Limbaje formale si tehnici de compilare

Laborator 8

Pentru simplificarea implementarii, generarea de cod (GC) are loc concomitent cu fazele de analiza sintactica, analiza de domeniu si de tipuri. Din acest motiv GC se foloseste de aceste faze, asa cum au fost ele descrise in laboratoarele anterioare. La implementarea actiunilor semantice se va tine cont de explicatiile din comentarii, care arata unde anume vor fi inserate acele actiuni (ex: „//before "crtFunc=NULL;"”). Actiunile necesare sunt detaliate in pagina „GC”.

Programul principal (implementat in predicatul „unit”) consta dintr-un apel „CALL” la functia „main”, urmat de „HALT” pentru oprirea executiei MV. Acest proces este analogic rularii unui program C. Fiecare predicat este raspunzator sa-si genereze propriul cod.

Important: in gramatica AtomC exista anumite secvente care presupun testarea pe rand a unor alternative, alternative din care se revine cu o secventa de forma „crtTk=startTk”, ce restaureaza atomul curent la valoarea pe care a avut-o la intrarea in predicat. In faza de GC, din cauza ca unele predicate vor genera cod atunci cand sunt apelate, secventa „crtTk=startTk” peste tot unde apare va trebui sa fie dublata de o secventa „deleteInstructionsAfter(startLastInstr);”, unde „startLastInstr” este valoarea lui „lastInstruction” la intrarea in predicat. Aceasta secventa are rolul de a sterge instructiunile generate de un anumit predicat, atunci cand acesta nu este indeplinit.

Argumentele de functii si variabilele locale se adreseaza relativ la registrul „FP”. Pentru aceasta la GC se folosesc doua variabile globale: „int sizeArgs,offset;”. La compilarea unei functii acestea devin 0 si apoi se incrementeaza cu dimensiunea fiecarui argument sau variabila locala intalnita. Dupa compilarea argumentelor de functie, ofsetul acestora trebuie modificat deoarece in MV ofsetul trebuie sa fie relativ la „FP”, care in stiva se afla dupa parametrii de apel si nu relativ la primul parametru de apel. Valoarea „offset” se seteaza si in TS pentru simbolurile locale. Pentru variabilele globale si pentru functii, adresele lor se stocheaza in TS folosind membrul „addr”.

Atat pentru calculul lui „addr” cat si pentru „offset” este necesar sa se cunoasca dimensiunea variabilelor, inclusiv a vectorilor si a structurilor. Tipurile de date primare si dimensiunile lor in AtomC sunt:

Tip	**Dimensiune**	**Dimensiune pentru limbaje fara sizeof**
char	sizeof(char)	1
int	sizeof(long int)	4
double	sizeof(double)	8
adresa (ex: transmitere vectori)	sizeof(void*)	dimensiunea unui tip intreg folosit ca index pentru adrese

Pornind de la aceste tipuri de baza, se pot calcula dimensiunile tipurile compuse:

- Dimensiunea unui vector – nrElemente*dimensiune(element)
- Dimensiunea unei structuri – suma dimensiunilor membrilor sai

Pentru calculul acestor valori, la GC se folosesc urmatoarele functii:

```
int             typeBaseSize(Type *type)
{
    int     size=0;
    Symbol      **is;
```

```
    switch(type->typeBase){
        case TB_INT:size=sizeof(long int);break;
        case TB_DOUBLE:size=sizeof(double);break;
        case TB_CHAR:size=sizeof(char);break;
        case TB_STRUCT:
            for(is=type->s->members.begin;is!=type->s->members.end;is++){
                size+=typeFullSize(&(*is)->type);
                }
            break;
        case TB_VOID:size=0;break;
        default:err("invalid typeBase: %d",type->typeBase);
        }
    return size;
}

int             typeFullSize(Type *type)
{
    return typeBaseSize(type)*(type->nElements>0?type->nElements:1);
}

int             typeArgSize(Type *type)
{
    if(type->nElements>=0)return sizeof(void*);
    return typeBaseSize(type);
}
```

La analiza de tipuri se incearca pastrarea pe cat posibil a valorilor ca „left-values” (lval), adica se opereaza cu adresele lor. Astfel, daca este nevoie sa se stocheze la aceste adrese o valoare, aceasta se poate realiza, operatie care nu ar fi fost posibila cu „right-values” (rval), deoarece acestea nu poseda o adresa. Din acest motiv, cand se intalnesc id-uri, elemente de vectori sau de structuri, pe stiva se depune adresa lor (lval) si nu valoarea lor (rval). Daca este necesar sa se foloseasca valoarea (rval) si deci sa se realizeze o conversie lval->rval, aceasta se realizeaza cu functia „Instr *getRVal(RetVal *rv)”, care genereaza conversia ceruta daca „rv” denota o lval. Din acest motiv nu se folosesc instructiuni de tip „PUSH” (care ar depune direct pe stiva o valoare), ci secvente de „PUSHADDR” (depune o adresa (lval)) urmate eventual de „LOAD” (pune pe stiva valoarea (rval) de la adresa (lval) deja existenta pe stiva).

Pentru conversiile implicite de tipuri (ex: int->double) se foloseste functia „void addCastInstr(Instr *after,Type *actualType,Type *neededType)”, care insereaza dupa instructiunea „after” codul necesar pentru conversia „actualType->neededType”.

Instructiunea „OFFSET” se foloseste pentru a se adauga o valoare (numar de octeti) la o adresa data. Astfel se pot calcula adresele elementelor vectorilor pornind de la adresa vectorului, dimensiunea unui element si indexul dorit si totodata se pot calcula adresele membrilor structurilor (ex: „pt.x”) pornind de la adresa structurii si ofsetul membrului.

Orice expresie, cu exceptia apelurilor de functii „void”, trebuie sa lase o valoare pe stiva. Astfel devin posibile operatii de genul „a=b=0”. Daca „b=0” nu ar lasa rezultatul pe stiva, nu ar fi nicio valoare disponibila pentru a seta „a”. Din acest motiv, operatiile care nu lasa nimic pe stiva (ex: atribuirea) trebuie sa duplice valoarea folosita inainte de a o scoate de pe stiva, in asa fel incat ea sa ramana disponibila pentru operatiile ulterioare. Duplicarea se realizeaza cu instructiunea „INSERT” care preia o valoare din varful stivei si o copiaza prin inserare la o locatie anterioara din stiva, realizandu-se astfel duplicarea ei.

Cand se termina evaluarea unei instructiuni de tip expresie (stm = … expr SEMICOLON …), pe stiva va ramane valoarea finala a expresiei „expr”. Daca tipul final nu este „void”, aceasta valoare trebuie stearsa din stiva. Aceasta se realizeaza folosind instructiunea „DROP”.

Instructiunile de genul „if(expr)stm” se traduc printr-un cod de forma:

```
//generare cod "expr"
JF L1
//generare cod "stm"
L1:
```

Se poate constata ca este necesar sa se genereze un „JF” la o instructiune care nu exista inca si care nici nu se stie cand va fi generata ulterior, dupa ce s-a terminat generarea codului pentru „if”. Din acest motiv, pentru a se putea finaliza generarea de cod pentru „if”, la sfarsit se genereaza si o instructiune „NOP” care actiuneaza ca destinatie pentru „JF”. In aceasta situatie sunt si instructiunile „while” si „for”, care ambele necesita un salt dupa codul lor.

Deoarece generarea de cod pentru instructiunea „for” este ceva mai complexa, o vom detalia in continuare (a se vedea pagina GC pentru detalii). Consideram forma „FOR LPAR expr:rv1? SEMICOLON expr:rv2? SEMICOLON expr:rv3 RPAR stm”. „for” se traduce printr-o instructiune „while” de forma:

```
expr1;
while(expr2){
stm;
expr3;
}
```

Daca „expr2” lipseste, se considera ca fiind „true” si rezulta o bucla infinita. Si „expr1”, „expr3” sunt optionale. Daca exista „expr2” se genereaza unui salt conditionat dupa „while” daca valoarea rezultata este falsa. Mai problematica este situatia creata de faptul ca ordinea generarii de cod conform gramaticii initiale este „expr1 expr2 expr3 stm”, pe cand noi dorim sa avem codul generat in ordinea „expr1 expr2 stm expr3”. Pentru aceasta a fost necesar ca dupa generarea de cod sa se inverseze in lista dublu inlantuita de instructiuni codul aferent lui „expr3” cu codul aferent lui „stm”.

Generarea de cod pentru instructiunea „break” se realizeaza creand inca de la inceputul generarii lui „for” sau „while” o instructiune „NOP”, care va fi adaugata doar la sfarsit in lista, dar care serveste ca destinatie pentru „break”. Aceasta instructiune se tine minte in variabila globala „Instr *crtLoopEnd;”. Fiecare instructiune „for” sau „while” va avea propria sa „crtLoopEnd”. Daca sunt mai multe asemenea instructiuni imbricate, fiecare salveaza vechea valoare „crtLoopEnd” pe durata generarii sale si o restaureaza la sfarsit. „crtLoopEnd” este folosita si ca test in „break” pentru a se semnala eroare in cazurile in care aceasta instructiune se foloseste fara a fi inclusa intr-un „while” sau „for”.

Pentru functiile „void” se genereaza automat o instructiune „RET”, fiindca pentru aceste functii programatorul nu este obligat sa scrie explicit un „return”. Pentru toate celelalte functii este necesar ca programatorul sa includa instructiunile „return” necesare.

Aplicatia 8.1: *Sa se implementeze toate functiile auxiliare necesare implementarii GC (ex: „Instr *appendInstr(Instr *i);”, care adauga instructiunea „i” la sfarsitul listei de instructiuni).*

<u>Tema:</u> sa se implementeze integral generarea de cod pentru AtomC