Limbaje formale si tehnici de compilare

Laborator 7

Masina Virtuala (MV) AtomC este bazata pe stiva. Toate operatiile si toate transferurile de memorie au loc prin intermediul stivei. Stiva in sine este implementata cu ajutorul unui vector de dimensiune constanta in felul urmator:

```
#define STACK_SIZE          (32*1024)
char      stack[STACK_SIZE];
char      *SP;              // Stack Pointer
char      *stackAfter;      // first byte after stack; used for stack limit tests
```

Nu exista o dimensiune predefinita pentru o celula a stivei, fiecare tip de date ocupand exact atatia octeti de cati are nevoie, fara aliniere. SP pointeaza intotdeauna la primul octet liber. Operatiile standard sunt „pushi, popi, pusha, popa, …". Sufixele indica tipul: a=adresa, c=caracter, d=double, i=int. Exemplu de implementare (pentru limbaje care nu au pointeri a se vedea Anexa 1 pentru discutie la implementarea fara pointeri):

```
void      pushd(double d)
{
    if(SP+sizeof(double)>stackAfter)err("out of stack");
    *(double*)SP=d;
    SP+=sizeof(double);
}

double      popd()
{
    SP-=sizeof(double);
    if(SP<stack)err("not enough stack bytes for popd");
    return *(double*)SP;
}

void      pusha(void *a)
{
    if(SP+sizeof(void*)>stackAfter)err("out of stack");
    *(void**)SP=a;
    SP+=sizeof(void*);
}

void      *popa()
{
    SP-=sizeof(void*);
    if(SP<stack)err("not enough stack bytes for popa");
    return *(void**)SP;
}
```

O instructiune este compusa dintr-un „opcode" si maxim doua argumente. Instructiunile formeaza o lista dublu inlantuita. Aceasta structura de date prezinta avantajele ca se pot insera/sterge noi instructiuni fara ca adresele celor existente sa se modifice si deci nu sunt afectate instructiunile de tip „CALL" sau „JMP" care au ca argument adresa unor alte instructiuni. O posibila implementare in care „instructions" este inceputul listei:

```
enum{O_ADD_C,O_ADD_D,O_ADD_I,…} // all opcodes; each one starts with O_
typedef struct _Instr{
    int     opcode;                 // O_*
    union{
        long int      i;            // int, char
```

```
        double      d;
        void        *addr;
        }args[2];
    struct _Instr   *last,*next;            // links to last, next instructions
}Instr;
Instr   *instructions,*lastInstruction;     // double linked list
```

Pentru operarea cu aceasta structura de date se folosesc functii de forma:

```
Instr   *createInstr(int opcode)
{
    Instr   *i;
    SAFEALLOC(i,Instr)
    i->opcode=opcode;
    return i;
}

void    insertInstrAfter(Instr *after,Instr *i)
{
    i->next=after->next;
    i->last=after;
    after->next=i;
    if(i->next==NULL)lastInstruction=i;
}

Instr   *addInstr(int opcode)
{
    Instr   *i=createInstr(opcode);
    i->next=NULL;
    i->last=lastInstruction;
    if(lastInstruction){
        lastInstruction->next=i;
        }else{
        instructions=i;
        }
    lastInstruction=i;
    return i;
}

Instr   *addInstrAfter(Instr *after,int opcode)
{
    Instr   *i=createInstr(opcode);
    insertInstrAfter(after,i);
    return i;
}
```

La acestea se mai adauga:

- `Instr *addInstrA(int opcode,void *addr)` – adauga o instructiune setandu-i si arg[0].addr
- `Instr *addInstrI(int opcode,long int val)` – adauga o instructiune setandu-i si arg[0].i
- `Instr *addInstrII(int opcode,long int val1,long int val2)` – adauga o instructiune setandu-i si arg[0].i, arg[1].i
- `void deleteInstructionsAfter(Instr *start)` – sterge toate instructiunile de dupa instructiunea „start”

Variabilele globale se afla intr-o zone speciala de memorie, implementata printr-un vector fix „globals”:

```
#define GLOBAL_SIZE     (32*1024)
char    globals[GLOBAL_SIZE];
int     nGlobals;
```

```
void	*allocGlobal(int size)
{
    void  *p=globals+nGlobals;
    if(nGlobals+size>GLOBAL_SIZE)err("insufficient globals space");
    nGlobals+=size;
    return p;
}
```

In pagina „MV” sunt prezentate toate instructiunile masinii virtuale. Felul in care se folosesc acestea va fi detaliat in laboratorul dedicat generarii codului. In acest laborator vom testa MV pe un cod scris direct. Un exemplu de cod care implementeaza „v=3;do{put_i(v);v=v-1;}while(v);” este:

```
void	mvTest()
{
    Instr	*L1;
    int		*v=allocGlobal(sizeof(long int));
    addInstrA(O_PUSHCT_A,v);
    addInstrI(O_PUSHCT_I,3);
    addInstrI(O_STORE,sizeof(long int));
    L1=addInstrA(O_PUSHCT_A,v);
    addInstrI(O_LOAD,sizeof(long int));
    addInstrA(O_CALLEXT,requireSymbol(&symbols,"put_i")->addr);
    addInstrA(O_PUSHCT_A,v);
    addInstrA(O_PUSHCT_A,v);
    addInstrI(O_LOAD,sizeof(long int));
    addInstrI(O_PUSHCT_I,1);
    addInstr(O_SUB_I);
    addInstrI(O_STORE,sizeof(long int));
    addInstrA(O_PUSHCT_A,v);
    addInstrI(O_LOAD,sizeof(long int));
    addInstrA(O_JT_I,L1);
    addInstr(O_HALT);
}
```

Functia „requireSymbol” este o varianta a „findSymbol” cu singura diferenta ca simbolul cerut trebuie neaparat sa existe in tabela de simboluri (TS), altfel se genereaza o eroare. Functiile predefinite gen „put_i” sunt implementate in C si ele se insereaza manual in TS, chiar de la inceputul compilarii programului. Aceste functii trebuie sa fie de forma „void put_i()”, adica nu au argumente si nu returneaza nimic. Ele vor putea accesa stiva prin intermediul functiilor „push*” si „pop*” sau chiar direct, manipuland SP. Ele trebuie sa descarce de pe stiva toti parametrii de apel si sa depuna valoarea returnata, daca este cazul. O implementare pentru „put_i” este:

```
void put_i()
{
    printf("#%ld\n",popi());
}
```

Pentru adaugarea in TS a acestor functii predefinite se adauga in structura „Symbol” campurile:

```
union{
    void	*addr;		// vm: the memory address for global symbols
    int		offset;		// vm: the stack offset for local symbols
    };
```

si se modifica functia „addExtFunc” din laboratorul 6 astfel:

```
Symbol	*addExtFunc(const char *name,Type type,void *addr)
{
    Symbol	*s=addSymbol(&symbols,name,CLS_EXTFUNC);
    s->type=type;
```

```
    s->addr=addr;
    initSymbols(&s->args);
    return s;
}
```

folosind aceasta functie, adaugarea in TS a unei functii predefinite se face:

```
Symbol  *s,*a;

s=addExtFunc("put_i",createType(TB_VOID,-1),put_i);
a=addSymbol(&s->args,"i",CLS_VAR);
a->type=createType(TB_INT,-1);
```

Codul generat se va apela cu instructiunea „run(instructions)”. „run” este functia principala a MV si are rolul de a executa toate instructiunile pana la intalnirea instructiunii „HALT”. Exemple de implementare a instructiunilor MV in functia „run”:

```
void run(Instr *IP)
{
    long int    iVal1,iVal2;
    double      dVal1,dVal2;
    char        *aVal1;
    char        *FP=0,*oldSP;
    SP=stack;
    stackAfter=stack+STACK_SIZE;
    while(1){
        printf("%p/%d\t",IP,SP-stack);
        switch(IP->opcode){
            case O_CALL:
                aVal1=IP->args[0].addr;
                printf("CALL\t%p\n",aVal1);
                pusha(IP->next);
                IP=(Instr*)aVal1;
                break;
            case O_CALLEXT:
                printf("CALLEXT\t%p\n",IP->args[0].addr);
                (*(void(*)())IP->args[0].addr)();
                IP=IP->next;
                break;
            case O_CAST_I_D:
                iVal1=popi();
                dVal1=(double)iVal1;
                printf("CAST_I_D\t(%ld -> %g)\n",iVal1,dVal1);
                pushd(dVal1);
                IP=IP->next;
                break;
            case O_DROP:
                iVal1=IP->args[0].i;
                printf("DROP\t%ld\n",iVal1);
                if(SP-iVal1<stack)err("not enough stack bytes");
                SP-=iVal1;
                IP=IP->next;
                break;
            case O_ENTER:
                iVal1=IP->args[0].i;
                printf("ENTER\t%ld\n",iVal1);
                pusha(FP);
                FP=SP;
                SP+=iVal1;
```

```
            IP=IP->next;
            break;
        case O_EQ_D:
            dVal1=popd();
            dVal2=popd();
            printf("EQ_D\t(%g==%g -> %ld)\n",dVal2,dVal1,dVal2==dVal1);
            pushi(dVal2==dVal1);
            IP=IP->next;
            break;
        case O_HALT:
            printf("HALT\n");
            return;
        case O_INSERT:
            iVal1=IP->args[0].i;      // iDst
            iVal2=IP->args[1].i;      // nBytes
            printf("INSERT\t%ld,%ld\n",iVal1,iVal2);
            if(SP+iVal2>stackAfter)err("out of stack");
            memmove(SP-iVal1+iVal2,SP-iVal1,iVal1); //make room
            memmove(SP-iVal1,SP+iVal2,iVal2);      //dup
            SP+=iVal2;
            IP=IP->next;
            break;
        case O_JT_I:
            iVal1=popi();
            printf("JT\t%p\t(%ld)\n",IP->args[0].addr,iVal1);
            IP=iVal1?IP->args[0].addr:IP->next;
            break;
        case O_LOAD:
            iVal1=IP->args[0].i;
            aVal1=popa();
            printf("LOAD\t%ld\t(%p)\n",iVal1,aVal1);
            if(SP+iVal1>stackAfter)err("out of stack");
            memcpy(SP,aVal1,iVal1);
            SP+=iVal1;
            IP=IP->next;
            break;
        case O_OFFSET:
            iVal1=popi();
            aVal1=popa();
            printf("OFFSET\t(%p+%ld -> %p)\n",aVal1,iVal1,aVal1+iVal1);
            pusha(aVal1+iVal1);
            IP=IP->next;
            break;
        case O_PUSHFPADDR:
            iVal1=IP->args[0].i;
            printf("PUSHFPADDR\t%ld\t(%p)\n",iVal1,FP+iVal1);
            pusha(FP+iVal1);
            IP=IP->next;
            break;
        case O_PUSHCT_A:
            aVal1=IP->args[0].addr;
            printf("PUSHCT_A\t%p\n",aVal1);
            pusha(aVal1);
            IP=IP->next;
            break;
        case O_RET:
            iVal1=IP->args[0].i;      // sizeArgs
```

```
                iVal2=IP->args[1].i;      // sizeof(retType)
                printf("RET\t%ld,%ld\n",iVal1,iVal2);
                oldSP=SP;
                SP=FP;
                FP=popa();
                IP=popa();
                if(SP-iVal1<stack)err("not enough stack bytes");
                SP-=iVal1;
                memmove(SP,oldSP-iVal2,iVal2);
                SP+=iVal2;
                break;
            case O_STORE:
                iVal1=IP->args[0].i;
                if(SP-(sizeof(void*)+iVal1)<stack)err("not enough stack bytes for SET");
                aVal1=*(void**)(SP-((sizeof(void*)+iVal1)));
                printf("STORE\t%ld\t(%p)\n",iVal1,aVal1);
                memcpy(aVal1,SP-iVal1,iVal1);
                SP-=sizeof(void*)+iVal1;
                IP=IP->next;
                break;
            case O_SUB_D:
                dVal1=popd();
                dVal2=popd();
                printf("SUB_D\t(%g-%g -> %g)\n",dVal2,dVal1,dVal2-dVal1);
                pushd(dVal2-dVal1);
                IP=IP->next;
                break;
            default:
                err("invalid opcode: %d",IP->opcode);
            }
        }
    }
```

La fiecare instructiune s-a prevazut afisarea executiei ei. Aceasta usureaza urmarirea executarii programului si depanarea acestuia.

Aplicatia 7.1: *Sa se implementeze toate functiile auxiliare necesare implementarii MV si toate functiile predefinite in AtomC (tabelul acestora este dat in laboratorul 6)*

Tema: sa se implementeze toate instructiunile MV AtomC si sa se testeze cu un cod produs manual.

Anexa 1: Implementarea MV in limbaje de programare care nu au pointeri (cu exemplificare Java)

Stiva poate pune unele probleme de implementare, deoarece este nevoie ca datele din ea sa fie accesate ca si cand ele sunt de mai multe tipuri (ex: popi() descarca 4 octeti intr-un int, popd() descarca 8 octeti intr-un double, alte instructiuni descarca/depun un numar „n” de octeti). Aceasta problema se poate rezolva in mai multe feluri:

- se declara stiva ca vector de tip „byte” si apoi: functiile „push” vor trebui sa impara valoarea in octetii constituenti si sa-i depuna in stiva; functiile „pop” vor trebui sa ia din stiva octetii constituenti si sa-i combine intr-o valoare de tipul cerut
- se declara stiva de tipul „java.nio.ByteBuffer” si se folosesc metode de genul „putDouble” si „getDouble” pentru serializarea/deserializarea valorii in acest buffer

Instructiunile vor putea fi implementate cu ajutorul unei clase „Instr”. Toate instructiunile care folosesc adresa unei instructiuni (CALL, JT, JF, JMP) vor avea posibilitatea sa accepte un argument de tipul clasei Instr (argumentele lui Instr trebuie printre valorile de genul „int” sau „double” sa accepte si valori de tipul Instr).

Functiile predefinite gen „put_i” pot fi implementate cu ajutorul unei clase abstracte „abstract class ExtFunc{abstract public void run();}” care va fi subclasata pentru fiecare functie predefinita. Instructiunea „CALLEXT” va trebui sa poata accepta un parametru de tip „ExtFunc”.

Toate adresele care provin de la valori globale (PUSHCT_A) sau de la valori de pe stiva (PUSHFPADDR) se pot modela ca indecsi intr-un vector si **trebuie sa faca parte din acelasi spatiu de adrese**. Din acest motiv vectorul „globals” si vectorul „stack” trebuie sa fie de fapt acelasi vector (un vector care sa cumuleze globals cu stack). Aceasta se poate realiza folosind acest vector in faza de generare de cod pentru a se aloca spatiu pentru variabilele globale. Dupa variabilele locale (tinand cont ca la executia MV dimensiunea acestora nu se modifica), la executia MV spatiul ramas se va folosi pentru stiva.