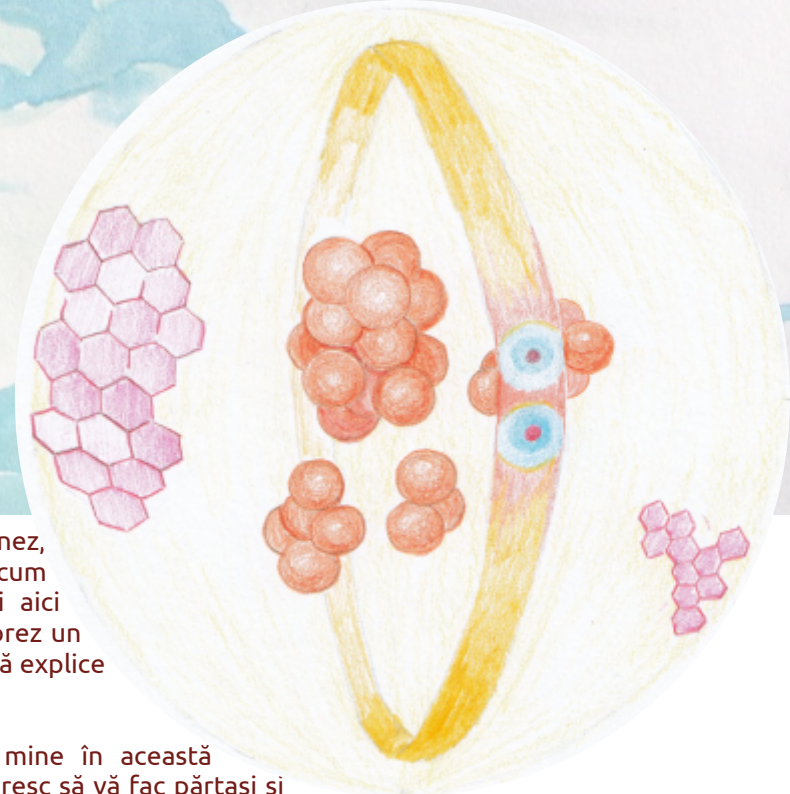


Introducere



Pentru voi!

Dedic acest efort copiilor și vouă, celor curioși, care veți învăța mai întâi prin a înțelege.

Este pentru voi, cei care lucrați în biblioteci și în alte instituții de memorie. Motivul pentru care vi se adresează vouă este nevoia unei noi abordări a practicii în contactul permanent cu informația și cu reprezentările atât de bogate ale acesteia.

Speranța mea se îndreaptă către cei care vor reuși să stăpânească limbajul foarte bine pentru a intra în etapa creativă fără întârziere. Nu este îndeajuns să folosești ceea ce îți este oferit de ceilalți. Pur și simplu este nevoie să poți rezolva problemele reale cu care te confrunți, fără a încerca adaptarea soluțiilor altora. Nu te îndemn să reinventezi roata, ci să-i înțelegi principiile pentru a reuși să faci un vehicul.

Mai dedic acest material celor care trăiesc în solitudine sau sunt privați de libertate, celor cărora viața nu le-a fost ușoară și tuturor celor care având timp la îndemână, ar dori să încerce ceva deosebit, ceva ce le-ar aduce satisfacții mari și care i-ar apropia de un tărâm magic cu infinite posibilități.

O resursă pentru a înțelege ECMAScript, adică JavaScript

Oricine învață mai bine dacă scrie lucrurile pe care dorește să le înțeleagă, iar această carte mă va ajuta și pe mine să înțeleg mai bine concepte și practici care se însușesc uneori cu dificultate, în timp. De cele mai multe ori, atunci când am

de învățat ceva nou, desenez, fac scheme, măzgălesc cum ar arăta. Voi continua și aici pentru că doresc să elaborez un material viu ilustrat care să explice și prin imagine.

Am dorit să vă iau cu mine în această călătorie pentru că îmi doresc să vă fac părtași și unui nou model de a scrie cărți: cel incluziv, care expune totul înainte. Materialele pe baza cărora s-a editat cartea pot fi consultate și pe web sub o licență deschisă. Aceasta este o promisiune către comunitatea celor care doresc să învețe.

Ținta este realizarea unui material de învățare pentru limbajul de programare JavaScript, care să fie eficient în înțelegerea aspectelor dificil de pătruns. Și, acestea nu sunt puține.

Sunt cuprinse experiențe și note strânse după ce am citit și am vizionat multe alte lucrări dedicate acestui limbaj de programare. Acesta nu este primul pe care l-am folosit pentru a rezolva cerințe punctuale. Am început pe vremuri să învăț Pascal dintr-o carte xeroxată. Deh, pe vremea aia nu erau nici computere prea multe, iar cartea tehnică în limba română rară... hmmm... ca și acum, parcă. Apoi am trecut prin BASIC folosind primul meu calculator, un HC (o clonă Zilog) și apoi m-am reapucat de programare din nou odată cu avântul Internetului în anii 2000. PHP-ul a fost experiența de programare care m-a pregătit pentru JavaScript și de aici împreună cu voi ce va urma. Nu te abandona gândului că trebuie să fi avut experiență în computere înainte. O minimă familiaritate cu utilizarea lor este îndeajuns.

Ceea ce am realizat după un efort de câțiva ani de acumulare personală, este faptul că nu există materiale de învățare prea multe în domeniul programării care să explice și cu ajutor vizual extins concepte și situații pe care anumite abstractizări ale unui limbaj le expune celor nefamiliarizați. Vizualizare nu înseamnă să te filmezi cum scrii cod pe care îl povestești. Aici mă gândesc la faptul că este nevoie să desenezi, să vezi cu ochii minții posibilele reprezentări ale abstractizărilor. Această lucrare dorește să ofere îndeajuns de mult suport vizual pentru a realiza scopul de învățare.

Acesta este și un efort de a lărgi baza de acces în limba română către un nou instrument de expresie: programarea. Chiar dacă limbajul de programare își găsește expresia prin cuvintele limbii engleze, nu există niciun motiv să nu pornești pentru că există o barieră de limbă. Setul acestora este mic și nu este nimic dificil în a le înțelege. Amânarea pentru momentul când vei învăța engleza este neproductivă și ce-ar fi o experiență de învățare a unui lucru nou fără mici provocări? Toate cuvintele rezervate în limba engleză vor fi traduse ca înțeles și voi puncta acolo unde este necesară lămurirea termenilor. Nu vei fi singur, te voi secunda acolo unde știu că e greu, unde nu face sens din prima.

Este posibil ca multe dintre interpretările mele sau felul în care explic să nu fie cel canonic, cel predat în școală sau în mediile academice dedicate. Nu este o lucrare orientată către comunitatea de cercetare a domeniului calculatoarelor. Audiența include și pe aceștia, dar baza o constituie cei care au nevoie de o ușă deschisă pentru a prinde curaj să scrie primele programe, pentru „a prinde gustul”, pentru a realiza un potențial pe care doar îl intuiau.

Ținta este de a căuta cea mai bună asamblare a cunoștințelor în dozele cele mai echilibrate pentru ca doritorii să ajungă la nivelul cel mai bun de înțelegere. Finalitatea este dobândirea încrederii pentru a scrie ușor cod funcțional și pentru a lucra cu diferitele biblioteci de cod existente.

Limbajul adoptat este unul dedicat celui care dorește să înțeleagă fenomenul și să ajungă la contextualizare rapidă a anumitor concepte sau abstractizări cu un prag mai ridicat. Pentru a realiza cât mai multe punți între subiectele care au o legătură directă, am ales să repet în anumite puncte cheie câteva informații necesare în defavoarea unor trimiteri seci care să aibă ca efect cât mai puține salturi între segmentele de cunoștințe.

Materialele pot servi și ca date prelucrabile pentru un posibil sistem de învățare dinamic și adaptat pe subiect. În acest sens, unele materiale includ secțiuni intitulate „dependințe cognitive” sau

„alonje”. Mai toate subiectele tratate conțin o secțiune intitulată „mante”, care are scopul de a rememora, consolida și enumera caracteristicile, atributele și efectele cele mai importante.

Pentru că de curând a apărut o nouă versiune a standardului ECMAScript care introduce noi structuri sintactice, am preferat să le introduc în economia explicațiilor chiar dacă pentru unele suportul browserelor nu este încă finisat. Pentru cei care veți citi în timp această carte, bucurați-vă! Veți vedea că în majoritatea cazurilor sunt oferite sintaxele alternative conform ultimelor specificații, care stau alături de cele familiare din ECMAScript 5.

Cred că v-am zăpăcit nițel cu cele două denumiri: ECMAScript și JavaScript. Nu sunt două lucruri distincte. JavaScript este o marcă înregistrată iar ECMAScript la fel, dar aceasta din urmă aparține creatorilor limbajului. JavaScript este ceea ce întreaga comunitate de programatori înțelege a fi iterațiile standardului ECMAScript. Închei povestea ECMAScript / JavaScript spunându-vă că începând cu ECMAScript 6, toate iterațiile vor purta marca anului în care s-a făcut incrementul. De exemplu, în acest an standardul se numește ECMAScript 2017.

Dacă veți căuta instrumentul perfect pentru a învăța, această lucrare nu este locul potrivit. Chiar dacă tot codul a fost testat, chiar dacă fragmentele care explică funcționalitatea au dimensiuni care să permită înțelegerea, sunt conștienți că se poate și mai bine. Mă voi strădui pe viitor.

Fii răbdător!

Te invit să citești pentru a înțelege. Oferă-ți timp așa cum ai proceda cu cea mai dragă activitate a ta.

Ai nevoie de timp!

Dacă poți, caută un loc fără distrageri. Focalizează-te înainte de a te apuca de lucru și ori de câte ori întâlnești greutate în înțelegere, respiră adânc, cu ochii închiși eliminând toate gândurile. Inspiră și expiră de cinci ori, fiind foarte atent doar la respirație și nimic altceva. Dacă tot nu înțelegi ceea ce citești, desprinde-te! E semnul că e nevoie de o pauză ca mintea ta să prelucrez tot ce ai acumulat până la momentul blocajului. Întoarce-te fără amânare și fii răbdător cu tine, cu erorile pe care le faci. Fii răbdător, ai nevoie de timp!

Astăzi întrebarea este cum, nu de ce

Pentru că această lucrare se adresează și specialiștilor din științele informării și prin extensie tuturor celor din domeniul umanișoarelor digitale, accentuez faptul că nu mai poate fi despărțită nevoia de a dobândi noi abilități, de cerința de a le pune în practică. Ceea ce doresc să subliniez este că aproape toate serviciile moderne ale unei instituții de memorie sau de cercetare nu mai pot fi



gândite fără o formă sau alta de prelucrare a datelor și informațiilor, fie că este remodelare, regăsire după șabloane sau prezentarea lor către comunitate ori interconectarea cu alte seturi ori sisteme de gestiune.

Întrebarea pentru toți profesioniștii domeniului este în acest moment **cum?** Cum să învăț să gestionez date, cum să le manipulez, cum să le prezint pentru a fi mai ușor de înțeles celor care au nevoie de ele. Cum să transform catalogul meu, cum să interconectez baze de date europene și globale la acesta, cum să convertesc dintr-un format în altul, dar mai ales cum să exploatez colecția pentru a o face și mai atractivă din perspectiva unui instrument de lucru care să devină permanentă... obișnuință!? Cum să transform instrumentul creat de mine într-un serviciu continuu?

Această carte dorește să ofere o cale prin care să fie dobândite cunoștințe în domeniul programării îndeajuns de avansate încât să permită o a doua natură celor care au nevoie să lucreze cu datele.

Una din țintele acestei cărți este aceea de a explica cunoștințele necesare pentru a manipula, transforma, și a genera dinamic date, precum și care sunt bazele pentru a le încadra într-o formă de prezentare.

Această lucrare îți este adresată și ție specialistul în umanioare digitale - **digital humanities**. Îți va oferi cunoștințele necesare să pui cap la cap diferite scripturi pentru a eficientiza munca curentă, pentru a înțelege și oferi înțelesuri noi unor seturi de date discrete pe care vei ști să le pui cap la cap și să le exploatezi.

Îți va veni ușor să înțelegi de ce funcționează o colecție de software, cum se leagă un framework de un altul și de ce funcționează acest lucru. Capacitatea de expresie și de prelucrare îți va fi ușurată semnificativ prin înțelegerea multor aspecte obscure ale funcționării JavaScript-ului.

Voi încheia pledoaria pentru acest nou drum cu o afirmație care se va dovedi adevărată pe măsură ce veți descoperi bucuria de a lucra cu structuri de cod ce permit prelucrarea.

Textul în dimensiunea lui digitală este o colecție de date în sine. El se agregă și poate fi jalonat pentru a crea structuri inteligibile pentru om cât și pentru mașină.

Textul este o colecție structurată de caractere cu înțeles doar pentru om, care așteaptă un agent software pentru a o reordona și interconecta cu alte surse cu scopul de a oferi o nouă dimensiune valorică. De fapt, acesta este și scopul final pentru care învățăm programare: **extragerea valorii indiferent de formă și destinație pentru a ajunge la noi înțelesuri**.

Pentru că acest material țintește pe cei care lucrează în domeniul umanioarelor, pentru că urmărește un salt la un nou nivel al celor care doresc

să stăpânească arta de a manipula date, vă invit să faceți un efort de a înțelege forma și formatele acestora. Cel mai ades veți vedea că exemplele vor fi create folosind HTML5, care este la rândul său văr bun cu XML-ul pe baza căruia multe dintre datele domeniilor noastre sunt „împachetate” și distribuite. De multe ori, de cele mai multe ori vom folosi pentru „împachetare”, transmitere și manipulare formatul JSON, care este un subset chiar a limbajului de programare JavaScript.

Anti-introducere

La începutul secolului 20 al mileniului trecut, matematicianul Gottlob Frege a căutat „un limbaj al gândului pur”, care să poată fi folosit pentru a prezenta demonstrațiile matematice fără a se apela în vreun fel la mecanismele intuiției, ci doar cele ale logicii. Ceea ce căuta Gottlob Frege era o sintaxă care să fie precisă ea în sine pe care să o poți folosi fără a o interpreta. De fapt, acesta a fost pasul către realizarea visului lui Gottfried Leibniz de a găsi un set de reguli stricte pentru procesarea simbolurilor, ceea ce el numea „limbajul gândurilor” ca reducere a ideilor la un set de concepte nucleu, care prin combinare, să ofere expresivitatea tuturor ideilor indiferent de complexitatea lor. Pentru asta Leibniz ar fi avut nevoie de o anumită algebră cu ajutorul căreia să poată reduce orice propoziție la o valoare de adevăr. Contribuțiile sale în algebra logicii sub forma operatorilor logici au fost completate un secol mai târziu de Gottlob Frege.

George Boole este figura care dezvoltă un set limitat al „legilor gândului”, în fapt o aritmetică care să permită reducerea expresiilor, fie la „adevărat”, fie la „fals”. Cu introducerea operațiilor logice and, notat cu $\wedge$, în română **ȘI**, or, notat cu $\vee$, în română **SAU** și **NOT**, notat cu $\neg$, se poate vorbi despre algebra Booleană ca fundament al domeniului computației moderne. Gottlob Frege face un pas înainte și în căutările sale pentru a descoperi un limbaj formal artificial cu o sintaxă care să nu necesite interpretare, va completa în mod fericit algebra booleană introducând cuantificatori noi precum $\forall$ însemnând oricare sau cu sensul de tot sau $\exists$ însemnând fie, există, pentru o parte. Deodată cu aceste cunoștințe acumulate, domeniul matematicii era pe drumul de a formaliza un limbaj exact de care avea nevoie pentru a algoritmiza demonstrațiile și astfel logica matematică a căpătat noile mijloace de expresie. Am menționat deja termenul de algoritm. Un algoritm (combinația dintre latinul **algorismus** în onoarea matematicianului persan Muhammad ibn Mūsā al-Khwārizmī și grecescul **arithmos**, care înseamnă număr) este o metodă care urmărește o rezolvare pas cu pas a unei

HTML5

Hyper Text Markup
Language<https://www.w3.org/html/>

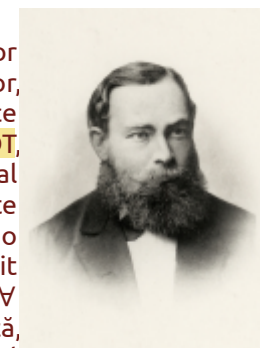
XML

Extensible Markup
Language<https://www.w3.org/TR/xml11/>

Gottfried Wilhelm
Leibniz

1 iul. 1646 – 14 nov. 1716

Domeniul Public

Wikipedia: https://en.wikipedia.org/wiki/Gottfried_Wilhelm_Leibniz

Gottlob Frege

8 nov. 1848 – 26 iul. 1925

Domeniul Public

Wikipedia: https://upload.wikimedia.org/wikipedia/commons/9/99/Young_frege.jpg

probleme. Un exemplu antic de algoritm este metoda lui Euclid de a calcula cel mai mare divizor comun.

Mai târziu, mult mai târziu, în 1936, matematicianul Alan Mathison Turing propune o mașină de calcul automată, care propunea stocarea algoritmilor ca programe, iar în 1937, Claude E. Shannon aplica logica booleană pe circuitele electronice. Din acest moment evoluția domeniului computerelor a accelerat până la nivelul actual.

Efortul de a învăța un limbaj de programare este răsplătit prin însușirea unui set de reguli formalizat cu ajutorul căruia să poți manipula date, informații și să extinzi gândul în lumea materială dacă dorești. Câștigul este cel al căruia care are la îndemână instrumentarul a cărui aplicare poate părea a fi cel mai apropiat lucru de magie.

Orice tehnologie îndeajuns de avansată nu se poate distinge de magie. (Arthur C. Clarke)

Mică anatomie a limbajului



Când vorbim respectarea standardul international.org/

Programarea, ca și activitate, ar fi util să fie înțeleasă ca o limbă străină nouă, pe care ai nevoie să o înveți pentru a comunica cu un străin - computerul. De aceea, se numește și limbaj. ECMAScript, JavaScript sau JS ca și limbaj are o gramatică proprie cu toate regulile sale, de la modul în care înșiri caracterele, până la modul în care faci enunțurile pentru a avea sens și pentru un computer atunci când le evaluează.

Vă va ajuta să înțelegeți că scrierea unui fragment de cod este precum scrierea unei fraze foarte lungi constituită din propoziții separate prin punct și virgulă. Dar această frază este forma cristalizată a unui algoritm. Să lămurim câteva lucruri de la bun început.

Ai o problemă pe care dorești să o rezolvi și în acest sens, creezi pași necesari. Nu te sfii să scrii acești pași pe o coală de hârtie. Adevărata programare începe de la faza de creion și hârtie. În plus, este dovedit științific că soluțiile se văd

JavaScript sau ECMAScript (titlul standardului) a pornit ca motorul dinamicii paginilor web, a prins viteză și a ajuns să devină un limbaj de programare cu uz general. Acest lucru înseamnă că poate fi folosit la mult mai multe lucruri în afara intențiilor sale originare. Avantajele folosirii JS pornesc de la server (Node.js), până la aplicațiile rulate în browserul web al utilizatorului.

vorbim de JavaScript, de fapt despre o implementare, adică de tuturor regulilor pe care le impune ECMAScript - <<http://www.ecma-memento/TC39.htm>>.

din lucrul cu obiecte, din interacțiunea gândurilor cu lumea fizică. Succesiunea tuturor pașilor identificați de tine se numește **algoritm**.

Transpunerea unui algoritm într-o soluție tehnică, se numește **program**, care este o succesiune de **instrucțiuni** pe care computerul le înțelege. Pentru a scrie un program, alegi un limbaj de programare, iar în cazul nostru, am ales deja: ECMAScript, adică JavaScript. Mai adaug doar că o transpunere a unui algoritm într-un program este o transpunere într-un **limbaj formal** (regulile sale interne îl formalizează).

Gata, începem! Avem nevoie de o privire generală asupra limbajului. Aaa, eram să uit. De ceva vreme a ieșit o nouă versiune a standardului. Comunitatea de programatori îi spune ECMAScript 6 iar celei anterioare ECMAScript 5. În cuprinsul materialelor veți găsi referințe la standard așa sau mai simplu ES5 și ES6. ECMAScript, ES5, ES6, JavaScript, JS, ECMAScript 2017, toate, fiecare vorbesc de aceeași realitate: limbajul de programare ECMAScript.

Vom pune sub microscop atomii, moleculele și însăși substanța din care este alcătuit tărâmul ECMAScript.

Am identificat o definiție a ceea ce este un limbaj de programare și vom debuta cu ea pentru a vă oferi imaginea completă de la bun început.

Un ansamblu de primitive și o mulțime de reguli care guvernează modul în care aceste primitive pot fi combinate pentru reprezentarea ideilor mai complexe, constituie un limbaj de programare. (Brookshear, J. Glenn. Introducere în informatică. Editura Teora. 1998)

Standardul ECMAScript 2016 este un „limbaj de programare” cu aplicativitate largă. Inițial ECMAScript a fost dezvoltat ca un limbaj de scriptare (fragmente de cod de mici dimensiuni cu aplicativitate strictă pentru a dinamiza paginile web), dar a evoluat într-unul care este aplicabil de la pagini web la roboți.

JavaScript este un limbaj de programare creat de Brendan Eich în perioada când lucra la compania Netscape. ECMAScript, adică numele standardului în baza căruia avem JavaScript, este rezultatul unui efort colaborativ care a pornit în 1996, un an mai târziu fiind

publicată prima ediție. În aprilie 1998 devine standardul internațional ISO/IEC 16262.

Orice limbaj de programare este caracterizat de o sintaxă proprie (un set de reguli care îmbină cuvintele astfel încât să le înțeleagă compilatorul), un înțeles al combinațiilor de cuvinte (semantică) care să reflecte ceea ce intenționezi cu programul și un set de cuvinte pe care compilatorul să le înțeleagă a fi ale limbajului de programare (vocabular sau cuvintele cheie).

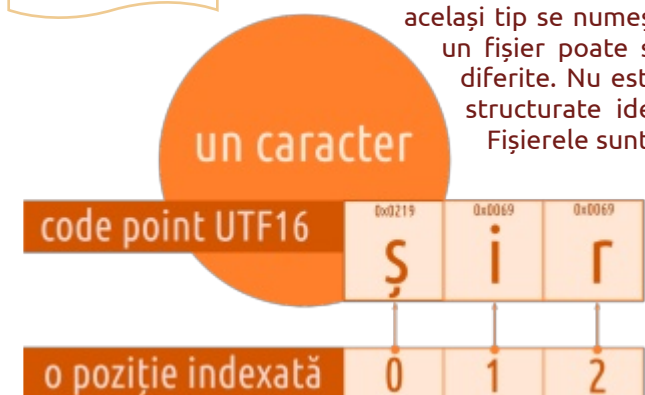
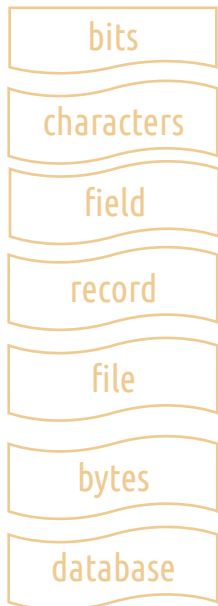
Pentru a face o anatomie, vom construi tărâmul JavaScript de la cele mai mici componente precum un fizician din domeniul particulelor pornește de la atomi.

Date, date, date

Scrierea unui program are drept țintă manipularea unor date pentru a le transforma în diverse scopuri.

Datele pe care un computer le poate manipula au o structură și dezvoltă o adevărată ierarhie pe măsură ce urmărim felul în care sunt reprezentate într-un computer. Cea mai mică unitate de date este bitul - digitul binar, care poate fi 0 sau 1 (în lb. engleză **bits**). Urcând o treaptă mai sus, avem caracterele despre care știm că sunt reprezentări pe 16 biți codate după standardul UTF (în lb. engleză **characters**). Pășind încă o treaptă ajungem la câmpuri de date (în lb. engleză „fields”). Mai multe câmpuri de date puse împreună formează o înregistrare (în lb. engleză **record**). O înregistrare specifică este gruparea a mai multor câmpuri înrudite. Cel mai bun exemplu este o înregistrare de catalog bibliografic indiferent de formatul de reprezentare. Toate câmpurile unei înregistrări unice sunt în relație directă cu aceeași entitate pe care o descriu; căreia îi aparțin. Gruparea mai multor înregistrări înrudite, care sunt de același tip se numește fișier (în lb. engleză **file**).

Totuși un fișier poate strânge date diferite după scheme diferite. Nu este obligatoriu ca toate datele să fie structurate identic sau să conțină aceleași date. Fișierele sunt organizate pe mediul de stocare ca secvențe de 8 biți numite în limba engleză **bytes**. Ultimul stadiu de ordonare a datelor este baza de date organizată pe tabele. Tabelele conțin înregistrări, care la rândul lor conțin câmpurile de date.



Caracterele folosite

E timpul să intrăm în rolul de programator deja. Ești în fața editorului de text preferat gata să redactezi primul tău program. În fereastra editorului, pentru a te face înțeles computerului, vei redacta codul prin înșiruirea de caractere în cuvinte, sintagme, enunțuri și așa mai departe.

Literele, pentru computer, nu sunt decât niște coduri alfanumerice în baza căora poate afișa un anumit caracter pe ecran.

Caracterele folosite pentru a scrie codul sursă ca și codare, respectă standardul internațional Unicode, care asociază coduri alfanumerice individuale pentru fiecare (Latin, Chirilic, etc.). Mai exact, fiecare secvență alfanumerică de codare se numește **punct de cod specific UTF16** (Unicode Transformation Format). Reține faptul că toate caracterele de lucru pentru un computer sunt pur și simplu coduri convenite la nivel internațional.

Caracterele cu rol special

JavaScript este un limbaj de programare folosit îndeosebi la manipularea șirurilor de caractere indiferent ce reprezintă acestea pentru noi oamenii.

Există câteva caractere care necesită chiar acum, la început de drum, o atenție specială. Acestea sunt: ' (ghilimele simple), " (ghilimele duble), \n (new line - linie nouă), \r (carriage return - retur de car), \t (tabulator orizontal, acest caracter apare când apeși tasta TAB), \v (tabulator vertical), \b (backspace - înapoi spre stânga cu ștergerea unui caracter), \f (form feed - salt pagină nouă la dispozitivul de imprimare), / (slash) și \ (backslash).

De ce necesită o atenție specială? Pentru că intră în componența șirurilor de caractere de lucru. Le vom întâlni în analiza textelor și vor crea probleme prin obținerea unor rezultate neașteptate dacă acum, la acest moment de început, nu le dăm cea mai mare atenție. De ce sunt speciale? Pentru că aceste caractere au însemnătate pentru înțelegerea textului, dacă acestea fac parte dintr-un fragment de text, dar în egală măsură au valoare și pentru motorul JavaScript la momentul analizei codul sursă.

Să spunem că avem următorul fragment de text pe care dorim să-l prelucrăm cu un program.

Acesta este un text demonstrativ care va enumera caracterele cu înțeles special pentru JavaScript. Problema apare când în text folosim 'cită în engleză cu ghilimele simple', poate "un citat în engleză între ghilimele duble", poate folosim un slash: /, ori un backslash \ sau vorbim despre carriage return \r sau despre new line \n, etc.

Acum avem o problemă. Dacă am introduce acest text într-o variabilă în

vederea prelucrării, folosind ghilimelele simple, să zicem, motorul JavaScript, la momentul în care va citi conținutul variabilei, va întâlni primul caracter ghilimele simple și va considera că acolo se închide fragmentul de text pentru analiză. Ceea ce se va întâmpla este că restul, pur și simplu va fi trunchiat și

```
var textDeAnalizat = 'Acesta este un text demonstrativ care va enumera caracterele cu înțeles special pentru JavaScript. Problema apare când în text folosim "cită în engleză cu ghilimele simple", poate "un citat în engleză între ghilimele duble", poate folosim un slash: /, ori un backslash \ sau vorbim despre carriage return \r sau despre new line \n, etc.';
```

ignorat ca ceva neinteligibil. De aceea este nevoie să marcăm în text aceste caractere cu însemnătate

specială printr-un **caracter de exceptare** (în engleză **escape sequence**), care să semnaleze motorului faptul că acel caracter face parte din text și nu este un semnal adresat lui să facă o anumită operațiune. **Caracterul de exceptare este un backslash** care se aplică înaintea caracterelor cu înțeles special. Da, da, chiar și în cazul lui backslash pui un backslash înainte pentru ca JavaScript să-l considere un caracter al resursei text și nu un semnal. Pe românește îi spui computerului: prietene, caracterul precedat de backslash, te rog să nu-l interpretezi la valoarea sa specială pentru limbajul de programare, ci consideră-l parte a fragmentului de text analizat.

Dacă veți încerca exemplul de mai sus în **Console** (deschide browserul preferat, apasă **F12**, mergi în tabul **Console**), adică dacă veți pune acest fragment în Console, chiar veți genera și o eroare de sintaxă pentru că toate caracterele acelea cu rol special pentru JavaScript, au darul să creeze erori dacă sunt întâlnite în fragmentele de text pe care le alegeți pentru a lucra cu ele. Hai, poți spune că ai intrat bine în pâine: ai aflat ce-i o variabilă și că aceasta poate ține fragmente de text și ai experimentat puțin și cu instrumentul care-ți va fi cel mai bun prieten de acum înainte: **Console**.

Cum ar arăta exemplul nostru corectat?

Acum, că nu avem nicio eroare, aflându-te în Console, pur și simplu apelează numele variabilei pentru a vedea textul și ce-a mai rămas din el (pur și simplu scrii numele variabilei **textDeAnalizat** și dai ENTER).

Am corectat doar până la **\r** pe care l-am lăsat așa înadins. Pune așa fragmentul și vezi ce se întâmplă. Da, combinația **\r** pur și simplu a dispărut din text. De ce? Pentru că indică faptul că în cazul în care se va trimite la imprimată acel text, el fa fi tăiat de dinaintea lui **\r** și reluat pe următoarea linie. Asta înseamnă retur de car, adică capul de imprimare trece pe o nouă linie. Uneori, citind literatură de specialitate veți mai vedea și secvența CR/LF (Carriage return/Line feed), care este aceeași comandă trimisă unei imprimante: pune capul de imprimare (carriage) la capul liniei (adică la stânga - imprimarea caracterelor se face de la stânga la dreapta) și ridică pagina cu o linie, adică cu un rând (line feed).

Acum corectează tu textul fiind aflat în Console. Ca să aduci în vizor variabila, chemi din istoricul operațiunilor linia în care introduceai textul în variabilă

(apăsarea de două ori a tastei săgeată sus ar trebui să aducă în linia de comadă variabila) și pur și simplu corectează **\r** adăugându-i un backslash în față chiar lângă cel existent deja și dă ENTER. Variabila va fi rescrisă ca și conținut și apelându-i numele încă o dată vei vedea la locul său combinația **\r**.

Ce ai mai observat? Că am omis să pun backslash la combinația **\n**. Acest lucru a produs un efect interesant: **\n** a dispărut din text, dar restul textului a trecut pe o linie nouă. Acesta este efectul **new line**, dacă nu are backslash în față. Dă comandă computerului să înlocuiască combinația **\n** cu o rupere a textului afișat pe o nouă linie de ecran.

Să trecem în continuare prin câteva cazuri fără de care nu poți scrie cod care să și funcționeze corect.

Cazul ghilimelelor. În JavaScript, fragmentele de text pot fi introduse în variabile folosind două tipuri de ghilimele care marchează modul de redactare a citatelor în limba engleză, adică ghilimele simple și ghilimele duble. Ai observat și tu că nu am insistat la exemplul analizat asupra combinației tipurilor de ghilimele. De ce? Pentru că atunci când avem un caz simplu în care avem un text introdus ca resursă de lucru într-o variabilă în interiorul ghilimelelor simple, vom avea grijă ca doar ghilimelelor simple din text ca parte a sa, să fie **exceptate**. Ghilimelele duble nu au nevoie de exceptare, dar buna practică spune ca toate aceste caractere cu înțeles special să fie exceptate. Eu am fost mai leneș la exemplificare pentru că m-am bazat pe mecanismul de lucru al JavaScript și anume că în cazul în care în text am doar ghilimele duble pot să le las fără a le excepta cu backslash.

În cazul ghilimelelor, ca bună practică, se vor folosi ghilimele simple pentru declararea șirurilor de caractere pentru că, fiind un limbaj strâns legat de markup-ul paginilor web, ar putea cuprinde ghilimele duble ca parte a fragmentelor de pagină web construite dinamic.

Exemplul cu fragmentul de cod HTML este exemplificarea perfectă cu imbricarea acceptată a unor perechi de ghilimele duble între ghilimelele simple care delimitează resursa de text.

```
var fragmentHTML = '<a href = "http://ceva.ro">Siteul Ceva.ro</a>';
```

Exemplul corectat

```
var textDeAnalizat = 'Acesta este un text demonstrativ care va enumera caracterele cu înțeles special pentru JavaScript. Problema apare când în text folosim "cită în engleză cu ghilimele simple", poate "un citat în engleză între ghilimele duble", poate folosim un slash: /, ori un backslash \ sau vorbim despre carriage return \r sau despre new line \n, etc.';
```

Separatoarele sintactice pentru delimitarea fragmentelor de cod

Se pune întrebarea: cum știe JavaScript să citească corect codul sursă, care este la rândul său un text pentru a face inteligibile unitățile sale funcționale, cele care produc înțelese pentru computer. În acest sens, nu numai JavaScript, ci toate limbajele de programare folosesc ceea ce se cheamă **separatoare**. Poți să ți le imaginezi precum separatoarele colorate din dosare, care fragmentează vizual părțile componente ale conținutului. Să vedem care sunt acestea.

Acoladele `{ }` au rolul de a indica mediul în care se va executa codul în JavaScript. Cel mai adesea veți vedea că indică **blocurile de cod ale funcțiilor**. Între acolade veți introduce liste de instrucțiuni și expresii specifice JavaScript separate prin punct și virgulă `;`. Acoladele și parantezele rotunde împreună cu punct și virgulă împart codul în calupuri ușor de urmărit vizual, dar mai mult de atât oferă „sens”, face „lizibil” codul și pentru computer prin **separarea** fragmentelor componente. Acoladele mai au și un rol special: delimitează spații cu un rol special. Creează niște „grădini private” care sunt **blocuri de cod** la care accesul se poate face doar dacă ești în acea grădină.

JavaScript este un limbaj atent la diferența între majuscule și minuscule

JavaScript este un limbaj de programare **case sensitive**, adică ține cont dacă scrii folosind majuscule. Pe scurt, șirul de caractere **ceva** este fundamental diferit de șirul de caractere **Ceva**. Și să-ți spun un mic secret de ce este diferit. Mai ții minte când mai sus spuneam că fiecare caracter pentru computer este un număr? În cazul nostru codul din spatele lui **c** mic de tipar este diferit de codul din spatele lui **C** ca majusculă.

Textul sursă / cod sursă

Vorbeam mai devreme despre codul sursă, despre textul pe care-l introduci tu într-un fișier și pe care-l consideră computerul a fi codul

sursă pentru programul creat. După felul în care este **consumat** de motorul JavaScript, textul sursă sau **codul sursă** poate fi de două tipuri: **Script** (un script) sau un **Module** (modul). Codul sursă, sau mai simplu **sursa** este scrisă de tine în editorul preferat, pe care ai salvat-o ca fișier cu extensia **.js**: **ceva.js**.

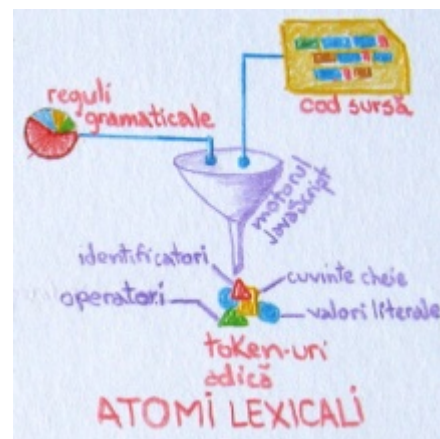
Codul pe care-l scrii poate fi redactat pe mai multe linii deoarece pentru JavaScript **spațiile**, **taburile** și **line breaks**-urile (trecherile pe linie nouă) sunt pur și simplu considerate a fi **spații albe** care nu au puterea să influențeze evaluarea codului în niciun fel.

Elemente de input - elemente de intrare în vederea evaluării

Spune standardul că mai întâi textul codului sursă este parcurs pentru a-l „converti într-o succesiune de elemente de input” folosindu-se regulile lexicale. Aceste elemente de input sunt: **atomii lexicali**, **semnele de încheiere a rândului** și **spațiile albe**.

Atomi lexicali

Toate elementele lexicografice care constituie codul în sine, cu excepția spațiilor albe și a comentariilor, se numesc **token-uri**, adică, am zis eu pe românește **atomii lexicali**. Acești **atomii lexicali** (token-ii) sunt rezultatul parcurgerii unui fragment de cod (codul sursă) căruia i se aplică regulile lexicale specifice gramaticii impuse de standardul ECMAScript.



Ca să-ți vină ușor să înțelegi, îți poți imagina un giuvaergiu, care dintr-un maldăr de pietre prețioase (codul sursă), ia una câte una (fragmente de cod), pentru a-i identifica caracteristicile și în final pentru a le pune pe fiecare după sortare în cutiuțele pregătite special înaintea asamblării într-o diademă deosebită (programul nostru care tocmai a făcut ceva spectaculos).

Rezultatul apare în urma aplicării regulilor de identificare a componentelor ce formează fragmentele „inteligibile” pentru computer din șirul de text de

intrare. Dacă-ți vine mai ușor este ca o analiză gramaticală în care identifici părțile de propoziție, ce sunt acestea din punct de vedere al părților de vorbire și așa mai departe.

Atomii sunt de mai multe tipuri: **cuvintele cheie**, **operatorii**, **identificatorii**, **valorile literale** și **template-urile** (fragmente de text mai lungi afișate pe mai multe linii).

Trebuie să te avertizez de faptul că toate **cuvintele rezervate** folosite de JavaScript sunt în limba engleză. Fondul lexical este cel al limbii engleze.

Line terminators - combinațiile semnal pentru încheierea rândurilor

Capetele de rând sunt folosite pentru a mări lizibilitatea codului și pentru a separa atomii lexicali unii de ceilalți. În general, între doi atomi lexicali stă un capăt de rând.

Ne-am mai întâlnit cu aceste combinații de caractere atunci când am explorat caracterele cu rol special. Vom suplimenta informațiile pe care le avem deja prin detalii privind natura și comportamentul fiecărei combinații în parte după cum urmează:

line-feed s-ar traduce în română ca o comandă: **mergi pe line nouă** este combinația de caractere `\n`, abreviat uneori ca LF (prescurtare de la **Line Feed**) sau NL (prescurtare de la **New Line**). Acesta este un caracter special (combinația dintre cele două caractere `\` și litera **n** este considerat în acest caz un singur caracter), care marchează faptul că motorul trebuie să continue afișarea sau prelucrarea aplicată unui fragment de text începând cu o nouă linie pe ecran. Hai, rulează în Console următorul fragment pentru a vedea cu ochii tăi ce se petrece: `print("ceva\nccapat");`. Fragmentul **ceva** a rămas pe o linie în timp ce fragmentul **ccapat** a ajuns pe o nouă linie sub cel anterior.

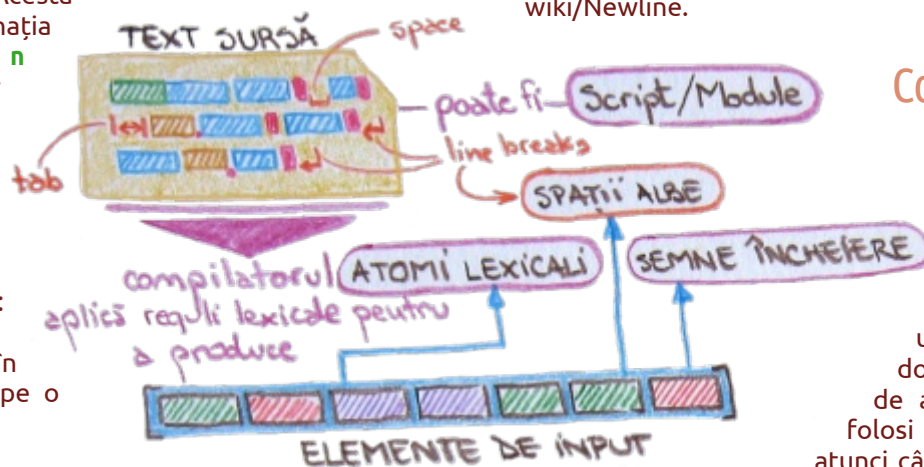
carriage return este reprezentat prin combinația `\r` și mai este întâlnit ca abrevierea CR - **carriage return**). Se comportă ca o comandă directă care spune: **trage înapoi carul de imprimare** și începe să imprimi textul începând cu o nouă linie. Acest caracter este o reminiscență a utilizării vechilor mașini de scris, care migrând la cele electronice, încă aveau

nevoie de un caracter special care să spună mecanismelor electromecanice să se întoarcă pe același rând și apoi „săltând” pagina de imprimare cu un rând: `print("ceva\rccapat");`. Efectul este vizibil, dacă fragmentul care conține caracterul special ar fi trimis către o imprimantă. S-ar produce aceeași rupere ca în cazul caracterului special `\n` numai că de această dată pe hîrtia imprimată. Efectul în Console este că `\r` dispăre din text iar cele două cuvinte vor fi lipite. Efectul vizibil pe ecran este alipirea fragmentelor de text acolo unde era `\r`. Caracterul s-a păstrat și în limbajele de programare cu toate că îl veți întâlni mai rar în cazul conținutului unor fișiere precum cele cu extensia TXT mai vechi. Dacă ești curios, poți replica acest lucru salvând din LibreOffice ca **.txt** cu menționarea formatului ASCII. Da, știu, este greu de crezut, dar nu am avut UTF dintodeauna. Ce-i UTF? Citește în continuare și vei afla minunea pe care o oferă acest standard de codare a caracterelor.

line separator: uneori este reprezentat prin combinația de caractere `\n`, iar alte ori în funcție de sistemul de operare este `\r\n`. Efectul este trecerea pe o nouă linie a textului.

paragraph separator este o combinație de caractere care depinde de sistemul de operare utilizat. De exemplu, pe Windows ai un CR urmat de un LF (**line feed**).

Pentru a înțelege mai bine, accesați și materialul explicativ de la <https://en.wikipedia.org/wiki/Typewriter> și <https://en.wikipedia.org/wiki/Newline>.



Comentariile

Acestea sunt utile pentru a documenta codul. La momentul executării codului, comentariile nu sunt luate în considerare, fiind supuse aceluiași tratament precum spațiile albe. Aceste sunt utile doar pentru noi cu scopul de a documenta codul. Sunt două moduri de a introduce comentarii. Se poate folosi dublu slash `//` **comentariu** sau atunci când ai nevoie de comentarii pe mai multe linii `/* comentariu */`. Backslash urmat de steluță urmat de comentarii. Dacă un comentariu pe mai multe linii se extinde pe mai mult de una, acesta va fi considerat în întregime ca un line terminator.



Wikipedia
Typewriter



Wikipedia Newline

Spațiile albe - whitespace

Acestea sunt caracterele „invizibile” cum ar fi spațiul, tasta space, pentru a separa vizual anumite fragmente de cod. Cel mai adesea sunt folosite spațiile (introduse de tasta SPACE) pentru a separa cuvintele pentru a da înțelesul semantic al acestora și taburile pentru a introduce „pauze” vizuale pe ecran cu scopul de a crește gradul de înțelegere și lizibilitate.

Sunt considerate a fi spații albe următoarele: **tab** (`\t` - tabulatorul orizontal, fiind un spațiu cu o anumită întindere), **space** (spațiu), **non-breakable space** (spațiu care nu poate fi fracționat), **line tabulation** (sau vertical tab: `\v`, referindu-se la mișcarea pe verticală a liniilor), **form feed** (se referă la trecerea pe ceea ce este înțeles a fi o pagină nouă `\f`).

Imediat după faza de constituire a **elementelor de input**, acestea mai sunt parcurse încă o dată, aplicându-se din nou regulile gramaticale pentru a identifica cine și ce funcție îndeplinește: care sunt **identificatorii**, **cuvintele rezervate** limbajului, etc. Să analizăm împreună ce conțin elementele de intrare.

Cuvintele cheie

ECMAScript are un set de **cuvinte**

rezervate din limba engleză, care nu pot fi folosite decât în scopul pentru care au fost rezervate. Cuvintele cheie sunt unul din tipurile de **atomi lexicali** și se scriu întotdeauna fără majuscule.

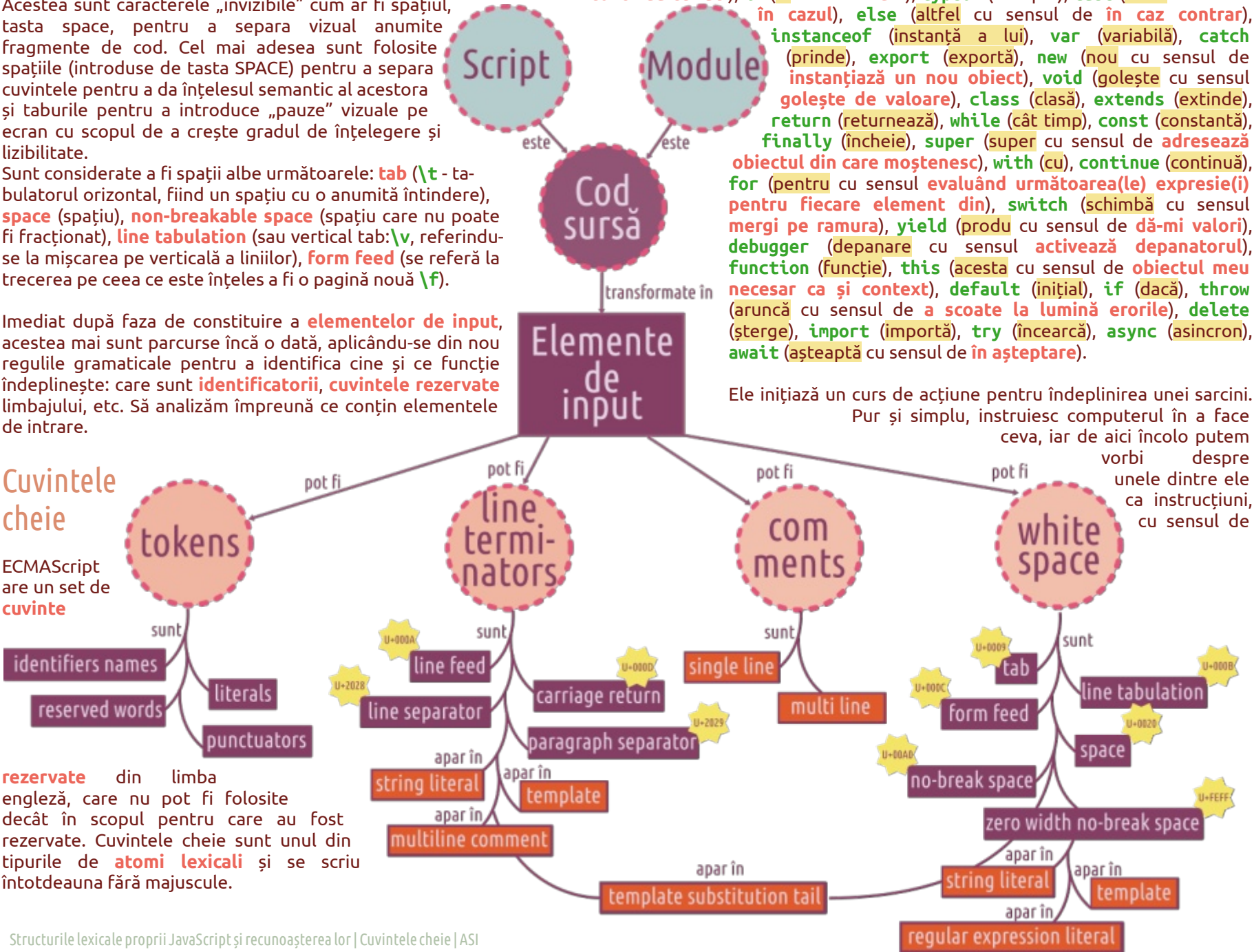


Cuvintele rezervate poartă în sine o acțiune

- async (asincron)
- await (în așteptare)
- break (ieși din execuție)
- case (în cazul)
- catch (prinde)
- class (clasă)
- const (constantă)
- continue (continuă)
- debugger (depanator)
- default (inițial)
- delete (șterge)
- do (continuă ce faci)
- else (în caz contrar)
- export (exportă)
- extends (extinde)
- finally (încheie)
- for (pentru element din)
- function (funcție)
- if (dacă)
- import (importă)
- in (din)
- instanceof (instanța lui)
- new (un nou obiect)
- return (returnează)
- super (obiectul părinte)
- switch (mergi pe ramura)
- this (obiectul context)
- throw (scoate erorile)
- try (încearcă)
- typeof (de tipul)
- var (variabilă)
- void (golește de valoare)
- while (cât timp)
- with (cu)
- yield (dă-mi valori)

Le vom enumera aici cu traducerea lor pentru a vă familiariza la un prim contact: **break** (întrerupe cu sensul de **ieși din execuție**), **do** (fă cu sensul de **continuă ce faci**), **in** (în cu sensul **din**), **typeof** (de tipul), **case** (cazul cu sensul **în cazul**), **else** (altfel cu sensul de **în caz contrar**), **instanceof** (instanță a lui), **var** (variabilă), **catch** (prinde), **export** (exportă), **new** (nou cu sensul de **instanțiază un nou obiect**), **void** (golește cu sensul **golește de valoare**), **class** (clasă), **extends** (extinde), **return** (returnează), **while** (cât timp), **const** (constantă), **finally** (încheie), **super** (super cu sensul de **adresează obiectul din care moștenesc**), **with** (cu), **continue** (continuă), **for** (pentru cu sensul **evaluând următoarea(le) expresie(i) pentru fiecare element din**), **switch** (schimbă cu sensul **mergi pe ramura**), **yield** (produ cu sensul de **dă-mi valori**), **debugger** (depanare cu sensul **activează depanatorul**), **function** (funcție), **this** (acesta cu sensul de **obiectul meu necesar ca și context**), **default** (inițial), **if** (dacă), **throw** (aruncă cu sensul de **a scoate la lumină erorile**), **delete** (șterge), **import** (importă), **try** (încearcă), **async** (asincron), **await** (așteaptă cu sensul de **în așteptare**).

Ele inițiază un curs de acțiune pentru îndeplinirea unei sarcini. Pur și simplu, instruiesc computerul în a face ceva, iar de aici încolo putem vorbi despre unele dintre ele ca instrucțiuni, cu sensul de



comenzi ferme.

Buna practică spune că toate exprimările intenției programatorului, hai să le numim **enunțuri** (iar liniile de cod cu instrucțiuni - **statements**), în JavaScript trebuie să fie încheiate prin punct și virgulă (;), chiar dacă motoarele care implementează ECMAScript, la momentul evaluării codului, introduc automat prin mecanismul de **automatic semicolon insertion** acest caracter.

Da, da. Programatorii sunt creaturi comode și motoarele permit anumite facilități. Unii aleg această practică înadins. Personal, mă feresc și pun semnele de punctuație pentru că astfel, codul devine lizibil, ochii deprind automatisme de citire și de aici și o mai mare eficiență.

Bine, bine. Da' care-i treaba cu **momentele ZEN**? Pe parcursul acestei călătorii de descoperire și autodescoperire, voi jalona conținutul cu astfel de momente, care vor fi propoziții sau fraze cu o sarcină precisă: să fie chintesența informației analizată defalcat. De ce moment ZEN? Pentru că este ca un exercițiu de meditație, care conduce la identificarea cu informația prin asimilarea ei.

Automatic semicolon insertion - introducerea automată a lui punct și virgulă

În JavaScript, enunțurile (**statements**), care sunt echivalentul propozițiilor din limbajul nostru de zi cu zi, trebuie să se termine cu punct și virgulă, iar acolo unde semnul grafic nu a fost scris de programator, codul se supune mecanismului ASI - **Automatic Semicolon Insertion**, care introduce de la sine putere semnul grafic.

Reguli de introducere:

- imediat înaintea acoladei de închidere **}**,
- atunci când șirul de token-uri nu poate fi tratat unitar **x - y**, de exemplu,
- imediat după operatorii și sintagmele care nu mai permit altă dezvoltare la nivelul expresiei sau a programului: sufixurile de operare **++** și **--** și cuvintele cheie **continue**, **break**, **return**, **yield** și **yield*** și **module**.

Declarațiile și instrucțiunile beneficiare ale acestui mecanism sunt:

- instrucțiuni simple,
- declarații de variabile: **var**, **let**, **const**,
- declarațiile de module: **import**, **export**,
- declararea expresiilor,
- declararea intenției de a folosi depanatorul de cod (**debugger**),
- instrucțiunile **continue**, **break**, **throw**
- și **return**.

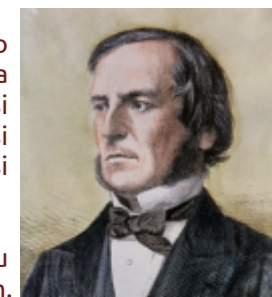
Există mai multe curente de opinii care au condus la diferite stiluri de redactare a codului sursă. Veți întâlni foarte mult cod scris fără punct și virgulă care să marcheze finalitatea enunțului. Unii consideră acest lucru acceptabil, dar vă invit în partea cealaltă, a celor care scriu foarte corect codul și care vor pune întotdeauna punct și virgulă la încheierea unui enunț după cum ne-a intrat în sânge ca atunci când încheiem o propoziție în scris să punem punct.

Operanzii

Pentru a ajunge la un rezultat avem nevoie mai întâi de niște valori, de niște date cu care să lucrăm. Operanzii, ca denumire, vin din matematică. Mda, știu, nu scăpăm... Nu te descuraja așa ușor, pur și



simplic programarea este o dezvoltare a matematicii și de acolo își trage și denumirile pentru „chestiile” cu care operăm. Am zis operăm, nu?!



George Boole

1815-1864

Este creatorul logicii matematice moderne și a algebrei booleene.

Domeniu public

[https://commons.wikimedia.org/wiki/](https://commons.wikimedia.org/wiki/File:George_Boole_color.jpg)

[File:George_Boole_color.jpg](https://commons.wikimedia.org/wiki/File:George_Boole_color.jpg)

g

Păi ce poți face cu niște operanzi altceva în afară de a opera cu ele? Ce? Nu știu. O adunare, o înmulțire... mai multe operațiuni, unele grupate cu paranteze. Hai că mai vedem ce și cum putem combina în expresiile formate.

Operatorii (operators)

Sunt caractere sau combinații de caractere care au rolul de a stabili o relație între doi operanzi. Este exact ca în matematică. Rolul operatorilor este de a ajunge la un rezultat în urma „evaluării”. De exemplu, când ai expresia **1 > 0**, operatorul de comparație **>** va oferi rezultatul, care este o valoare boolean **true** (ești încă aici? nu te-am pierdut, nu?!).

În esență, putem spune că majoritatea rezultatelor, atunci când scriem cod, provin din astfel de evaluări. Ăăă, cum **boolean**, prietene? Da. E o valoare care testează adevărul și poartă numele de boolean în memoria cercetătorului George Boole, care a dezvoltat algebra booleană. Nu uita că într-un computer, la nivelul

cel mai de jos, totul este 1 și 0, **adevărat SAU fals, adevărat ȘI fals, adevărat NU fals**. Apropo de Boole și de operatori. Tot de la Boole avem și regula comutativității pentru adunare $x + y = y + x$ și regula distributivității $z(x + y) = zx + zy$.

De cele mai multe ori, operatorii stabilesc o evaluare a expresiilor de la stânga la dreapta. Putem verbaliza, de exemplu $1 + 1$ ca „unu plus unu”, ceea ce înseamnă că am citit enunțul de la stânga la dreapta. Am spus de cele mai multe ori, pentru că avem și cazurile când un operator, de exemplu **egal (=)**, care este interpretat la evaluarea codului de la dreapta la stânga. De exemplu, în enunțul $a = 1$, citim: „valoarea 1 este atribuită variabilei a”. Operatorul egal împarte cei doi operanzi în expresii aflate în partea stângă (**left-hand-side**) și expresii aflate în partea dreaptă (**right-hand-side**).

Expresiile (expressions)

O expresie este o combinație rezolvabilă de operatori și operanzi.

Finalizarea rezolvării unei expresii se numește **evaluare**. Asta înseamnă că la momentul evaluării combinației, aceasta se va finaliza cu obținerea unei valori.

Dicționarele explicative spun că o **expresie** este un **grup de numere, litere etc. legate între ele prin simboluri de operații matematice (adunare, înmulțire etc.)**. (DEX 98) sau **formulă care exprimă raporturi matematice** (NODEX 2002).

O mică paranteză utilă pentru curiozitatea ta. Sunt convins că te-ai întrebat cum s-a ajuns la forma actuală de scriere a codului. Am găsit un răspuns.

Odată, demult, prin anii 60, la apariția limbajului de programare ALGOL, a debutat un efort colaborativ de formalizare a sintaxei limbajelor de programare. Rezultatul a fost o formă de exprimare sintactică cunoscută acum sub titulatura de **Backus Naur**.

Conform **Backus Naur Form** (BNF) - notația care formalizează sintaxa unui limbaj de programare indiferent care ar fi el - o expresie **este definită ca** un „termen”, care poate fi la rândul său urmat de alt termen și așa mai departe. Așa arată formalizarea BNF: **expression ::= term { "|" term }**. Simbolul **::=** înseamnă „este definit ca”, iar **|** (caracterul **pipe**) înseamnă „ȘI-ul” logic.

Expresiile mai complexe cer folosirea unor semne grafice care să indice motorului unde se încheie acestea. Aceste semne grafice sunt **separatorii**.

Structurile lexicale proprii JavaScript și recunoașterea lor | Categorii de expresii

Separatorii

I-am amintit mai sus. Sunt folosiți pentru a **separa** fragmentele cu înțeles pentru compilator. De exemplu, cel mai simplu separator este **un spațiu** (adu-ți mereu aminte că un spațiu este și el un caracter) sau **un TAB**, care sunt folosite pentru a despărți cuvintele între ele. Un alt separator este **punct și virgulă**, care este ca punctul din limbajul natural.

Enter-ul pe care-l dai pentru a trece pe o nouă linie, de fapt este tot un separator.

Categorii de expresii

JavaScript are următoarele categorii de expresii:

- **aritmétique**, care se rezumă la un număr. Este și cazul folosirii operatorilor aritmetici;
- de **șiruri de caractere**, care se rezumă chiar la o înșiruire de caractere. Este și cazul folosirii operatorilor pe șiruri;
- **logice**, care se rezumă ori la **true** ori la **false**;
- **expresii de bază** cum ar fi cuvintele cheie sau expresiile de uz general și
- **expresiile din partea stângă a operatorului de atribuire (=)**, adică la ce trebuie să se lege evaluarea a ceea ce este în partea dreaptă.

Cea mai simplă expresie este o **valoare literală** scrisă direct, ori o variabilă, dacă e mai pe gustul tău.

După cum spuneam, combinarea operanzilor cu operatorii, creează expresii.

Este necesară o mică precizare. Valorile de lucru sunt de două feluri. Cele care sunt fixe, care așa cum le-ai scris, așa rămân. Mai sunt numite și **literale**. Și valorile care se pot modifica în funcție de dinamica programului și care se numesc **variabile**, care pornesc de la o valoare dată sau nu.

În exemplul dat, avem litera **x**, care ține locul unei valori ce va apărea în urma evaluării expresiei $1 + 1$. Tehnic, **x** este definit prin cuvântul special **var** ca fiind o variabilă. Litera **x** o numim identificator al variabilei. E ca o etichetă pe un borcânel (variabila) a cărui conținut se va schimba când expresia din partea dreaptă a egalului va fi evaluată la executarea codului.

Aceasta este adunată cu o valoare de sine stătătoare numită **valoare literală**.

```
var x = 1 + 1;
// expresie de atribuire a unei expresii aritmetice
```

Am lămurit deja mai sus că o valoare literală



Wikipedia
Forma
Backus-Naur



Pentru că expresiile sunt evaluate la o valoare, o expresie poate fi combinată cu altă expresie pentru a forma o expresie mai complexă și așa mai departe.

este pur și simplu valoarea introdusă direct prin reprezentarea sa literală, adică cifre pentru numerele și caractere între ghilimele pentru text. Deci, ca să indicăm computerului că folosești o variabilă care este inițializată cu valoarea trei vei scrie cifra: **var trei = 3;**. Ce se întâmplă când îl pui pe trei între ghilimele? Da, ai intuit perfect, se transformă în text: **var text = '3';**.

Mai sunt și altele, dar le vom lămurii pe parcurs. De ce este nevoie de o precizare de acest fel? Pentru că de nu ai pune între ghilimele textul, motorul nostru de JavaScript ar înțelege că faci o referință către un identificator al unei variabile, constante, funcții sau obiect. Reține acest aspect foarte important. Pe scurt, cifrele sunt evidente în sine, iar textul trebuie între ghilimele simple sau duble.

Adu-ți mereu aminte că **var ceva = "altceva";** este o variabilă care identifică valoarea text **"altceva"**, ceea ce este complet diferit de **var ceva = altceva;**, care transformă variabila cu identificatorul **ceva** într-o referință către identificatorul **altceva**. Acesta poate fi o altă variabilă, o funcție sau un obiect.

Revenim acum la matematică, la momentul când respectam regulile dictate de prioritatea operatorilor. Mai ții minte? Mai întâi ce-i în paranteze; dacă ai înmulțiri sau împărțiri, acestea primează, apoi adunările și scăderile...

O mică mențiune: în cazul programării acoladele și parantezele pătrate pe care le foloseam în matematică

```
1 + 1 * (5 - 1); // 5
// enunț format din mai multe expresii
```

pentru a separa expresiile imbricate, sunt numai paranteze rotunde. De exemplu, pentru expresia: **{1 + [2 - (2 * 3)]}** din notația convențională matematică, în programare este scris astfel: **1 + (2 - (2 * 3))**.

Ne focalizăm pe enunțul de mai sus. Începem de la stânga spre dreapta să facem evaluarea expresiei. Buuuun! Și avem: 1 plus 1 egal? Stop joc! Cel de-al doilea unu (cel din dreapta operatorului plus) este implicat într-o operațiune pe care va trebui să o rezolvăm mai întâi pentru că așa spune prioritatea operatorului ori (*). Pentru moment, lăsăm în suspans prima operațiune de adunare și sărim să facem înmulțirea. Surpriză majoră: 1 este înmulțit cu o altă expresie care este între parantezele rotunde. Deci, abandonăm și înmulțirea și facem operațiunea dintre parantezele rotunde pentru a ajunge la o valoare. Gata! Avem valoarea 4. Perfect, acum că avem valoarea, putem face înmulțirea: **1 * 4**. În urma evaluării ajungem la valoarea 4. Acum este permisă evaluarea primei operațiuni de la care am plecat: **1 + 4**. Ajungem la rezultatul 5. Hai că nu a fost greu, doar nițică matematică... știi, de mate nu scăpăm, dar nu ne lăsăm.

Expresii cu operanzi diferiți - transformarea (coercion)

JavaScript este un limbaj de programare care oferă o flexibilitate fantastică. În ceea ce privește operanzii, aceștia pot fi chiar de tipuri diferite iar JavaScript este forțat să ofere totuși un răspuns.

Hai să ne uităm la următoarea expresie: **true + 10;** pe care o dăm motorului JavaScript spre evaluare. Ce crezi că se va întâmpla pentru că în acest moment operezi cu o valoare boolean și un număr?

JavaScript va recurge la un mecanism de transformare (în limba engleză **coercion**) aplicat valorilor și în acest caz, va „transforma” valoarea boolean în echivalentul său numeric, adică 1. Da, da, **true** este 1, iar **false** este 0.

Bun, acum a fost rezolvată dilema. Se va face evaluarea care va avea drept rezultat valoarea 11. Nebunie, nu?! Vom trata mai atent aceste „transformări”.

Pentru că expresiile sunt evaluate la o valoare, acestea pot fi asignare unor variabile, constante sau proprietăți ale unui obiect: **var ceva = 2 + 2;**

Dacă ai amestecat, e perfect normal. Respiră de cinci ori foarte adânc cu ochii închiși concentrându-te adânc la fiecare respirație. Dacă nimic nu se leagă, mergi într-un parc. Eu te aștept aici.

Enunțuri (statements)

Este echivalentul unei propoziții în limbaj uman. Componentele unui enunț pot fi **valori**, **operatori**, **expresii**, **cuvinte cheie** și **comentarii**.

Cel mai simplu enunț este introducerea unei **valori literale** (literal înseamnă că menționezi prin caractere valoarea - cifre pentru numere, șiruri de caractere între ghilimele pentru text):

```
3;
```

Dicționarele explicative spun că un enunț este o **regulă după care se face un calcul sau se aplică o construcție matematică; executare a unui calcul**. (DEX 09).

Cel mai simplu enunț, care nu este folosit pentru că nu are utilitate, dar care este bun pentru a ilustra, este cel al unei simple expresii. Deocamdată înțelege expresia ca pe o combinație de operanzi și operatori care evaluată oferă o valoare.

```
a + 1; // enunțul unei expresii (expression statement)
```

Tot ce este în partea dreaptă a egalului, este o valoare.

Dacă textul din partea dreaptă nu este între ghilimele, acesta este o referință către o altă valoare.

Citirea expresiilor se face de la stânga la dreapta.

Expresiile sunt evaluate după reguli.

O expresie nu va fi tratată niciodată ca operand, ci rezultatul evaluării sale

Totul în JavaScript este evaluat în final la o valoare boolean, fie ceva care poate fi considerată a fi o valoare adevărată, fie ceva care poate fi considerat a fi o valoare falsă - truthy și falsey, cum ar zice în engleză.

Un program este o listă de enunțuri.

Enunțurile sunt încheiate cu punct și virgulă cu excepția notabilă aplicată prin mecanismul ASI.

Crede-mă, nu vreau să te zăpăcesc, dar o expresie poate fi în același timp un enunț așa cum avem cazul simplu de mai sus. O expresie este un enunț care este evaluat întotdeauna la o valoare. Enunțul este ca fraza din analiza gramaticală cu diferența în cazul nostru că nu se termină cu punct, ci cu punct și virgulă.

Exemplul oferit nu este cel mai util. În schimb, există un enunț al celei mai utile expresii întâlnite în întreg limbajul: **enunțul expresiei de apelare** al unei funcții (**call expression**): **făCeva()**; . Chiar dacă nu am învățat nimic despre funcții, ține minte că apelarea acestora este nimic mai mult decât un **enunț al unei expresii**, dar care declanșează executarea acelei funcții. Nu te las așa în suspans... consideră un obiect în JavaScript ca pe un pistol iar gloanțele lui ca pe niște proprietăți. Folosind această imagine, o funcție este un cartuș care a fost puscat iar apelul funcției este trăgaciul. Apăsând trăgaciul (apelezi funcția), puști capsă cartușului și iese glonțul pe țeavă (funcția returnează un rezultat celui care a apelat-o). Cam militaristă comparația, dar e târziu în noapte și asta mi-a venit la îndemână pe ecranul minții.

Ei, abia acum te-am zăpăcit. Nu?!

Hai să facem o mică „analiză gramaticală” pe „text” dacă tot știm în acest moment ce sunt expresiile și ce sunt enunțurile.

```
var x; // enunț declarativ - declaration statement
x = 2 * 3;
```

Pe prima linie avem un enunț declarativ pentru variabila x. Pe a doua linie avem două expresii. Una a înmulțirii și alta a asignării valorii rezultate.

Enunțurile pot sta singure sau pot fi adunate într-un bloc distinct. Acest bloc distinct se numește în limba engleză **block statements** și se înțelege în limba română a fi un set de enunțuri grupate într-un bloc delimitat de acolade.

```
{
  var mesaj = 'Salut!';
  console.log(mesaj);
}
```

Vei vedea mai târziu cât de utile sunt în cazul scrierii enunțurilor care controlează execuția codului, cum ar fi deciziile prin **if...else** sau buclele, cum ar fi **while(expresie){bloc de enunțuri}**. Este nevoie acum să punem ordine în ideile pe care le-am explorat cu privire la enunțuri.

Care sunt enunțurile în JavaScript?

- enunțul blocurilor de cod **{...}**,
Structurile lexicale proprii JavaScript și recunoașterea lor | Categoriile de expresii

- enunțul declarațiilor de variabile și constante **let x = 10; const y = 9.8;**,
- enunțuri goale. Pur și simplu nu ai nimic, dar un terminator: **;**,
- enunțul unei expresii,
- enunțul **if**,
- enunțul **break**,
- enunțul **continue**,
- enunțul **return**,
- enunțul **with**,
- enunțul **switch**,
- enunțul **throw**,
- enunțul **try** și
- enunțul **debugger**.

Mai sunt un set care se numesc „enunțuri ale iterabilelor”:

- **do-while**,
- **while**,
- **for**,
- **for-in**,
- **for-of**.

Ce nu poate constitui un enunț al unei expresii? Orice începe cu:

- **{**,
- **function**,
- **async**,
- **function**,
- **class**,
- **let** [**.**

Enunțurile pot fi indentate (pui spații albe înaintea fragmentului de cod și ca efect vizual se vor deplasa spre dreapta). Despre indentare spune **Marele Dicționar pentru Neologisme** din 2000: „plasare a programelor pe linii, pentru scrierea cât mai clară a acestora”.

Declarațiile

Folosind cuvintele cheie ale limbajului faci declarații, care, de fapt, ceea ce menționează este tipul datelor cu care vei lucra în programul tău. Un exemplu foarte simplu este declararea unei variabile: **var x = 10;**. Un program JavaScript este o colecție de declarații de variabile și funcții.



Blocurile de cod

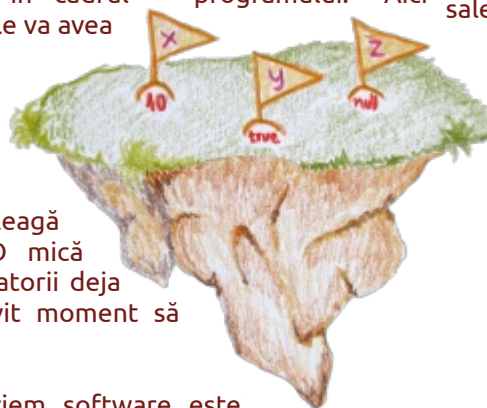
Blocurile grupează enunțurile. În JavaScript poți declara un bloc de cod foarte simplu deschizând acolade. Scrii codul între acolade și poți considera că acest cod aparține unui bloc de cod distinct. Blocurile au rolul de a grupa una sau mai multe enunțuri.

Gruparea declarațiilor într-un bloc, se comportă ca o **unitate de cod** menită să evalueze expresii, să apeleze funcții și metode, să afișeze rezultate și să returneze rezultate.

```
{ var ceva = 'Salut!' }
```

Cel mai adesea vei vedea blocurile de cod ca parte a unor sintaxe mai elaborate cum ar fi deciziile ori ca parte a funcției, chiar indicând corpul funcției.

Partea cea mai valoroasă în gruparea cu ajutorul blocurilor este că se realizează și o separare a fragmentelor de cod în cadrul programului. Aici mă refer la faptul că declararea unei variabile va avea ca „domeniu de existență” acel bloc de cod.



Despre identificatori

Declararea variabilelor și a funcțiilor se leagă organic de conceptul identificatorilor. O mică introducere în ceea ce reprezintă identificatorii deja am făcut, dar acesta este cel mai potrivit moment să explorăm mai mult ceea ce înseamnă.

Adu-ți aminte că scopul pentru care scriem software este pentru a manipula valori. Operațiunile au efecte în manipularea anumitor resurse.

Reperete de mai sus sunt identificatorii, care odată înțelegi, vor permite accesul la ceea ce înseamnă variabilele ca și concept.

Să ne imaginăm că avem o hartă imaginară pe care avem marcate prin fanioane diferite locații. Locațiile reprezintă valorile pentru care avem nevoie de un nume, de un toponim. De exemplu, pentru orașul (percepem orașul ca fiind valoarea) din centrul regiunii Moldova avem numele Bacău, care este identificatorul. Adică, identificăm orașul ca valoare administrativă cu un toponim.

Putem să ne închipuim că identificatorii sunt toponime ale „tărâmului” JavaScript. Identificatorii pot fi orice secvență de caractere care poate să

înceapă cu semnul dollar \$, sau cu liniuță jos _ (**underscore**) urmate de orice puncte de cod codate numeric respectând schema de codare a caracterelor UTF16.

Am putea spune foarte simplu că un identificator este numele unei valori, iar identificatorul este o înșiruire de caractere. După cum observi, identificatorii poți fi cuvinte românești cu diacritice. De ce? Pentru că un computer se uită la reprezentarea numerică a caracterelor iar numerele din spatele fiecărui caracter în parte ce alcătuiește numele identificatorului face parte din setul de numere acceptat de UTF16. Vom insista mai mult pe această corespondență numerică pentru că, mai târziu, vom vedea că stă la baza multor operațiuni pe șiruri. Înțelegerea acestui aspect este o cheie foarte importantă.

Aceasta a fost mica lecție de anatomie aplicată codului sursă JavaScript pentru a înțelege foarte bine și cele mai mici părți ale sale.

```
var mâncare = 'vinete coapte';
```

Identificatorii sunt denumirile după care putem accesa valorile.