Laborator 5

Arbori dezechilibrați și interpretor de expresii postfix

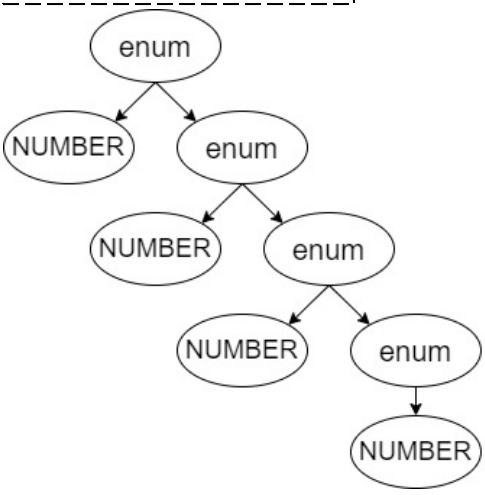
În laboratorul al treilea am aflat cum se stochează simbolurile pe stivă și cum se procesează și se propagă valorile semantice (cu ajutorul pseudo-variabilelor dolari) bottom-up în arborele sintactic (parse tree). În cazul expresiilor aritmetice, pe stiva de valori erau stocate doar valori de tip întreg, care în urma unei reduceri se substituiau tot cu o valoare întreagă. La o reducere, partea dreaptă a unei producții se înlocuia cu (se reducea la) partea stângă.

În laboratorul al patrulea am lucrat și cu liste, fapt pentru care a fost nevoie să definim un %union cu un câmp pentru numere întregi și un câmp pentru liste. Pe stiva de valori ajung să coexiste și numere și liste, de aceea trebuie să facem stiva omogenă, cu elemente de același tip, și anume union. Acest union, în cazul de față, este un fel de wrapper, care conferă o formă comună numerelor și listelor. Deși la citirea unui token din input se pune pe stiva de valori doar câte un întreg, în urma unei reduceri (când se recunoaște în input o structură sintactică agregată), ajunge să se pună pe stiva de valori o valoare de tip agregat. În cazul de față, structura de date agregată este lista.

Așadar, ne putem imagina cum arată arborele sintactic la citirea și crearea unei liste. La fiecare pas, pe stiva de valori se pune câte un element de tip întreg dintr-o listă. Când în stiva de valori există un întreg SAU un întreg și o listă, acestea vor fi reduse la o singură listă. În regula **enum** din setul de producții (din laboratorul 4) este descrisă compunerea unei liste intern, ca structură de date. Regula poate fi recursivă dreapta sau recursivă stânga. În primul caz, se introduce un element la începutul listei, iar arborele este dezechilibrat pe dreapta. În al doilea caz, se introduce un element la finalul listei, iar arborele este dezechilibrat pe stânga. Introducerea unui element la finalul unei liste este o metodă ineficientă, deoarece trebuie parcursă toată lista.

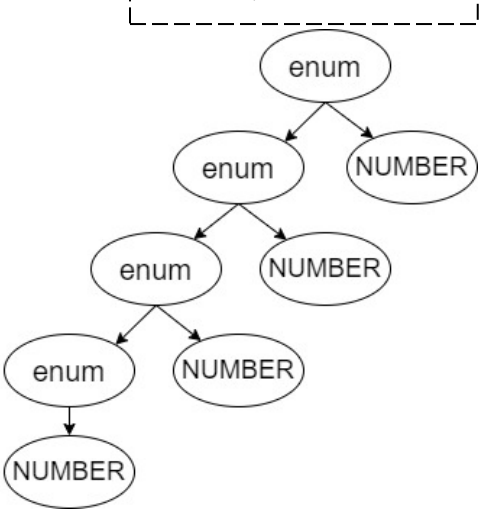
Recursivitate dreapta:

```
enum -> NUMBER enum
      | NUMBER
```



Recursivitate stânga:

```
enum -> enum NUMBER
      | NUMBER
```



În primul caz, pe stiva de valori se stochează la fiecare pas toate elementele citite până acum din listă. După stocarea ultimului element, începe extragerea de pe stivă și adăugarea efectivă în listă a elementelor, începând cu ultimul (cel mai din dreapta) și terminând cu primul (cel mai din stânga). Adică se adaugă elementele în lista finală în ordine, de la dreapta la stânga, printr-o simplă adăugare pas cu pas în capul unei liste parțiale. În al doilea caz, se păstrează în stiva de valori doar elementul care se introduce în listă la pasul curent.

Să luăm ca exemplu compunerea listei **'(1 2 3 4)**. În tabelul următor putem vedea cum evoluează stiva de valori la compunerea unei liste recursiv dreapta, respectiv recursiv stânga.

Recursivitate dreapta	Recursivitate stânga
Stiva de valori: 1 1 2 1 2 3 1 2 3 4 1 2 3 1 2 1	Stiva de valori: 1 2 3 4
Compunerea listei: 4 3 4 2 3 4 1 2 3 4	Compunerea listei: 1 1 2 1 2 3 1 2 3 4

Următoarea provocare este să creăm un analizor lexico-sintactic pentru a transforma expresii aritmetice din formă infix în formă postfix.

Exemple:
1 + 2 => 1 2 +
(a – 5 + c) * 2 => a 5 – c + 2 *
a / 2 + – (a – 5 + c) * 2 ^ b => a 2 / a 5 – c + – 2 b ^ * +

Observăm că inputul poate să conțină variabile, pe lângă numere și operatori. Având în vedere că nu putem calcula expresiile, ci trebuie doar să le modificăm forma, le putem stoca sub formă de string-uri. În acest caz, canalul de comunicare semantică între analizorul lexical și parserul sintactic, **yylval**, trebuie să accepte string-uri, deci poate fi reprezentat ca un %union cu un câmp de string-uri.

În acest exercițiu sunt foarte importante prioritățile operațiilor, pentru a asigura o transpunere corectă a expresiilor. Ne amintim din laboratorul al treilea că prioritățile operațiilor sunt specificate în partea de declarații a fișierului .y, cu %left, %right sau %nonassoc. Ordinea în care sunt declarate indică importanța lor: prima are cea mai mică prioritate, iar ultima are cea mai mare prioritate. Dacă asociativitatea este pe stânga (%left), atunci o serie de operații cu

aceleași priorități va fi tratată de la stânga spre dreapta. Dacă asociativitatea este pe dreapta (%right), atunci seria va fi tratată din dreapta spre stânga. Există situații în care este nevoie să definim priorități particulare. De exemplu, dacă tratăm numere negative, minusul lor (minus unar) nu are aceeași prioritate ca minusul de la scădere (minus binar) și nici nu este tratat la fel. Așa că vom defini o prioritate specială pentru minusul unar al numerelor negative.

```
%left '+' '-'
%left '*' '/'
%left MINUSUNAR
%right '^'
```

În setul de producții trebuie să specificăm că minusul numerelor negative are prioritatea specială, definită de noi. Prin "%prec MINUSUNAR" se suprascrie precedența care ar fi fost dată implicit de '-'. Știm deja că precedența unei reguli este importantă în conflictele shift-reduce, când se compară precedența regulii care s-ar reduce cu precedența simbolului care s-ar shifta.

```
expr : expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | expr '^' expr
     | '-' expr %prec MINUSUNAR
     | '(' expr ')'
     | NUMBER
     | VAR
     ;
```

Exerciții propuse:

1. Creați un analizor lexico-sintactic care să preia din input expresii aritmetice scrise în formă postfix și să le transforme în formă prefix. Exemplu: 1 2 + => + 1 2
2. Creați un analizor lexico-sintactic care să citească din input expresii aritmetice în formă postfix, fără variabile, doar cu numere, și să afișeze rezultatul calculului. Exemplu: 1 2 + => 3
   10 5 – 3 + 2 * => 16