'
>>> tenyek1[0]
'laptop'
```

[11] https://awesomeopensource.com/project/buguroo/pyknow
[12] https://github.com/nilp0inter/experta/tree/develop/docs
[13] http://pyke.sourceforge.net/index.html
[14] https://awesomeopensource.com/projects/expert-system/python
[15] https://www.jetbrains.com/pycharm/

```
>>> tenyek2
Fact(laptop=3850, monitor=2450, keyboard=350)
>>>>> tenyek2['laptop']
3850
>>> tenyek2['keyboard']
350
>>>
```

Alosztályok is létrehozhatók a `Fact` alatt, melyekhez tényeket tudunk hozzárendelni:

```
class Vallalkozas(Fact):
    """Vallalkozás tények."""
    pass
class Partner(Fact):
    """Üzleti prtnerek."""
    pass

teny1 = Vallalkozas('Madmax KFT')
teny2 = Partner('Stabile')

>>> teny1
Vallalkozas('Madmax KFT')
>>> teny2
Partner('Stabile')
>>>
```

A CLIPS-ből ismert `deffacts` szerkezethez nagyon hasonló az Expertaban használt `DefFacts`, amellyel egyszerre deklarálhatunk egy sor tényt, a `yield` utasítással a megfelelő mezőkkel. A Python-generátorok témakörünknél hosszabb tanulmányozást igényelnek, maradjunk azonban annyiban, hogy a `yield`, minden generátor-függvény esetében, a „hagyományos" függvények `return`jéhez hasonlóan meghatározza, hogy a függvény milyen értéket kell hogy visszaadjon. Esetünkben a tény mezőit.

```
@DefFacts()
def needed_data():
    yield Fact(premium_termek='laptop1')
    yield Fact(reducedprice_termek='laptop2')
    yield Fact(return_termek='keyboard1')
    yield Fact(select_termek='keyboard2')
```

Ezen logika mentén a tudásbázis a generátorfüggvénnyel nem jöhet létre. Tehát a szakértői rendszer tudásbázisa, az Experta-ban, csak akkor jön létre, amikor az

engine.reset() utasítással "visszaállítjuk" a kezdeti tudásbázist egy `class`-ban, ahogyan alább a tudásbázis szerkesztésénél majd látni fogjuk.
Nézzük meg a vállalkozások esetében, hogy néz ki a CLIPS `slot`jainak megfelelő `yield`en belül a `Field`ek szerkezete. Az előző fejezetből ismert CLIPS `deffacts` szerkezetében a vállalatok nevét, forgalmát, profitját és alkalmazottainak számát tartalmazza, a 2-es számú Mellékletben megtalálható `vallalkozas.clp`-ben. Az alábbi példában ugyanezt szerkesztettük meg az Expertaban:

```
vallalkozas.py
from experta.engine import KnowledgeEngine
from experta.deffacts import DefFacts
from experta.fact import Fact, Field

class vallalkozas(Fact):
    nev = Field(str, mandatory=True)
    forgalom = Field(int, mandatory=True)
    profit = Field(int, mandatory=True)
    alkalmazottak = Field(int, mandatory=True)

class DefFact(KnowledgeEngine):
    @DefFacts()
    def init_vallalkozas(self):
        yield vallalkozas(nev='IT Kreatív KFT',
                          forgalom=6400000,
                          profit=280000,
                          alkalmazottak=32)
        yield vallalkozas(nev='ABC Kft',
                          forgalom=6400000,
                          profit=56000,
                          alkalmazottak=26)
        yield vallalkozas(nev='Próba KFT',
                          forgalom=1250000,
                          profit=12000,
                          alkalmazottak=6)
        yield vallalkozas(nev='Havasigyopár Kft',
                          forgalom=7500000,
                          profit=35000,
                          alkalmazottak=8)
        yield vallalkozas(nev='IT Szervíz Kft',
                          forgalom=175000,
                          profit=14000,
                          alkalmazottak=3)
```

A szabályoknak az Expertaban is, hasonlóan az eddig tárgyalt nyelvekhez, két összetevője van: a logikai feltételeket tartalmazó LHS (bal oldal) és a szabály

aktiválásával sorra kerülő, végrehajtást tartalmazó RHS (jobb oldal). Ahhoz, hogy a szabály teljesüljön, a feltételként megadott tényeknek igaznak kell lenniük.

```
    @Rule <<LHS - feltételek>>
    def <<szabály neve>>:
        <<LHS - utasítások>>
```

A logikai feltételek szerkesztésénél felhasználhatjuk a logikai ÉS, VAGY és NEM utasításokat:

```
@Rule(
    AND(
        OR(Termek('premium'),
           Termek('selected')),
        NOT(Fact('non-selection'))
    )
)
def kiemelt_termek():
    """ Prémium termék """
    termek_felar()
```

A következőkben egy tudásbázisban összefoglalva mutatjuk be a tények és szabályok használatát. A tudásbázis általános szerkezetével kapcsolatban fontos megjegyezzük a Pythonban használt bekezdések[16] (angolul: `indentation`) jelölését, amellyel a kódok kezdetét jelöljük, és a függvények, utasítások alegységeinél kiemelt szerepe van. Tehát az Expertaban, a `class`, `@DefFacts`, `@Rule` utasítások szerkezetében, a fenti példákban, illetve alább a tudásbázispéldában látható a bekezdések helyes használata, amely négy szóközből áll.

Első lépésként egy `Vallalkozasok` nevű tudásbázist hozunk létre, melynek osztályán belül négy szabály van, három a tények beolvasására (`beolvas_hely`, `beolvas_forgalom`, `beolvas_nev`, `kiir`) és egy a kiírásra. A példa az Experta dokumentáció alapján van megszerkesztve[17]:

```
from experta import *

class Vallalkozasok(KnowledgeEngine):
    @DefFacts()
    def _initial_action(vallalat):
        yield Fact(action="beolvas")
```

[16] https://pythonexamples.org/python-indentation/

[17] Forrás: https://experta.readthedocs.io/en/latest/thebasics.html#knowledgeengine

```
    @Rule(Fact(action='beolvas'),
          NOT(Fact(hely=W())))
    def beolvas_hely(vallalat):
        vallalat.declare(Fact(hely=input("Címe: ")))

    @Rule(Fact(action='beolvas'),
          NOT(Fact(forgalom=W())))
    def beolvas_forgalom(vallalat):
        vallalat.declare(Fact(forgalom=input("Forgalom:
")))

    @Rule(Fact(action='beolvas'),
          NOT(Fact(nev=W())))
    def beolvas_nev(vallalat):
        vallalat.declare(Fact(nev=input("Vállalkozás neve:
")))

    @Rule(Fact(action='beolvas'),
          Fact(nev=MATCH.nev),
          Fact(hely=MATCH.hely),
          Fact(forgalom=MATCH.forgalom))
    def kiir(vallalat, nev, hely, forgalom):
        print("A %s vállalat címe: %s, forgalma: %s"  %
(nev, hely, forgalom))

engine = Vallalkozasok()
engine.reset()  # Visszaállítjuk az eredeti tényeket.
engine.run()  # Futtatás.
```

A parancssorból futtatva bevisszük egy vállalat adatait, majd kiírjuk a tények listáját:

```
>>>
==================== RESTART: vallalkozasok.py
====================
Vállalkozás neve: Madmax Kft
Címe: Csíkszereda, Szabadság tér, Nr 1
Forgalom: 7500000
A Madmax Kft vállalat címe: Csíkszereda, Szabadság tér, Nr
1, forgalma: 7500000
>>>
>>> engine.facts
FactList([(0, InitialFact()), (1, Fact(action='beolvas')),
(2, Fact(nev='Madmax Kft')), (3, Fact(hely='Csíkszereda,
Szabadság tér, Nr 1')), (4, Fact(forgalom='7500000'))])
```

A szabály jobb oldali részében, a logikai feltételeknél hasonlóan működik a `TEST` utasítás, mint a CLIPS esetében láttuk. A forgalom beolvasott értékét át kell alakítsuk az `int()` függvénnyel, és ez alapján változtattunk a `kiir` szabályon:

```
    @Rule(Fact(action='beolvas'),
          Fact(nev=MATCH.nev),
          Fact(hely=MATCH.hely),
          Fact(forgalom=MATCH.forgalom),
          TEST(lambda forgalom: int(forgalom) > 200000))
    def kiir(vallalat, nev):
        print("A %s vállalat forgalma nagyobb, mint
200000"  % (nev))
```

Mint látjuk, ebben a példában teljesül a szabály feltétele, a 200 000-nél nagyobb forgalmú vállalatokat listázza ki:

```
Forgalom: 280000
Vállalkozás neve: IT Kreatív Kft
Címe: Csíkszereda, Szabadság tér, Nr 14
A IT Kreatív Kft vállalat forgalma nagyobb, mint 200000
>>>
```

Az alábbi példában egy CLIPS-ben megszerkesztett, hét vállalatra vonatkozó szakértői rendszert (3. melléklet) szerkesztünk meg a Ptyhon Experta moduljában. Az első nyolc sorban az Experta moduljainak az importálása történik. A `vallalat` nevű osztályban megszerkesztjük az `ID`, `forg`, `profit`, `alkalm` és `caen` nevű mezőket, amelyeket majd a `VallalatElemzes` osztályban töltünk fel az E2-ben látható vállalatoknak megfelelő értékeivel. Az 55-64 sorok (`@Rule` utasítással) tartalmazzák a három szabályt.

```
from experta import rule
from experta.engine import KnowledgeEngine
from experta.fieldconstraint import L
from experta.rule import Rule
from experta.fact import Fact, Field
from experta.deffacts import DefFacts
from experta.shortcuts import MATCH
from experta.conditionalelement import TEST

class vallalat(Fact):
    ID = Field(int, mandatory=True)
    forg= Field(int, mandatory=True)
    profit = Field(int, mandatory=True)
```

```
    alkalm = Field(int, mandatory=True)
    caen = Field(int, mandatory=True)

class VallalatElemzes (KnowledgeEngine):
    @DefFacts()
    def init_vallalat(self):
        yield  vallalat(ID = 1,
                            forg = 2458000,
                            profit = 58000,
                            alkalm = 4,
                    caen = 6201)
        yield  vallalat(ID = 2,
                            forg = 3721700,
                            profit = 62000,
                            alkalm = 5,
                    caen = 6201)
        yield  vallalat(ID = 3,
                            forg = 3789000,
                            profit = 27500,
                            alkalm = 12,
                    caen = 4729)
        yield  vallalat(ID = 4,
                            forg = 6897500,
                            profit = 85700,
                            alkalm = 15,
                    caen = 4729)
        yield  vallalat(ID = 5,
                            forg = 4258000,
                            profit = 38500,
                            alkalm = 11,
                    caen = 3109)
        yield  vallalat(ID = 6,
                            forg = 3797500,
                            profit = 67700,
                            alkalm = 10,
                    caen = 3109)
        yield  vallalat(ID = 7,
                            forg = 2679000,
                            profit = 68500,
                            alkalm = 8,
                    caen = 9602)
    @Rule(vallalat(caen=MATCH.caen))
    def IT_vallalat(self, caen=MATCH.caen):
        if caen == 9602:
            return print('Van IT-ban tevékenykedő vállalat
a ténylistában.')
```

```
    @Rule(vallalat(ID=MATCH.ID,      alkalm=MATCH.alkalm),
TEST(lambda alkalm: alkalm<10))
    def ar_elemzes(self, ID, alkalm):
            return   print(ID,   'számú   kisvállalkozás,
amelyben az alkalmazottak száma.', alkalm)
    @Rule(vallalat(ID=MATCH.ID,          forg=MATCH.forg),
TEST(lambda forg: forg > 4000000))
    def forgalom_elemzes(self, ID, forg):
            return print(ID, 'számú vállalkozás, forgalma
nagyobb, mint 4000000: ', forg)

engine = VallalatElemzes()
engine.reset()
engine.run()
```

A futtatás eredményeként az 3.b. mellékletben látható eredményhez hasonlóan kiírja, hogy talált IT-ban tevékenykedő vállalatot, azonosítja az 1-es és 7-es számú kisvállalatokat, illetve a 4-es és 5-ös 4000000 fölötti forgalmú vállalatokat:

```
= RESTART: C:/Python/Experta/vallalatok_.py
Van IT-ban tevékenykedő vállalat a ténylistában.
7 számú kisvállalkozás, amelyben az alkalmazottak száma. 8
5  számú  vállalkozás,  forgalma  nagyobb,  mint  4000000:
4258000
4  számú  vállalkozás,  forgalma  nagyobb,  mint  4000000lej:
6897500
2 számú kisvállalkozás, amelyben az alkalmazottak száma. 5
1 számú kisvállalkozás, amelyben az alkalmazottak száma. 4
>>>
```

A következő termékre vonatkozó példában Lego játékok jellemzői alapján alakítottuk ki a tények argumentumait. A tudásbázis megszerkesztése, majd beolvasása:

```
from experta import rule
from experta.engine import KnowledgeEngine
from experta.fieldconstraint import L
from experta.rule import Rule
from experta.fact import Fact, Field
from experta.deffacts import DefFacts
from experta.shortcuts import MATCH
from experta.conditionalelement import TEST

class lego_jatek(Fact):
    ID = Field(str, mandatory=True)
    tipus = Field(str, mandatory=True)
```

```
    nev = Field(str, mandatory=True)
    korosztaly = Field(int, mandatory=True)
    ar = Field(int, mandatory=True)

class LegoElemzes(KnowledgeEngine):
    @DefFacts()
    def init_lego_jatek(self):
        yield  lego_jatek(ID = '42139',
                            tipus = 'Technic',
                            nev = 'Terepjáró',
                            korosztaly = 10,
                    ar = 289)
        yield  lego_jatek(ID = '60265',
                            tipus = 'City',
                            nev = 'Óceánkutató bázis',
                            korosztaly = 6,
                    ar = 222)
        yield  lego_jatek(ID = '21178',
                            tipus = 'Minecraft',
                            nev = 'A rókaházikó',
                            korosztaly = 8,
                    ar = 70)
        yield  lego_jatek(ID = '31116',
                            tipus = 'Creator',
                            nev = 'Szafari lombház a
vadonban',
                            korosztaly = 7,
                    ar = 160)
        yield  lego_jatek(ID = '76191',
                            tipus = 'Super Heroes -
Marvel',
                            nev = 'Végtelen Kesztyű',
                            korosztaly = 18,
                    ar = 304)
        yield  lego_jatek(ID = '71043',
                            tipus = 'Harry Potter',
                            nev = 'Roxfort kastély',
                            korosztaly = 16,
                    ar = 1680)
        yield  lego_jatek(ID = '75257',
                            tipus = 'Star Wars',
                            nev = 'Millennium Falcon',
                            korosztaly = 9,
                    ar = 639)
engine = LegoElemzes()
engine.reset()
engine.run()
```

A tudásbázis megszerkesztése, majd beolvasása után, a szabályok, az életkor és ár intervallum alapján keressük meg a megfelelő termékeket:

```
@Rule(lego_jatek(ID=MATCH.ID,tipus=MATCH.tipus,nev=MATCH.n
ev,korosztaly=MATCH.korosztaly), TEST(lambda korosztaly:
korosztaly > 10))
    def eletkor(self, ID, tipus, nev, korosztaly):
            return print(ID,' tipus: ', tipus, '\n név: ',
nev, ', 10 évnél idősebbeknek. Korosztály: ', korosztaly,
sep='')
```

Eredmény:

```
71043 tipus: Harry Potter
 név: Roxfort kastély, 10 évnél idősebbeknek. Korosztály:
16
76191 tipus: Super Heroes - Marvel
 név: Végtelen Kesztyű, 10 évnél idősebbeknek. Korosztály:
18
```

Második szabály:

```
@Rule(lego_jatek(ID=MATCH.ID,tipus=MATCH.tipus,nev=MATCH.n
ev,ar=MATCH.ar), TEST(lambda ar: 500 > ar > 100))
    def ar_intervallum(self, ID, tipus, nev, ar):
            return print(ID,'  tipus: ', tipus, 'név: \t',
nev, '\n 100 fölötti és 500 alatti termék. Ára: \t', ar
sep='')
```

Eredménye:

```
76191  tipus: Super Heroes - Marvelnév:     Végtelen
Kesztyű
 100 fölötti és 500 alatti termék. Ára:     304
31116  tipus: Creatornév:  Szafari lombház a vadonban
 100 fölötti és 500 alatti termék. Ára:     160
60265  tipus: Citynév:     Óceánkutató bázis
 100 fölötti és 500 alatti termék. Ára:     222
42139  tipus: Technicnév:  Terepjáró
 100 fölötti és 500 alatti termék. Ára:     289
```

A következő programhoz az 4. mellékletben található vállalati adatokat olvassuk be, Excelből. Tevékenységenként átlagokat számolunk, majd ezek értelmezésére szerkesztünk Experta-szabályokat.

```
import pandas as pd
import statistics
from experta import rule
from experta.engine import KnowledgeEngine
from experta.fieldconstraint import L
from experta.rule import Rule
from experta.deffacts import DefFacts
from experta.shortcuts import MATCH
from experta.conditionalelement import TEST
import statistics

vall = pd.read_excel('vallalatok adatai04.xlsx',
sheet_name='vallalk2022')

forg = vall['Forgalom']
profit = vall['Profit']
alkalm = vall['Alkalmazottak']
caen = vall['CAEN']

atlg_forg = statistics.mean(forg)
atlg_profit = statistics.mean(profit)
atlg_alkalm = statistics.mean(alkalm)

print('A ', len(forg), 'vállalat átlag forgalma:',
atlg_forg)
print('Átlag profitjuk:', atlg_profit)
print('Átlag alkalmazott számuk:', atlg_alkalm)
```

Eredménye:

```
>>>
= RESTART: C:/Python/ vallalatok_elemzes04.py
A 28 vállalat átlagforgalma: 5649866.428571428
Átlag profitjuk: 52801.78571428572
Átlag alkalmazottszámuk: 17.178571428571427
>>>
```

Megszerkesztjük a tudásbázist a 28 vállalat értékeire és az első szabályt, amely a kisvállalkozásokat keresi meg. Kisvállalat kritériumai: 10 személynél kevesebb alkalmazott, illetve kétmillió eurónál kevesebb forgalom[18].

[18] Forrás: http://publications.europa.eu/resource/cellar/1bd0c013-0ba3-4549-b879-0ed797389fa1.0014.02/DOC_2

```
class vallalat(Fact):
    forg = Field(int, mandatory=True)
    profit = Field(int, mandatory=True)
    alkalm = Field(int, mandatory=True)
    caen = Field(int, mandatory=True)

class VallalatElemzes(KnowledgeEngine):
    @DefFacts()
    def init_vallalat(self):
        yield from [(vallalat(forg = int(row.Forgalom),
                             profit = int(row.Profit),
                             alkalm = int(row.Alkalmazottak),
                             caen = int(row.CAEN))) for
index, row in vall.iterrows() ]


@Rule(vallalat(forg=MATCH.forg,alkalm=MATCH.alkalm,caen=MA
TCH.caen),TEST(lambda forg: forg<9860000),TEST(lambda
alkalm: alkalm<10))
    def kisvallalat(self, forg, caen, alkalm):
            return print('Forgalom: ', forg, ' Ágazat: ',
caen, ', kisvállalkozás ', alkalm, ' alkalmazottal',
sep='')

engine = VallalatElemzes()
engine.reset()
engine.run()
```

Eredményként, az alábbi nyolc vállalat esetében, teljesült a `kisvallalat` szabály feltétele:

```
Forgalom: 2679000 Ágazat: 9602, kisvállalkozás 8
alkalmazottal
Forgalom: 890000 Ágazat: 6201, kisvállalkozás 1
alkalmazottal
Forgalom: 1758000 Ágazat: 6201, kisvállalkozás 3
alkalmazottal
Forgalom: 6750000 Ágazat: 6201, kisvállalkozás 9
alkalmazottal
Forgalom: 8420000 Ágazat: 6201, kisvállalkozás 8
alkalmazottal
Forgalom: 1270000 Ágazat: 6201, kisvállalkozás 2
alkalmazottal
Forgalom: 3721700 Ágazat: 6201, kisvállalkozás 5
alkalmazottal
```

```
Forgalom: 2458000 Ágazat: 6201, kisvállalkozás 4
alkalmazottal
>>>
```

A Python környezetben elvégzett bármilyen számítás eredménye beilleszthető az Experta szakértői rendszerében. Vizsgáljuk meg a következő szabállyal, mely vállalatok esetében magasabb az átlagnál a forgalom és a profit.

```
@Rule(vallalat(forg=MATCH.forg,profit=MATCH.profit,caen=MA
TCH.caen),TEST(lambda forg: forg>atlg_forg),TEST(lambda
profit: profit>atlg_profit))
    def atlag_folottiek(self, forg, profit, caen):
            return print('A ' ,caen, ' ágazatban működő
vállalat, forgalma: ', forg, ' és profitja: ', profit, '
átlag fölötti.', sep='')
```

Alább látható eredményként az a négy vállalat, melyekre teljesült az `atlag_folottiek` szabály feltétele:

```
A 9602 ágazatban működő vállalat, forgalma: 6897500 és
profitja: 85700 átlag fölötti.
A 4729 ágazatban működő vállalat, forgalma: 6897500 és
profitja: 85700 átlag fölötti.
A 6201 ágazatban működő vállalat, forgalma: 6750000 és
profitja: 245000 átlag fölötti.
A 6201 ágazatban működő vállalat, forgalma: 8420000 és
profitja: 157000 átlag fölötti.>>>
```

4.2. Szakértői rendszerek összehasonlítása

Az Experta szakértői rendszerének tanulmányozása után nézzük meg az eddig tárgyalt két programozási nyelv (Prolog és CLIPS) közötti különbségeket, illetve a Pythonban modulként elérhető Experta szakértői rendszerét, egy rövid öszefoglaló keretében.

A 9.-es táblázatban, az első oszlopban látható a tények jelölése, a másodikban a szabály szerkezete.

9. táblázat. Predikátumok a Prolog, CLIPS és Experta környezetben

Nyelv	Tények létrehozása	Példák
Prolog	predikátum(attribútum).	vallalat_id(1).
CLIPS	(assert (predikátum attribútum)) (deftemplate név(slot slotnév)) (deffacts tény-lista (név(slotnév érték)))	(assert (vallalat_id 1)) (deftemplate vallalat(slot id)) (deffacts vallalkozas-tenyek (vallalat (id 1)))

Expert
a
	from experta import Fact
	
	class predikatum(Fact):
	  név = Field()
	
	class
	Predikatumok(KnowledgeEngine)
	:
	    @DefFacts()
	    def init_predikatum
	(self):
	        yield predikatum
	(név = 'Név',
		teny1 =
		Fact(vallalat_id =
		1)
		
		class
		vallalat(Fact):
		    ID = Field(int,
		mandatory=True)
		
		class
		VallalatElemzes
		(KnowledgeEngine):
		    @DefFacts()
		    def
		init_vallalat(self)
		:
		        yield
		vallalat(ID = 1)

A következő táblázatban a szabály szerkezete, a három nyelvben, rövid példákkal.

10. táblázat. A szabály szerkezete a Prolog, CLIPS és Experta környezetben

Nyelv	Szabály szerkezete	Példák

Prolog	`szabaly_neve(Változó)`
	`:-`
	`<logikai-feltételek>.`
		`tobb_alkalmazott(CegA, CegB) :-`
		`alkalmazottak(CegA, AlkA),`
		`alkalmazottak(CegB, AlkB),`
		`AlkA > AlkB.`

CLIPS	`(defrule szabaly_neve`
	`<logikai-feltételek>`
	`=>`
	`<utasítás/utasítások>)`
		`(defrule kiir`
		`(innovacio igen)`
		`(vallalkozas (nev ?nev))`
		`=>`
		`(printout t ?nev " innováló vállalat." crlf)`
		`)`

Experta	`@Rule (predikatum(<argumentumok és ezek logikai-feltételei>))`
	`def szabály-neve(argumentumok):`
	`<műveletek>`
		`@Rule(vallalat(ID=MATCH.ID, alkalm=MATCH.alkalm), TEST(lambda alkalm: alkalm > 50))`
		`def alkalm_elemzes(self, ID, alkalm):`
		`return print(ID, 'számú vállalkozásban az alkalmazottnak száma, nagyobb mint 50: ', alkalm)`

Összehasonlítva a három általunk tanulmányozott szakértői rendszert, nézzük meg ezek előnyeit és hátrányait.

11. táblázat. A Prolog, CLIPS és Experta előnyei és hátrányai

	Előnyök	Hátrányok
Prolog	Könnyen értelmezhető nyelvi logika és jól leírható kapcsolatok. Ingyenes. Nagyszámú dokumentáció elérhető. Könnyen szerkeszthető. A keresési algoritmusok könnyen adaptálhatóak benne.	Klasszikus, hagyományos, részben meghaladott.
Clips	Ingyenes. Legelterjedtebb szakértői rendszer. Könnyű szerkesztési mód. Jól	Klasszikus, hagyományos, részben meghaladott.

	Előnyök	Hátrányok
	megjeleníti az amúgy könnyen szerkeszthető logikai összefüggéseket.	
Experta	Relatív könnyen szerkeszthető. Jól megjeleníti a logikai összefüggéseket. Ingyenes. Könnyen összekapcsolható egyéb Python-programokkal.	Nem önálló program, végül is egy modul. Ebből következően kevésbé elterjedt, mint a többi hasonló szakértőirendszer-modul.

Az összehasonlításból jól látszik a szakértői rendszerek viszonylag hosszú fejlődéstörténete is. A logikai programozás jelenleg is fontos szerepet kap a különböző rendszerek fejlesztésében. A jövő elsősorban a beépíthető rendszerekben van, ilyen értelemben a szakértői rendszerek valamilyen komplexebb (vagy MI-alapú) rendszernek a szerves részét képezhetik.
Illetve külön fejlődési irányt jelentenek a fuzzy logika alapú szakértői rendszerek, amelyre példát a következő fejezetben láthatunk. A szakértői rendszerek jövőjét a fuzzy alapú, mesterséges intelligenciát felhasználó rendszerek jelentik. Ennek számos területen látjuk felhasználását a gazdaságban és az üzleti életben, olyan területeken, mint tőzsdeportfóliót kezelő rendszerek, ügyfélkapcsolati, diagnosztikai rendszerek, termékek árazása stb.

Kérdések

1. Szerkesszen olyan szakértői rendszert Expertában, amelynek argumentumait egy termék jellemzői jelentik. Majd három szabályt, amely a gyártó, ár és terméktípus szerint keresi meg a megfelelő termékeket.
2. Szerkesszen egy szakértői rendszert termékek elemzésére hasonló következtetési szabályokkal, CLIPS-ben és Expertában, és röviden mutassa be az eltérő szintaxis miatti előnyöket, illetve hátrányokat.

5. FUZZY SZAKÉRTŐI RENDSZEREK

5.1. A fuzzy logika és a fuzzy halmazok meghatározása

A bizonytalanság kezelésének legmegfelelőbb módszerét a valószínűségszámításon alapuló következtetési rendszerek jelentik.

A fuzzy logika alapján történő következtető rendszerek megszerkesztésére a Python programozási nyelv a legalkalmasabb, itt a fuzzy halmazokat, változókat és szabályokat megfelelően meg tudjuk határozni, és a következtetési eljárás a Python-környezetben lefuttatható.

A fuzzy logika sajátossága, hogy a Boole-algebrával ellentétben a „tények" vagy „logikai kijelentések" nem csak Igaz és Hamis (1 és 0) értékkel írhatóak le, hanem a fuzzy név (angolul az eredeti jelentése: homályos, elmosódott), ahogy utal rá: a fuzzy változók, jól leírják az „átmeneteket" is.

A fuzzy halmazokat és az erre épült elméletet Lotfi Zadeh dolgozta ki, aki 1965-ben a Berkeley-i Egyetem tanára volt. Az 1980-as évek végétől elsőként Japánban készítettek fuzzy logika alapján működő szakértői rendszereket.

A fuzzy logikai halmazok legfontosabb jellemzője, hogy a benne található változók valamilyen „átmenetet", gradienst, fokozatot vagy skálát ábrázolnak. Ezek lehetnek fizikai változók (mint a sebesség, hőmérséklet, nyomás stb.), illetve bármilyen más tudományterületről származó változók (pl. egy vállalat számlakifizetési hajlandósága vagy személyek magassága stb.). A Zadeh által kidolgozott modell, olyan módon képes a bizonytalanság kezelésére, hogy bővíti a Boole-logika által értelmezett 0 vagy 1 értékeket. Például egy üzleti partner, egy beszállító késését, a „hagyományos" Boole-algebra 0 – nem késő, 1 – késő, bővítheti a 0.6 – többnyire késik (0.6 valószínűséggel késni fog az input termék) vagy 0.3 – többnyire nem késik típusú változóval. Más hasonló kérdések: Mit jelent, ha egy elektronikai rendszer „nagymértékben" túlterhelt? Mit jelent, hogy kissé magas a fogyasztása egy berendezésnek? Hogyan tudjuk értékelni a fogyasztói preferenciákat bizonyos termékek vagy szolgáltatások kapcsán?

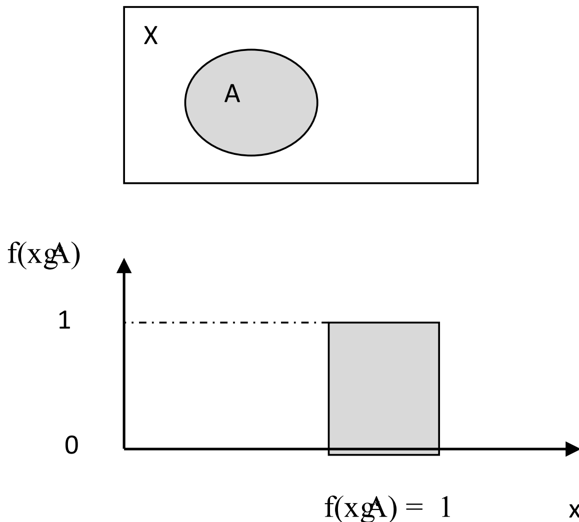
A fuzzy halmaz a hagyományos matematikai halmazokról van levezetve. Legyen x elem, amelynek az A halmazhoz való tartozására két eset lehetséges:

1. x ∈ A, vagyis tagja az A halmaznak, illetve

2. x ∉ A nem tagja az A halmaznak.

Ezt a két esetet a Boole-algebrával úgy tudjuk leírni, hogy ha tagja a halmaznak, (első esetben) az 1 (Igaz) értéket rendeljük hozzá, illetve ha nem tagja a 0 (Nem) értéket. A fuzzy logikában az alapkérdés úgy van feltéve, hogy „egy elem milyen mértékben tagja az adott halmaznak”.
A fent meghatározott A halmaz a fuzzy logikában az éles halmaz (crisp set), és a 0 és 1 értékeket nevezzük tagsági foknak (angolul: degree of membership) vagy tagsági értéknek (membership value).
A fuzzy halmaz meghatározásában tehát az X halmazból, vagyis az összes x elem halmazából indulunk ki, melynek A egy részhalmaza. Az A halmaz éles részhalmazon, az f(x|A)-t nevezzük az A halmaz karakterisztikus függvényének (characteristic function), amely a 0 vagy 1 értékeket veheti fel, aszerint, hogy f(x|A) = 1, ha x ∈ A, vagy f(x|A) = 0, ha x ∉ A. (9. ábra)

9. ábra. A fuzzy függvény éles halmazának és karakterisztikus függvényének meghatározása



A fuzzy halmazon meghatározott tagsági függvénynek (membership function) azt az m(x|A) függvényt nevezzük, amely a következőképpen írja le x-nek az A halmazhoz való tartozását:

m(x|A) = 1, ha x ∈ A,

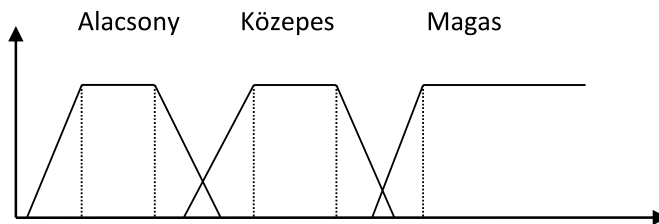
$m(x|A) = 0$, ha $x \notin A$,

$0 < m(x|A) < 1$, ha x bizonyos fokig tagja az A függvénynek.

Az $m(x|A)$, más néven tagsági fok (degree of membership) vagy tagsági érték (membership value), képes leírni az átmeneti fokozatokat, olyan helyzeteket, amikor az x csak „részben” tagja az A halmaznak.

Jó példa erre a magas fogyasztású járművek halmaza, ahol azt mérjük, hogy 100 kilométeren hány liter üzemanyagot fogyaszt egy személygépkocsi. A tagsági függvény fenti három esetéből a harmadik írja le a 0-1 átmenetet, mivel bármilyen (0,1) intervallumon értelmezett valós számot felvehet. Vagyis a tagsági függvény arra a kérdésre ad választ, hogy „mennyire tekinthető magas fogyasztású járműnek?”, vagy „mennyire tekinthető alacsony fogyasztásúnak?” egy gépkocsi. (10. ábra)

10. ábra. Az üzemanyag-fogyasztás tagsági függvénye



5.2. Fuzzy szakértői rendszer

A fuzzy szakértői rendszer a szakértői rendszer és a fuzzy logika kombinációja. Amíg a „hagyományos” szakértői rendszer a Boole-logikára épül, a fuzzy szakértői rendszer a fuzzy logikát használja. Tehát a fuzzy szakértői rendszer fuzzy szabályok és tagsági függvények gyűjteménye, amelyeket adatok következtetésére használnak. A hagyományos szakértői rendszerben használt szimbolikus érvelés helyett a fuzzy szakértői rendszerben numerikus feldolgozás történik.

A fuzzy szakértői rendszernek alapvetően három fő összetevője van (11. ábra):

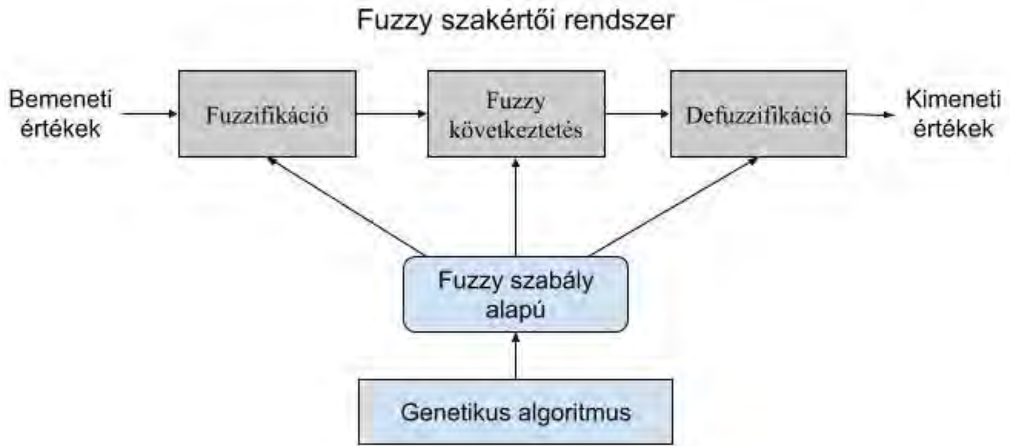
1. Fuzzifikáció: az értékek átalakítása fuzzy bemenetekké, felhasználva a tagsági függvényeket.

2. Fuzzy következtetés: fuzzy kimenet meghatározása, fuzzy szabályok segítségével.

3. Defuzzifikáció: a tagsági függvények használatával a rendszer kimeneti értékének előállítása. (Siler és Buckley, 2005)

Számos defuzzifikációs módszer ismert, úgymint a maximális tagsági értékű elem keresésének módszere, súlypont keresésének módszere (más néven centroid), maximumok átlagolásának módszere és a többiek[19]. Ezeket foglalja össze a fuzzy szabályalapú következtetési rész. (Dubois és Prade, 1988; Siler és Buckley, 2005; Sun és társai, 2008; Temur és társai, 2014)

11. ábra. A fuzzy szakértői rendszer szerkezete



Forrás: Temur és társai (2014)

Hasonlóan a Boole-algebra szabályaihoz, itt is két fő része van a fuzzy szabálynak:

- inputként az *előzménynek* (antecedent, angolul) szolgálnak, illetve
- outputként a *következménynek* (consequent) nevezünk.

Mindkettőben több kifejezés előfordulhat, a logikai Vagy (Or): `|`, És (And): `&`, illetve Nem (Not): `~` műveletek felhasználásával.

A fuzzy szakértői rendszer szabálya tehát az alábbi formájú:

```
Ha x alacsony és y magas, akkor z = közepes.
```

Kód „megfogalmazásban":

```
if {antecedent clauses} then {consequent clauses}
```

Ahol:

- x és y bemeneti változók (ismert értékek), z egy kimeneti változó (egy kiszámítandó érték neve);
- `alacsony` egy x-re definiált tagsági függvény (fuzzy részhalmaz);

[19] Forrás: http://vlabs.iitb.ac.in/vlabs-dev/labs/machine_learning/labs/explist.php

- `magas` egy y-ra definiált tagsági függvény, és
- `közepes` egy z-re definiált tagsági függvény.

A szabály „ha" és „akkor" (If… Then, angolul) közötti része a szabály _előfeltétele_ vagy _előzménye_, (utasításként, általában: `_antecedent_`). Ez egy fuzzy logikai kifejezés, amely leírja, hogy a szabály milyen mértékben alkalmazható. A szabály „akkor" utáni része a szabály _következtetése_ vagy _következménye_ (utasításként, általában: `_consequent_`). A szabály ezen része egy vagy több kimeneti változóhoz rendel egy tagsági függvényt. A legtöbb fuzzy szakértői rendszer szabályonként egynél több következtetést tesz lehetővé.

Számos gazdasági alkalmazása van, amelyek közül felsorolunk néhányat. Fuzzy elemző rendszert használt Mikhailov (2002) a több kritérium alapján történő, üzleti partnerek kiválasztására. Olyan kritériumok szerepeltek, mint az ár, a minőség, a pénzügyi stabilitás, illetve fogyasztói szolgáltatások. Altinoz (2008) tanulmánya fuzzy szabályalapú szakértői rendszert használ, amely a beszállító-kiválasztási kritériumok alapján attribútumpontozással keresi meg a legmegfelelőbb beszállítót, Sun és társai (2014) eladási előrejelzést a divatiparban, illetve Temur és társai (2014) szintén fuzzy szakértői rendszert szerkesztettek, a visszáru mennyiségének előrejelzésére. Rafigh és társai (2021) hasonló rendszert szerkesztettek a környezetbarát beszállítók kiválasztására.

5.3. Pythonban szerkesztett fuzzy szakértői rendszer

A következőkben a Python `skfuzzy` moduljában megszerkesztett fuzzy logika alapú szakértői rendszert mutatjuk be. A könyv írásakor a `skfuzzy 0.2` verziója volt elérhető[20], de a Pythonban számos más, hasonló fuzzy modul elérhető, úgymint a: `fuzzywuzzy`[21], `fuzzyset`[22], `Fuzzy`[23], `Fuzzy-Math`[24] vagy a kifejezetten szakértői rendszer, a `fuzzy-expert`[25].
Telepítése az Anaconda-környezetben:

```
conda install -c conda-forge scikit-fuzzy
```

[20] Forrás: https://pythonhosted.org/scikit-fuzzy/
[21] Forrás: https://github.com/seatgeek/fuzzywuzzy
[22] Forrás: https://pypi.org/project/fuzzyset/
[23] Forrás: https://pypi.org/project/Fuzzy/
[24] Forrás: https://pypi.org/project/Fuzzy-Math/
[25] Forrás: https://pypi.org/project/fuzzy-expert/

Vagy más környezetben a Githubról letöltött[26] kittel lehetséges:

```
python setup.py install
```

A fuzzy halmazok meghatározása Pythonban a következő paraméterekkel lehetséges:

- név, a halmaz megnevezése;
- minimumérték;
- maximumérték;
- felbontás, vagyis a lépések száma a minimum- és maximumértékek között.

A fuzzy tagsági függvényeket könnyen meg tudjuk szerkeszteni a `skfuzzy`ban. A fuzzy függvény argumentumait az alábbi példa szerint adjuk meg, ahol egy vállalat beszállítóit vizsgáljuk. Bemeneti változók:

- a vállalati stratégiai tervezés szintje (százalékban), illetve
- az üzleti partnerek száma. Ezt százalékban fejezzük ki, hogy a vállalat összes beszállítójára a maximális üzleti partnerrel rendelkező szám felel meg 100%-nak[27].

Kimeneti változó:

- az outsourcing együttműködésre való hajlandóság (százalékban).

A leíró logikai változók tagsági függvényeit,[28] az első részben határoztuk meg. A `skfuzzy`n kívül, a felbontáshoz a `numpy`t, illetve az ábrázoláshoz a `matplotlib`et használjuk.

```
import numpy as np
import skfuzzy as fuzz
import matplotlib.pyplot as plt

x_outsrc = np.arange(0, 100, 1)
x_straterv = np.arange(0, 100, 1)
x_uzlpart = np.arange(0, 100, 1)

outsrc_al = fuzz.trimf(x_outsrc, [0, 0, 50])
outsrc_kzp = fuzz.trimf(x_outsrc, [0, 65, 100])
outsrc_mgs = fuzz.trimf(x_outsrc, [75, 100, 100])
```

[26] Forrás: https://github.com/scikit-fuzzy/scikit-fuzzy

[27] Az üzleti partnerek száma a vállalati együttműködésekre vonatkozó elemzésekben használt fontos mutatószám, de számos más is szóba jöhet, mint a vállalat életkora, ágazat, intézményekkel történő együttműködés stb. (György és Madaras, 2020)

[28] Forrásként a plot_tipping_problem_newapi.py-t használtuk, innen: https://pythonhosted.org/scikit-fuzzy

```
straterv_al = fuzz.trimf(x_straterv, [0, 0, 50])
straterv_kzp = fuzz.trimf(x_straterv, [0, 50, 100])
straterv_mgs = fuzz.trimf(x_straterv, [50, 100, 100])

uzlpart_al = fuzz.trimf(x_uzlpart, [0, 0, 50])
uzlpart_kzp = fuzz.trimf(x_uzlpart, [0, 50, 100])
uzlpart_mgs = fuzz.trimf(x_uzlpart, [50, 100, 100])
```

A billentyűzetről olvassuk be a vizsgált vállalatra a stratégiai tervezés és üzleti partnerek száma értékeket. És ezek alapján határozzuk meg a `fuzz.interp_membership` tagsági függvények paramétereit.

```
straterv_f = input("Adja meg a vállalati stratégiai
tervezés szintjét (1 és 100 között):\n")
straterv_f = int(straterv_f)
uzlpart_f = input("Adja meg üzleti partnerek számát (1 és
100 között):\n")
uzlpart_f = int(uzlpart_f)

straterv_szint_al = fuzz.interp_membership(x_straterv,
straterv_al, straterv_f)
straterv_szint_kzp = fuzz.interp_membership(x_straterv,
straterv_kzp, straterv_f)
straterv_szint_mgs = fuzz.interp_membership(x_straterv,
straterv_mgs, straterv_f)

uzlpart_szint_al = fuzz.interp_membership(x_uzlpart,
uzlpart_al, uzlpart_f)
uzlpart_szint_kzp = fuzz.interp_membership(x_uzlpart,
uzlpart_kzp, uzlpart_f)
uzlpart_szint_mgs = fuzz.interp_membership(x_uzlpart,
uzlpart_mgs, uzlpart_f)
```

Három szabályt szerkesztettünk meg:

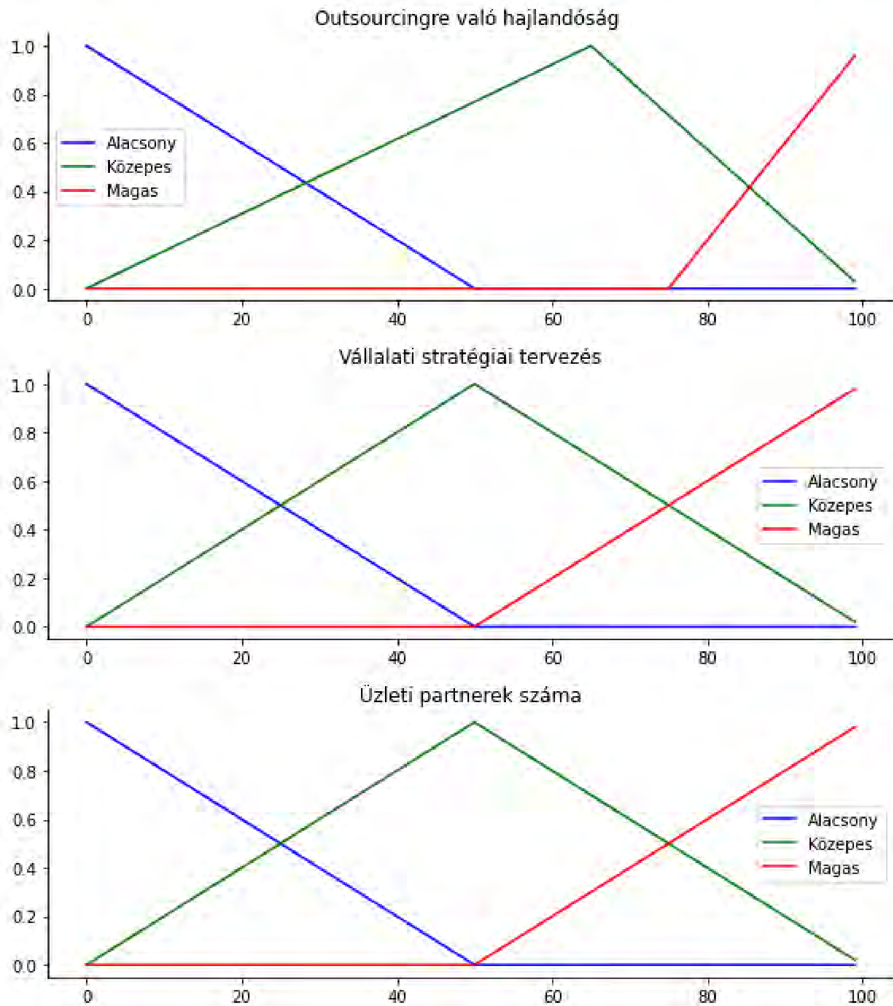
1. Ha a vállalati stratégiai tervezés hiányos, vagy az üzleti partnerek száma alacsony, akkor az outsourcing együttműködésre való hajlandóság alacsony lesz.
2. Ha a vállalati stratégiai tervezés szintje átlagos, akkor az outsourcingra való hajlandóság közepes lesz.
3. Ha a vállalati stratégiai tervezés magas fokú, és az üzleti partnerek száma magas, akkor az outsourcingra való hajlandóság magas lesz.

Felhasználjuk a `numpy` csomag, `fmax` és `fmin` függvényeit a maximum- és minimumértékek meghatározására.

```
akt_szabaly1 = np.fmax(straterv_al, uzlpart_al)
szabaly1 = np.fmin(akt_szabaly1, outsrc_al)

szabaly2 = np.fmin(straterv_szint_kzp, outsrc_kzp)

akt_szabaly3 = np.fmax(straterv_szint_mgs,
uzlpart_szint_mgs)
szabaly3 = np.fmin(akt_szabaly3, outsrc_mgs)
```

A tagsági függvények ábrázolása: a defuzzifikáció a súlypont keresésének módszerével történt.

```
outsrc0 = np.zeros_like(x_outsrc)
eredm.fill_between(x_outsrc, outsrc0, akt_szabaly1,
facecolor='b')
eredm.plot(x_outsrc, outsrc_al, 'b' )
eredm.fill_between(x_outsrc, outsrc0, szabaly1,
facecolor='c')
eredm.plot(x_outsrc, outsrc_kzp, 'm')
eredm.fill_between(x_outsrc, outsrc0, szabaly3,
facecolor='r')
eredm.plot(x_outsrc, outsrc_mgs, 'r')
eredm.set_title('Tagsági függvény')
plt.tight_layout()
```



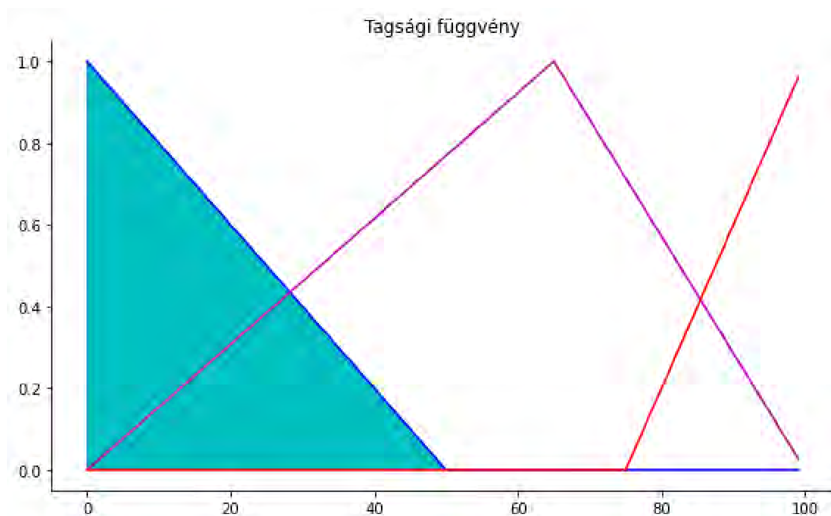
Az outsourcingra való képességet ábrázoltuk, illetve kiírtuk az értékét:

```
aggr = np.fmax(akt_szabaly1,
                         np.fmax(szabaly2, szabaly3))

outsrc = fuzz.defuzz(x_outsrc, aggr, 'centroid')
outsrc_akt = fuzz.interp_membership(x_outsrc, aggr,
outsrc)
print('Az outsourcing együttműködésre való hajlandóság: ',
outsrc)

fig, eredm = plt.subplots(figsize=(8, 5))

eredm.plot(x_outsrc, outsrc_al, 'b', )
```

```
eredm.plot(x_outsrc, outsrc_kzp, 'g')
eredm.plot(x_outsrc, outsrc_mgs, 'r')
eredm.fill_between(x_outsrc, outsrc0, aggr,
facecolor='silver')
eredm.plot([outsrc, outsrc], [0, outsrc_akt], 'k')
eredm.set_title('Aggregált tagság és eredmény-vonal ')
```

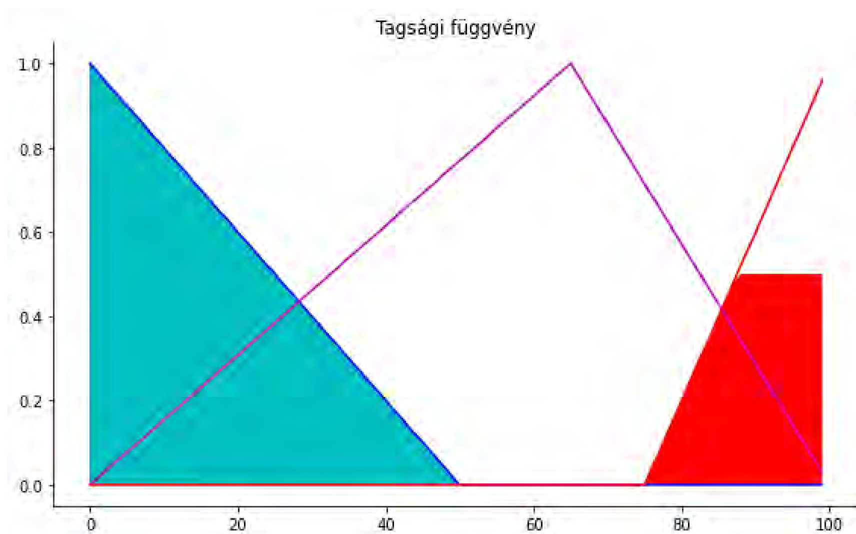
Két vállalat esetében vizsgáltuk az outsourcingra való hajlandóságot, amelyeknél a stratégiai tervezés 45%-os, illetve 56%-os, az üzleti partnerek számának aránya 32%-os, illetve 75%-os. Az első vállalat:

```
Adja meg a vállalati stratégiai tervezés szintjét (1 és
100 között):
45
Adja meg üzleti partnerek számát (1 és 100 között):
32
Az outsourcing együttműködésre való hajlandóság:
44.79250793475307
```



A második vállalat:

```
Adja meg a vállalati stratégiai tervezés szintjét (1 és
100 között):
56
Adja meg üzleti partnerek számát (1 és 100 között):
75
Az outsourcing együttműködésre való hajlandóság:
47.43284881228724
```



Az eredmények értelmezéseként elmondhatjuk, hogy a második vállalat esetében magasabb az outsourcingra való hajlandóság, amelynek értékét 47,43%-ra becsültük.

Kérdések

3. Szerkessze meg egy termék (vagy szolgáltatás) fogyasztói megítélésére vonatkozóan, az „Alacsony", „Közepes" és „Magas" értékkel jellemezhető fuzzy halmazokat, illetve ezek tagsági függvényeit és grafikus reprezentációját.
4. Szerkesszen egy fuzzy szakértői rendszert a Python `skfuzzy` moduljában, a vásárlási hajlandóságra, egy termék (vagy szolgáltatás) fogyasztói megítélése alapján, „Alacsony", „Közepes" és „Magas" értékkel jellemezhető fuzzy halmazokkal. Tesztelje néhány termék esetében.

FELHASZNÁLT IRODALOM

1. Abdelkader R., Pérez M. (2019) *Experta Documentation*, online elérhető (1/27/2022): https://readthedocs.org/projects/experta/downloads/pdf/stable/
2. Alor-Hernández G., Valencia-García R. (2017) *Current Trends on Knowledge-Based Systems*, Springer International Publishing AG 2017
3. Altinoz C. (2008) *Supplier selection for industry: a fuzzy rule-based scoring approach with a focus on usability*, Int. J. Integrated Supply Management, Vol. 4, Nos. 3/4, 2008
4. Aszalós, L. (2014) *Prolog programozási nyelv*, Debreceni Egyetem, online: https://gyires.inf.unideb.hu/GyBITT/18/index.html
5. Blackburn P., Bos J. és Striegnitz K. (2006) *Learn Prolog Now!,* Saarland University, online: https://www.coli.uni-saarland.de/~kris/learn-prolog-now/lpnpage.php?pageid=online
6. Bramer M. (2005) *Logic Programming with Prolog*, Springer
7. Bryant N. (1989) *Szakértői rendszer felépítése*, Novotrade, Budapest
8. Bogulya I. (1995) *Szakértői rendszerek, technikák és alkalmazások*, ComputerBooks. Budapest
9. Bowen K. A. (1991) *Prolog and Expert Systems*, McGraw-Hill, Inc.
10. Bratko I. (2012) *Prolog Programming for Artificial Intelligence*. Fourth edition, Pearson Education Limited, 2012
11. Clocksin W. F., Mellish C. S. (2003) *Programming in Prolog*, Fifth edition, Springer
12. Demolombe R. (1988) *Architecture of management expert systems*, in: Ernst C. J. (editor) (1988) *Maganement Expert Systems,* Addison-Wesley Publishers Ltd.
13. Dillard J. F. és Mutchler J. F. (1988) *Knowledge-based expert systems in auditing*, in: Ernst C. J. (editor) (1988) *Maganement Expert Systems,* Addison-Wesley Publishers Ltd.
14. Dubois D, és Prade H. (1988) *Processing of imprecision and incertanity in expert systems reasoning models*, in: Ernst C. J. (editor) (1988) *Maganement Expert Systems*, Addison-Wesley Publishers Ltd.
15. Endriss U. (2005) *An Introduction to Prolog Programming*, University of Amsterdam, Amsterdam
16. Ernst C. J. (1988) *Maganement Expert Systems*, Addison-Wesley Publishers Ltd.

17. Futó Iván (2019) *Mesterségesintelligencia-eszközök – szakértői rendszerek alkalmazása a közigazgatásban*, Dialóg Campus Kiadó, Budapest
18. Gal A., Lapalma G., Saint-Dizier P., Somers H. (1991) *Prolog for natural language processing*, John Wiley and Sons Ltd.
19. Giarratano J. (1998) *Expert Systems Principles and Programming*, Third Edition, PWS Publishing Company
20. György, O., Madaras. Sz. (2020) *Factors Influencing SME Outsourcing: Evidence from Romania*, Acta Universitatis Sapientiae Economics and Business 8(1):5-18
21. Hayes – Roth F. (1988) *Knowledge-based Expert Systems: the state of the Art*, in: Ernst C. J. (editor) (1988) Maganement Expert Systems, Addison-Wesley Publishers Ltd.
22. Hein J. L. (2005) *Prolog Experiments in Discrete Mathematics, Logic, and Computability*, Portland State University
23. Jakab Z., Kovács M. A. és Kiss G. (2006) *Mit tanultunk? A jegybanki előrejelzések szerepe az inflációs cél követésének első öt évében Magyarországon*, Közgazdasági Szemle, vol. LIII, issue 12, 1101-1134
24. Kandel A. (1991) *Fuzzy Expert Systems*, CRC Press
25. Krzysztof R. A. (1997) *From Logic Programming to Prolog*, Prentice Hall Europe.
26. Leondes C. T. (2002) *Expert systems: the technology of knowledge management and decision making for the 21st century*, Elsevier
27. Li S.,Li J. Z., He H., Ward P., Davies B. J. (2011*) WebDigital: A Web-based hybrid intelligent knowledge automation system for developing digital marketing strategies*, Expert Systems with Applications 38 (2011) 10606–10613
28. Liao S.-H. (2005) *Expert system methodologies and applications—a decade review from 1995 to 2004*, Expert Systems with Applications 28 (2005) 93–103
29. Liebowitz J. (1998) *The Handbook of Applied Expert Systems*, CRC Press
30. Merritt D. (1990) *Adventure in Prolog*, Springer
31. Merritt D. (1989) *Building Expert Systems in Prolog,* Springer-Verlag
32. Mersnissi V. El. (1988) *A financial management assistant systems*, in: Ernst C. J. (editor) (1988) Maganement Expert Systems, Addison-Wesley Publishers Ltd.
33. Nugues P. M. (2006) *An Introduction to Language Processing with Perl and Prolog*, Springer-Verlag, Berlin Heidelberg
34. Năstase P., Zaharie D., (1999), *Sisteme expert: teorie şi aplicaţii*, Dual Tech, Bucureşti
35. Orzan G. (2007) *Sisteme expert de marketing*, Editura Uranus, 2007
36. Pereira F. C. N., Shieber S. M. (2002) *Prolog and Natural-Language Analysis*, Microtome Publishing, Brookline, Massachusetts
37. Pólos L., és Ruzsa I., (1987) *A logika elemei*, Tankönyvkiadó, Budapest

38. Ross P. (1989*) Advanced Prolog. Techniques and Examples*, Addison-Wesley Publishers Ltd.
39. Rozenholc M., (1988) *EvEnt asseses risk taking, in: Ernst C. J.* (editor) (1988) Maganement Expert Systems, Addison-Wesley Publishers Ltd.
40. Russell S., Norvig P. (2005) *Mesterséges Intelligencia modern megközelítésben*, Második, átdolgozott, bővített kiadás, Panem Könyvkiadó, Budapest
41. Schnupp P., Nguyen Huu C.T., Bernhard L.W. (1989) *Expert Systems Lab Course*, Springer-Verlag Berlin Heidelberg, 1989
42. Shue L.-Y., Chen C.-W., Shiue W. (2009) *The development of an ontology-based expert system for corporate financial rating*, Expert Systems with Applications 36 (2009) 2130–2142
43. Siler W., Buckley J. J. (2005) *Fuzzy expert systems and fuzzy reasoning*, John Wiley & Sons, Inc.
44. Sterling L., Shapiro E. (1994) *The Art of Prolog*, The MIT Press, Cambridge, England
45. Stobo J. (2005) *Problem Solving with Prolog*, Taylor & Francis
46. Sun, Z.L., Choi, T.M., Au, K.F., Yu, Y., 2008. *Sales forecasting using extreme learning machine with applications in fashion retailing*. Decision Support Systems 46(1), 411–419 and
47. Temur, G.T., Balcilar, M., Bolat B., 2014. *A fuzzy expert system design for forecasting return quantity in reverse logistics network*. Journal of Enterprise Information Management 27(3), 316–328.
48. Zaharia M., Cârstea C., és Sălăgean L. (2003) *Inteligenţa artificială şi sistemele expert în asistarea deciziilor economice,* Editura Economică, Bucureşti
49. Wiig K. (1990) *Expert Systems*. A manager's guide. ILO Geneva
50. *** www.caen.ro
51. *** http://publications.europa.eu/resource/cellar/1bd0c013-0ba3-4549-b879-0ed797389fa1.0014.02/DOC_2
52. *** https://www.swi-prolog.org/
53. *** http://andrewprice.me.uk/geek/prolog2/
54. *** https://clipsrules.net/
55. *** https://docs.python.org/3/
56. *** https://www.pythontutorial.net/
57. *** https://python-textbok.readthedocs.io/en/1.0/index.html
58. *** https://experta.readthedocs.io/en/latest/index.html
59. *** https://www.jetbrains.com/pycharm/
60. *** https://www.cpp.edu/~jrfisher/www/prolog_tutorial
61. *** https://github.com/Kirito56/ExpertaMadman/blob/main/DefFact.py
62. *** https://pandas.pydata.org/pandas-docs/stable/pandas.pdf

63. *** https://sites.google.com/site/prologsite/prolog-problems
64. *** https://awesomeopensource.com/project/buguroo/pyknow
65. *** https://github.com/nilp0inter/experta/tree/develop/docs
66. *** http://pyke.sourceforge.net/index.html
67. *** https://anaconda.org/conda-forge/scikit-fuzzy
68. *** https://awesomeopensource.com/projects/expert-system/python
69. *** https://pythonhosted.org/scikit-fuzzy/
70. *** https://pypi.org/project/Fuzzy/
71. *** https://pypi.org/project/fuzzy-expert/
72. *** https://www.w3schools.com/python/default.asp
73. *** http://github.com/scikit-fuzzy/scikit-fuzzy
74. ***https://towardsdatascience.com/fuzzy-inference-system-implementation-in-python-8af88d1f0a6e
75. ***http://vlabs.iitb.ac.in/vlabs-dev/labs/machine_learning/labs/explist.php

KÖSZÖNETNYILVÁNÍTÁS

Ezúton is köszönöm a könyv megírásában nyújtott szakmai segítséget **dr. Bíró Piroskának** (adjunktus, Sapientia EMTE, Csíkszeredai Kar), aki a lektorálás során fogalmazta meg értékes javaslatait, illetve **Szabó Árpádnak** (informatikus-programozó, Budapest), aki ötleteivel és tanácsaival járult hozzá a könyv végső formájához.

Dr. Madaras Szilárd

2010-ben szerzett közgazdaságtan PhD-t a kolozsvári Babeş–Bolyai Tudományegyetemen. A Sapientia Erdélyi Tudományegyetem Csíkszeredai Karának oktatója.

Oktatásban és kutatásban használt számítógépes programok és programozási nyelvek: SPSS, Eviews, STATA, MATLAB, Prolog, CLIPS, Python stb. Oktatott tantárgyak: információs rendszerek menedzsmentje, térinformatika, mesterséges intelligencia és szakértői rendszerek, vállalatirányítási rendszerek, innovációmenedzsment és többváltozós statisztikai modellek. Kutatási területek: vállalkozói tevékenység, munkaerő-foglalkoztatás, idősorelemzés, ökonometriai modellek, vidékfejlesztés és regionális gazdaságtan. Számos magyar és angol nyelvű tudományos cikk és könyvfejezet szerzője vagy társszerzője.

MELLÉKLETEK

1. melléklet. Egy IT-vállalat alkalmazottai (Prologban)

```
tapasztalat(janos,4).
tapasztalat(istvan,7).
tapasztalat(arpad,12).
tapasztalat(huba,6).
tapasztalat(jolan,9).

prognyelv(janos,java).
prognyelv(istvan,c).
prognyelv(istvan,c_sh).
prognyelv(arpad,c_sh).
prognyelv(huba,php).
prognyelv(jolan,java).
prognyelv(jolan,php).

nyelv(janos,angol).
nyelv(janos,nemet).
nyelv(istvan,angol).
nyelv(arpad,angol).
nyelv(huba,magyar).
nyelv(jolan,angol).
nyelv(jolan,nemet).
```

2. melléklet. Vállalkozások (CLIPS-ben)

```
vallalkozas.clp

(deftemplate vallalkozas (slot nev) (slot forgalom) (slot
profit) (slot alkalmazottak))

(deffacts vallalkozas-tenyek
(vallalkozas (nev „IT Kreatív KFT”) (forgalom 6400000)
(profit 280000) (alkalmazottak 32))
(vallalkozas (nev „ABC Kft”) (forgalom 6400000) (profit
56000) (alkalmazottak 26))
(vallalkozas (nev „Próba KFT”) (forgalom 1250000) (profit
12000) (alkalmazottak 6))
(vallalkozas (nev „Havasigyopár Kft”) (forgalom 7500000)
(profit 35000) (alkalmazottak 8))
```

```
(vallalkozas (nev „IT Szerviz Kft”) (forgalom 175000)
(profit 14000) (alkalmazottak 3))
)
```

3. mlléklet. Vállalkozásokra szerkesztett CLIPS program (a) és futtatásának eredménye (b)

```
(deftemplate vallalat (slot id) (slot forg) (slot profit)
(slot alkalm) (slot caen) )

(deffacts vallalkozasok
(vallalat (id 1) (forg 2458000) (profit 58000) (alkalm 4)
(caen 6201))
(vallalat (id 2) (forg 3721700) (profit 62000) (alkalm 5)
(caen 6201))
(vallalat (id 3) (forg 3789000) (profit 27500) (alkalm 12)
(caen 4729))
(vallalat (id 4) (forg 6897500) (profit 85700) (alkalm 15)
(caen 4729))
(vallalat (id 5) (forg 4258000) (profit 38500) (alkalm 11)
(caen 3109))
(vallalat (id 6) (forg 3797500) (profit 67700) (alkalm 10)
(caen 3109))
(vallalat (id 7) (forg 2679000) (profit 68500) (alkalm 8)
(caen 9602))
)

(defrule it_vallalat
(vallalat (caen 9602))
=>
(printout t "Van IT-ban tevékenykedő vállalat a
ténylistában.", crlf)
)

(defrule ar_elemzes
(vallalat (id ?id) (alkalm ?meret))
(test (< ?meret 10))
=>
(printout t ?id " ID-val, kisvállalkozás, amelyben az
alkalmazottak száma: " ?meret crlf)
)

(defrule forgalom_elemzes
(vallalat (id ?id) (forg ?forg))
(test (> ?forg 4000000))
=>
```

```
(printout t ?id " ID-val, vállalkozás, forgalma nagyobb,
mint 4000000 : " ?forg crlf)
)
```

(a)

```
CLIPS> (load "C:/E/Egyetem/Tantargyfejlesztes/szakertoi
rendszerek/CLIPS/vallalat_CLIPS.clp")
Defining deftemplate: vallalat
Defining deffacts: vallalkozasok
Defining defrule: it_vallalat +j+j
Defining defrule: ar_elemzes +j+j
Defining defrule: forgalom_elemzes +j+j
TRUE
CLIPS> (reset)
CLIPS> (run)
Van IT-ban tevékenykedő vállalat a ténylistában.
7 ID-val, kisvállalkozás, amelyben az alkalmazottak száma:
8
5 ID-val, vállalkozás, forgalma nagyobb, mint 4000000 :
4258000
4 ID-val, vállalkozás, forgalma nagyobb, mint 4000000 :
6897500
2 ID-val, kisvállalkozás, amelyben az alkalmazottak száma:
5
1 ID-val, kisvállalkozás, amelyben az alkalmazottak száma:
4
CLIPS>
```

(b)

4. melléklet. A vállalkozások adatai[29]

	A	B	C	D	E
1	Forgalom	Profit	Alkalmazottak	CAEN	
2	2458000	58000	4	6201	
3	3721700	62000	5	6201	
4	1270000	12500	2	6201	
5	8420000	157000	8	6201	
6	6750000	245000	9	6201	
7	1758000	80000	3	6201	
8	890000	45750	1	6201	
9	3789000	27500	12	4729	
10	6897500	85700	15	4729	
11	4875820	32580	22	4729	
12	7850000	25800	24	4729	
13	8564200	35870	32	4729	
14	2685000	25000	14	4729	
15	10895000	28950	34	4729	
16	3789000	27500	12	3109	
17	6897500	85700	15	3109	
18	4875820	32580	22	3109	
19	7850000	25800	24	3109	
20	8564200	35870	32	3109	
21	2685000	25000	14	3109	
22	10895000	28950	34	3109	
23	3789000	27500	12	9602	
24	6897500	85700	15	9602	
25	4875820	32580	22	9602	
26	7850000	25800	24	9602	
27	8564200	35870	32	9602	
28	2685000	25000	14	9602	
29	10895000	28950	34	9602	

vallalk2022 | 4729 | 3109 | 9602

Forrás: Saját szerkesztés

[29] 6201 – Szoftverfejlesztési tevékenységek (ügyfélközpontú szoftverek). A kategóriához tartoznak a szoftvertermékek programozási, szerkesztési, tesztelési és támogatási tevékenységei. (Forrás: www.caen.ro)
4729 – Élelmiszerek kiskereskedelme, szaküzletekben. A kategóriához tartozik a tejtermékek és tojások kiskereskedelme, illetve egyéb élelmiszerek kiskereskedelme. (Forrás: www.caen.ro)
3109 – Bútorgyártás. (Forrás: www.caen.ro)
9602 – Fodrászat, szépségápolás. (Forrás: www.caen.ro)

A KÖNYV ELSŐSORBAN a gazdasági informatika szakos hallgatók számára készült, de minden olyan informatika iránt érdeklődő olvasó talál benne érdekes dolgokat, akit érdekel a formális logika alapján történő programozás, a szabályalapú szakértői rendszerek működése, illetve a fuzzy logika alapján megszerkesztett szakértői rendszerek.
A könyvben tárgyalt, Prolog, CLIPS és Python nyelvekben megszerkesztett, elsősorban gazdasági jellegű példák lehetőséget nyújtanak e programozási nyelvek összehasonlítására a szabályalapú rendszerek működésének mélyebb megértésére. A könyv egyben áttekintést nyújt a szakértői rendszerek fejlődéséről és felhasználási irányairól is. A gyakorlati megértést szolgálja, hogy több mint 50 alkalmazást mutatunk be, remélve ezáltal, hogy a tisztelt olvasó átfogó képet kap a szakértői rendszerek programozásáról.
A szakértői rendszerek készítésénél használt sajátos logika elsajátítása hasznos a diákok és az olvasók számára, mert hozzájárul a logikai rendszerek mélyebb megértéséhez, a tudásreprezentáció lehetőségeinek megismeréséhez és végső soron a programozási képességek és készségek kifejlesztéséhez.

SAPIENTIA
ERDÉLYI MAGYAR
TUDOMÁNYEGYETEM
Csíkszeredai Kar