Limbaje formale și tehnici de compilare

implementarea analizorului sintactic

Vom implementa analizorul sintactic pentru limbajul AtomC folosind ca algoritm *analiza sintactică descendent recursivă (ASDR)*. Această modalitate de implementare se pretează foarte bine la o implementare manuală (direct de către programator, fără unelte care să genereze analizoare sintactice) și transpune gramatica sintactică în cod într-o manieră directă.

Implementarea regulilor sintactice folosind algoritmul ASDR

ASDR implementează o gramatică conform următoarelor reguli:

- Se folosește un iterator care parcurge lista de atomi lexicali. Spunem că un atom este *consumat*, dacă iteratorul trece la următorul atom.
- Fiecare regulă sintactică (RS) se implementează printr-o funcție proprie (FIRS - funcție care implementează o regulă sintactică)
- O FIRS nu are niciun parametru și returnează *true/false*, dacă RS implementată de ea s-a îndeplinit sau nu. De exemplu, RS *unit* poate fi implementată prin funcția: "**bool** unit(){...}". În limbajul C, pentru a avea acces la tipul de date **bool**, precum și la constantele *true/false*, se poate include antetul <*stdbool.h*>.
- Dacă o FIRS s-a îndeplinit (returnează *true*), atunci ea trebuie să-și consume toți atomii din care este compusă. De exemplu, dacă s-a consumat RS *structDef*, atunci trebuie consumați toți atomii ei, de la STRUCT și până la SEMICOLON, inclusiv.
- Dacă o FIRS nu este îndeplinită (returnează *false*), atunci ea nu trebuie să consume niciun atom (se aplică principiul "totul sau nimic"). Este interzis ca o FIRS să consume atomi și să returneze *false*. Această regulă se poate implementa simplu, memorând la începutul FIRS iteratorul în lista de atomi, iar apoi, când este necesar să se returneze *false*, înainte de aceasta se va reface iteratorul cu poziția memorată.
- Implementarea propriu-zisă a unei RS în interiorul FIRS se face astfel:
    - Dacă există, se elimină *recursivitatea stângă* din RS
    - Fiecare componentă a RS se implementează folosind o secvență de cod specifică

Analiza sintactică se realizează inițializând iteratorul pe primul atom din lista de atomi, iar apoi se apelează regula de start a gramaticii (*unit* pentru AtomC). Această regulă cuprinde în ea toți atomii lexicali, inclusiv END, deci, dacă este îndeplinită, s-a realizat verificarea sintactică a întregului program de intrare.

Eliminarea recursivității stângi

O RS este recursivă la stânga dacă există ramuri ale ei care încep cu regula însăși. De exemplu, regula:

exprOr: **exprOr** OR **exprAnd** | **exprAnd**

este recursivă la stânga pentru că pe prima ramură a lui SAU (|) se începe tot cu ea însăși (*exprOr: exprOr …*). RS care sunt recursive la stânga nu se pot implementa folosind algoritmul ASDR, deoarece se crează o recursivitate infinită:

```
bool exprOr(){
    if(exprOr()){ // recursivitate infinită
        ...
```

Din acest motiv, este necesar ca prima oară să se elimine din RS recursivitatea stângă. Pentru a se elimina recursivitatea stângă, se folosește următoarea formulă:

$$A: A\,\alpha_1 \mid \ldots \mid A\,\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n \quad \rightarrow \quad \begin{array}{l} A: \beta_1\,A' \mid \ldots \mid \beta_n\,A' \\ A': \alpha_1\,A' \mid \ldots \mid \alpha_m\,A' \mid \varepsilon \end{array}$$

Această formulă are următoarele componente:

- A - Numele RS (ex: *exprOr*)
- A' - O nouă RS care se va introduce în gramatică. Aceasta poate avea orice nume valid, de exemplu "*exprOrPrim*".
- $\alpha_1 \ldots \alpha_m$ - Toate fragmentele de pe ramurile care sunt recursive la stânga, care urmează imediat după numele RS. Pentru *exprOr*, avem o singură ramură recursivă la stânga, deci avem doar α_1 = *OR exprAnd*
- $\beta_1 \ldots \beta_n$ - Toate componentele de pe ramurile care nu sunt recursive la stânga. Pentru *exprOr*, avem o singură ramură care nu este recursivă la stânga, deci doar β_1 = *exprAnd*
- ε (epsilon) - Alternativa vidă. ε se îndeplinește fără să se consume niciun caracter. Este echivalent cu faptul că întreaga RS devine opțională (A: e | ε ⇔ A: e?).

Pentru a fi mai ușor de reținut formula, se poate constata că în interiorul ei există doar A', nu și A. În A rămân doar ramurile care nu sunt recursive la stânga, iar în A' doar ramurile care sunt recursive la stânga, la care se adaugă și ε. Atât în A cât și în A', după fiecare fragment din RS originară (α sau β) se pune A'.
Conform formulei, RS originară se va înlocui cu alte două RS. După aplicarea acestei reguli, *exprOr* devine:

exprOr: **exprOr** OR **exprAnd** | **exprAnd** → **exprOr**: **exprAnd exprOrPrim**
exprOrPrim: OR **exprAnd exprOrPrim** | ε

Se poate constata că noua RS introdusă, *exprOrPrim*, este recursivă (prima ramură a ei se încheie tot cu ea însăși). Această recursivitate nu este deranjantă, deoarece ea nu este infinită. Ca demonstrație, pentru a se ajunge la apelul recursiv *exprOrPrim*, trebuie să se consume cel puțin 2 atomi lexicali (*OR* și atomii din *exprAnd*, care are minim un atom). Astfel, la fiecare apel recursiv ne vom apropia de sfârșitul listei de atomi, deci recursivitatea nu este infinită.

Implementarea componentelor regulilor sintactice

O RS se poate compune din următoarele componente:

A. Atomi lexicali (terminale, notați cu litere mari)
B. Reguli sintactice (încep cu o literă mică)
C. Secvențe: $e_1\ e_2 \ldots e_n$
D. Alternative: $e_1 \mid e_2 \mid \ldots \mid e_n$
E. Repetiție opțională: e*
F. ε (epsilon)
G. Alte construcții: opționalitate (e?), repetiție obligatorie (e+)

În ASDR aceste componente se implementează în felul următor:

A. Atomii lexicali

Atomii lexicali se consumă folosind funcția *consume*. Această funcție primește ca parametru un cod de atom. Dacă la poziția curentă a iteratorului în lista de atomi se află un atom cu același cod, atomul respectiv se consumă, iar *consume* returnează *true*. Dacă la poziția curentă se află un atom cu un alt cod decât cel cerut, *consume* returnează *false* și iteratorul rămâne pe poziția curentă.

```
Token *iTk; // iteratorul în lista de atomi. Inițial pointează la primul atom din listă.
Token *consumedTk; // atomul care tocmai a fost consumat. Va fi folosit în etapele următoare ale compilatorului.
```

```
bool consume(int code){
    if(iTk->code==code){// dacă la poziția curentă avem codul cerut, consumăm atomul
        consumedTk=iTk;
        iTk=iTk->next;
        return true;
        }
    return false; // dacă la poziția curentă se află un atom cu un alt cod decât cel cerut, nu are loc nicio acțiune
    }
```

Funcția *consume* se poate folosi în felul următor:

```
// stmCompound: LACC ( varDef | stm )* RACC
bool stmCompound(){
    if(consume(LACC)){ // se testează dacă la poziția curentă este {
        // dacă da, atunci { este deja consumat și se continuă cu interiorul blocului {...}
        ...
```

B. Reguli sintactice

Regulile sintactice se consumă apelând FIRS care le implementează. De exemplu, dacă vrem să testăm dacă există un tip de bază, apelăm *typeBase*. În caz că la poziția curentă se află un tip de bază, atunci *typeBase* îl va consuma în întregime și va returna *true*. Dacă la poziția curentă nu este un tip de bază, atunci *typeBase* va returna *false*, iar iteratorul în lista de atomi va rămâne nemodificat.

```
// fnParam: typeBase ID arrayDecl?
bool fnParam(){
    if(typeBase()){ // se testează dacă la poziția curentă se poate consuma regula typeBase
        // dacă da, atunci typeBase este deja consumat și se continuă cu restul regulii fnParam
        ...
```

C. Secvențe: e_1 e_2 ... e_n

O secvență este echivalentul lui **ȘI** între toate componentele sale. Din acest motiv, o secvență se îndeplinește doar dacă se îndeplinește fiecare componentă. Vom implementa secvențele folosind *if-uri imbricate*, astfel încât să ajungem la următoarele if-uri doar dacă primele au fost testate cu succes.

```
// fnParam: typeBase ID arrayDecl?                    ( secvență de 3 componente)
bool fnParam(){
    Token *start=iTk; // se salvează poziția inițială a iteratorului
    if(typeBase()){
        if(consume(ID)){ // se ajunge la ID doar dacă s-a trecut de typeBase
            if(arrayDecl()){ // se ajunge la arrayDecl doar daca s-a trecut de ID
                }
            return true; // arrayDecl este opțional, deci "return true;" nu depinde
```

```
de existența sa
            }
        }
    // în caz că oricare dintre componentele secvenței nu se îndeplinește, atunci se
ajunge la ”return false;”
    // se reface poziția inițială a iteratorului, în caz că unele if-uri exterioare
au consumat atomi
    iTk=start;
    return false;
    }
```

Din implementare, se poate constata că se ajunge la sfârșitul RS (*”return true;”*), doar dacă sunt îndeplinite toate if-urile. Înainte de *”return false;”* se reface poziția inițială (de la intrarea în funcție) a iteratorului în lista de atomi, pentru cazul că unele if-uri exterioare au consumat atomii de început ai regulii *fnParam*. Astfel se respectă principiul ”totul sau nimic” (o FIRS, dacă returnează *true*, atunci trebuie să-și consume toți atomii componenți, altfel, dacă returnează *false*, atunci nu trebuie să consume niciun atom).

D. Alternative: e_1 | e_2 | ... | e_n

Vom implementa alternativele folosind if-uri succesive, câte unul pentru fiecare ramură. Dacă o ramură se îndeplinește, nu vom mai testa următoarele ramuri. Dacă niciuna dintre ramuri nu este îndeplinită, atunci rezultă *false*.

```
// typeBase: TYPE_INT | TYPE_DOUBLE | TYPE_CHAR | STRUCT ID
bool typeBase(){
    if(consume(TYPE_INT)){
        return true;
        }
    if(consume(TYPE_DOUBLE)){
        return true;
        }
    if(consume(TYPE_CHAR)){
        return true;
        }
    if(consume(STRUCT)){
        if(consume(ID)){
            return true;
            }
        }
    return false;
    }
```

E. Repetiție opțională: e*

Putem implementa repetiția opțională printr-o buclă infinită din care se iese atunci când nu se mai poate consuma expresia repetată.

```
// unit: ( structDef | fnDef | varDef )* END
```

```
bool unit(){
    for(;;){                // buclă infinită
        if(structDef()){}
        else if(fnDef()){}
        else if(varDef()){}
        else break;         // dacă nu se poate consuma nimic la iterația curentă, se
iese din buclă
        }
    if(consume(END)){       // se ajunge la END chiar și dacă nu se consumă nimic în
repetiție, deci ea este opțională
        return true;
        }
    }
```

Dacă expresia care se repetă este simplă (ex: o secvență), se poate folosi o variantă simplificată de implementare, în care primul membru al secvenței se folosește direct ca și condiție pentru repetare:

```
// ( COMMA expr )*
while(consume(COMMA)){
    if(expr()){
        }
    }
```

F. ε (epsilon)

Epsilon se îndeplinește fără să se consume niciun caracter. Din acest motiv, dacă este vorba de o întreagă RS, se implementează ca "**return true;**". Dacă *epsilon* apare într-o subexpresie a RS, atunci se lasă pur și simplu codul să continue, fără niciun test.

```
// exprOrPrim: OR exprAnd exprOrPrim | ε
// echivalent cu: exprOrPrim: ( OR exprAnd exprOrPrim )?
bool exprOrPrim(){
    if(consume(OR)){    // prima alternativă: OR exprAnd exprOrPrim
        if(exprAnd()){
            if(exprOrPrim()){
                return true;
                }
            }
        }
    return true;    // ε - exprOrPrim returneaza true chiar dacă nu consumă nimic
    }
```

G. Alte construcții: opționalitate (e?), repetiție obligatorie (e+)

Opționalitatea (e?) se poate considera ca o alternativă în care ultima ramură este ε (e | ε). Din acest motiv, se poate implementa prin testarea lui *e*, iar apoi codul continuă indiferent dacă s-a consumat sau nu *e*.

```
// arrayDecl? SEMICOLON
```

```
if(arrayDecl()){}
if(consume(SEMICOLON)){}    // se ajunge la SEMICOLON indiferent dacă s-a consumat
sau nu arrayDecl
```

Repetiția obligatorie (e+) se transformă într-o secvență formată din *e* (pentru a necesita apariția obligatorie), urmată de repetiția opțională a lui *e* (e+ → e e*), apoi se implementează ca atare.

Tratarea erorilor

Erorile sintactice apar atunci când analiza sintactică se blochează în interiorul unei reguli și ea nu mai poate să continue în niciun fel folosind atomul curent. De exemplu, în gramatică, după atomul IF trebuie neapărat să urmeze LPAR. Dacă am consumat IF, iar după el nu apare LPAR, atunci nu există nicio posibilitate să se avanseze cu analiza sintactică. În această situație, vom emite o eroare sintactică de forma "*lipsește (după if*".

Pentru a emite erorile sintactice, folosim funcția *tkerr*, astfel încât să se afișeze și linia din codul sursă unde a survenit eroarea:

```
if(consume(IF)){
    if(consume(LPAR)){
        ...
        }else tkerr("lipseste ( dupa if");
    }
```

Pentru a diferenția cazurile în care trebuie să emitem erori, față de cazurile în care trebuie să avem "*return false;*", vom ține cont de următoarele reguli:

- "*return false;*" se folosește atunci când mai există și alte alternative de a continua analiza sintactică, chiar dacă regula curentă nu este îndeplinită. De exemplu, regula *structDef* nu trebuie să emită eroare dacă nu se îndeplinește, deoarece în regula *unit*, de unde este apelată *structDef*, mai există și alte variante de continuare a analizei sintactice (ex: *fnDef*, *varDef* sau END), chiar dacă nu se îndeplinește *structDef*. Din acest motiv, apariția lui *structDef* este opțională, deci ea va returna *false* dacă nu se îndeplinește, fără să emită eroare.
- Emiterea unei erori se face atunci când nu mai există nicio variantă de a se continua analiza sintactică, chiar dacă se abandonează regula curentă pentru a se încerca alte variante din regulile părinți. De exemplu, dacă în *structDef* s-a consumat deja RACC, atunci este obligatoriu să urmeze SEMICOLON. Nu există nicio altă variantă în toată gramatica să existe construcții de forma "STRUCT ID LACC varDef* RACC", după care să nu urmeze SEMICOLON.

Se poate constata că, în general, la primul membru al unei secvențe, nu trebuie să emitem eroare dacă acesta lipsește, pentru a se da posibilitatea testării și altor variante. Dacă în schimb s-a început o secvență, atunci este foarte posibil ca deja de la al doilea membru al ei, dacă acesta lipsește și nu este opțional, să trebuiască se emitem erori.

Câteva sugestii pentru textele erorilor sintactice:

- Când concepem textele erorilor sintactice, trebuie să ne punem în postura unui programator care utilizează compilatorul implementat de noi. Acest programator nu știe nume de atomi sau de RS, deci nu vom folosi aceste nume în mesajele de eroare. De exemplu, nu vom spune "*lipsește LPAR după IF*", ci "*lipsește (după if*".
- În general, erorile sintactice pot apărea din două motive:

    - **în mod evident lipsește ceva** - de exemplu, "(" după *if* - în această situație, când este clar că ceva lipsește, textul erorii poate începe cu "*lipsește…*".
    - **o componentă există, dar ea este invalidă** - de exemplu, în codul sursă apare "*if(a b)*", adică programatorul nu a pus operatorul dintre *a* și *b*. În această situație, condiția de fapt va fi formată doar din *a*, și compilatorul se va plânge că după *a* nu urmează ")". Pentru acest gen de erori, în care este mai probabil să avem componente eronate, decât ca acestea să nu existe, putem folosi formulări de genul "*condiție invalidă pentru if sau lipsește)*", pentru a se insista astfel pe cea mai probabilă eroare, atunci când testăm pentru existența ")".
- Când avem de-a face cu alternative, este destul de complicat să emitem un mesaj de eroare foarte exact, fiindcă sunt multe posibilități care pot duce la eroare. În aceste situații, vom emite un mesaj de eroare generic, de exemplu "*eroare de sintaxă*". O asemenea situație apare de exemplu în regula *unit*, când s-a ieșit din repetiția opțională, dar cu toate astea nu s-a ajuns la END.

Depanarea analizorului sintactic

Implementarea ASDR duce la destul de multe apeluri recursive, directe sau indirecte. Din acest motiv, uneori este mai greu să căutăm erorile folosind doar facilitățile de debugging care există în diverse IDE (breakpoints, step over, step into, …).

Pentru a facilita depanarea analizorului sintactic, putem folosi următoarele metode:

- La începutul fiecărei FIRS afișăm numele acesteia (ex: *puts("# varDef");*). În acest fel vom vedea prin ce succesiune de reguli trece analiza sintactică. Această metodă nu ne spune și când se încheie o regulă (pentru că putem avea în ea mai multe return-uri), dar oricum este un ajutor semnificativ.
- Vom modifica funcția *consume*, în așa fel încât să afișeze atât ce atom se dorește a fi consumat, cât și care a fost rezultatul execuției lui *consume*. Pentru aceasta, va trebui să avem o funcție care returnează numele asociat unui cod de atom, funcție pe care putem să o refolosim de la implementarea afișării atomilor lexicali din analizorul lexical.

```
// const char *tkCodeName(int code) - o funcție care primește ca parametru un cod de
atom și îi returnează numele
bool consume(int code){
      printf("consume(%s)",tkCodeName(code));
      if(iTk->code==code){
            consumedTk=iTk;
            iTk=iTk->next;
            printf(" => consumed\n");
            return true;
            }
      printf(" => found %s\n",tkCodeName(iTk->code));
      return false;
      }
```

Această variantă de *consume*, dacă dorim de exemplu să consumăm atomul IF, iar acesta se află la poziția curentă, va afișa:

```
consume(IF) => consumed
```

Dacă în schimb la poziția curentă avem atomul RETURN, se va afișa:

```
consume(IF) => found RETURN
```

În caz de erori, vom urmări în primul rând izolarea regulii în care apare eroarea. Pentru aceasta, ștergem tot ce putem din codul sursă, astfel încât să rămânem cu cât mai puțini atomi și reguli care se execută.

O posibilitate de dezvoltare a analizorului lexical este să testăm pe rând câte o regulă. Pentru acesta folosim un cod sursă de test cât mai simplu, în care să avem doar construcția pe care o testăm. De exemplu, prima oară testăm definițiile de variabile (iar atunci codul sursă va consta doar dintr-o definiție de variabilă), apoi definițiile de structuri, apoi funcții vide, după care inserăm în funcție câte o instrucțiune (stm) etc. Pentru fiecare dintre aceste situații vom testa atât în varianta corectă, cât și în cazuri de eroare, dacă analizorul funcționează corect. De exemplu, putem să introducem mai multe sau mai puține paranteze, acolade, virgule etc. De asemenea, putem sa omitem diverse componente, cum ar fi operanzi, condiții, nume, tipuri, separatori etc. În acest fel, din aproape în aproape, ne vom asigura că fiecare regulă este implementată corect.