Limbaje formale și tehnici de compilare

mașina virtuală

Pentru a executa programele scrise în AtomC, vom folosi o **mașină virtuală** (MV). Compilatorul va compila codul sursă AtomC generând instrucțiuni pentru MV, iar apoi aceste instrucțiuni vor fi executate. Va fi folosită o MV bazată pe stivă, deoarece implementarea lor este simplă și în același timp generarea de cod pentru ele este relativ ușoară. Pentru mai multe detalii despre acest gen de MV, a se vedea cursul.
Tipurile de date și declarațiile funcțiilor care implementează MV se găsesc în fișierul **mv.h**, iar implementarea se află în **mv.c**.
O instrucțiune (**Instr**) se compune dintr-un cod de operație (**opcode** - operation code) și un **argument**. Opcode-ul specifică instrucțiunea propriu-zisă (ex: OP_HALT este codul instrucțiunii care oprește execuția MV). Unele instrucțiuni au nevoie de un argument (ex: instrucțiunea PUSH.i, care depune o constantă întreagă pe stivă, are nevoie să stocheze în argument constanta pe care o va depune pe stivă). Argumentul unei instrucțiuni este de tipul **Val**, iar el constituie o așa-numită *valoare universală*, capabilă să stocheze orice tip de date din AtomC. Instrucțiunile sunt implementate simplu, folosind un vector de instrucțiuni, în care se adaugă noi instrucțiuni folosind funcția **addInstr**. Funcțiile **addInstrWithInt** și **addInstrWithDouble** se folosesc pentru a adăuga mai comod instrucțiuni care au un argument de tip **int**, respectiv **double**.
Stiva este implementată ca fiind un vector cu celule de tipul **Val**. Astfel, fiecare celulă din stivă poate stoca orice tip de valoare. Depunerea unei valori în stivă se face cu una dintre instrucțiunile **push**, iar extragerea unei valori din stivă se face cu una dintre instrucțiunile **pop**. Există mai multe instrucțiuni **push** și **pop**, pentru diverse tipuri de date. De exemplu, **pushi** depune pe stivă o valoare de tip **int**.
În această implementare există unele mici diferențe față de instrucțiunile MV din curs, în sensul simplificării acestora. De exemplu, în implementare există o singură instrucțiune FPLOAD, care se poate folosi pentru orice tip de date, spre diferență de curs, unde fiecare tip de date are nevoie de o instrucțiune FPLOAD specifică (ex: FPLOAD.i, FPLOAD.f).

MV se inserează în compilator astfel:

```
pushDomain();
mvInit();    // initializare masina virtuala
parse(tokens);
//showDomain(symTable,"global");
genTestProgram();    // genereaza cod de test pentru masina virtuala
run();    // executie cod masina virtuala
dropDomain();
```

După ce s-a creat domeniul global cu *pushDomain*, se apelează **mvInit** pentru inițializarea MV. În această etapă se adaugă în tabela de simboluri (TS) unele funcții predefinite, de exemplu **put_i**, care afișează o valoare de tip **int**. Aceste funcții trebuie adăugate în TS înainte de a se începe parsarea codului sursă, pentru că altfel ele nu vor fi găsite la analiza de tipuri și vor rezulta erori de genul *"simbol nedefinit"*.

Funcții externe

Funcțiile predefinite cum este **put_i**, care sunt implementate în interiorul compilatorului (în **mv.c**), le considerăm ca fiind **externe** (*external functions*) față de codul sursă AtomC, deoarece ele nu au un corp format din instrucțiuni MV, ca restul funcțiilor care vor fi generate din codul sursă AtomC. De exemplu, dacă compilatorul este scris în C, funcțiile externe vor fi compilate odată cu restul compilatorului la cod mașină. Din acest motiv, funcțiile externe vor trebui apelate altfel decât funcțiile generate din codul sursă AtomC, care vor avea un corp format din instrucțiuni MV. Pentru

a se diferenția între cele două tipuri de funcții (externe și MV), s-a modificat fișierul **ad.h** pentru a include în structura de date **Symbol** două câmpuri: **extFnPtr** și **instrIdx**. Aceste câmpuri se iau în considerare doar pentru simboluri de felul SK_FN. Dacă **extFnPtr** este diferit de NULL, atunci el păstrează un pointer la codul funcției externe respective. Dacă **extFnPtr** este NULL, atunci înseamnă că avem de-a face cu o funcție cu codul în MV, iar în acest caz **instrIdx** va indica indexul primei instrucțiuni a funcției din vectorul de instrucțiuni. Tot în **ad.h** și **ad.c** au fost adăugate două funcții noi, **addExtFn** și **addFnParam**, pentru a se simplifica adăugarea funcțiilor externe în TS.

Execuția codului MV

Implementarea execuției codului MV se află în funcția **run**. **IP** (Instruction Pointer) este un pointer la instrucțiunea curentă din vectorul de instrucțiuni. Într-o buclă infinită se execută pe rând fiecare instrucțiune, până la apariția instrucțiunii HALT, iar atunci se revine din funcția **run**. Pentru a se vedea mai bine execuția, la fiecare instrucțiune se afișează pe ecran o linie de forma:

008/6 LESS.i // 1<2 -> 1

La început este indexul instrucțiunii în vectorul de instrucțiuni (8), iar apoi numărul de valori din stivă (6). Urmează numele instrucțiunii (LESS.i) și posibilele sale argumente (LESS.i nu are argumente). În comentariu se afișează date relevante despre execuția instrucțiunii, în acest caz valorile din stivă care s-au comparat (1 și 2) și rezultatul comparării (1).

Unele instrucțiuni cum este FPLOAD, deoarece acționează la nivel de celule de stivă (**Val**) și nu știu ce tip de valoare există atunci în stivă, afișează valoarea curentă conform mai multor posibilități de tip. De exemplu, o afișare de forma "*// i:2, f:9.88131e-324*", înseamnă că dacă este vorba de un **int** (i), atunci valoarea este 2, iar dacă este vorba de un **double** (f), atunci valoarea este 9.88131e-324.

Testarea MV

Pentru a se testa MV, înainte de **run** se apelează funcția **genTestProgram**. Această funcție generează cod MV pentru un program simplu, descris în comentariile la **genTestProgram**. Rezultatul execuției acestui program de test trebuie să fie similar cu ceea ce se află în fișierul "rezultat-executie.txt".

Temă de laborator

Pentru a fi notați la acest laborator, va trebui să creați o funcție similară cu **genTestProgram**, care să genereze codul de mai jos:

```
f(2.0);
void f(double n){
    double i;
    for(i=0.0;i<n;i=i+0.5){
        put_d(i);
        }
    }
```

Față de **genTestProgram**, în loc de **int** se va folosi **double**, incrementul va fi de 0.5, iar pentru afișare se va folosi funcția **put_d**. Funcția **put_d** va trebui implementată după modelul **put_i**, iar apoi inițializarea pentru ea inclusă în **mvInit**.

Va fi necesar să fie adăugate câteva instrucțiuni noi în MV, de exemplu LESS.f, după modelul lui LESS.i. Acestea se vor adăuga în lista de opcode-uri din **mv.h**, iar apoi vor fi implementate în funcția **run**.

Atenție: deoarece în activitatea următoare (generarea de cod), se vor mai adăuga unele instrucțiuni, iar din această cauză fișierele **mv.h** și **mv.c** vor avea adăugiri, este bine ca instrucțiunile MV pe care le adăugați să fie puse la sfârșitul instrucțiunilor existente, pentru a fi ușor de migrat în versiunile noi de **mv.h** și **mv.c**.