Limbaje formale si tehnici de compilare

Laborator 5

Actiuni semantice

Un analizor sintactic pur nu face altceva decat sa verifice daca succesiunea atomilor lexicali corespunde regulilor sintactice. Daca exista erori sintactice se genereaza erori. In caz de succes se returneaza succes. Pentru a se mai putea intreprinde si alte actiuni pe langa verificarea sintactica, in interiorul analizorului se insereaza actiuni care vor fi executate cand se indeplineste regula respectiva. Aceste actiuni, care de fapt reprezinta secvente de cod incluse in gramatica, se numesc actiuni semantice. O gramatica insotita de actiuni semantice se numeste **schema de traducere**.

Pentru a se putea interfata actiunile semantice cu analizorul sintactic, acestea trebuie sa dispuna de diverse informatii cum ar fi atomul lexical care tocmai a fost consumat. In general ele trebuie sa poata fi capabile sa transfere informatie intre diversele reguli sintactice, pentru a se putea stoca si procesa rezultatele partiale. Aceasta informatie este stocata sub forma **atributelor**. Exista doua mari tipuri de atribute:

- **atribute mostenite** – isi capata valoarea in exteriorul unei reguli si sunt transmise ca atare acesteia
- **atribute sintetizate** – sunt calculate in interiorul unei reguli (si in regulile apelate de acestea) dupa care sunt returnate la incheierea executiei regulii respective

Analogic vorbind, atributele mostenite echivaleaza cu parametrii de apel ai unei functii, pe cand atributele sintetizate echivaleaza cu valorile returnate de acea functie.

Exista mai multe notatii posibile pentru actiunile semantice si pentru atribute, de exemplu o notatie atribuie fiecarei componente a unei reguli un index si acceseaza atributele acelor componente pe baza indexului respectiv. In acele ce urmeaza vom folosi o notatie mai flexibila in care vom putea atribui direct nume tuturor atributelor de interes:

- **head (attrDef1,attrDef2,…) ::=** … – declara pentru regula curenta (nonterminalul avand numele **head**) o serie de atribute. Fiecare atribut este definit conform urmatoarei sintaxe:
  - **in nume:tip** – declara atributul mostenit avand numele „nume” si tipul „tip”
  - **out nume:tip** – declara atributul sintetizat avand numele „nume” si tipul „tip”
- **nonterminal:attr** sau **nonterminal:(attr1,attr2,…)** – in interiorul unei reguli apeleaza nonterminalul respectiv cu atributul sau cu atributele specificate. Tipurile atributelor se vor deduce automat din definitia nonterminalului.
- **token:attr** – „attr” va fi o referinta la acel token si prin intermediul lui vor putea fi accesate campurile dorite
- **{…}** – o actiune semantica. In interiorul acoladelor va fi cod scris in limbajul in care este implementat compilatorul
- **{…}?** – un predicat semantic. In interior se afla o conditie care va putea fi testata cu ajutorul instructiunii „if”. Se trece mai departe in corpul regulii doar daca este indeplinita conditia.

Pentru a nu se repeta diverse secvente de cod, actiunile semantice vor putea fi specificate si in exteriorul regulilor sintactice si aceasta facilitate va fi folosita pentru a se defini functii sau variabile.

Exemplu: Sa se implementeze un calculator care dupa fiecare linie introdusa de la tastatura va evalua expresia respectiva si va afisa rezultatul ei.

Gramatica adnotata cu actiuni semantice si cu atribute pentru acest calculator poate arata in urmatorul mod:

calculator ::= (*expr*:**v** NL **{printf("%g\n",v);}**)*
expr(**out ret**:double) ::= *term*:**ret** (ADD *term*:**t** **{ret+=t;}** | SUB *term*:**t** **{ret-=t;}**)*
term(**out ret**:double) ::= *factor*:**ret** (MUL *factor*:**t** **{ret*=t;}** | DIV *factor*:**t** **{t!=0}? {ret/=t;}**)*
factor(**out ret**:double) ::= REAL:**tk** **{ret=tk->realVal;}** | LPAR *expr*:**ret** RPAR

In gramatica de mai sus se pot constata urmatoarele aspecte:

- „expr", „term" si „factor" au fiecare cate un atribut sintetizat de tip „double". Acest atribut este folosit pentru a calcula valoarea numerica a subexpresiei respective si pentru a o returna in finalul predicatului.
- atomul lexical „REAL" este referit de atributul „tk" si prin intermediul acestui atribut i se acceseaza campul „realVal", care contine valoarea numerica a acelui atom, asa cum a fost ea calculata la analiza lexicala.
- „term" contine si un predicat semantic ceea ce permite sa se efectueze impartiri doar daca impartitorul este diferit de 0.
- de exemplu in „expr", daca s-a evaluat un „term", atributul sintetizat „ret" va primi valoarea returnata de acesta. Daca ulterior apare „+" si dupa el un al doilea „term", lui „ret" i se adauga valoarea celui de-al doilea „term". Acest procedeu se repeta atata timp cat exista operatorii „+" sau „-". In final „ret" va contine valoarea finala a subexpresiei si aceasta va fi returnata de „expr".

La implementarea atributelor trebuie tinut cont de faptul ca un predicat trebuie sa returneze atat „true/false" (daca regulile sale sintactice au fost indeplinite sau nu de atomii de intrare), cat si atributele sintetizate. Deoarece in majoritatea limbajelor de programare nu se permite ca o functie sa returneze mai multe valori, se pot aplica urmatoarele metode:

- in limbajele care permit transfer prin referinta/adresa (C/C++/C#) atributele sintetizate se pot implementa ca argumente suplimentare transmise prin referinta
- in limbajele care nu permit transfer prin referinta (Java) atributele sintetizate se pot returna prin intermediul unui obiect transmis ca parametru suplimentar, obiect care va fi de tipul unei clase care are ca membru o variabila de tipul atributului dorit. O alta varianta este ca o structura cu toate atributele sintetizate sa fie returnata de functie in loc de o valoare logica, aplicandu-se suplimentar conventia ca „null" inseamna si „false", iar un obiect returnat inseamna in acelasi timp „true".

Tinand cont de cele expuse anterior, implementarea lui „factor" poate fi de forma:

```
int factor(double *ret)
{
if(consume(REAL)){
    Token    *tk=consumedTk;
    *ret=tk->realVal;
    }
else if(consume(LPAR)){
    if(!expr(ret))tkerr(crtTk,"invalid expression after (");
    if(!consume(RPAR))tkerr(crtTk,"missing )");
    }
else return 0;
return 1;
}
```

In acesta implementare atributul sintentizat „ret" s-a implementat ca un argument suplimentar, transmis prin adresa. „tk" ar fi putut fi inlocuit direct cu „consumedTk", dar s-a implementat in acest fel pentru a se pastra forma din gramatica. Se poate constata ca pentru expresiile intre paranteze, „ret" a fost setat direct prin apelul lui „expr" si nu a mai fost necesara nicio alta prelucrare ulterioara.

Analiza de domeniu (AD)

Determina pentru o anuma pozitie din program care simboluri sunt vizibile si care nu. **Un simbol este orice definitie care are un nume** (variabile, functii, tipuri, …).

De exemplu urmatorul program, desi este corect atat din punct de vedere lexical cat si sintactic, are unele erori de domeniu:

```
struct Pt    p;          //Gresit: "struct Pt" nu exista
float        x;
int          f(int x)    //Corect: argumentele pot redefini variabile globale
{
    int    x;            //Gresit: o variabila locala in blocul functiei nu poate avea acelasi
                                 nume ca un argument
    {
        int          x;  //Corect: in interiorul un bloc {...} se pot redefini orice variabile
        double       x;  //Gresit: redefinirea lui "x" definit anterior in acest bloc
    }
}
```

Se poate constata ca un program poate fi corect sintactic dar cu toate acestea sa fie gresit semantic (din punct de vedere al intelesului). Erorile care tin de simboluri ce sunt nedefinite, redefinite sau nu sunt accesibile intr-un anumit domeniu (cum ar fi cazul membrilor „private” dintr-o clasa in afara acelei clase) se depisteaza in faza de analiza de domeniu. Domeniul mai este denumit si „context”.

Pentru a se putea realiza AD, toate simbolurile trebuie colectate intr-o structura de date numita „tabela de simboluri” (TS) si pe baza informatiei din TS se va desfasura analiza propriu-zisa.

Tabela de simboluri

Pentru limbajul AtomC un simbol poate fi stocat sub urmatoarea forma:

```
enum{TB_INT,TB_DOUBLE,TB_CHAR,TB_STRUCT,TB_VOID};
typedef struct{
    int          typeBase;      // TB_*
    Symbol       *s;            // struct definition for TB_STRUCT
    int          nElements;     // >0 array of given size, 0=array without size, <0 non array
}Type;

enum{CLS_VAR,CLS_FUNC,CLS_EXTFUNC,CLS_STRUCT};
enum{MEM_GLOBAL,MEM_ARG,MEM_LOCAL};

typedef struct _Symbol{
    const char  *name;          // a reference to the name stored in a token
    int         cls;            // CLS_*
    int         mem;            // MEM_*
    Type        type;
    int         depth;          // 0-global, 1-in function, 2... - nested blocks in function
    union{
        Symbols         args;           // used only of functions
        Symbols         members;        // used only for structs
        };
}Symbol;
Symbols         symbols;
```

Prima oara este definit tipul pe care il poate lua un simbol in AtomC. Acest tip este definit de structura „Type” si poate fi o variabila simpla de tipul „int”, „double”, „char”, poate fi o structura (in acest caz campul „s” detaliaza despre ce structura e vorba), poate fi „void” (doar pentru tipul functiilor) sau poate fi un vector (unidimensional) de

elemente avand unul dintre tipurile anterioare (cu exceptia „void”). In cazul vectorilor campul „nElements” va pastra numarul de elemente. Daca simbolul nu este un vector, acest camp va contine o valoare negativa.

Structura „Symbol” pastreaza toate informatiile ce definesc un simbol cum ar fi numele lui, clasa din care face parte, tipul, etc.

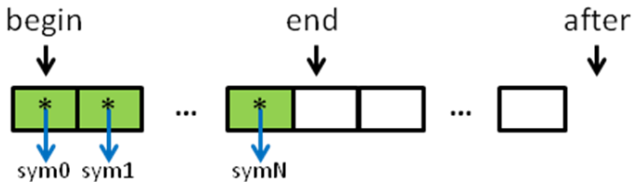
In cazul simbolurilor care cuprind definitii catre alte simboluri (argumentele unei functii sau membrii unei structuri) s-au prevazut campuri speciale care contin simbolurile interioare. Campul „args” contine doar argumentele functiei, nu si variabilele ei locale, acestea fiind tratate separat.

Tabela de simboluri in sine este stocata in variabila „symbols”. In C, deoarece nu exista vectori de dimensiune variabila, s-a implementat tipul „Symbols” ca un vector dinamic de pointeri la „Symbol” in modul urmator:

```
struct _Symbol;
typedef struct _Symbol Symbol;
typedef struct{
    Symbol          **begin;      // the beginning of the symbols, or NULL
    Symbol          **end;        // the position after the last symbol
    Symbol          **after;      // the position after the allocated space
}Symbols;

void            initSymbols(Symbols *symbols)
{
    symbols->begin=NULL;
    symbols->end=NULL;
    symbols->after=NULL;
}
```

„Symbols” este analogul lui „ArrayList<Symbol>” in Java sau a lui „vector<Symbol*>” in C++:



Pentru a nu se realoca memorie de fiecare data cand se adauga un nou pointer, cand intervine necesitatea unei redimensionari se aloca spatiu pentru mai multe pozitii, dublandu-se de fiecare data dimensiunea totala. „end” este prima pozitie libera din vector iar „after” pointeaza la memoria de dupa spatiul de memorie alocat pentru vector. In acest fel, atunci cand se adauga un nou element se testeaza prima data daca „end==after” si in aceasta situatie inseamna ca vectorul este plin si trebuie redimensionat. Altfel, se pune noul element la pozitia „end” si se incrementeaza „end”. Numarul de elemente este „n=end-begin”. Iterarea se poate face cu indecsi:

```
for(i=0;i<n;i++)if(symbols->begin[i]->cls==CLS_FUNC)…
```

sau cu pointeri, analogic iterarilor standard folosite de biblioteca standard „std” din C++:

```
for(p=symbols->begin;p!=symbols->end;p++)if((*p)->cls==CLS_FUNC)…    // Symbol **p
```

Cu aceasta structura de date, o posibila functie care creaza un simbol nou cu un nume si clasa date si il adauga la un vector specificat, arata in felul urmator:

```
Symbol          *addSymbol(Symbols *symbols,const char *name,int cls)
{
    Symbol      *s;
    if(symbols->end==symbols->after){    // create more room
        int     count=symbols->after-symbols->begin;
        int     n=count*2;          // double the room
        if(n==0)n=1;                // needed for the initial case
```

```
            symbols->begin=(Symbol**)realloc(symbols->begin, n*sizeof(Symbol*));
            if(symbols->begin==NULL)err("not enough memory");
            symbols->end=symbols->begin+count;
            symbols->after=symbols->begin+n;
            }
        SAFEALLOC(s,Symbol)
        *symbols->end++=s;
        s->name=name;
        s->cls=cls;
        s->depth=crtDepth;
        return s;
    }
```

Cautarea unui simbol in TS se face cu functia „Symbol *findSymbol(Symbols *symbols,const char *name)”, care cauta simbolul cu numele dat in vectorul specificat si returneaza un pointer catre el daca l-a gasit sau NULL in caz contrar. Cautarea se face de la dreapta la stanga, deoarece exista cazuri in care un simbol poate fi redefinit si atunci este necesar sa se foloseasca ultima definitie a sa (cea mai apropiata de contextul in care acesta e folosit).

Cand se compileaza o functie, parametrii acesteia se stocheaza atat in TS cat si in membrul „args”. Variabilele ei locale se stocheaza doar in TS. Aceasta are urmatoarea motivatie: atat argumentele cat si variabilele locale trebuie sterse din TS la sfarsitul functiei, deoarece ele nu mai sunt valide dupa functie. Parametrii in schimb trebuie sa ramana disponibili undeva, pentru a se verifica daca functia este apelata corect atunci cand este folosita.

Regulile in ce priveste AD sunt date intr-o pagina separata, numita „definitii”. In aceasta pagina sunt specificate atat regulile AtomC in privinta definitiilor, cat si regulile semantice relevante pentru implementarea acestor reguli.

AD trebuie sa populeze TS cu toate simbolurile intalnite. Se vor produce mesaje de eroare in cazurile specificate. Daca nu exista erori, in final se vor afisa toate numele simbolurilor din TS insotite de: cls, mem, type.

Aplicatia 5.1: *Sa se implementeze functiile necesare pentru regulile semantice folosite in pagina „definitii”: findSymbol, deleteSymbolsAfter, …*

Tema: sa se implementeze analiza de domeniu completa pentru limbajul AtomC.