Limbaje formale și tehnici de compilare

analiza de domeniu

În cadrul analizei de domeniu (AD) se preiau din textul sursă definițiile de simboluri (variabile, funcții, structuri, …) și se depun într-o structură de date denumită **tabela de simboluri**. Tot în această etapă se verifică dacă un simbol este redefinit în mod incorect.

Domeniul unui simbol este partea din program în care acel simbol este vizibil. De exemplu, un simbol global este vizibil din tot programul, iar un simbol declarat în cadrul unei funcții este vizibil doar în acea funcție. Domeniul primului simbol este global, iar domeniul celui de-al doilea este domeniul local funcției în care a fost definit. În general, la limbajele de programare cu structură de bloc, fiecare bloc definește un nou domeniu, imbricat în domeniul părinte. De exemplu, acoladele unui **if** definesc un subdomeniu în care putem defini noi variabile.

Din acest punct de vedere, tabela de simboluri (TS), poate fi organizată ca o stivă de domenii: baza stivei este domeniul global, iar vârful stivei este domeniul curent. De fiecare dacă când intrăm într-un nou domeniu (ex: când intrăm într-o funcție), adăugăm la TS acel domeniu în vârf. Când s-a încheiat domeniul respectiv (ex: când ieșim din funcție), ștergem din TS domeniul, împreună cu toate simbolurile care au fost declarate în el.

În exemplul următor se poate vedea cum funcționează analiza de domeniu pentru un program simplu. TS este organizată ca o stivă de domenii, incluse între {...}, cu vârful la dreapta. Fiecare domeniu din stivă este figurat între [...].

```
// TS={[]} - inițial în TS se află un singur domeniu, vid, cel global
int x,y;          // {[x,y]} ; s-au adăugat variabilele în domeniul curent (global)
void f(           // {[x,y,f]} ; la '(' începe domeniul local al funcției
      int x,      // {[x,y,f], [x]} ; x global poate fi redefinit ca argument,
fiindcă este în alt domeniu
      int n)      // {[x,y,f], [x,n]}
{
    int n;        // eroare: redefinire n (n există deja în domeniul funcției)
    if(x<y){      // {[x,y,f], [x,n], []} ; acoladele încep un nou domeniu
        int n=x-1;      // {[x,y,f], [x,n], [n]}
        }         // {[x,y,f], [x,n]} ; la '}' se iese din domeniul lui if
}     // {[x,y,f]} ; la '}' se iese din domeniul funcției și rămân doar simbolurile
din domeniul global
```

Pentru limbajul AtomC vom realiza AD simultan cu analiza sintactică. În această situație, este necesar ca în cadrul regulilor sintactice să avem posibilitatea de a executa cod și de a primi/returna diverse valori în/din regula sintactică executată. Regulile sintactice devin astfel un fel de funcții, care pot primi parametri, pot executa cod și pot returna valori.

Atribute moștenite și sintetizate

Pentru ca o regulă sintactică să poată avea parametri și să poată returna valori, îi vom defini **atribute**. Aceste atribute pot fi:

- **moștenite** - sunt trimise regulii de la regulile care o apelează. Atributele moștenite sunt echivalentul parametrilor unei funcții.
- **sintetizate** - sunt computate (sintetizate) în interiorul regulii și apoi returnate regulilor care o apelează. Atributele sintetizate sunt echivalentul valorilor returnate de o funcție.

Specificarea atributelor se face punând paranteze drepte după numele regulii, iar apoi specificând în interiorul acestora felul, tipul și numele atributului:

rule[**in** int n, **out** char *ch]: …

Regula *rule* are un atribut moștenit (notat cu **in**) de tip *int*, denumit *n* și un atribut sintetizat (notat cu **out**) de tip *char**, denumit *ch*. Uneori mai este folosit și specificatorul **inout**, pentru a denota atribute care sunt folosite atât ca parametri, cât și ca valori returnate.

Deoarece atributele sintetizate trebuie să returneze valori afară din regulă, este necesar să se folosească o metodă care permite aceasta. De exemplu, pentru limbajul C se poate folosi transfer prin adresă (ca în exemplul de mai sus), pentru C++ transfer prin referință etc. Pentru limbaje care nu permit transferul parametrilor prin referință (ex: Java), se pot folosi mai multe metode, de exemplu împachetarea valorii într-o clasă, vectori cu un singur element, variabile globale etc.

Apelul unei reguli care are definite atribute se face punând după ea paranteze drepte și în interiorul lor valorile de apel:

rule2: … rule**[21, &c]** …

În cazul atomilor lexicali, se poate specifica la consumarea acestora un nume de parametru, cu convenția că acel parametru va conține atomul respectiv:

rule3: … ID**[name]** …

Dacă atomul ID a fost consumat, *name* va conține tot acel atom (nu doar câmpul *text* de exemplu). Pentru implementarea noastră, când trebuie să memorăm atomi consumați, vom folosi variabila globală *consumedTk*, pe care o setează funcția *consume* de fiecare dată când a consumat un atom. Pentru exemplul de mai sus putem avea următorul cod: “Token *name=consumedTk;”, care trebuie inserat imediat după apelul lui ”consume(ID)”.

Acțiuni semantice

Acțiunile semantice sunt secvențe de cod care se inserează direct în corpul unei reguli sintactice. În acest fel, o regulă sintactică devine capabilă să execute diverse acțiuni. Acțiunile semantice pot apărea oriunde în interiorul regulii și se specifică între acolade:

line: ID[name] ASSIGN CT_INT[val] {printf(“%d was assigned to %s\n”,val->i,name->text);} SEMICOLON

Dacă o acțiune semantică apare singură pe o ramură a unei alternative, atunci se consideră că acea ramură este de tip *epsilon* (se îndeplinește fără a se consuma nimic).

Tipuri, structuri de date și funcții necesare pentru AD

Toate tipurile, structurile de date și funcțiile specificate mai jos sunt deja implementate în fișierele **ad.h** și **ad.c**.

Implementarea unui simbol

Pentru a memora un simbol în TS, avem nevoie de următoarele date:

- **numele** simbolului
- **felul** - dacă este variabilă, parametru, funcție sau structură
- **tipul** - implementat de structura *Type*, care conține tipul de bază (inclusiv pointer la definirea structurii pentru definiții de structuri) și, în caz de vectori, dimensiunea vectorului
- **owner** - un pointer la simbolul (funcție sau structură) în interiorul căruia a fost definit simbolul curent. Pentru simboluri globale, *owner* este *NULL*
- **varIdx** - se folosește pentru variabile, pentru a se ști unde anume este stocată variabila respectivă

- varMem - se folosește pentru variabile globale; pointează la zona de memorie (alocată dinamic) care conține valoarea acelei variabile
- **paramIdx** - se folosește pentru parametri de funcții și indică indexul parametrului în lista de parametri
- **structMembers** - se folosește pentru structuri și este o listă cu membrii structurii respective (variabile)
- **fn** - se folosește pentru funcții și stochează o listă cu parametrii funcției și o listă cu variabilele sale locale, indiferent de subdomeniul în care au fost definite

Deoarece în implementarea propusă stocăm simbolurile sub forma unei liste simplu înlănțuite, folosim câmpul **next** pentru această înlănțuire.

Managementul simbolurilor se face cu ajutorul următoarelor funcții:

- *Symbol *newSymbol(const char *name,SymKind kind)* - alocă dinamic un nou simbol, având numele și felul specificate
- *Symbol *dupSymbol(Symbol *symbol)* - duplică simbolul dat ca parametru
- *Symbol *addSymbolToList(Symbol **list,Symbol *s)* - adaugă simbolul la sfârșitul listei
- *int symbolsLen(Symbol *list)* - returnează numărul de simboluri din listă
- *void freeSymbol(Symbol *s)* - eliberează memoria ocupată de un simbol
- int typeSize(Type *t) - returnează dimensiunea în octeți a unui tip

Implementarea tabelei de simboluri (TS)

TS se implementează ca o stivă de domenii. Fiecare domeniu conține o listă cu toate simbolurile definite în el și o înlănțuire la domeniul părinte. Managementul TS va fi realizat prin funcții de forma:

- *Domain *pushDomain()* - crează și adaugă un domeniu în vârful stivei de domenii
- *void dropDomain()* - șterge domeniul din vârful stivei de domenii
- *Symbol *addSymbolToDomain(Domain *d,Symbol *s)* - adaugă simbolul *s* în domeniul *d*
- *Symbol *findSymbolInDomain(Domain *d,const char *name)* - caută un simbol cu numele *name* în domeniul *d*
- *Symbol *findSymbol(const char *name)* - caută un simbol cu numele *name* în toate domeniile, începând cu domeniul curent.
- *void showDomain(Domain *d,const char *name)* - afișează conținutul domeniului *d*, dându-i numele *name*

Implementarea AD pentru AtomC

Toate regulile și acțiunile necesare pentru AD sunt descrise în fișierul ”AtomC - analiza de domeniu”. Aici apar toate regulile sintactice pe care va trebui să le modificăm în analizorul sintactic, pentru a extrage simbolurile din codul sursă și a le depune în TS. Practic va trebui la unele reguli sintactice să adăugăm atribute și cod, în locurile specificate. Toate aspectele care trebuie introduse în corpul regulilor sunt marcate cu bold. Dacă în corpul unei reguli apare ”...”, înseamnă că în acea poziție corpul regulii rămâne la fel ca și până acum.

Variabila *owner* este globală și are tipul *Symbol**. Ea este inițial *NULL* și va indica simbolul în interiorul căruia suntem (funcție sau structură).

Ca exemplu de implementare, putem folosi regula *arrayDecl*:

```
arrayDecl[inout Type *t]: LBRACKET
    ( CT_INT[tkSize] {t->n=tkSize->i;} | {t->n=0;} )
    RBRACKET
```

Pornind de la această definiție, o posibilă implementare este următoarea:

```
bool arrayDecl(Type *t){
    if(consume(LBRACKET)){
        if(consume(CT_INT)){
            Token *tkSize=consumedTk;
            t->n=tkSize->i;
            }else{
            t->n=0;           // array fara dimensiune: int v[]
            }
        if(consume(RBRACKET)){
            return true;
            }else tkerr("missing ] or invalid expression inside [...]");
        }
    return false;
    }
```

Se pot constata următoarele:

- funcția are acum un parametru, corespunzător atributului *t*
- Corpul regulii rămâne practic la fel ca la analizorul sintactic, doar că trebuie să inserăm acțiunile semantice.
- pentru construcția *CT_INT[tkSize]*, s-a definit variabila "Token *tkSize", căreia i-a fost atribută valoarea *consumedTk* imediat după ce atomul dorit (CT_INT) a fost consumat
- Există o ramură care nu consumă nimic (de tip *epsilon*) și care are pe ea o acțiune semantică. Această ramură provine de la faptul că în regula inițială *arrayDecl*, CT_INT este opțional. Ramurile de tip *epsilon* se implementează simplu, inserând pe ele acțiunea semantică necesară și considerând că returnează *true*.

Inserarea AD în compilator se realizează astfel:

```
// afisare atomi lexicali (sfarsitul analizei lexicale)

pushDomain();    // creaza domeniul global in tabela de simboluri

// analiza sintactica (apelare analizor sintactic)

showDomain(symTable,"global");    // afisare domeniu global
dropDomain();    // sterge domeniul global
```

Pentru a se testa AD, se poate folosi fișierul de test din activitate. În acest caz, pe ecran vor trebui afișate simbolurile într-un mod similar cu conținutul fișierului "simboluri-rezultate.txt".