**COLEGIUL NAȚIONAL “A.T.Laurian” BOTOȘANI**

**Profilul matematică-informatică - intensiv engleză**

***LUCRARE DE ATESTAT PROFESIONAL***

**COORDONATOR, CANDIDAT,**

**Prof.Cardaș Daniela Cerasela Băcăuanu Gabriel-Alexandru**

**~ MAI 2019~**

***Tema lucrării:***

**Vindictus:**Magazin de calculatoare în Lisp

Cuprins:

1. Argument – pag. 4
2. Despre limbaj – pag. 4
3. Vindictus – ce este și ce oferă – pag. 6
4. Inițializarea programului – pag. 7
5. Database-backend.lisp – pag. 7
6. Database-informations.lisp – pag. 21
7. Database-user-commands – pag. 25
8. Database-repl.lisp – pag. 31
9. Bibliografie – pag. 35

**Argument:**

Lisp a fost la început o curiozitate foarte mare pentru mine, un limbaj despre care tot auzeam prin glume, de cele mai multe ori legate de numarul insuportabil de paranteze care pot să apară. Am reușit să prind într-un bundle mai multe cărți despre limbaje de programare, printre care se afla una despre Common Lisp. Am început să studiez acest limbaj, cu fiecare pagină simțind cum devine tot mai natural și mai confortabil, so much so încât a devenit limbajul în care prefer să scriu și în care mă simt bine când îl folosesc.

Urmând cursurile cărții Land of Lisp am ajuns la hash tables, structuri de date complexe în care scrierea, prelucrarea și citirea datelor este rapidă și eficientă. Eu sunt genul care învăț experimentând, așa că am dorit să încep un proiecțel în care să am ocazia de a mă distra jucându-mă cu aceste hash table-uri. Așa am ajuns să lucrez la Vindictus, un magazin de calculatoare în care principalul mod de stocare a datelor îmi permite să folosesc din plin aceste structuri de date.

**Despre limbaj:**

Lisp este un limbaj nu prea întâlnit în jurul nostru, la prima vedere, sau în discuţii. Acesta este un limbaj foarte vechi, mai precis, al doilea limbaj ca vechime dupa FORTRAN, ambele fiind încă folosite şi în ziua de azi. Lisp a apărut prima dată ca limbaj teoretic in 1957, scris de matematicianul John McCarthy la MIT, inspirat de notația limbajului lambda calculus, ajungând să fie un limbaj de programare real un an mai târziu. Acesta a fost original construit pentru a ajuta profesorul şi echipa sa să studieze inteligenţa artificială, McCarthy fiind un pioner al domeniului si Lisp uneltele care l-au ajutat să urce muntele. Principalul feature al acestui străvechi animal este LISt Processing, care îi şi oferă numele și structura sa bizara. Limbajul a ajuns să fie îngropat din cauza comodității, un cod în Lisp fiind de dimensiuni mai mici și la fel de eficient atunci când fiecare variabilă este declarată cu un tip, și trecerea de la studiul inteligenței artificiale la cel al Machine Learning-ului. Totuși, Lisp nu este mort, ci încă mai este folosit în jurul nostru, de exemplu, în multe sisteme bancare uneltele pentru detectarea fraudelor sunt scrise in Lisp. Un exemplu mai precis ar fi Xanalys Ltd, a cărui unelte în domeniul securității au fost scrise în Common Lisp.

În Lisp codul se află mereu între paranteze rotunde, deoarece orice comandă este privită drept o listă, principalul tip de dată folosit atât în scrierea de cod, cât și în scrierea Lisp-ului în sine. Un apel este alcătuit din funcție și argumente, un calcul de forma ”2+3” fiind scris sub forma ”(+ 2 3)”. Deși pare o abordare foarte urâtă, felul în care este aranjat totul oferă șansa de a construi arbori într-un mod foarte ușor: (defparameter foo ‘((a (b c)) (d (e f)) (g (h i)))). Acesta este un arbore care are 10 noduri, ”rădăcina” fiind foo, care are cei trei descedenți direcți ”a,b,c”, fiecare având câte doi descendenți.

Lisp suportă un număr destul de mare de tipuri de date: simboluri, tipuri numerice (numere întregi, raționale, reale, complexe), liste, array-uri și diferite variații de hash-table-uri. Variabilele pot să fie declarate fără un tip de dată, acesta fiind alocat în mod automat în funcție de ce conține și de ce va conține. De exemplu, pot declara variabila foo prin (defparameter foo ‘(a b c)), aceasta conținând o listă de simboluri, dar după ce o folosesc și nu mai am nevoie de ea o pot redefini cu (setf foo 5). Acum foo este un număr de care poate voi avea nevoie ceva mai târziu și din moment ce aveam variabila respectivă liberă de ce să irosesc spațiu declarând alta? Lisp de asemenea oferă posibilitatea de a declara tipul de dată în interiorul unei funcții pentru a scurta timpul de execuție, prin (declare (type \*tipul dorit\* \*variabilele\*)) sau în exterior prin (check-type \*variabila\* \*tipul dorit\*).

Limbajul suportă operatorii normali ca ”+,-,\*,/,=,!”, având și o serie de operatori specializati precum cons, pe care îl putem folosi în alcătuirea unei liste.

Vindictus

Bine aţi venit la Vindictus, singurul magazin de calculatoare scris in Lisp! Vindictus este un proiect născut din pasiune şi facut prin pasiune, care, deşi mai are mult de crescut, se află deja într-un stadiu destul de avansat.

**Ce este şi ce oferă?**

Vindictus este o aplicaţie scrisă în Lisp, cu o interfaţă grafică destul de modestă, dar care îşi ascunde adevărata frumuseţe în putere şi simplitate. Aplicaţia oferă cumpărătorilor comenzi pentru a-şi crea un cont, sau a se loga într-unul deja existent, pentru a construi un pc, a vedea comenzile precedente sau pentru a şterge o comandă/contul.

Pentru admini, Vindictus are o serie de comenzi speciale care le permit să adauge sau să şteargă un admin, să adauge/şteargă o componentă hardware în/din şi comenzile destinate publicului, dar care pot fi folosite asupra oricărui cont deja existent.

**De ce se află Lisp în spatele aplicaţiei?**

Am ales Lisp ca limbaj pentru backend + frontend deoarece oferă o modularitate și o flexibilitate aparte. Funcţiile puse la îndemână de standardul limbajului, uşurinţa de a crea noi funcţii şi macro-uri, împreună cu anumite trucuri (care e posibil să nu fie prea inteligente în această lucrare de atestat) rezultă în programe foarte compacte, Vindictus având mai puţin de 350 de linii de cod, incluzând custom REPL-ul, care serveşte drept interfaţă.

**Disclaimer!**

Anumite alegeri în cod n-au fost făcute având eficienţa ca scop, ci pentru a arăta anumite feature-uri ale limbajului sau pentru a păstra un aspect un pic mai curat. De asemenea, programul încă prezintă un bug micuţ, anume apariţia în consolă a cuvântului “#\newline” după anumite funcţii.

**Iniţializarea programului**

(load "Proiecte/Database/database-backend.lisp")

(load "Proiecte/Database/database-informations.lisp")

(load "Proiecte/Database/database-user-commands.lisp")

(load "Proiecte/Database/database-repl.lisp")

(start-screen)

Pentru a porni Vindictus trebuie incărcat fişierul “load-base.lisp” într-un compiler de Lisp prin comanda (load “Vindictus/load-base.lisp”). Recomand copierea folderului cu codul sursă și a filei load-base.lisp în folderul compiler-ului (preferabil CLisp sau SBCL). După încărcarea acestuia aplicaţia va porni în mod automat.

Pe liniile 7-11 se găsesc o serie de conturi şi parole pentru a putea testa toate comenzile, numele fiind cel de după linie, iar parola fiind cea din paranteză.

În continuare vom lua fiecare fişier cu cod sursă în ordinea apariţiei lor in load-base.lisp pentru a discuta funcțiile și comenzile ce se află în interior, această ordine reprezentând într-o oarecare măsură dependenţa şi importanţa lor.

**database-backend.lisp**

Probabil cel mai important fişier dintre toate, database-backend.lisp reprezintă scheletul programul, conţinând o mare parte din funcţiile folosite de întreg programul.

(defmacro push\_back(the-list el)

`(setf ,the-list (concatenate 'list ,the-list (list ,el))))

(defmacro gremove(el the-list)

`(setf ,the-list (remove ,el ,the-list)))

Un lucru foarte curios este faptul că pe primele 5 linii de cod nu se află functii aşa cum ne-am aștepta, ci macro-uri. Macro-urile sunt unelte foarte puternice în Lisp, oferind puterea de a customiza limbajul după placul programatorului. În cazul meu, le-am folosit pentru a evita să setez (cu ajutorul funcţiei setf) de fiecare dată o listă când doresc s-o modific în contextul operațiilor respective, lucru care a putut să fie făcut doar prin intermediul acestor mici trucuri magice. De asemenea, sintaxa lor este una extrem de specială dacă privim le comparăm cu una dintre funcţiile de mai jos.

În Lisp codul si datele sunt construite din aceleași materiale, acesta având un code mode si un data mode, semnalat cu ‘ sau cu `. La al doilea semn există un strop de magie prin faptul că poate să-i fie pusă “pauză” folosind virgula. Să presupunem că avem salvat în variabila \*name\* numele “Andrei”, iar în variabila \*vârstă\* 15, dacă scriem într-un terminal de Lisp ‘(\*nume\* are \*vârstă\* ani) sau `(\*nume\* are \*vârstă\* ani) vom primi ‘(\*nume\* are \*vârstă\* ani), adică lucrul pe care i l-am oferit ca dată, dar daca scriem `(,\*nume\* are ,\*vârstă\* ani) vom primi (ANDREI ARE 15 ANI) (REPL-ul obişnuit returnează orice nu este string cu litere mari, simbolurile fiind case-insensitive. Lucrul acesta îl vom evita folosind un custom REPL). Acesta este un lucru uimitor, avem bucăţi de cod compilat în interiorul unei bucăţi de dată. Caracteristica aceasta, de a avea codul si datele alcătuite din aceleaşi lucruri, face din Lisp un meta limbaj de programare.

**Cassoc**

(defun cassoc (el alist)

(if alist

(if (string= el (caar alist))

(car alist)

(cassoc el (cdr alist)))

NIL))

Subprogramul cassoc –c(har) assoc- am scris-o foarte aprope de final, atunci când finalizam funcţia a-delete-order. Prezenţa acesteia a fost necesară deoarece funcţia assoc şi funcţia member nu merge pentru string-uri, aşa cum am avut nevoie, ci doar pentru simboluri sau numere. Cassoc este un subprogram recursiv care trece prin toate elementele unei liste de asociatie, căutând în car-ul (primul element) fiecărei liste care compun alist-ul, până dă peste elementul căutat sau până ajunge la finalul alist-ului, caz în care returnează nil (nul). Cu alte cuvinte, caută o cheie pe care i-o ofer până dă de prima apariție a acesteia. Egalitatea dintre elementul căutat şi car-ul fiecărei liste este verificată prin “string=”, o funcţie specializată în testarea egalităţii dintre 2 string-uri. La apelul (cassoc “b” ‘((“a” 1”) (“b” 2) (“c” 3))) va returna (“b” 2).

**List-to-string + string-to-list**

(defun list-to-string (lst)

(when lst

(concatenate 'string (write-to-string (car lst)) " " (list-to-string (cdr lst)))))

(defun string-to-list(str) (read-from-string (concatenate 'string "(" str ")")))

Funcţille list-to-string şi string-to-list sunt două dintre trucurile despre care vorbeam la început. Acestea doua fac conversia de la listă la string si vice-versa, lucru extrem de util la citirea şi la scrierea datelor. List-to-string este o funcţie recursivă care ia fiecare element dintr-o listă, îl transformă în string, concatenându-i un spaţiu după şi ataşând elementul următor. String-to-list este un truc şi mai frumos prin care doar adaug paranteze la început şi la sfârşitul string-ului şi îl trec prin funcţia read-from-string (aceasta primeşte ca parametru un string şi returnează un simbol sau o listă).

**Hash-it + sum-list + find-index**

(defun hash-it(pass index)

(cond

((equal (car pass) nil) nil)

((oddp (car pass)) (cons (+ (ash (car pass) -1) (\* 2 index)) (hash-it (cdr pass) index)))

(t (cons (- (+ (- (car pass) (ash index 1)) (ash (ash (ash (car pass) -1) -1) -1)) (\* 2 index)) (hash-it (cdr pass) index)))))

(defun sum-list (lst)

(if(not (equal lst nil))

(+ (car lst) (sum-list (cdr lst)))

'0))

(defun find-index (lst)

(if (> lst 10)

(+ (mod lst 10) (find-index (truncate (/ lst 10))))

(mod lst 10)))

Despre următoarele 3 funcţii n-o să vorbesc prea mult deoarece acestea au fost scrise mai mult pentru a simula un algoritm pentru a face hash unei parole. Funcţia hash-it este atracţia principală a acestei părţi, aceasta primeşte parola, sub formă de listă cu coduri ASCII a fiecarui caracter, şi un index care este găsit de funcţia find-index, care primeşte un număr şi face suma cifrelor acestuia. Numărul pe care-l primeşte subprogramul find-index este transmis prin intermediul funcţiei sum-list, care ia o listă cu codurile ASCII a literelor ce alcătuiesc numele utilizatorului şi returnează suma acestora. În acest fel, chiar dacă doi utilizatori folosesc aceeaşi parolă, în baza de data vor apărea parole diferite.

**Is-admin**

(defun is-admin(name)

(if (member name \*admin-list\*)

(setf \*admin-logged\* t)))

Subprogramul is-admin doar verifică daca numele celui care se loghează aparţine unui admin, caz în care modifică un flag pentru a anunţa REPL-ul că un admin este logat și are acces la comenzile de admin, sau dacă este un simplu utilizator, caz în care nu face nimic.

**Userp**

(defun userp(name)

(if (member name \*users\*) t nil))

Userp doar primeşte un nume şi returnează adevărat (T) dacă acesta există în baza de date sau fals (NIL) în caz contrar. Deşi nu este nimic interesant cu funcţia in sine, aceasta conţine un detaliu ceva mai special. Userp este un predicat, lucru indicat prin convenţie de p-ul de la urmă. Un predicat este o funcţie al cărei unic scop este să facă o verificare şi să returneze T sau NIL (ex: alphanumericp, symbolp, numberp, listp etc.).

**Ask-password**

(defun ask-password(name)

(princ "Introduce your password: ")

(let ((ind-name (find-index (sum-list (map 'list #'char-code (string name))))))

(flet ((hash-pass(psw) (coerce (mapcar #'code-char(hash-it (map 'list #'char-code psw) ind-name)) 'string)))

(let ((pass (read-line)))

(if (string= (gethash 'password (eval name)) (hash-pass pass))

(progn (setf \*logged\* t) (setf \*logged-account\* name) (is-admin name) `(welcome back ,\*logged-account\* at "Vindictus!"))

'(password incorrect!))))))

Următoarea funcţie poate părea înfricoşătoare, dar dacă o luăm pas cu pas, o să vedem că nu e atăt de zmeu pe cât se dă. Prima linie este ocupată doar de o afişare a unui mesaj. Special form let este un alt lucru nou peste care dăm, acesta având două părţi: o listă de asociere prin care putem declara o serie de variabile locale şi un corp, în interiorul căruia variabilele sunt declarate. Prima dată transformăm numele în string, acesta fiind salvat ca simbol din motive de eficienţă şi claritate a codului, după care folosim o funcţie numită map. Aceasta primeşte un simbol prin care se precizează formatul rezultatului, o funcţie care este aplicată fiecărui element dintr-un sequence (sequence este o denumire folosită foarte des în programarea generică, acest termen însemnând: listă, string sau array). În acest caz, map îmi transformă string-ul meu într-o listă de caractere, pe care le transformă in coduri ASCII. Această listă este transmisă funcţiei sum-list care apoi furnizează rezultatul funcţiei find-index, salvând la final index-ul în ind-name.

Asemenea lui let, mai avem un special form numit flet. Flet are aceeaşi structură ca let, doar ca acesta este folosit pentru a declara subprograme locale. O nebunie, ştiu, dar o nebunie bună. Abilitatea de a avea funcţii în interiorul altor funcţii este extrem de utilă, mai ales în programe care folosesc o abordare divide et impera. hash-pass primeşte ca parametru o parolă pe care o trece prin acelaşi proces ca numele, transformând-o într-o listă de coduri ASCII cu care este hrănită funcţia hash-it, împreună cu indexul pe care l-am găsit cu o linie mai sus. La următoarea linie mai declarăm o variabilă în care salvăm parola introdusă de la tastatură de către utilizator prin funcţia read-line, care returnează input-ul ca string, spre diferentă de read, care returnează input-ul ca simbol.

Fiecare utilizator are informaţiile stocate într-un hash table, a cărui câmpuri pot fi apelate prin gethash, primul argument este cheia câmpului, al doilea fiind tabela în care caut. La început aveam o problemă care apărea din faptul că al doilea argument este tabela, nu numele tabelei. Prima dată mă gândeam că n-o să am cum să apelez astfel o tabelă, pe care o vreau, deoarece aveam doar numele oferit de utilizator, dar aici a apărut eroul care mi-a salvat ziua. Eval este o comanda prin care pot evalua argumentul pe care i-l ofer, deci evaluând numele pot fac rost de tabelă. Aceste lucruri puse la un loc îmi dau voie să fac rost de parola unui anumit cont, care a fost deja trecută prin “algoritmul” de hashing. Această linie comparând parola salvată cu parola pe care o trec prin acelaşi “algoritm”, cu acelaşi index. Dacă cele două parole se potrivesc înseamnă că el este proprietarul contului, deci îl las să intre, dar nu înainte de a folosi altă chestie nouă. Progn este un alt form care este prieten de cataramă cu if-ul din Lisp, deoarece if-ul este mai mult echivalent cu operatorul ternar din C++ (ex: (cond ? if-true : if-false)), decat cu if-ul normal. Acesta nu are separator propriu intre cele două cazuri, deci a doua funcţie din if este apelată doar atunci când o condiţie este falsă, iar orice se află după produce o eroare. Progn, în schimb, rezolvă acest downside prin faptul că primeşte o mulţime de funcţii între parantezele sale şi îi spune if-ului “Nu e nimic de văzut aici, sunt o simplă funcţie care aşteaptă sa fie apelată”, jucând astfel rolul de cal troian.

În cazul în care a fost introdusă parola corectă, este activat un alt flag pentru a spune REPL-ului că cineva este on şi trebuie să folosească funcţiile specifice user-ului. De asemenea se face o verificare cu funcţia is-admin pentru a activa sau nu şi celălalt flag. La urmă, dacă s-a conectat, utilizatorul este întămpinat printr-o manieră despre care am discutat mai devreme, folosind quasiquoting. În caz că parola introdusă este greşită va fi afişat un mesaj de avertizare.

**A-add-component**

(defun a-add-component()

(labels ((verify-name(category)

(let ((name (read)))

(if (member name (gethash 'the-list (eval category)))

(progn (princ "Sorry, there is already a component with this name") (princ #\newline)

(verify-name category))

(return-from verify-name name))))

(add-chars (name category clist)

(when clist

(database-print (car clist))

(let ((char (read)))

(push\_back (gethash name (eval category)) char)

(add-chars name category (cdr clist))))))

(princ "What kind of component do you want to add? ")

(princ "If the name is not valid then you'll have to type it again.") (princ #\newline)

(princ "(\*RAM\*, \*video-cards\*, \*CPU\*, \*storage\*, \*mother-boards\*, \*PSU\*)") (princ #\newline)

(let ((option (read)))

(loop while (not (member option \*components\*)) do(setf option (read)))

(princ "Name of the component:")

(let ((name (verify-name option))

(clist (gethash 'chars (eval option))))

(add-chars name option clist)

(push\_back (gethash 'the-list (eval option)) name)

(princ "Product added successfully!")))))

Aceasta este o mică bestie de funcţie fiind a doua ca număr de linii, dar este şi una ceva mai uşurică de explicat acum că despre multe chestii ce le folosesc aici am vorbit anterior, putând să mă concentrez doar pe detalii despre funcţia în sine. A-add-component este o comandă destinată adminilor, lucru marcat prin prefixul “a-“ şi are rolul de a introduce în baza de date o nouă componentă. Labels este acelaşi lucru ca şi flet, doar că permite subprogramelor locale să se apeleze între ele sau chiar pe ele. Prima din cele două este una recursivă şi are rolul de a citi un nume valid, pentru a nu avea două piese cu acelaşi nume. Aceasta verifică dacă deja există numele, caz în care afişează un mesaj corespunzător şi se reapelează, iar în cazul în care acesta nu este folosit returnează numele găsit. Era la fel de corect să scriu în al doilea caz “name”, returnându-l la finalul apelului, dar am păţit cândva să nu-mi returneze, lucru care m-a împins să folosesc return-from a cărui prim argument e numele funcţiei din care returnează ceva, iar al doilea este lucrul pe care îl returnează.

A doua funcţie locală doar trece prin lista de caracteristici a unei componente printr-o metodă recursivă şi le citeşte, afisănd caracteristica pe o aşteaptă şi folosind macro-ul push\_back declarat la început pentru a o pune la locul potrivit în câmpul noului produs.

Corpul principal prezintă o a doua modalitate prin care pot să verific validitatea unui nume, folosind de data asta o metodă iterativă. Într-adevăr, este mult mai eficientă, dar dacă aş avea de făcut mai mult de o operație ar avea un aspect ceva mai urât. După aceasta caut un nume valid folosind prima funcţie locală şi îmi pregătesc lista de caracteristici prin care trebuie să trec atunci cand vreau să adaug una nouă, chemând mai apoi cea de-a doua funcţie. La final mai folosesc o dată macro-ul push\_back pentru a introduce numele piesei în lista de componente, afişând un mesaj care să confirme că adăugarea s-a făcut cu succes.

**A-delete-component**

(defun a-delete-component()

(princ "What kind of component would you like to delete? Ex: ") (database-print \*components\*)

(princ "If the name is not valid then you'll have to type it again.") (princ #\newline)

(setf comp (read))

(loop while (not (member comp \*components\*)) do(setf comp (read)))

(princ #\newline)

(princ "Which one would you like to delete? If it's not correct you'll have to type again") (princ #\newline)

(setf piece (read))

(loop while (not (member comp (gethash 'the-list (eval comp)))) do(setf comp (read)))

(gremove piece (gethash 'the-list (eval comp)))

(update-database)

(princ "Component successfully deleted") (princ #\newline))

Următoarea funcţie este opusa ultimei, după cum ne spune și numele de a-delete-component. Aceasta afişează pe ecran toate categoriile de componente după care foloseşte aceeaşi modelitate de verificare ca la funcţia precedentă. Aceasta procedură o repet şi pentru numele piesei, folosind apoi cel de-al doilea macro declarat la inceput pentru a o şterge, afişând un mesaj de confirmare. Consider interesant modul prin care este ştearsă piesa. Eu nu fac altceva decât să-i şterg numele din lista de piese, fiind asemănător cu ştergerea datelor de pe un mediu de stocare unde nu sunt şterse datele în sine, ci pointerii prin care se poate ajunge la ele.

**A-view-history**

(defun a-view-history (acc-name)

(if (not (userp acc-name)) (return-from a-view-history "The name is not associated with an account"))

(princ "Here are your orders:")

(gethash 'orders (eval acc-name)))

Funcţia a-view-history doar afişează comenzile anterioare, verificând doar dacă numele primit ca parametru este unul valid, folosind predicatul declarat ceva mai sus, şi afişând lista din câmpul de comenzi al contului.

**A-delete-order**

(defun a-delete-order (acc-name)

(if (not (userp acc-name)) (return-from a-delete-order "The name is not associated with an account"))

(labels ((verify-order()

(let((order (read-line)))

(setf aux (cassoc order (gethash 'orders (eval acc-name))))

(if aux

(progn (gremove aux (gethash 'orders (eval acc-name)))

'(The order was removed!))

(progn

(database-print `(there is no ,order order! Do you want to try again? Yes/no))

(if (eq (read) 'yes) (verify-order)))))))

(verify-order)))

Începem să ne apropiem de finalul acestei file, având în continuare a-delete-order care are rolul de a şterge o anumită comandă a unui cont. Face aceeaşi verificare de validitate a numelui şi sub ea se află funcţia care se ocupă propriu-zis de ştergerea comenzii. Aceasta citeşte numele comenzii, dacă există o comandă cu acel nume pe cont atunci o să fie ştearsă folosind macro-ul gremove, dacă nu atunci o să fie afişat un mesaj care să-l anunţe pe utilizator că nu a fost gasită o comandă cu acest nume şi să-l întrebe dacă doreşte să încerce din nou.

**A-delete-account**

(defun a-delete-account (acc-name)

(if (not (userp acc-name)) (return-from a-delete-account "The name is not associated with an account"))

(princ "Are you sure that you want to delete this account? 'Yes/no'") (princ #\newline)

(let ((option (read)))

(if (eq option 'yes)

(progn (princ "This account will be deleted") (princ #\newline)

(gremove acc-name \*users\*)

(when (eq acc-name \*logged-account\*)

(setf \*logged\* nil)

(setf \*logged-account\* nil)

(setf \*admin-logged\* nil)

(update-database)

(start-screen))))))

O funcţie destul de necesară oricărui magazin sau platforme este cea de ştergere a contului. Vindictus are una simpluţă, aş spune eu, pentru a face lucrul acesta. Prima dată face verificarea normală a numelui şi apoi cere o confirmare a dorinţei de a şterge acel cont. În caz afirmativ contul, asemenea pieselor, este şters numele din lista în care se afla, iar daca acel cont care a fost şters era logat totul este resetat, user-ul fiind trimis înapoi la ecranul de început. When este o altă structură de decizie logică, doar că aceasta are un caz, pentru când condiţia este adevărată. Totuşi, avantajul acesteia este că acceptă oricâte comenzi între parantezele sale.

**A-make-admin + a-delete-admin**

(defun a-make-admin(acc-name)

(if (not (userp acc-name)) (return-from a-make-admin "The name is not associated with an account"))

(if (member acc-name \*admin-list\*) (database-print `(,acc-name is already an admin))

(push\_back \*admin-list\* acc-name)))

(defun a-delete-admin(acc-name)

(if (not (userp acc-name)) (return-from a-delete-admin "The name is not associated with an account"))

(if (member acc-name \*admin-list\*) (gremove acc-name \*admin-list\*)

(progn (princ "This account isn't an admin") (princ #\newline))))

Funcţiile a-make-admin and a-delete-admin sunt aproape identice, fiind diferite doar când vine vorba de ce se întâmplă după if. În cazul celei de oferire a rangului de admin, dacă numele este asociat deja unui admin este afişat un mesaj pentru a atenţiona utilizatorul, iar în caz contrar numele este adăugat la lista de admini. Cea de-a doua dacă dă peste un nume care aparţine unui admin îl scoate pe acesta din listă, iar dacă nu este, afişează un mesaj de atenţionare.

**Print-components**

(defun print-components(comp)

(labels ((pres-comps (component clist)

(when clist

(database-print (gethash (car clist) (eval component)))

(pres-comps component (cdr clist)))))

(princ #\newline)

(database-print (list-to-string (gethash 'the-list (eval comp))))

(database-print (list-to-string (gethash 'chars (eval comp))))

(pres-comps comp (gethash 'the-list (eval comp)))))

Suntem într-un final la sfârşitul acestei file. Ştiu că a fost o călătorie destul de lungă, dar de aici este smooth sailing. Ultima funcţie care ne interesează de aici este print-components, start-screen fiind doar o serie de mesaje la finalul căreia este chemat REPL-ul. Print-components este o funcţie care este chemată doar atunci când un utilizator doreşte să construiască un sistem, având rolul de a afişa componentele dintr-o anumită categorie, ce caracteristici au şi lista de proprietăți la fiecare dintre ele. Avem prima dată funcţia locală pres-comps care, printr-o abordare recursivă, trece pe rând prin componente şi afişează datele din dreptul numelui lor. Sub aceasta avem două afişări, prima arătând utilizatorului piesele disponibile, iar apoi lista de caracteristici, în ordinea în care vor apărea.

**database-informations.lisp**

Acum că avem scheletul proiectului putem trece la partea din program care se ocupă cu citirea şi scrierea în baza de date.

Primele 17 rânduri sunt ocupate de o serie de declarări necesare stocării de informaţie: flag-uri, liste şi tabele pentru componente și stream-ul de unde o să citesc datele. Liniile 19 şi 20 sunt două citiri din fişier folosind comanda read-line, care returnează un string, ce este transformat apoi în listă.

**Initialize**

(defun initialize (user-list)

(when user-list

(progn

(setf (symbol-value (car user-list)) (make-hash-table))

(setf (gethash 'password (eval (car user-list))) (read-line file))

(setf (gethash 'orders (eval (car user-list))) (string-to-list (read-line file)))

(initialize (cdr user-list)))))

(initialize \*users\*)

Funcţia initialize primeşte lista de useri şi trece pe rând prin ea, declarând câte un hash-table pentru fiecare persoană, punând în câmpurile “password” şi “orders” informaţiile ce se află în bază. De subliniat aici ar fi faptul că nu folosesc (eval (car user-list)) ci symbol-value, deoarece numele nu este deja o variabilă şi trebuie să-l privesc ca un simbol cu care voi declara una. Sincer să vă spun, nu ştiu de ce merge, reuşesc într-o măsură să înţeleg conceptul, dar dacă e să evaluez (symbol-value (car user-list)) aş primi o eroare, una care nu apare când este în interiorul acelui setf.

**Stock-initialize**

(defun stock-initialize()

(labels ((read-details(destination a-list)

(when a-list

(setf (gethash (car a-list) (eval destination)) (string-to-list (read-line file)))

(read-details destination (cdr a-list)))))

(loop for x in \*components\* do(progn

(setf (gethash 'the-list (eval x)) (string-to-list (read-line file)))

(setf (gethash 'chars (eval x)) (string-to-list (read-line file)))

(read-details x (gethash 'the-list (eval x)))))))

(stock-initialize)

(close file)

Ultima funcţie de citire este stock-initialize, care se ocupă strict de încărcarea datelor despre componente. Cam aceeaşi şmecherie ca la cea de citit detalii despre conturi. Am o funcţie locală care primeşte ca parametri tipul de componente şi apoi lista de componente, prin care trece citind date referitoare la piesa respectivă. Partea interesantă se află abia după aceasta, în corpul principal unde cu ajutorul unei instrucţiuni repetitive, (echivalentul lui for(int i : a) din C++, unde a este un array) trec prin lista de tipuri de componente, pe care am declarat-o mai sus, şi citesc ce piese am în acea categorie, împreună cu ce caracteristici prezintă. În ambele cazuri, citirea detaliilor despre conturi şi despre componente, fix după scrierea acestora se află apelul lor, fiind chemate imediat după începerea programului. După ce am terminat de citit inchid fişierul, deoarece voi avea nevoie să-l redeschid atunci când voi vrea să scriu în el.

O apropiere similară este folosită şi la scrierea înapoi în fişier, în ambele cazuri. Vindictus a fost un proiect început acum câteva luni când abia învăţam despre Lisp şi la anumite linii de cod se vede vârsta pe care o au. La o parte am reuşit să le aduc mai la zi, folosind lucruri pe care le-am învăţat ceva mai târziu, pe când altele le-am lăsat aşa cum erau deoarece nu deranjau cu nimic, sau erau suficient de eficiente. Foarte bine se observă la funcţiile de citire/scriere pentru conturi şi apoi pentru piese. Cele pentru conturi le-am scris la început şi se bazează integral pe metode recursive (pe care le consider încă foarte plăcute vizual), pe când cele destinate componentelor combină recursivitatea cu iterația pentru a-mi face o viaţă mai uşoară.

**Update-database**

(defun update-database()

(setf file-output (open "Vindictus/database-data.txt" :direction :output :if-exists :supersede))

(labels ((update-clients (user-list)

(when user-list

(format file-output (gethash 'password (eval (car user-list)))) (terpri file-output)

(format file-output (list-to-string (gethash 'orders (eval (car user-list))))) (terpri file-output)

(update-clients (cdr user-list))))

(update-components (component clist)

(when clist

(format file-output (list-to-string (gethash (car clist) (eval component)))) (terpri file-output)

(update-components component (cdr clist)))))

(format file-output (list-to-string \*admin-list\*)) (terpri file-output)

(format file-output (list-to-string \*users\*)) (terpri file-output)

(update-clients \*users\*)

(loop for x in \*components\*

do(progn

(format file-output (list-to-string (gethash 'the-list (eval x)))) (terpri file-output)

(format file-output (list-to-string (gethash 'chars (eval x)))) (terpri file-output)

(update-components x (gethash 'the-list (eval x)))))

(close file-output)))

Aceasta este funcţia de scriere în fişier, de data aceasta nu mai am două, ci una singură din cauza acelei prime linii. Când deschid fişierul eu specific că acesta este de ieşire prin “:direction :output” şi dacă există deja ceva în el, să scrie peste (”:if-exists :supersede”). Asta înseamnă ca nu pot să updatez o parte din bază iar pe cealaltă s-o las aşa. E foarte posibil să existe o metodă pentru a face lucrul acesta, dar nu am ajuns încă la un stadiu la care să fiu conştient de ea. Format este pur şi simplu o funcţie de scriere care primeşte un stream si un argument pe care să-l afişeze în acel stream. De asemenea, numele său, “format”, îl cam dă de gol că acesta poate să formateze într-un anumit fel scrisul, doar că acesta este un alt lucru pe care n-am ajuns să-l studiez prea mult. Terpri este un fel de #\newline, pentru a putea scrie pe linia următoare.

Funcția de scriere merge în sens invers al celei de scriere, punând în fişier lista de admini, apoi lista de useri, urmând să scrie informaţiile referitoare la fiecare dintre ei. La fel şi partea pentru componente, scrie în fişier lista de componente, caracteristicile şi apoi informaţiile referitoare la fiecare piesă. După ce am scris toate aceste date închid stream-ul şi funcţia, care la fel ca această filă, se termină.

**database-user-commands.lisp**

Deja ne apropiem tot mai mult de suprafaţă, fiind deja la un nivel destul de apropiat de utilizatorul normal, această filă fiind destinată comenzilor prin care user-ul interacţionează cu Vindictus.

**Login**

(defun login()

(princ "Introduce your name: ")

(let ((login-name (read)))

(if(userp login-name) (ask-password login-name) '(the name introduce is not associated with an account. try again!))))

Prima funcţie este cea de login, care citeşte numele şi verifică dacă exista un cont asociat acestuia, în caz afirmativ continuând procesul de logare cu ask-password.

**Create-account**

(defun create-account()

(princ "If the name you picked is already used you'll have to try again until you are asked to select a password") (princ #\newline)

(princ "Type the name you want to use: ")

(labels ((verify-name()

(let ((name (read)))

(if(userp name) (verify-name) name))))

(let ((name (verify-name)))

(setf (symbol-value name) (make-hash-table))

(princ "Select a new password: ") (setf temp-pass (read-line))

(setf ind-name (find-index (sum-list (map 'list #'char-code (string name)))))

(let((perm-pass (coerce (mapcar #'code-char(hash-it (map 'list #'char-code temp-pass) ind-name)) 'string)))

(setf (gethash 'password (eval name)) perm-pass)

(setf \*logged\* t)

(setf \*logged-account\* name)

(push name \*users\*)

(push nil (gethash 'orders (eval name)))

`(welcome ,\*logged-account\* at "Vindictus!")))))

Create-account este un pic mai mare, necesitând mai multe operaţii pentru crearea unui cont. Prima dată avem o funcţie de verificare a numelui, pe care am lăsat-o special pentru a sublinia cum verificam validitatea unui nume printr-o metodă recursivă versus cum o fac acum printr-o metodă iterativă. Funcţia citeşte un nume şi dacă există deja un utilizator cu acest nume se reapelează până când un nume valid este introdus. După ce variabila “name” memorează un nume valid, este creat un nou hash-table cu acel nume şi în următoarele 4 linii este creată şi salvată o parolă printr-un proces identic cu atunci când o generez pentru logare. După ce contul este creat, setez flag-ul ca să ştie sistemul că este un cont logat şi pun în \*logged-account\* numele noului cont pentru a putea interacţiona cu Vindictus direct, prin contul abia făcut. Poate pare bizar că în câmpul pentru comenzi pun un NIL, dar acesta este pentru a-mi rezerva o linie în fișierul cu informații pentru comenzi, ca să nu fie ocupată de parola altcuiva şi să ajungă astfel să fie dat peste cap totul. La final îi urez noului client un călduros bun venit, folosind din nou quasiquoting.

**Build-pc**

(defun build-pc()

(labels ((add-to-pc (lst-pc price comp piece)

(when piece

(let ((pieces-list (concatenate 'list (gethash 'the-list (eval comp)) (list nil))))

(when (not (eq piece 'blnk))

(if (not (eq (elt (gethash piece (eval comp)) 1) 0))

(progn

(push\_back lst-pc piece)

(decf (elt (gethash piece (eval comp)) 1))

(setf price (+ price (elt (gethash piece (eval comp)) 0))))

'(this component is no longer in stock! please select another!)))

(let ((aux (read)))

(loop while (not (member aux pieces-list)) do (setf aux (read)))

(setf piece aux))

(multiple-value-bind (a b) (add-to-pc lst-pc price comp piece)

(setf lst-pc a) (setf price b))))

(values lst-pc price)))

(princ "Pick a name for your pc: ")

(let ((name (read-line))

(pc-list '())

(price 0))

(push\_back pc-list name)

(princ "Type the name of the piece you want and hit enter.")

(princ "If the component you typed is not available just type the name of another one that is.")

(princ "After you picked enough components of a type use 'NIL' to go to the next component.")

(loop for comp in \*components\* do(progn

(print-components comp)

(multiple-value-bind (a b) (add-to-pc '() 0 comp 'blnk)

(push\_back pc-list a) (setf price (+ price b)))))

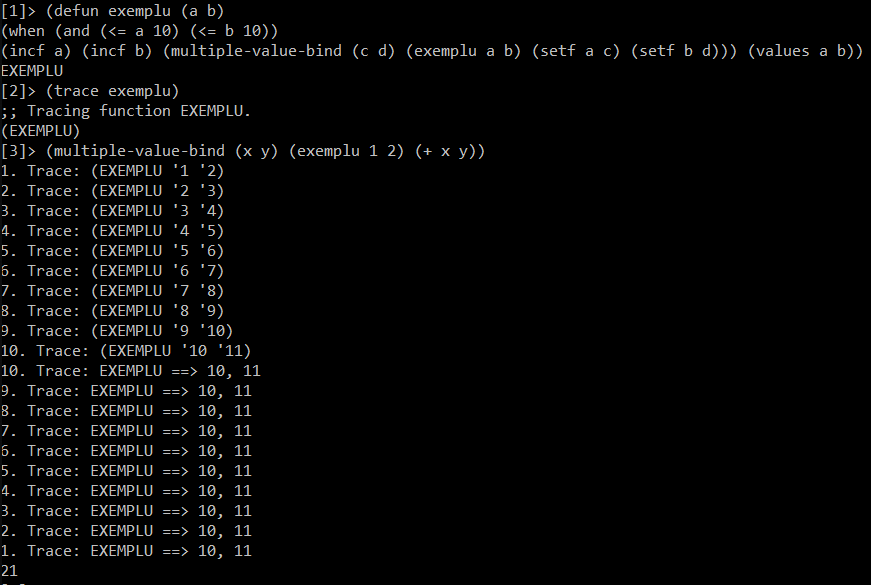
(princ "You purchased the pc: ") (database-print pc-list)

(push\_back (gethash 'orders (eval \*logged-account\*)) pc-list)

(princ "With the price of: ") (database-print price))))

Acum, pentru principala bestie a programului, build-pc este funcţia în jurul căreia se învârte programul, fiind cea prin care utilizatorii pot achiziţiona un PC custom, făcut după piesele alese de ei. Partea recursivă a funcţiei e probabil cea mai urâtă dintre toate deoarece e mai lungă și încurcată decât mi-ar plăcea mie. Prima dată verifică dacă piesa pe care o primeşte este NIL, acesta fiind modul prin care utilizatorul poate să iasă din loop şi să treacă la următoarea componentă, aşa cum este specificat la liniile 47-49. După îmi declar o listă de componente din care pot alege, asigurându-mă că pot alege inclusiv NIL pentru a trece mai departe. Deoarece eu la apel chem funcţia cu o piesă denumită “ ‘blnk ” mă asigur prima dată că nu încerc s-o adaug în lista de componente, ci o ignor. Dacă este o piesă validă verific dacă mai este în stoc (acest detaliu aflandu-se mereu pe a doua poziţie din câmpul unei piese, iar preţul fiind mereu primul detaliu), iar dacă mai este, o adaug (elt este o comandă care primeşte o listă şi un nr mai mic decât lungimea acesteia şi returnează elementul de pe poziţia i, unde i e indexul oferit ca parametru), pe cand dacă nu mai este, afişez un mesaj de atenţionare.

Subprogramul la final va ajunge cu o listă de piese şi un preţ, dar pentru că este recursivă se va întoarce la valorile pe care le avea înainte de apel, ajungând ca la finalul apelului să-mi întoarcă o listă goală şi un preţ de 0 lei. De asta am folosit o nouă chestie numită multiple-value-bind, care primeşte o listă de parametri pe care îi uneşte cu valorile returnate de o funcţie, putând să fie folosiţi în corpul acestui form. Folosind funcţia values Lisp oferă libertatea de a întoarce parametri multipli. Într-un context normal doar prima valoare ar fi folosită, dar în tandem cu multiple-value-bind pot folosi toate valorile întoase. Funcţia merge pe principiul următor: cât timp utilizatorul tot adaugă piese şi eu adaug în lista de componente şi în preţ valorile respective, dar la urmă chem funcţia cu valorile pe care le am şi folosind multiple-value-bind, iau valorile pe care mi le intoarce funcţia şi le bag în lista de componente şi în preţ, când utilizatorul introduce NIL subprogramul întoarce lista completă, împreună cu preţul corect, prin toate apelurile.



Vedem, în acest exemplu menit să prezinte coneptul subprogramului add-to-pc, cum atât timp cât condiţia de a intra in “when” este respectată cele două numere tot cresc, iar când un număr nu mai respectă condiţia se ajunge la acel (values a b) care întoarce din al 10-lea apel la al 9-lea 10 şi 11, valori care sunt atribuite lui a şi b, urmând să i le întoarcă celui de-al 8-lea apel şi tot aşa până la primul apel care le returnează.

Sub funcţie se află o serie de variabile locale, prima fiind numele sistemului care va fi pus la începutul listei şi folosit în principal la acţiunea de ştergere a unei comenzi. Numele este pus în listă, după care se trece prin categoriile de componente fiind construită o listă şi un preţ pentru fiecare, ce vor fi adăugate la finalul apelului în configuraţie şi în preţul total. La final este afişată comanda făcută, adăugată în câmpul de comenzi al contului şi afişat preţul configuraţiei.

**Delete-account + delete-order + view-history + log-off**

(defun delete-account() (a-delete-account \*logged-account\*))

(defun delete-order() (a-delete-order \*logged-account\*))

(defun view-history() (a-view-history \*logged-account\*))

(defun log-off ()

(princ "Are you sure you want to exit? 'Yes/no'") (princ #\newline)

(let ((answer (read)))

(when (eq answer 'yes)

(setf \*logged-account\* nil)

(setf \*logged\* nil)

(setf \*admin-logged\* nil)

(update-database)

'(we hope that we will see you soon!))))

Acestea sunt ultimele 4 funcţii de pe aceasta filă, pe lângă cea de help pe care n-are rost s-o descriu. Dar avem parte de o surpriză plăcută când 3/4 funcţii sunt pur şi simplu apeluri la versiunile destinate adminilor, doar că având ca parametru numele contului logat. Am vrut să evit să las utilizatorilor normali funcţii ce trebuiesc apelate cu vreun parametru, deoarece dacă nu este oferit parametrul, utilizatoul primește o eroare care scoate din REPL-ul custom şi îi oferă acces la orice funcţii din program, inclusiv la variabilele unde sunt memorate datele. Acest bug este evitat în cazul adminilor prin o a doua verificare, ce se află în database-repl.lisp și se asigură că dacă este apelată o funcţie care necesită un parametru, input-ul conţine exact două elemente (funţie+parametru).

Ultimul subprogram despre care vom discuta din această filă este log-off. Prima dată acesta va cere confirmarea acţiunii dorite, în urma căreia va trece printr-un proces identic ca atunci când cineva îşi şterge contul: contul logat o să fie golit, flag-ul pentru online este dezactivat şi baza de date este actualizată.

**database-repl.lisp**

Ultima filă conţine cod referitor la REPL-ul (Read-Eval-Print-Loop) custom pe care îl utilizează Vindictus ca “frontend”, asigurând o experienţă mai plăcută celor ce nu sunt obişnuiţi cu Lisp. Această ultimă parte a fost făcută urmând instrucţiunile din Land of Lisp, de la pagina 93.

**Database-repl**

(defun database-repl()

(let((cmd (database-read)))

(unless (and (eq (car cmd) 'quit) \*admin-logged\*)

(database-print (database-eval cmd))

(database-repl))))

Prima funcţie este REPL-ul în sine unde prima dată este declarată o variabilă cmd, care apelează funcţia database-read. După ce primeşte comanda, verifică dacă nu cumva un admin doreşte să iasă din REPL, pentru a face nişte modificări sau teste, folosind comanda “quit”. Dacă un user ar încerca să iasă n-ar avea permisiunea de a o face, lucrul acesta l-am făcut, din nou, pentru a mă asigura că un user nu poate să iasă din REPL şi să aibă acces la întreg programul. Dacă totul este bine este apelată funcţia database-print pentru a afişa rezultatul evaluării comenzii de către database-eval, după care se reapelează.

**Database-read**

(defun database-read()

(let((cmd (read-from-string (concatenate 'string "(" (read-line) ")"))))

(flet ((quote-it(x)

(list 'quote x)))

(cons (car cmd) (mapcar #'quote-it(cdr cmd))))))

Urmează database-read a cărui scop principal este de a permite utilizatorului să introducă o comandă fără ca să fie obligat să o pună intre paranteze. Funcţia foloseşte read-line, care întoarce un string şi la care concatenează paranteze la început şi la sfârşit, transformându-l într-o listă. Funcţia quote-it este una care doar pune un apostrof în faţa unui element. Folosind cons (în Lisp o listă este alcătuită din cons cells, deci cu ajutorul lui cons, care primește două elemente și le leagă împreună, putem construi liste) alcătuiesc o listă în care primul element (funcţia) rămâne neschimbat, pe când folosind mapcar pun apostrof înaintea celorlalte elemente, pentru a înştiinţa Lisp-ul că eu transmit simboluri, nu variabile.

**Database-eval**

(defun database-eval(input)

(if (member (car input) (if \*logged\* (if \*admin-logged\* \*admin-commands\* \*consumer-commands\*) \*guest-commands\*))

(if (and (member (car input) \*param-req-com\*) (not (= (length input) 2)))

'(this function requires one parameter to access)

(eval input))

'(i do not know the command)))

Database-eval verifică dacă primul element al inputului este o comandă accesibilă. Dacă flag-ul pentru cont logat este activ atunci se mai face o verificare pentru a determina dacă utilizatorul are acces la comenzile de utilizator sau de admin, iar dacă nu este activat flag-ul pentru on, utilizatorul are acces doar la login și create-account. Dacă inputul e o comandă validă atunci este evaluat, iar dacă nu, este afişat un mesaj de informare.

**Tweak-text**

(defun tweak-text (lst caps lit)

(when lst

(let ((item (car lst))

(rest (cdr lst)))

(cond ((eql item #\space) (cons item (tweak-text rest caps lit)))

((member item '(#\! #\? #\.)) (cons item (tweak-text rest t lit)))

((eql item #\") (tweak-text rest caps (not lit)))

(lit (cons item (tweak-text rest nil lit)))

(caps (cons (char-upcase item) (tweak-text rest nil lit)))

(t (cons (char-downcase item) (tweak-text rest nil nil)))))))

Tweak-text este un subprogram care are rolul de a înfrumuseţa output-ul. Aşa cum am amintit în treacăt ca simbolurile sunt case insensitive şi afişate cu toate literele mari, lucru foarte neplăcut vizual. Această funcţie primeşte ca parametru o listă de caractere prin care trece caracter cu caracter pentru a face ajustările necesare. Aceasta foloseşte operatorul logic cond care ar fi echivalentul switch-ului din C++, fiecare caz începând cu câte o condiţie care trebuie îndeplinită pentru a accesa blocul respectiv de cod. Rezultatul acesteia este o noua listă, construită cu ajutorul comenzii cons, care urmează să fie transformată în string. Dacă se află pe un caracter #\space codul trece mai departe fără a face ceva. Când dă peste un semn de punctuaţie schimbă un flag pentru a pune următorul caracter cu literă mare. Atunci când în listă apare #\” codul schimbă un flag pentru a-şi da seama dacă a intrat în interiorul unui citat şi trebuie să lase lucrurile aşa, sau dacă tocmai a ieşit şi trebuie să continue cu schimbările. Dacă niciunul dintre flag-uri nu este adevărat atunci doar va trece caracterul cu literă mică, acesta fiind cazul general.

**Database-print**

(defun database-print (lst)

(princ (coerce (tweak-text (coerce (string-trim "()" (prin1-to-string lst)) 'list) t nil) 'string))

(fresh-line))

Ultimul, dar nu şi cea din urmă, este subprogramul database-print, care primeşte o listă de elemente, foloseşte funcţia prin1-to-string, care primeşte un simbol sau o listă şi returnează un string, pentru a avea mesajul sub formă de şir de caractere, căruia îi sunt îndepărtate parantezele şi transformat prin funcţia coerce în listă de caractere. Această listă este hrănită subprogramului tweak-text, care va returna tot o listă de caractere, dar modificate aşa cum trebuie, aceasta fiind transformată într-un string şi afişat pe ecran, trecându-se la o linie nouă.

**Bibliografie:**

1. Land of Lisp: Learn to Program in Lisp, One Game at a Time! by Conrad Barski
2. <http://www.lispworks.com/products/myths_and_legends.html>
3. <https://www.xanalys.com/>
4. <http://www.lispworks.com/documentation/HyperSpec/Front/X_Master.htm>
5. <https://www.gnu.org/fun/jokes/eternal-flame.en.html>

**Poezie:**

For God wrote in Lisp code

When he filled the leaves with green.

The fractal flowers and recursive roots:

The most lovely hack I've seen.

And when I ponder snowflakes,

never finding two the same,

I know God likes a language

with its own four-letter name.

Now, I've used a SUN under Unix,

so I've seen what C can hold.

I've surfed for Perls, found what Fortran's for,

Got that Java stuff down cold.

Though the chance that I'd write COBOL code

is a SNOBOL's chance in Hell.

And I basically hate hieroglyphs,

so I won't use APL.

Now, God must know all these languages,

and a few I haven't named.

But the Lord made sure, when each sparrow falls,

that its flesh will be reclaimed.

And the Lord could not count grains of sand

with a 32-bit word.

Who knows where we would go to

if Lisp weren't what he preferred?

And God wrote in Lisp code

Every creature great and small.

Don't search the disk drive for man.c,

When the listing's on the wall.

And when I watch the lightning burn

Unbelievers to a crisp,

I know God had six days to work,

So he wrote it all in Lisp.