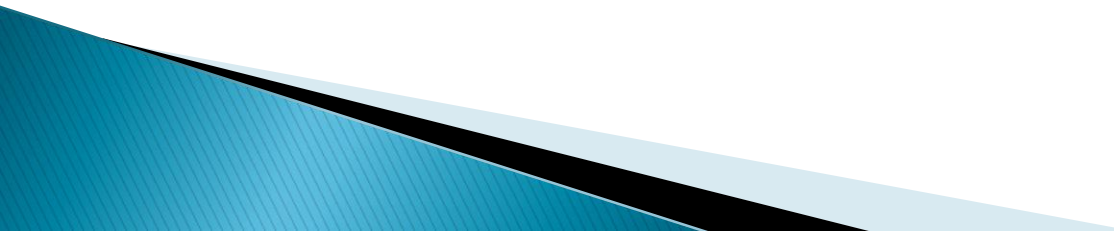


# Limbaje formale și tehnici de compilare

Curs 7: analizorul sintactic descendent recursiv;  
eliminarea recursivității stângi; tratarea erorilor

# Implementarea analizorului sintactic

- ▶ Există două metode generale de implementare a unui analizor sintactic (parser):
  - ▶ **Analiza sintactică descendentă (top-down parsing)** – se pornește de la regula de start a gramaticii și se derivă regulile până când se ajunge la terminale. Arborele sintactic este construit de la rădăcină către frunze.
  - ▶ **Analiza sintactică ascendentă (bottom-up parsing)** – se pornește de la terminale și acestea se reduc la reguli sintactice, până când se ajunge la regula de start. Arborele sintactic este construit de la frunze către rădăcină.
- 

# Integrarea ANLEX în ANSIN

- ▶ Analiza sintactică se poate desfășura după ce a avut loc analiza lexicală și s-a produs lista de atomi lexicali. Această metodă permite decuplarea ANLEX de ANSIN, este mai rapidă, deoarece atomii sunt parsați o singură dată, dar necesită memorie pentru păstrarea tuturor atomilor.
- ▶ Analiza sintactică se poate desfășura concomitent cu cea lexicală: de fiecare dată când ANSIN are nevoie de un atom, va apela ANLEX pentru a i-l furniza. Această metodă nu necesită memorie pentru păstrarea listei de atomi, dar necesită o metodă de revenire în șirul de intrare, ceea ce uneori (ex: date citite din rețea) rezultă în necesitatea implementării unui buffer. Dacă ANSIN revine dintr-o alternativă greșită, un atom lexical se va parsa de mai multe ori.

# Analizorul sintactic descendent recursiv

- ▶ ASDR este o metodă de implementare de tip descendentă (top-down) a ANSIN
- ▶ ASDR se pretează foarte de bine la implementarea manuală, de către programator, a gramaticilor
- ▶ Codul rezultat are o structură foarte asemănătoare cu cea a gramaticii
- ▶ Dacă este nevoie de o viteză mai mare de analiză, în special în cazul gramaticilor cu multe alternative, există metode de optimizare a ASDR
- ▶ Modalitatea de implementare a ASDR prezentată în curs este adaptată gramaticilor **PEG** (Parsing Expression Grammar). Diferența între acestea și formalismul GIC (Gramatică Independentă de Context), este faptul că în PEG alternativa (  $e_1 \mid e_2$  ) este ordonată, la fel ca în expresiile regulate: dacă  $e_1$  este recunoscută (match), iar regula se îndeplinește, atunci nu se mai analizează și  $e_2$ .

# Funcția *consume*

- ▶ Funcția **consume** furnizează ANSIN următorul atom lexical de prelucrat
- ▶ **consume** primește ca parametru un cod de atom
- ▶ Dacă atomul curent are codul cerut, el se consumă (se trece la următorul atom), iar **consume** va returna *true*
- ▶ Dacă atomul curent are alt cod decât cel cerut, **consume** va returna *false*, rămânând la poziția curentă

```
#include <stdbool.h> // în C, pentru bool, true, false
```

```
Atom *pCrtAtom; // iterator atom curent
```

```
bool consume(int cod){  
    if(pCrtAtom->cod==cod){  
        ++pCrtAtom; // considerăm că atomii sunt păstrați într-un vector  
        return true;  
    }  
    return false;  
}
```

# Algoritm implementare ASDR

- ▶ Se elimină recursivitatea stângă din gramatică
- ▶ Fiecare regulă se va implementa printr-o funcție separată
- ▶ O funcție nu are niciun parametru și returnează *true/false*, dacă ea a putut fi aplicată (match) sau nu cu succes pe atomii începând cu poziția curentă
- ▶ Dacă o funcție s-a aplicat cu succes, ea consumă toți atomii corespunzători corpului ei; dacă funcția nu s-a putut aplica, ea nu trebuie să consume niciun atom (*total sau nimic*). Acest mecanism se poate implementa salvând la intrarea în funcție poziția curentă din șirul de atomi. De fiecare dată când funcția trebuie să returneze *false*, poziția salvată inițial va fi refăcută.

```
bool regula(){  
    Atom *pStartAtom=pCrtAtom;           // salvare poziție inițială  
    ... return true; ...  
    pCrtAtom=pStartAtom;                  // refacere poziție inițială  
    return false;  
}
```

# Implementarea corpului unei reguli

- ▶ **Atomii lexicali (terminalele)** se consumă folosind funcția **consume**
- ▶ **Neterminalele** se consumă apelând funcțiile care le implementează
- ▶  $e_1 e_2$  – se consumă cu *if*-uri imbricate, deci se ajunge la următoarea componentă doar dacă prima a fost consumată
- ▶  $e_1 \mid e_2$  – se consumă cu succesiuni de *if*-uri, fiecare *if* testând o variantă din alternativă. Dacă o variantă a fost îndeplinită, nu se mai trece la următoarea variantă. Pentru aceasta se poate folosi un *return* în fiecare *if* sau succesiuni de *if/else if/.../else*. Dacă este posibil ca într-o alternativă să se fi consumat atomi și alternativa nu se aplică, trebuie refăcută poziția inițială.
- ▶  $e^*$  – se implementează printr-o buclă infinită, din care se iese atunci când nu se mai poate consuma  $e$
- ▶  $e^+$  – se aplică identitatea  $e^+ = e e^*$
- ▶  $e?$  – se consumă fără a se ține cont de rezultatul consumării, astfel încât  $e$  poate să lipsească
- ▶  $e \mid \epsilon$  – este echivalentă cu  $e?$

# Implementare $e_1$ $e_2$

- ▶ *if*-uri imbricate, astfel încât se ajunge la următoarea componentă doar dacă prima a fost consumată

instrWhile = WHILE LPAR expr RPAR instr

```
bool instrWhile(){
    Atom *pStartAtom=pCrtAtom;
    if(consume(WHILE)){
        if(consume(LPAR)){
            if(expr()){
                if(consume(RPAR)){
                    if(instr()){
                        return true;
                    }
                }
            }
        }
    }
    pCrtAtom=pStartAtom;
    return false;
}
```



# Implementare $e_1$ | $e_2$

- ▶ succesiuni de *if*-uri, fiecare *if* testând o variantă din alternativă. Fiecare alternativă trebuie să refacă poziția inițială, dacă e posibil să se fi consumat în ea atomi.

factor = ID | LPAR expr RPAR | NR

```
bool factor(){
    Atom *pStartAtom=pCrtAtom;
    if(consume(ID)){
        return true;
    }
    if(consume(LPAR)){
        if(expr()){
            if(consume(RPAR)){
                return true;
            }
        }
        pCrtAtom=pStartAtom;
    }
    if(consume(NR)){
        return true;
    }
    return false;
}
```

# Implementare e\*

- ▶ se implementează printr-o buclă infinită, din care se iese atunci când nu se mai poate consuma *e*

varDef = type ID ( COMMA ID )\* SEMICOLON

```
bool varDef(){
    Atom *pStartAtom=pCrtAtom;
    if(type()){
        if(consume(ID)){
            for(;;){
                if(consume(COMMA)){
                    if(consume(ID)){
                        else{ /*eroare*/ }
                    }
                }
                else break;
            }
            if(consume(SEMICOLON)){
                return true;
            }
        }
    }
    pCrtAtom=pStartAtom;
    return false;
}
```

```
// variantă pentru ( COMMA ID )*
while(consume(COMMA)){
    if(consume(ID)){
        else{ /*eroare*/ }
    }
}
```

# Implementare e?

- ▶ se consumă fără a se ține cont de rezultatul consumării, astfel încât *e* poate să lipsească
- ▶ În fazele ulterioare ale compilatorului va fi necesar să se știe dacă s-a consumat *e*. De aceea, la consumarea lui *e* se folosește un *if*, care deocamdată nu are instrucțiuni atașate.

exprUnary = SUB? factor

```
bool exprUnary(){
    Atom *pStartAtom=pCrtAtom;
    if(consume(SUB)){
    }
    if(factor()){
        return true;
    }
    pCrtAtom=pStartAtom;
    return false;
}
```

# Eliminarea recursivității stângi (1)

- ▶ la implementarea analizorului sintactic folosind ASDR, recursivitatea stângă duce în cod la recursivitate infinită

**expr** = **expr** ( ADD | SUB ) termen | termen

```
bool expr() {
    Atom *pStartAtom = pCrtAtom;
    if(expr()) {
        if(consume(ADD) || consume(SUB)) {
            if(termen()) {
                return true;
            }
        }
        pCrtAtom = pStartAtom;
    }
    if(termen()) {
        return true;
    }
    pCrtAtom = pStartAtom;
    return false;
}
```

# Formulă eliminare recursivitate stângă

- ▶ Pentru eliminarea recursivității stângi, se poate folosi următoarea formulă de transformare a unei reguli gramaticale, în care:
  - ▶  $A$  – numele regulii inițiale
  - ▶  $A'$  – numele unei reguli nou introdusă
  - ▶  $\alpha_1, \alpha_2, \dots, \alpha_m$  – fragmente finale de expresii pe ramurile cu recursivitate stângă
  - ▶  $\beta_1, \beta_2, \dots, \beta_n$  – expresiile de pe ramurile fără recursivitate stângă


$$A = A \alpha_1 \mid A \alpha_2 \mid \dots \mid A \alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$



$$\begin{aligned} A &= \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A' \\ A' &= \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon \end{aligned}$$

# Exemplu eliminare recursivitate stângă

```
postfix = postfix LBRACKET expr RBRACKET  
        | postfix INC  
        | postfix DEC  
        | postfix DOT ID  
        | factor
```



```
postfix = factor postfixPrim  
postfixPrim = LBRACKET expr RBRACKET postfixPrim  
            | INC postfixPrim  
            | DEC postfixPrim  
            | DOT ID postfixPrim  
            | ε
```

# Eliminarea recursivității stângi (2)

**expr** = **expr** ( ADD | SUB ) termen | termen

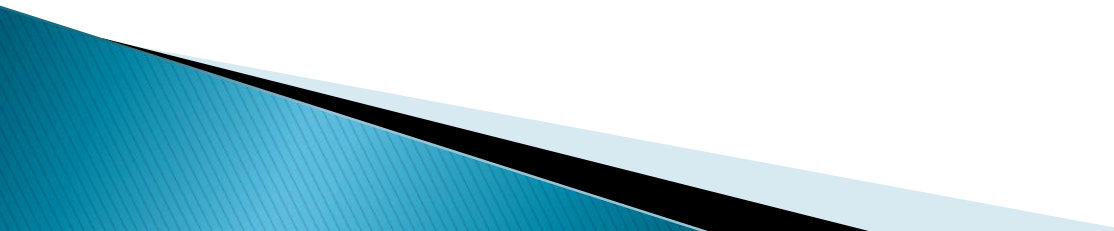


expr = termen exprPrim  
exprPrim = ( ADD | SUB ) termen exprPrim |  $\epsilon$

```
bool exprPrim(){
    Atom *pStartAtom=pCrtAtom;
    if(consume(ADD)||consume(SUB)){
        if(termen()){
            if(exprPrim()){
                return true;
            }
        }
    }
    pCrtAtom=pStartAtom;
    return true; //  $\epsilon$ 
}
```

```
bool expr(){
    Atom *pStartAtom=pCrtAtom;
    if(termen()){
        if(exprPrim()){
            return true;
        }
    }
    pCrtAtom=pStartAtom;
    return false;
}
```

# Tratarea erorilor sintactice

- ▶ Una dintre funcțiile ANSIN este generarea de mesaje de eroare relevante, dacă a avut loc o eroare la parsarea șirului de atomi
  - ▶ Mesajele de eroare trebuie să fie:
    - ▶ **localizate** – trebuie să conțină suficiente informații pentru a localiza de unde anume s-a generat eroarea
    - ▶ **destinate celor care folosesc compilatorul** – textul mesajului de eroare trebuie conceput ca să aibă înțeles din perspectiva celor care folosesc compilatorul, nu din perspectiva celor care îl implementează
    - ▶ **specifice** – secvențe diferite din gramatică trebuie să genereze erori diferite, care să permită înțelegerea regulilor care nu au fost respectate
- 



# Localizarea erorilor

- ▶ Pentru a localiza o eroare, mesajul generat poate conține:
  - ▶ numele fișierului de intrare
  - ▶ linia și coloana din fișier
  - ▶ marginile din fișierul de intrare ale regulii în interiorul căreia s-a generat eroarea
- ▶ Pentru generarea erorilor se poate folosi o funcție (ex: `err(mesaj_de_eroare)`), care să adauge automat la mesajul generat informațiile de localizare, pe baza atomului curent)

```
exprAdd = exprAdd ( ADD | SUB ) exprMul | exprMul
```

```
// test1.c  
...  
for(i=1;i<n;i++)  
    if(v[i-]<v[i]){  
...  
    ↑
```

test1.c [249,13]: expresie invalidă după -

# Mesaje pentru cei care folosesc compilatorul

- ▶ Perspectiva celor care implementează un compilator este diferită de perspectiva celor care îl vor folosi
- ▶ Mesajele de eroare întotdeauna vor trebui să implementeze perspectiva celor care vor folosi compilatorul

```
declVar = type ID SEMICOLON  
exprAdd = exprAdd ( ADD | SUB ) exprMul | exprMul
```

// perspectiva celor care implementează compilatorul

- lipsește ID după tipul variabilei
- în exprAdd lipsește exprMul după + sau -

// perspectiva celor care folosesc compilatorul

- lipsește numele variabilei
- lipsește expresia de după + sau -

# Mesaje specifice

- ▶ Erorile care apar în locuri diferite trebuie să aibă texte diferite
- ▶ Textele trebuie să reflecte cât mai bine semnificația construcțiilor gramaticale care le-au generat

```
declVar = type ID SEMICOLON
```

```
declFn = type ID LPAR arg ( COMMA arg )* RPAR LACC instr* RACC
```

```
arg = type ID
```

```
instr = exprWhile | LACC instr* RACC
```

// mesaje vagi sau prea generale

- lipsește numele // numele cui? variabilei, funcției, argumentului?
- lipsește { // care acoladă? de la funcție sau de la instrucțiune?

// mesaje specifice

- lipsește numele variabilei
- lipsește { după antetul funcției

# Determinarea erorilor posibile

- ▶ Prin analiza gramaticii se determină ce erori pot fi generate pe baza ei
- ▶ Regulă generală: **dacă într-un anumit punct din gramatică există posibilitatea ca analiza să se blocheze, în acel punct este posibilă o eroare.**
- ▶ Algoritm pentru tratarea erorilor posibile care pot apărea într-o gramatică:
  - ▶ regulile în sine nu vor genera erori, ci doar vor returna *false* dacă nu sunt îndeplinite.
  - ▶ în cadrul fiecărei reguli se determină care sunt punctele în care analiza se poate bloca
  - ▶ pentru fiecare dintre aceste puncte se va prevedea un mesaj de eroare specific

# Analiza $e_1$ $e_2$

- ▶ Dacă un element din secvență nu se poate consuma, rezultă eroare
- ▶ În general erorile vor fi de forma *"lipsește ... "* sau *"invalid"*, deoarece acea construcție nu a fost găsită la poziția curentă fie din cauză că nu există, fie din cauză că este greșită și deci nu poate fi consumată
- ▶ Mesajul trebuie să reflecte situația cea mai probabilă: *lipsește/invalid*

```
bool instrWhile(){
    Atom *pStartAtom=pCrtAtom;
    if(consume(WHILE)){
        if(consume(LPAR)){
            if(expr()){
                if(consume(RPAR)){
                    if(instr()){
                        return true;
                    } else err("instrucțiune invalidă pentru corpul while");
                } else err("condiție while invalidă sau lipsește ");
            } else err("condiție invalidă pentru while");
        } else err("lipsește ( după while)");
    }
    pCrtAtom=pStartAtom;
    return false;
}
```

instrWhile = WHILE LPAR expr RPAR instr

# Analiza primului element din $e_1$ $e_2$

- ▶ Primul element dintr-o secvență va putea genera eroare doar dacă secvența este obligatorie

```
program = linie+ FINISH  
linie = ID COMMA NR
```

```
bool program(){  
    if(linie()){ // linie+ -> linie linie*  
        while(linie()){  
            if(consume(FINISH)){  
                return true;  
            }else err("eroare de sintaxa");  
        }else err("fisier vid");  
    }return false;  
}
```

# Analiza $e_1$ | $e_2$

- ▶ În general mesajele nu vor putea fi foarte specifice, deoarece nu se știe care alternativă s-a apropiat cel mai mult de forma corectă
- ▶ Se poate mări specificitatea mesajelor dacă în cursul analizei se salvează informații cu privire la calea care a fost urmată până în punctul curent

// mesaje posibile dacă nu se consumă *instr*: "*instrucțiune invalidă*" sau "*eroare de sintaxă*"

program = instr+ FINISH

instr = instrWhile | instrIf | expr SEMICOLON

// mesaj posibil: "*expresie invalidă după +/-*"

exprAdd = exprAdd ( ADD | SUB ) exprMul | exprMul

// se poate crește specificitatea mesajului dacă se ține minte operatorul consumat și în mesaj se va afișa acel operator: "*expresie invalidă după –*"

# Analiza e? și e\*

- ▶ nu se vor genera mesaje de eroare, deoarece se poate considera că atunci când nu poate fi consumată nu este vorba de o eroare ci de lipsa elementului
- ▶ **Atenție:** chiar dacă "e?" sau "e\*" nu generează mesaje de eroare, este posibil ca "e" să genereze mesaje, deci va trebui analizată separat

// *MUL?* nu va genera niciodată mesaje de eroare

declVar = type MUL? ID SEMICOLON

// ( *declVar* | *declFunc* )\* nu va genera niciodată mesaje de eroare

program = ( declVar | declFunc )\* FINISH

// *"ELSE instr"* este o secvență opțională, dar ea trebuie analizată separat: dacă s-a consumat *ELSE* și nu urmează *instr*, se generează mesaj de eroare

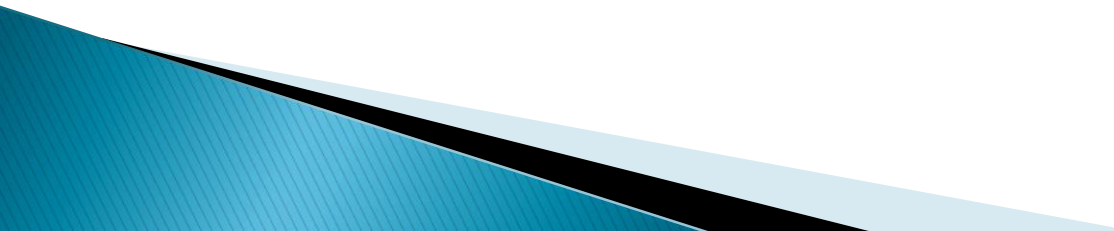
instrIf = IF LPAR expr RPAR instr ( ELSE instr )?

// *"COMMA ID"* este o secvență cu repetiție opțională, dar se analizează separat: dacă s-a consumat *COMMA* și nu urmează *ID*, se generează mesaj de eroare

declVar = type ID ( COMMA ID )\* SEMICOLON



# Revenirea din eroare

- ▶ În cazul cel mai simplu de implementare, un compilator se oprește după ce raportează prima eroare
  - ▶ Dezavantajul acestei metode este că dacă există mai multe erori, programatorul va trebui să compileze de mai multe ori un program, pentru a le depista pe toate, ceea ce poate lua timp
  - ▶ Din acest motiv, un compilator poate încerca să compileze în continuare un program, pentru a raporta cât mai multe erori la o singură compilare
  - ▶ Pentru a compila în continuare un program, după ce a apărut o eroare, compilatorul trebuie să revină din eroarea respectivă
- 

# Necesitatea revenirii din eroare

- ▶ Considerăm cazul unui compilator care la apariția unei erori doar afișează mesajul corespunzător și apoi continuă compilarea
- ▶ În această situație, din același punct al programului se vor genera mai multe erori, pe măsură ce se revine în regulile apelante
- ▶ Rezultă că trebuie să existe o strategie prin care să se evite mesajele de eroare ce sunt de fapt doar consecințe ale primei erori

```
program = expr+ FINISH  
expr = expr ( ADD | SUB ) termen | termen  
termen = termen ( MUL | DIV ) factor | factor  
factor = ID | NR | LPAR expr RPAR
```

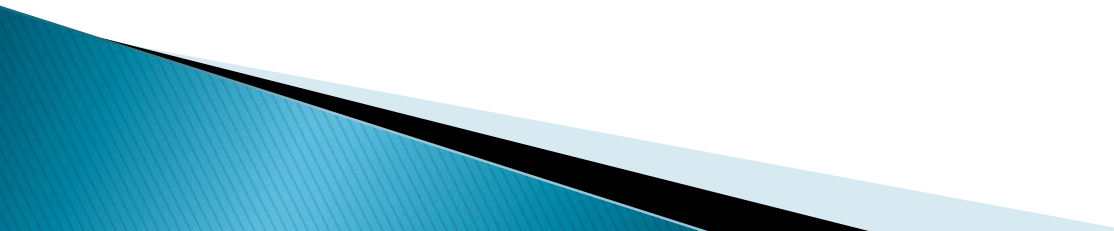
// mesaje de eroare generate pentru: **a \* ( b + 1**

- lipsește paranteza dreaptă după expresie
- expresie invalidă după \*
- expresie invalidă

# Revenire prin completarea regulii

- ▶ Există mai multe strategii de revenire din eroare, dar ele se pot împărți în două categorii:
  - ▶ renunțarea la parsarea regulii curente și consumarea tuturor atomilor care țin de regulile dependente de ea, până când se ajunge la un punct de sincronizare
  - ▶ continuarea parsării regulii curente prin introducerea de elemente care să ducă la îndeplinirea ei cu succes
- ▶ Un compilator poate să combine mai multe strategii, în funcție de cele mai probabile cauze de apariție a erorilor:
  - ▶ Erorile care apar de obicei prin greșeli sintactice să fie tratate prin renunțarea la regula curentă
  - ▶ Erorile care de obicei provin din omisiuni să fie tratate prin completarea regulii curente

# Revenire folosind puncte de sincronizare

- ▶ La apariția unei erori se revine din toate regulile care depind de regula curentă, până la o regulă care se consideră că include toată construcția eronată, regula însăși încheindu-se în mod corect. Punctul de sincronizare este chiar atomul care încheie regula la care s-a oprit revenirea.
  - ▶ Se consumă toți atomii până la punctul de sincronizare inclusiv, iar apoi se reia analiza sintactică
  - ▶ Exemple de puncte de sincronizare: punct și virgulă, acoladă închisă, cuvinte cheie, ...
- 

# Exemplu revenire cu puncte de sincronizare

```
instr = instrWhile | expr SEMICOLON
```

```
// expr -> ... -> exprAdd -> exprMul -> ... -> factor
```

// functia *err* va folosi *throw* pentru a genera o excepție care va face revenirea din toate apelurile intermediare, până la *try {...} catch()*

// se consideră *instr* ca fiind o regulă care conține în întregime erorile

```
bool instr(){
    try{
        if(instrWhile()){return true;}
        if(expr()){
            if(consume(SEMICOLON)){
                return true;
            }else err("lipsește ; după expresie");
        }
    }catch(exception &e){
        skipToSyncPoint(); // consumă toți atomii până la și inclusiv primul atom
        // care este considerat ca atom de sincronizare: punct și virgulă, acoladă închisă
        return true;        // consideră că instr s-a încheiat cu succes
    }
    return false;
}
```

# Revenire prin completarea regulii

- ▶ Conform regulii curente, se consideră ce elemente (atomi sau reguli) trebuie să existe pentru ca regula să se completeze cu succes
- ▶ Se consideră că aceste elemente există în șirul de intrare, astfel încât se continuă parsarea regulii
- ▶ Exemple de elemente care se presupune a fi existente:
  - ▶ după o expresie se consideră că există punct și virgulă
  - ▶ după un *if* există paranteză deschisă
  - ▶ după un *break* există punct și virgulă

# Exemplu revenire cu completarea regulii

factor = ID | NR | LPAR expr RPAR

// functia err are doar rol de afișare a erorii, fără a ieși din program

```
bool factor(){
    if(consume(ID)){return true;}
    if(consume(NR)){return true;}
    if(consume(LPAR)){
        if(expr()){
            if(!consume(RPAR))err("expresie invalidă după ( sau lipsește ");
        }else{
            err("expresie invalidă după (");
            skipTo(RPAR);          // consumă toți atomii pana la și inclusiv RPAR;
            testează și atomii de sincronizare, pentru cazul în care RPAR lipsește
        }
        return true;              // se returnează true chiar dacă a avut loc o eroare
    }
    return false;
}
```