Limbaje formale si tehnici de compilare

Laborator 6

Se considera urmatorul program C:

```
double   abs(double x)
{
    if(x<0)return -x;
    return x;
}
...
int     v[10]=abs(-1);        // Eroare: un "double" nu se poate atribui unui vector
double r=abs(0.5,0.7);        // Eroare: "abs" are un singur parametru
abs=0;                        // Eroare: unei functii nu i se poate atribui o valoare
```

Se poate constata ca desi programul este corect lexical, sintactic si din punct de vedere al analizei de domeniu (toate simbolurile folosite sunt definite in mod corect), el totusi nu este corect din punct de vedere al tipurilor. Pentru analiza erorilor care apar din nerespectarea regulilor de tipuri, se foloseste analiza tipurilor.

Componente necesare pentru analiza tipurilor

Pentru analiza tipurilor (AT) vom avea nevoie ca fiecare predicat ce intervine in tratarea expresiilor sa returneze anumite atribute sintetizate. Aceste atribute sunt grupate intr-o structura de date care contine:

- **tipul propriu-zis**, cum ar fi „int” sau „char[]”
- **daca valoarea este o lval**
- **daca valoarea este o constanta**; in acest caz se pastreaza si valoarea constantei

O **valoare stanga (lval – left value)** este o expresie care se poate afla in stanga unei operatii de atribuire. O lval poate furniza pe langa valoarea ei propriu-zisa si o adresa unde se poate depozita o valoare. Exemple: „a”, „v[i]”, „*p”, „pt.x”.

O **valoare dreapta (rval – right value)** este o expresie care poate fi in dreapta unei operatii de atribuire. O rval nu poate furniza o adresa si deci nu se poate stoca o valoare in interiorul ei. Exemple: „3.14”, „a+2”, „cos(x)”.

Cu cele explicate mai sus devine clara diferenta intre urmatoarele expresii:

```
int a=10;       // OK: in stanga este o lval
2=3;            // Eroare: in stanga este doar o rval care nu furnizeaza o adresa
```

In AtomC un tip de baza poate fi „int”, „double”, „char”, „void” sau o structura. Un tip complet poate fi un tip de baza simplu sau un vector unidimensional al unui tip de baza. Toate acestea sunt definite in structura „Type” (a se vedea laboratorul cu analiza de domeniu). Pentru usurinta vom defini o functie care creeaza un nou tip:

```
Type     createType(int typeBase,int nElements)
{
    Type     t;
    t.typeBase=typeBase;
    t.nElements=nElements;
    return t;
}
```

Algoritm pentru analiza tipurilor

Toate predicatele care trateaza expresii (ex: exprAdd, exprCast, exprPrimary) vor avea un nou atribut sintetizat de tipul „RetVal”:

```
typedef union{
    long int        i;      // int, char
    double          d;      // double
    const char      *str;   // char[]
}CtVal;

typedef struct{
    Type    type;           // type of the result
    int     isLVal;         // if it is a LVal
    int     isCtVal;        // if it is a constant value (int, real, char, char[])
    CtVal   ctVal;          // the constat value
}RetVal;
```

In pagina „tipuri” se afla toate regulile la care se supun tipurile de date in AtomC si totodata actiunile semantice care implementeaza aceste reguli.

La evaluarea unei expresii se evalueaza prima oara componentele acelei expresii (operanzii sai). Daca este necesar sa se realizeze anumite conversii implicite (ex: adunarea dintre un numar cu virgula si unul intreg necesita ca numarul intreg sa fie convertit la numar cu virgula), acestea se vor realiza dupa evaluarea operanzilor.

Constantele (ex: CT_INT) au din start tipurile corespunzatoare asociate lor si in acelasi timp li se salveaza valoarea propriu-zisa.

Identificatorii sunt cautati in tabela de simboluri (TS) si astfel li se afla tipul.

Din aceste tipuri initiale (constante si identificatori) fiecare operatie isi va determina tipul rezultat precum si celelalte componente din „RetVal” si le va propaga mai departe. De exemplu operatia „v[i]” (indexare de vector) va propaga mai departe pe langa tipul rezultat o lval pe cand operatia „a*10” va propaga mai departe o rval. In ambele cazuri valoarea propagata nu este constanta.

Cand este nevoie sa se realizeze o conversie (cast) la un tip destinatie indicat (ex: cand o functie are in TS un parametru de tip „double” dar este apelata cu un parametru de tip „int”), pentru aceste conversii se foloseste functia „cast”:

```
void    cast(Type *dst,Type *src)
{
    if(src->nElements>-1){
        if(dst->nElements>-1){
            if(src->typeBase!=dst->typeBase)
                tkerr(crtTk,"an array cannot be converted to an array of another type");
            }else{
            tkerr(crtTk,"an array cannot be converted to a non-array");
            }
        }else{
        if(dst->nElements>-1){
            tkerr(crtTk,"a non-array cannot be converted to an array");
            }
        }
    switch(src->typeBase){
        case TB_CHAR:
        case TB_INT:
        case TB_DOUBLE:
```

```
            switch(dst->typeBase){
                case TB_CHAR:
                case TB_INT:
                case TB_DOUBLE:
                    return;
                }
        case TB_STRUCT:
            if(dst->typeBase==TB_STRUCT){
                if(src->s!=dst->s)
                    tkerr(crtTk,"a structure cannot be converted to another one");
                return;
                }
            }
        tkerr(crtTk,"incompatible types");
    }
```

Functia „cast” incearca sa converteasca tipul „src” la tipul „dst”. Daca nu este nicio conversie posibila se genereaza eroare. In faza de generare de cod aceasta functie va fi completata cu codul specific de conversie de la un tip la altul.

Pe langa functia „cast” mai este nevoie de inca o functie de conversie, care primeste doua tipuri ca parametri si incearca sa gaseasca tipul lor rezultat (pe care-l returneaza), conform regulilor aritmetice din AtomC. De exemplu daca se aduna un „char” cu un „int”, indiferent in ce ordine, rezultatul va trebui sa fie de tipul „int”. Aceasta functie va avea urmatorul antet:

```
Type    getArithType(Type *s1,Type *s2);
```

Se vor modifica si actiunile semantice din regula „arrayDecl” pentru a se stoca numarul elementelor vectorului, daca acesta este dat.

Actiuni semantice in predicate recursive la stanga

In cazul predicatelor care se modifica pentru eliminarea recursivitatii stangi, actiunile semantice se pot implementa astfel:

- fiecare actiune semantica se considera o componenta a predicatului respectiv si va fi tratata ca facand parte din secventa **α** sau **β** despre care este vorba in formula de eliminare a recursivitatii stangi
- in predicatul initial (A) atributele isi pastreaza modul de folosire (mostenite sau sintetizate)
- in predicatul nou format (A’) atributele sintetizate din predicatul initial (A) devin atribute mixte (atat mostenite cat si sintetizate). Aceste atribute vor fi preluate din A la apelul lui A’ (mostenire). A’ este posibil sa le modifice si apoi le va returna lui A (sintetizare)

Exemplu din regulile de analiza de tipuri pentru AtomC:

```
exprOr(out rv:RetVal): exprOr:rv OR exprAnd:rve {/*use (rv,rve); return rv*/} | exprAnd:rv ;
```

Aceasta regula contine o actiune semantica pentru ramura **α** a regulii de transformare. Aplicand formula si tinand cont de cele de mai sus, se obtine:

```
exprOr(out rv:RetVal): exprAnd:rv exprOrP:rv ;
exprOrP(inout rv:RetVal): OR exprAnd:rve {/*use (rv,rve); return rv*/} exprOrP:rv | ε ;
```

Se observa ca in „exprOrP” „rv” este atat atribut mostenit (care se transmite la apelul lui „exprOrP” din „exprOr”) cat si atribut sintetizat in „exprOrP”, deoarece este posibil sa fie modificat in acest predicat.

O explicatie intuitiva este urmatoarea: daca avem o expresie care nu contine „OR”, atunci in „exprOr” „rv” va lua valoarea sa din apelul lui „exprAnd” si va fi returnat cu aceasta valoare („exprOr” se reduce la „exprAnd”). Daca in schimb avem „OR” dupa „exprAnd” atunci va trebui sa combinam valoarea primului „exprAnd” (preluata deja din

„exprOr”) cu valoarea celui de-al doilea „exprAnd” (cel care urmeaza dupa „OR” si care valoare va fi stocata in „rve”). In acest caz se combina „rv” (atribut mostenit din „exprOr”) cu „rve” (atribut sintetizat) si valoarea rezultata va deveni noul „rv” care se va returna (atribut sintetizat). Acest proces de combinare continua atata timp cat mai sunt si alti operatori „OR” (apelul recursiv al lui „exprOrP”).

Functii predefinite

In AtomC exista o serie de functii predefinite, conform tabelului de mai jos. Aceste functii sunt disponibile din start, fara a fi nevoie sa se includa un antet pentru aceasta. Din acest motiv, ele trebuie adaugate in TS de catre compilator, la inceputul compilarii (altfel vor aparea erori de tipul „simbol nedefinit”). Adaugarea se poate face intr-o functie „addExtFuncs()” care va contine secvente de genul:

```
s=addExtFunc("put_s",createType(TB_VOID,-1));
addFuncArg(s,"s",createType(TB_CHAR,0));
```

unde „addExtFunc” si „addFuncArgs” sunt de forma:

```
Symbol	*addExtFunc(const char *name,Type type)
{
	Symbol	*s=addSymbol(&symbols,name,CLS_EXTFUNC);
	s->type=type;
	initSymbols(&s->args);
	return s;
}

Symbol	*addFuncArg(Symbol *func,const char *name,Type type)
{
	Symbol	*a=addSymbol(&func->args,name,CLS_VAR);
	a->type=type;
	return a;
}
```

Functie	**Efect**
void put_s(char s[])	Afiseaza sirul de caractere dat
void get_s(char s[])	Cere de la tastatura un sir de caractere si il depune in s
void put_i(int i)	Afiseaza intregul „i”
int get_i()	Cere de la tastatura un intreg
void put_d(double d)	Afiseaza numarul real „d”
double get_d()	Cere de la tastatura un numar real
void put_c(char c)	Afiseaza caracterul „i”
char get_c()	Cere de la tastatura un caracter
double seconds()	Returneaza un numar (posibil zecimal pentru precizie mai buna) de secunde. Nu se specifica de cand se calculeaza acest numar (poate fi de la inceputul rularii programului, de la 1/1/1970, …)

Deocamdata aceste functii vor avea doar antetele puse in TS, urmand ca implementarea lor sa fie discutata in cadrul masinii virtuale.

Aplicatia 6.1: *Sa se implementeze functia „getArithType”*

<u>Tema:</u> sa se implementeze analiza de tipuri completa pentru limbajul AtomC.