

# Proiect de diplomă

## Utilizarea continuărilor în compilarea programelor Scheme

autor: Zoran Constantinescu  
coordonator: prof. Irina Athanasiu

Iunie 1997

# Cuprins

# Capitolul 1

## Introducere

Lucrarea face parte dintr-un proiect mai mare, al cărui subiect este studierea posibilităților de realizare a unui compilator al limbajului Scheme. Codul generat de compilator a fost ales a fi byte-code pentru mașina virtuală Java.

Scheme este un limbaj de programare orientat funcțional, apropiat de limbajele de programare funcționale "moderne". El este un limbaj înrudit cu Lisp, însă în comparație cu acesta este mult mai simplu. Pe lângă aceasta are meritul susținerii prin funcții predefinite a unor tehnici avansate de programare, cum ar fi: întreruperea și continuarea proceselor de calcul, evaluarea leneșă a expresiilor - "lazy evaluation". Scheme implementează mecanismul de gestiune automată a memoriei. Programatorul nu se mai ocupă de alocarea și eliberarea de memorie pentru obiectele create în timpul execuției programului. Alocarea trebuie făcută explicit, dar este treaba mediului de execuție unde se va alocă obiectul, programatorul neavând nici un control asupra acestei decizii. Eliberarea memoriei nu mai trebuie făcută de loc, o procedură de colectare a memoriei disponibile (garbage collector) va colecta toate obiectele ce nu mai pot fi referite prin program.

Codul rezultat în urma compilării a fost ales a fi byte-code pentru mașina virtuală Java (Java Virtual Machine - JVM). Motivul alegerii este portabilitatea oferită de Java la nivel de fișier cu conținut executabil, precum și faptul că și mașina virtuală Java implementează mecanismul de garbage-collection. Astfel rularea unui program scris pentru această mașină virtuală este posibilă pe orice sistem de operare pe care poate rula un emulator al unei astfel de mașini virtuale. Codul generat de compilator, împreună cu biblioteca de clase ce implementează primitivele de bază **Scheme** va putea

fi executat pe orice implementare a unei mașini virtuale Java.

Implementările de limbaj **Scheme** trebuie să suporte apelurile recursive. Aceasta înseamnă că limbajul trebuie să poată efectua calcule iterative în spațiu de memorie constant, chiar dacă acestea sînt descrise sintactic în mod recursiv. În acest fel, calculele iterative pot fi descrise doar prin apeluri normale de proceduri, fără a folosi construcții speciale, specifice limbajelor de programare clasice. Dificultatea în implementarea procedurilor recursive constă în faptul că, atunci cînd este apelată o astfel de procedură, este nevoie să păstrăm operațiile care mai trebuiesc efectuate după întoarcerea din procedura recursivă. Această informație, care este folosită pentru controlul viitor al calculului, se numește informație de control. O metodă pentru implementarea recursivității este să facem explicită informația de control. Procedurile recursive sînt scrise în așa fel încît ele nu întorc niciodată valori direct, în schimb însă pasează aceste valori unui argument care este o procedură. Aceste argumente speciale sînt numite *continuări*.

Această tehnică este numită transmitere de continuări CPS ("Continuation Passing Style"). Un alt avantaj al utilizării continuărilor este posibilitatea scrierii de proceduri care pot întoarce ca rezultat mai multe valori, prin pasarea lor unei continuări care are mai multe argumenta. Expresiile **Scheme** sînt transformate în acest limbaj cu transmitere de continuări.

Scopul proiectului curent este proiectarea și implementarea unui generator de "byte-code" pentru mașina virtuală Java, pornind de la forma intermediară CPS a programelor **Scheme**, simplificate în prealabil.

## Capitolul 2

# Generalități despre compilarea limbajului Scheme

La prima vedere, deoarece sintaxa limbajul Scheme este în formă prefixat, cu paranteze, implementarea unui compilator pentru mașina virtuală Java de tip stivă ar fi foarte simplă. Scheme-ul însă are o mulțime de facilități care nu pot fi implementate printr-o simplă traducere în byte-code pentru mașina virtuală.

Una din problemele implementării limbajului **Scheme** este necesitatea eliminării totale a recursivității. Acest lucru înseamnă că o procedură **Scheme** recursivă va putea folosi un spațiu de memorie constant pentru efectuarea unui calcul. În limbajele clasice, ca de exemplu **C**, **Pascal**, în cazul apelului unei proceduri recursive se salvează pe stivă parametrii actuali, precum și adresa de îtoarcere în subrutina apelantă, urmînd ca refacerea stivei să se facă doar după terminarea apelului. În acest fel se poate ajunge destul de repede la o depășire de stivă. Soluția în cazul limbajului **Scheme** este transformarea programului într-o formă intermediară folosind ca limbaj intermediar limbajul cu transmitere de continuări CPS. O altă soluție pentru eliminarea recursivității ar putea fi ca procedura apelată (recursivă) să refacă parțial stiva, prin ștergerea din stivă a parametrilor actuali de apel a procedurii; în acest caz trebuie totuși păstrată adresa de revenire din funcție în procedura apelantă. În cazul implementării noastre a fost aleasă prima variantă, folosirea limbajului intermediar CPS.

Un alt motiv pentru această alegere este legat tot de limbajul **Scheme**. Acesta permite creare și implicit folosirea de către programator a continuărilor explicite. Cu ajutorul funcției `call-with-current-continuation` (`call/cc`), procesul de calcul curent poate

fi întrerupt și salvat, putînd fi reluat apoi mai tîrziu. Cu ajutorul acestui mecanism, în limbajul Scheme pot fi implementate: mecanismul de tratare a excepțiilor, utilizarea thread-urilor, folosirea back-tracking-ului. Printr-o simplă traducere a limbajului, acest mecanism nu ar putea fi implementat. În schimb însa, prin folosirea limbajului intermediar cu transmitere de continuări, implementarea continuărilor devine un lucru cert.

## 2.1 Arhitectura compilatorului

Compilatorul va realiza traducerea programului sursa Scheme în program executabil Java prin intermediul mai multor pași:

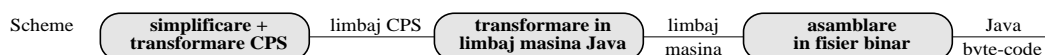


Figura 2-1: Arhitectura compilatorului.

- transformarea limbajului Scheme într-un limbaj Scheme simplificat, în care sînt folosite doar comenzile de bază ale Scheme; toate constantele au tipurile specificate; toate variabilele distincte au nume diferite. Această transformare are rolul de a simplifica programul pentru transformarea CPS.
- transformarea limbajului simplificat Scheme într limbajul cu transmitere de continuări - CPS (Continuation Passing Style). Acest limbaj va fi descris în secțiunea următoare.
- transformarea programului din limbajul CPS obținut, în cod ”de asamblare” pentru mașina virtuală Java. Am ales varianta transformării în limbaj de asamblare și nu direct în fișier binar .class deoarece în acesta din urmă există niște structuri de date destul de complexe, și ar fi îngreunat scrierea acestui modul. Am lăsat astfel în grija asamblorului crearea acestor structuri de date.
- transformarea codului rezultat în byte-code de Java, respectiv crearea de fișiere .class Java, cu ajutorul unui asamblor de byte-code Java. Pe lângă crearea structurilor de date necesare, acest modul face și conversia mnemonicelor instrucțiunilor în byte-code-urile corespunzătoare.

## 2.2 Simplificarea limbajului Scheme

Primul pas constă în aducerea programului **Scheme** la o formă simplificată în care să apară cât mai puține comenzi Scheme. Acest lucru este realizat prin transformări simple la nivelul programului sursă Scheme, rezultatul fiind tot un program Scheme, dar ceva mai complex, însă care folosește mai puține comenzi. De exemplu secvența:

```
(cond
  (test_1  consec_1)
  (test_2  consec_2)
  ...
  (test_n  consec_n)
  (else   altern. ))
```

va deveni în urma simplificării:

```
(if test_1
    consec_1
    ...
    (if test_n
        consec_n
        alternat) ... )
```

În mod asemănător se pot transforma și alte construcții, limbajul **Scheme** simplificat rezultat avînd doar cîteva comenzi elementare: `define`, `if`, `quote`, `lambda`, `set!`, `evaluare`.

## 2.3 Limbajul intermediar CPS

Transformarea în forma CPS are drept scop aducerea expresiilor într-o formă denumită *tail-form*. În această formă, funcțiile nu mai întorc nici un rezultat, în schimb însă vor transmite acest rezultat unuia din parametri. Acest parametru este de fapt o continuare. Fiecare funcție va prelua un număr de parametri, va efectua calcule, iar rezultatul îl va pasa mai departe unei continuări. Fiecare funcție va deveni de fapt o astfel de continuare, care va primi o valoare și va transmite mai departe noul rezultat. Avantajul față de metoda clasică de apel, în care se depune pe stivă adresa de întoarcere din funcția apelată, este că nu mai avem nevoie de această adresă. Pur și simplu transmitem o continuare funcției apelate, care la rîndul ei va transmite rezultatul ei

continuării. În felul acesta nu mai apare problema posibilității terminării stivei în urma unor apeluri recursive repetate.

Să considerăm ca exemplu următoarea expresie **Scheme**:

```
(define list-append
  (lambda (list_a list_b)
    (if (null? list_b)
        list_a
        (cons (car list_b) (list-append list_a (cdr list_b))))))
```

Această funcție primește ca argumente două liste, *list\_a* și *list\_b* și întoarce ca rezultat o nouă listă obținută prin concatenarea celor două.

$(\text{list-append } '(1\ 2\ 3) \ '(x\ y\ z)) \implies (1\ 2\ 3\ x\ y\ z)$

Forma echivalentă CPS este următoarea:

```
(define list-append-cps
  (lambda (list_a list_b k)
    (if (null? list_b)
        (k list_a)
        (list-append-cps list_a (cdr list_b)
                          (lambda (v)
                            (k (cons (car list_b) v)))))))
```

Se observă că în forma CPS, funcția primește un nou argument *k*, care reprezintă de fapt o continuare. În acest exemplu, funcțiile `null?`, `car` și `cdr` sînt considerate expresii simple, ceea ce înseamnă că evaluarea lor pot genera apeluri recursive. La apelul unei astfel de funcții, pur și simplu se întoarce rezultatul, fără a apela alte funcții. Pe ramura "else" a formei CPS se observă că este creată o nouă continuare, cu ajutorul construcției `lambda`.

Pentru apelul noii forme CPS, vom avea nevoie de o nouă construcție în care să creem continuarea inițială:

```
(define list-append
  (lambda (list_a list_b)
    (list-append-cps list_a list_b (lambda (v) v))))
```

## Capitolul 3

# Mașina Virtuală Java

Mașina Virtuală Java (JVM) este o mașină virtuală implementată prin emulare software pe o mașină reală. Codul executabil pentru mașina virtuală se află stocat în fișiere de tip '.class', fiecare astfel de fișier conținând codul pentru o singură clasă Java.

Datorită formatului compact și eficient al byte-code-ului există implementări de interpretoare pentru mașina virtuală Java pentru diverse platforme și sisteme de operare. Există posibilitatea realizării unor compilatoare în timp real, care să transforme byte-code-ul metodelor în instrucțiuni cod mașină pentru un anumit tip de procesor pe măsura execuției metodelor respective.

Recent au apărut chip-uri care execută direct setul de instrucțiuni al mașinii virtuale, eliminând complet necesitatea unui interpretor sau a unui compilator JIT (Just In Time compiler). Un astfel de procesor este *picoJava* de la Sun Microsystems, cu o arhitectură simplă de tip RISC, optimizat pentru viteză și care funcționează 100 % conform specificațiilor JVM.

### 3.1 Arhitectura JVM

Mașina virtuală este organizată ca o mașină stivă. Un emulator al unei astfel de mașini citește instrucțiunile din zona de cod, pe care le execută. În urma executării instrucțiunilor, se operează asupra stivei. Tipurile de date folosite de mașina virtuală sînt tipurile de bază ale limbajului Java. Aceste tipuri sînt: byte, short, int, long, float, double, char, object și returnAddress (v. tabel ??).

Aproape toate verificările pentru tipurile de baza (prima parte a tabelului) sînt

tip	lungime	descriere
byte	1-byte	întreg cu semn în complement față de 2
short	2-byte	întreg cu semn în complement față de 2
int	4-byte	întreg cu semn în complement față de 2
long	8-byte	întreg cu semn în complement față de 2
float	4-byte	real IEEE 754 simplă precizie
double	8-byte	real IEEE 754 dublă precizie
char	2-byte	caracter Unicode
object	4-byte	referință la un obiect Java
returnAddress	4-byte	adresă de întoarcere pt. instrucțiuni jsr,ret

Tabela 3.1: Tipuri de baza Java

efectuate în timpul compilării. Instrucțiunile byte-code care operează cu tipurile de bază indică pe lângă operație și tipurile operanzilor. De exemplu: instrucțiunea *imul* va înmulți două numere de tip `integer`. Nu există instrucțiuni specifice pentru tipul `boolean`, folosindu-se în loc instrucțiunile tipului `integer`. Pentru array-uri de tip `boolean` se folosesc cele de tip `byte`.

Mașina virtuală Java are patru registre, care sînt folosiți doar pentru controlul execuției și al operării stivei. Ei nu sînt folosiți pentru transferul parametrilor sau a valorilor de revenire, pentru aceasta utilizîndu-se stiva. Cei patru regiștrii sînt descriși în tabelul ???. Intrările stivei precum și dimensiunea regiștrilor sînt de 32 biți. Operanzii de dimensiune mai mare (8 octeți - `long`, `double`) ocupă două cuvinte consecutive. Stiva este folosită pentru transmiterea parametrilor actuali metodelor și pentru valorile întoarse de acestea.

Obiectele (instanțele de clase) create de programele Java sînt alocate în heap-ul sistemului. Aceasta este zona care este supusă colectării de memorie.

Fiecare metodă a unei clase Java folosește un număr fix de variabile locale, adresate ca offset față de registrul `vars`. Toate variabilele locale au dimensiunea de 32 biți. Întregii și realii dublă precizie se consideră că ocupă două variabile locale consecutive.

Stiva operanzilor este de 32 biți. Aceasta este folosită pentru transmiterea parametrilor metodelor și primirea rezultatelor acestora. Pe lângă aceasta, stiva este folosită pentru furnizarea parametrilor operațiilor elementare și pentru salvarea rezultatelor operațiilor.

O instrucțiune pentru JVM este formată dintr-un octet, opcode specificînd operația, urmat de zero sau mai mulți operanzi, reprezentînd parametrii sau datele utilizate de

registru	descriere
<b>pc</b>	adresa următoarei instrucțiuni
<b>vars</b>	începutul zonei de variabile locale
<b>optop</b>	vîrful stivei de operanzi
<b>frame</b>	începutul înregistrării de activare

Tabela 3.2: Regiștrii JVM

operație.

## 3.2 Descrierea fișierelor .class

Fiecare fișier .class conține codul binar compilat al unei clase Java sau a unei interfețe Java. Acest fișier conține toate datele despre o clasă sau o interfață. Fișierul este privit ca un stream de octeți, elementele de 16, respectiv 32 biți fiind considerate ca grupuri de 2, respectiv 4 octeți așezați în format big-endian (cu octetul semnificativ primul). În interiorul unui astfel de fișier se află byte-code-uri, adică implementări pentru fiecare metodă a clasei, scrise cu setul de instrucțiuni a mașinii virtuale.

### 3.2.1 Formatul fișierului

Formatul unui fișier .class este următorul:

```

ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count - 1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attribute_count];
}

```

Semnificațiile câmpurilor fișierului sînt următoarele:

- **magic** - semnătura fișierului .class; trebuie să aibă valoarea 0xCAFEBAFE.
- **minor\_version, major\_version** - specifică versiunea compilatorului care a generat fișierul .class respectiv. Valoarea curentă pentru **major\_version** este 45, iar pentru **minor\_version** este 3.
- **constant\_pool\_count** - indică numărul de intrări în **constant\_pool**, o tabelă cu toate constantele folosite în fișierul .class respectiv.
- **constant\_pool** - tabela cu toate constantele folosite în fișier. Acestea sînt valori ale constantelor de tip String, nume de alte clase, denumiri de metode, constante numerice și alte astfel de constante folosite în structura clasei sau de byte-code. Prima intrare din tabelă, **constant\_pool[0]** este întotdeauna nefolosită de către compilator, putînd fi folosită de către implementarea mașinii virtuale pentru orice scop. Fiecare intrare din tabelă este o intrare de lungime variabilă, formatul fiecăreia fiind determinat pe baza primului octet, denumit "tag byte".
- **access\_flag** - conține o constantă pentru accesul la clasă, conform tabelului ??:

Denumire flag	Valoare	Folosire	Descriere
ACC_PUBLIC	0x0001	CMV	Vizibil pentru oricine
ACC_PRIVATE	0x0002	.MV	Vizibil doar pentru clasa definită
ACC_PROTECTED	0x0004	.MV	Vizibil subclaselor
ACC_STATIC	0x0008	.MV	Variabila sau metoda este statică
ACC_FINAL	0x0010	CMV	Nu permite alte subclase, suprascriere sau atribuire după inițializare
ACC_SYNCHRONIZED	0x0020	.M.	Se folosește mecanismul de monitoare (lock) la acces
ACC_VOLATILE	0x0040	..V	Nu poate fi optimizat folosind cache-ul
ACC_TRANSIENT	0x0080	..V	Nu poate fi scris sau citit de un manager de obiecte persistente
ACC_NATIVE	0x0100	.M.	Implementat în alt limbaj, nu Java
ACC_INTERFACE	0x0200	C..	Este o clasă interfață
ACC_ABSTRACT	0x0400	CM.	Nu este specificat corpul clasei/metodei

Tabela 3.3: Modificatori de acces

unde C=clasă, M=metodă, V=variabilă.

- **this\_class** - este un index în **constant\_pool**, iar intrarea din tabel trebuie să fie de tipul **CONSTANT\_class**, reprezentînd denumirea clasei.
- **super\_class** - asemănător lui **this\_class**, dar este denumirea clasei imediat superioare; dacă indexul este zero, atunci clasa este implicit `java.lang.Object`, neavînd clasă superioară.
- **interfaces\_count** - reprezintă numărul de interfețe pe care clasa respectivă le implementează.
- **interfaces** - este o tabelă, fiecare element al ei fiind un index în **constant\_pool**, care trebuie să fie de tipul **CONSTANT\_class**, reprezentînd denumirea interfeței implementate.
- **fields\_count** - reprezintă numărul de cîmpuri implementate de explicit de clasă, fără a se lua în calcul cele moștenite.
- **fields** - fiecare valoare din această tabelă reprezintă o descriere a unui cîmp din clasă: denumirea, tipul și eventualele atribute.
- **methods\_count** - reprezintă numărul de metode, statice și dinamice, definite de această clasă.
- **methods** - fiecare valoare reprezintă descrierea unei metode definite explicit de această clasă: denumirea, semnatura (tipul parametrilor și a valorii întoarse), eventualele atribute. Metodele moștenite de clasă nu se află în această tabelă.
- **attributes\_count** - indică numărul de atribute suplimentare ale clasei.
- **attributes** - fiecare intrare în tabelă reprezintă un atribut al clasei. Un astfel de atribut este "Source", care specifică numele fișierului sursă din care a fost compilată clasa.

### 3.2.2 Structuri de date

O semnătură este un șir de caractere (string) care descrie tipul unei metode, al unui cîmp sau al elementelor unui tablou (array). O semnătură este reprezentată ca un șir de octeți conform regulilor de mai jos:

- **signatura unui câmp** descrie tipul unui argument al unei funcții sau al unei variabile:

```
<sign_cimp> ::= <tip_cimp>;
```

- **signatura unei metode** descrie tipul argumentelor funcției și tipul rezultatului:

```
<sign_metoda> ::= (<sign_argumente>)<sign_rezultat>;
```

unde

```
<sign_argumente> ::= <sign_argument>*;
<sign_argument> ::= <tip_cimp>;
<sign_rezultat> ::= <tip_cimp> | V;

<tip_cimp> ::= <tip_baza> | <tip_obiect> | <tip_tablou>;
<tip_baza> ::= B | C | D | F | I | J | S | Z;
<tip_obiect> ::= L<nume_clasa>;
<tip_tablou> ::= [<dimens_optionala>]<tip_cimp>;
<dimens_optionala> ::= [0-9]*;
```

iar semnificația tipurilor de bază este următoarea:

Caracter	Tip Java	Descriere
B	byte	octet cu semn
C	char	caracter
D	double	real IEEE dublă precizie
F	float	real IEEE simplă precizie
I	int	întreg
J	long	întreg dublă precizie
S	short	short
Z	boolean	adevărat sau fals
Lnume_clasa		un obiect de tipul clasă specificat
tip_cimp		tablou

Tabela 3.4: Tipuri de bază

### 3.3 Setul de instrucțiuni

Lungimea instrucțiunilor mașinii virtuale Java este variabilă. Există instrucțiuni fără nici un parametru și instrucțiuni cu unul, doi sau mai mulți parametri. Formatul unei instrucțiuni este următorul:

`<cod-operatie> operand_1 operand_2 ...`

lungimea codului operației fiind de un octet (de aici vine și denumirea de *byte-code*).

Există următoarele grupe de instrucțiuni:

- instrucțiuni pentru salvarea constantelor pe stivă: **bipush**, **sipush**, **ldc1**, **ldc2**, **ldcw**, **aconst\_null**, **iconst\_m1**, **iconst\_<n>**, **lconst\_<1>**, **fconst\_<d>**, **dconst\_<d>**;
- instrucțiuni pentru încărcarea conținutului variabilelor pe stivă: **iload**, **iload\_<n>**, **lload**, **lload\_<n>**, **fload**, **fload\_<n>**, **dload**, **dload\_<n>**, **aload**, **aload\_<n>**;
- instrucțiuni pentru încărcarea valorilor de pe stivă în variabile: **istore**, **iload\_<n>**, **lstore**, **lload\_<n>**, **fstore**, **fload\_<n>**, **dstore**, **dload\_<n>**, **astore**, **aload\_<n>**, **iinc**;
- instrucțiuni de lucru cu tablouri de obiecte (*array*): **newarray**, **anewarray**, **multianewarray**, **arraylength**, **iaload**, **laload**, **faload**, **daload**, **aaload**, **baload**, **caload**, **saload**, **istore**, **lstore**, **fstore**, **dstore**, **astore**, **bstore**, **cstore**, **sstore**;
- instrucțiuni pentru lucrul cu stiva: **nop**, **pop**, **pop2**, **dup**, **dup2**, **dup\_x1**, **dup2\_x1**, **dup\_x2**, **dup2\_x2**, **swap**;
- instrucțiuni aritmetice: **iadd**, **ladd**, **fadd**, **dadd**, **isub**, **lsub**, **fsub**, **dsub**, **imul**, **lmul**, **fmul**, **dmul**, **idiv**, **ldiv**, **fdiv**, **ddiv**, **irem**, **lrem**, **frem**, **drem**, **ineg**, **lneg**, **fneg**, **dneg**;
- instrucțiuni logice: **ishl**, **ishr**, **iushr**, **lshl**, **lshr**, **lushr**, **iand**, **ior**, **ixor**, **land**, **lor**, **lxor**;
- instrucțiuni de conversie de tipuri: **i2l**, **i2f**, **i2d**, **l2i**, **l2f**, **l2d**, **f2i**, **f2l**, **f2d**, **d2i**, **d2l**, **d2f**, **int2byte**, **int2char**, **int2short**;
- instrucțiuni pentru întoarcerea din funcții: **ireturn**, **lreturn**, **freturn**, **dreturn**, **areturn**, **return**, **breakpoint**;
- instrucțiuni pentru transferul controlului: **ifeq**, **ifnull**, **ift**, **ifle**, **ifne**, **ifnonnull**, **ifgt**, **ifge**, **if\_icmpeq**, **if\_icmpne**, **if\_icmplt**, **if\_icmple**, **if\_icmpgt**, **if\_icmpge**, **lcmp**, **fcmpl**, **fcmpg**, **dcmpl**, **dcmpg**, **if\_acmpeq**, **if\_acmpne**, **goto**, **jsr**, **ret**, **goto\_w**, **jsr\_w**, **ret\_w**;
- instrucțiuni de salt bazate pe tabele de valori: **tableswitch**, **lookupswitch**;
- instrucțiuni de lucru cu câmpurile obiectelor: **putfield**, **putstatic**, **getfield**, **getstatic**;
- instrucțiuni de apel a metodelor obiectelor: **invokevirtual**, **invokenonvirtual**, **invokestatic**, **invokenonstatic**;
- instrucțiuni pentru tratarea excepțiilor: **athrow**;
- instrucțiuni diverse pentru lucrul cu obiecte: **new**, **checkcast**, **instanceof**;
- monitoare: **monitorenter**, **monitorexit**;

## Capitolul 4

# Generarea claselor Java pe baza limbajului intermediar CPS

### 4.1 Structura programului compilat

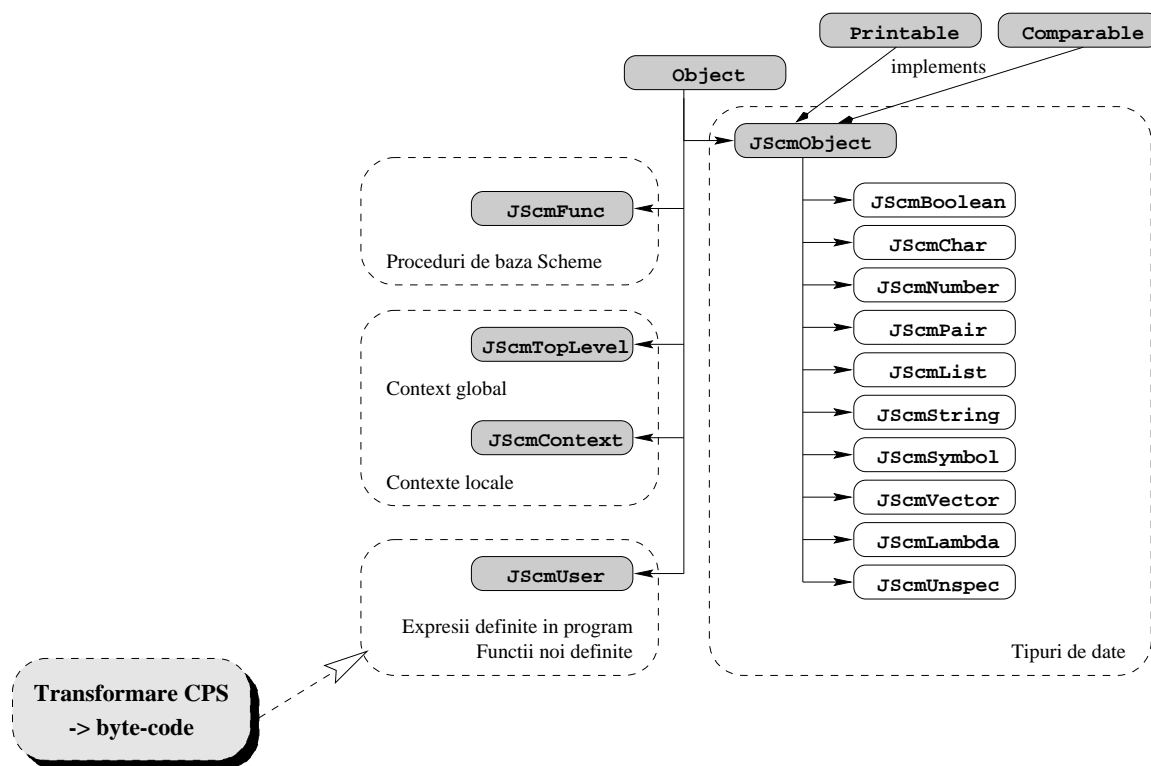


Figura 4-1: Ierarhia completă de clase.

Programul adus la forma intermediară în limbajul **CPS** este transformat, așa cum se va vedea mai jos, într-o clasă **JScmUser**. Această clasă conține toate expresiile și

definițiile de funcții care apar în programul **CPS**. Clasa rezultată va crea, în funcție de program, un număr de obiecte. Unele dintre aceste obiecte sînt create static, la pornirea programului, altele sînt create dinamic, pe parcursul execuției programului, în funcție de instrucțiunile executate. Aceste obiecte sînt instanțe ale claselor care apar în figura ??.

Există clase echivalente fiecărui tip de dată Scheme. Aceste clase implementează pe lîngă structurile de date necesare memorării datelor, și metodele pentru accesul la aceste date și modificarea lor. Clasele pentru tipurile de date sînt:

**JScmBoolean, JScmChar, JScmNumber, JScmPair**  
**JScmList, JScmString, JScmSymbol, JScmVector**

Obiectele reprezentînd instanțe ale acestor clase sînt pe de o parte constantele care apar în program, iar pe de altă parte sînt valorile variabilelor create pe parcursul execuției programului.

O altă categorie de clase folosite sînt cele pentru păstrarea contextelor. Acestea sînt:

**JScmContext, JScmTopLevel**

Clasa **JScmTopLevel** va păstra toate legăturile variabilelor la nivel top-level din cadrul programului. Va exista o singură instanță a acestei clase. Clasa **JScmContext** va fi folosită pentru pătrarea contextelor locale în cadrul apelurilor de funcții. Vor fi create în mod dinamic instanțe ale acestei clase, pentru fiecare evaluare a unei expresii top-level.

Clasa **JScmFunc** va conține toate metodele necesare apelului de proceduri **Scheme** predefinite. Va exista o singură instanță a clasei. Clasa va avea toate metodele statice, și nu va avea nici un cîmp.

Toate aceste clase sînt implementate în limbajul Java, singura excepție fiind clasa **JScmUser**, care va fi creată de către compilator, pe baza programului **Scheme** transformat în prealabil în limbajul intermediar CPS.

## 4.2 Tipuri de date

Fiind un limbaj "dynamically typed", în Scheme verificarea tipurilor se face la execuție și nu la compilare. Limbajul intermediar **CPS** va fi și el la fel, cu singura excepție că toate constantele au tipurile explicit specificate. Din acest motiv, în programul

compilat vom folosi pentru toate obiectele **Scheme** referințe la obiecte Java (clase Java).

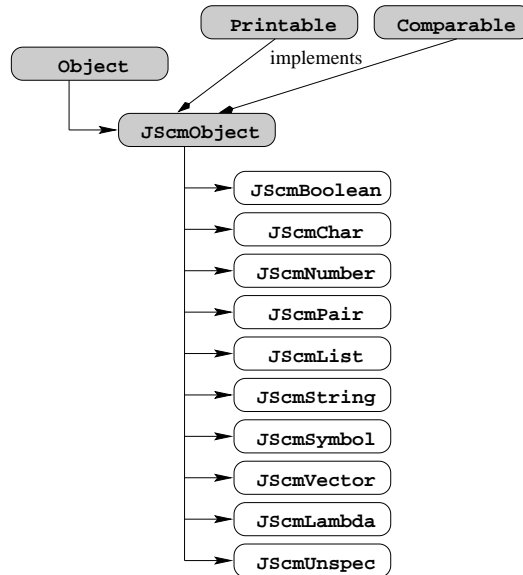


Figura 4-2: Ierarhia de clase pentru tipurile de date.

Fiecărui tip Scheme îi asociem câte o clasă Java, descendentă din clasa `JScmObject` (v. figura ??).

Fiecare din aceste clase va implementa două tipuri de metode:

- metode corespunzătoare funcțiilor Scheme specifice tipului respectiv (ex. `string-length` pentru tipul `string`; `car`, `cdr` pentru tipurile `pair`, `list`; etc.);
- metode corepunzătoare funcțiilor Scheme aplicabile tuturor tipurilor (ex. `char?`, `number?` - pentru verificarea apartenenței la un anumit tip; `eq?`, `eqv?`, `equal?` - pentru testarea egalității a două obiecte **Scheme**; etc.).

Metodele comune tuturor claselor sînt specificate în cele două clase de tip *interface*, pe care clasa `JScmObject` le implementează. Aceste două clase *interface* sînt:

```

public interface Comparable {
    boolean eqP (JScmObject obj);
    boolean eqvP (JScmObject obj);
    boolean equalP (JScmObject obj);
}
public interface Printable {
    void print ();
}
  
```

Tipurile Scheme și clasele corespunzătoare Java sînt specificate în tabelul ??.

Fiecare clasă definește un anumit tip de dată Scheme. Metodele implementate de fiecare clasă sînt specifice tipului definit. De exemplu, metodele `eq(...)`, `eqv(...)`,

Tip <b>Scheme</b>	Tip <b>CPS</b>	Clasă definiție tip
boolean ( <b>#t</b> , <b>#f</b> )	bool	clasa <b>JScmBoolean</b>
character	chr	clasa <b>JScmChar</b>
numbers	nmb	clasa <b>JScmNumber</b>
pairs	pair	clasa <b>JScmPair</b>
list	list	clasa <b>JScmList</b>
strings	str	clasa <b>JScmString</b>
symbols	id	clasa <b>JScmSymbol</b>
vectors	vect	clasa <b>JScmVector</b>
procedures	lambda	clasa <b>JScmLambda</b>
unspecified	unspec	clasa <b>JScmUnspec</b>

Tabela 4.1: Echivalențe tipuri **Scheme** - tipuri Java

`equal(...)` vor testa egalitatea obiectului primit ca parametru cu obiectul instanțiat. Metoda `print()` va tipări valoarea obiectului în funcție de tipul acestuia, respectând notația **Scheme**. Toate metodele implementate de clasele de definiție de tipuri vor fi apelate de metode din clasa **JScmFunc**, care aduce aceste metode la forma cerută de apelurile **Scheme** (subcapitolul următor).

Metodele implementate de fiecare clasă sînt descrise mai jos:

- clasa **JScmBoolean** implementează tipul de dată boolean, cu valorile posibile `true` (**#t**) și `false` (**#f**). Metodele implementate de această clasă sînt:
  - `JScmBoolean` (boolean b);  $\longrightarrow$  constructor
  - `JScmBoolean` (JScmBoolean b);  $\longrightarrow$  constructor
  - boolean `eqP` (JScmObject obj);  $\longrightarrow$  implementează predicatul de echivalență `eq?` din **Scheme**;
  - boolean `equivP` (JScmObject obj);  $\longrightarrow$  implementează predicatul de echivalență `equiv?` din **Scheme**;
  - boolean `equalP` (JScmObject obj);  $\longrightarrow$  implementează predicatul de echivalență `equal?` din **Scheme**;
  - void `print` ();  $\longrightarrow$  metodă pentru afișarea valorii obiectului: **#t** sau **#f**.
- clasa **JScmChar** implementează tipul de dată caracter (char): litere, cifre, caractere speciale. Pentru acest tip avem metode de comparație care fac diferențiere între literele mari și mici. Specificația standard **Scheme** ([?] pag. 24) impune următoarele reguli în stabilirea ordinii caracterelor:
  - literele mari sînt în ordine: (`char<? \#A \#B`) întoarce **#t**.
  - literele mici sînt în ordine: (`char<? \#a \#b`) întoarce **#t**.
  - cifrele sînt în ordine: (`char<? \#0 \#9`) întoarce **#t**.
  - toate cifrele preced ca ordine toate literele mari sau invers

- toate cifrele preced ca ordine toate literele mici sau invers

Ordinea aleasă a fost cea dată de ordinea caracterelor în Java. Metodele implementate de această clasă sînt:

- *JScmChar* (char c); → constructor;
  - *JScmChar* (JScmChar c); → constructor;
  - boolean *eq, lt, le, gt, ge* (JScmChar c); → metode pentru stabilirea unei ordini totale a caracterelor;
  - boolean *eq-ci, lt-ci, le-ci, gt-ci, ge-ci* (JScmChar c); → metode similare cu cele de la *eq*, cu diferența că nu se face diferențiere între litere mari și litere mici;
  - boolean *alpha, num, space, upper, lower* (); → metode pentru determinarea tipului de caracter;
  - *JScmNumber char2int* (); → metodă pentru conversia unui caracter într-un obiect de tip întreg;
  - boolean *eqP, eqvP, equalP* (JScmObject obj); → implementează predicatul de echivalență **Scheme**: *eq?*, *eqv?*, *equal?*;
  - void *print* (); → metodă pentru afișarea valorii obiectului: #<caracter>, sau *#space, #tab, #newline, #linefeed, #return, #page, #backspace, #rubout* pentru caractere speciale.
- clasa **JScmNumber** implementează tipul de dată numeric. Au fost implementate doar numerele întregi. Metodele clasei sînt:
    - *JScmNumber* (int n); → constructor;
    - *JScmNumber* (JScmNumber n); → constructor;
    - boolean *zero, positive, negative* (); → testarea semnului;
    - boolean *odd, even* (); → testarea parității;
    - *JScmNumber +, \** (JScmNumber n1...nk); → operații aritmetice;
    - boolean *eq, lt, le, gt, ge* (JScmNumber n1...nk); → operatori de comparare;
    - boolean *eqP, eqvP, equalP* (JScmObject obj); → implementează predicatul de echivalență **Scheme**: *eq?*, *eqv?*, *equal?*;
    - void *print* (); → metodă pentru afișarea valorii obiectului: <număr>, de exemplu 45.
  - clasa **JScmPair** implementează tipul pereche cu punct (*pair*), care are două câmpuri denumite *car* și *cdr*. Metodele create de clasă sînt:
    - *JScmPair* (JScmObject car, JScmObject cdr); → constructor;
    - *JScmPair* (JScmPair pair); → constructor;
    - JScmObject *car, cdr* (); → întorc valoarea câmpului *car*, respectiv *cdr*;
    - JScmUnspec *set\_car, set\_cdr* (JScmObject obj); → setează valoarea câmpului *car*, respectiv a câmpului *cdr*;
    - *JScmPair cons* (JScmObject car, cdr); → metodă pentru crearea unui nou obiect de tip *pair*;

- boolean *eqP*, *eqvP*, *equalP* (JScmObject obj); → implementează predicatul de echivalență **Scheme**: *eq?*, *eqv?*, *equal?*;
  - void *print* (); → metodă pentru afișarea valorii obiectului: (*car*) . (*cdr*), de exemplu (53 . "Hello").
- clasa **JScmList** implementează tipul listă; clasa are următoarele metode:
    - *JScmList* (); →() constructor listă vidă;
    - *JScmList* (JScmObject car, JScmList cdr); → constructor;
    - *JScmList* (JScmList lst); → constructor;
    - *JScmList* (JScmObject val[]); → constructor de creare a unei liste dintr-un array;
    - int *length* (); → determină numărul elementelor listei;
    - JScmObject *car* (); → întoarce valoarea primului element al listei (cîmpului *car* a listei);
    - JScmList *cdr* (); → întoarce ca rezultat o listă din care a fost eliminat primul element (cîmpul —it *cdr* al listei);
    - boolean *eqP*, *eqvP*, *equalP* (JScmObject obj); → implementează predicatul de echivalență **Scheme**: *eq?*, *eqv?*, *equal?*;
    - void *print* (); → metodă pentru afișarea valorii obiectului: (<elem1> <elem2> ... <elemn>) de exemplu (1 3 "d" 4).
  - clasa **JScmString** implementează tipul de dată șir de caractere (string). Metodele implementate sînt următoarele:
    - *JScmString* (String str); → constructor;
    - *JScmString* (JScmString obj); → constructor;
    - int *length* (); → ne dă lungimea șirului de caractere;
    - JScmChar *ref* (int index); → metoda întoarce caracterul de pe poziția <index> din șir;
    - JScmUnspec *set* (int index, char ch); → modifică un caracter specificat din șir;
    - boolean *eqP*, *eqvP*, *equalP* (JScmObject obj); → implementează predicatul de echivalență **Scheme**: *eq?*, *eqv?*, *equal?*;
    - void *print* (); → metodă pentru afișarea valorii obiectului sub forma: "**Șir de caractere**".
  - clasa **JScmVector** implementează tipul de dată vector cu elemente care, spre deosebire de marea majoritate a limbajelor de programare, pot fi de tipuri diferite. Un vector ocupă mai puțin loc decît o listă cu aceleași elemente, iar timpul de accesare al unui element este mai redus pentru vector decît pentru listă. Metodele implementate sînt următoarele:
    - *JScmVector* (JScmObject obj[]); → constructor;
    - *JScmVector* (int n); → constructor pentru un vector cu <n> elemente de tipul *unspecified*;

- *JScmVector* (int n, JScmObject obj); → constructor vector cu ⟨n⟩ elemente, avînd toate aceleași valoare ⟨obj⟩;
  - *JScmVector* (JScmVector vect); → constructor de copiere;
  - *JScmList make\_list* (); → crează o listă din vector;
  - *JScmList length* (); → întoarce ca rezultat numărul de elemente din vector;
  - *JScmObject ref* (int index); → metoda întoarce elementul de pe poziția ⟨index⟩ din vector;
  - *JScmUnspec set* (int index, JScmObject obj); → setează elementul de pe poziția specificată la valoarea ⟨obj⟩;
  - *JScmUnspec fill* (JScmObject obj); → setează toate elementele vectorului la valoarea dată ⟨obj⟩;
  - boolean *eqP*, *eqvP*, *equalP* (JScmObject obj); → implementează predicatul de echivalență **Scheme**: *eq?*, *eqv?*, *equal?*;
  - void *print* (); → metodă pentru afișarea elementelor vectorului sub forma: #⟨3 4 7 "y"⟩.
- clasa **JScmSymbol** implementează tipul de dată simbol din **Scheme**. Metodele implementate sînt:
    - *JScmSymbol* (JScmSymbol sym); → constructor de copiere;
    - *JScmSymbol* (String str); → constructor;
    - boolean *eqP*, *eqvP*, *equalP* (JScmObject obj); → implementează predicatul de echivalență **Scheme**: *eq?*, *eqv?*, *equal?*;
    - void *print* (); → metodă pentru afișarea valorii simbolului sub forma: '⟨symbol⟩'.
  - clasa **JScmUnspec** implementează tipul de dată unspecified din **Scheme**. Metodele folosite sînt:
    - *JScmUnspec* (); → constructor;
    - boolean *eqP*, *eqvP*, *equalP* (JScmObject obj); → implementează predicatul de echivalență **Scheme**: *eq?*, *eqv?*, *equal?*;
    - void *print* (); → metodă pentru afișarea valorii obiectului sub forma: #unspec.
  - clasa **JScmLambda** implementează procedurile definite de utilizator (expresiile lambda definite). Descrierea clasei va fi explicată într-unul din subcapitolele următoare.

### 4.3 Implementarea procedurilor de bază Scheme

Procedurile de bază ale limbajului **Scheme** sînt implementate în clasa **JScmFunc**. Fiecare din metodele acestei clase implementează cîte una din procedurile de bază. Marea majoritate a metodelor sînt apeluri la metode din clase Java corespunzătoare tipurilor specificate mai sus, făcîndu-se transformarea la sintaxa cerută de **Scheme**.

Clasa `JScmFunc` nu are variabile, doar metode. Pentru fiecare apel al unei astfel de proceduri de bază **Scheme**, în *byte-code*-ul generat rezultă un apel al unei metode din această clasă.

De exemplu pentru procedura **Scheme** (`string-length <şir>`), metoda corespunzătoare din clasa **JScmFunc** este:

```
public static JScmNumber stringslength (JScmString str) {
    return new JScmNumber (str.length ());
}
```

Procedurile **Scheme** de echivalență, `eq?`, `eqv?` și `equal?` sînt implementate în mod diferit față de celelele. În funcție de tipul primului parametru, se face apelul metodei corespunzătoare `eqP`, `eqvP` sau `equalP` din clasa tipului respectiv.

O parte din procedurile **Scheme** referitoare la tipul de dată *caracter* sînt date în exemplul de mai jos:

```
class JScmFunc {
    public JScmFunc () {
    }

/*===== CHAR type functions =====*/
    /* invoke copy constructor */
    public static JScmChar new_char (JScmChar obj) {
        return new JScmChar (obj);
    }

    /* R4RS essential procedure - char? */
    public static JScmBoolean charP (JScmObject obj) {
        return new JScmBoolean (obj instanceof JScmChar);
    }

    /* R4RS essential procedure - char=? */
    public static JScmBoolean char_eqP (JScmObject obj1, JScmObject obj2) {
        return new JScmBoolean ((obj1 instanceof JScmChar) &&
                                (obj2 instanceof JScmChar) &&
                                ((JScmChar)obj1).eq ((JScmChar)obj2));
    }
    ...
    /* R4RS essential procedure - char-ci=? */
    public static JScmBoolean char_ci_eqP (JScmObject obj1, JScmObject obj2) {
        return new JScmBoolean ((obj1 instanceof JScmChar) &&
                                (obj2 instanceof JScmChar) &&
                                ((JScmChar)obj1).eq_ci ((JScmChar)obj2));
    }
    ...
    /* R4RS essential procedure - char-alphabetic? */
    public static JScmBoolean char_alphabeticP (JScmObject obj) {
```

```

        return new JScmBoolean ((obj instanceof JScmChar) &&
                                ((JScmChar)obj).alpha ());
    }
    ...
    /* R4RS essential procedure - char-upcase */
    public static JScmChar char_upcase (JScmObject chr) {
        return new JScmChar (((JScmChar)chr).charUpValue ());
    }
    ...
    /* R4RS essential procedure - char->integer */
    public static JScmNumber char2integer (JScmObject chr) {
        return new JScmNumber (((JScmChar)chr).intValue ());
    }
}

```

## 4.4 Clasa generată JScmUser

Această clasă este generată de pe baza programului adus în forma intermediară **CPS**. Această clasă va implementa în cadrul metodelor sale codul corespunzător programului. În această clasă sînt implementate deasemeni toate funcțiile noi definite de programator. Modul în care aceste funcții noi sînt implementate va fi explicat în subcapitolul următor.

Structura unei astfel de clase este următoarea:

```

class JScmUser
extends java.lang.Object
{
1   /* constants used in the program */

    Field public static JScmObject const_0
    Field public static JScmObject const_1
    ...
    /* constants for error report */
    Field public static JScmObject const_err
    Field public static JScmObject const_null

2   /* class constructor */

    Method void <init> ()
    max_stack 1
    {
        aload_0
        invokenonvirtual void java.lang.Object.<init> ()
        return
    }
}

```

```

3  /* class initializer */

Method static void init ()
max_stack 4
{
4  /* initialize the constants */

  /* constant err */
  new JScmString
  dup
  ldc "*** not yet implemented"
  invokenonvirtual void JScmString.<init> (java.lang.String)
  putstatic JScmObject JScmUser.const_err

  /* constant null */
  new JScmString
  dup
  ldc "* null"
  invokenonvirtual void JScmString.<init> (java.lang.String)
  putstatic JScmObject JScmUser.const_null

  /* constant no. 0, type NUMBER */
  new JScmNumber
  dup
  ldc 11
  invokenonvirtual void JScmNumber.<init> (int)
  putstatic JScmObject JScmUser.const_0
  ...

  return
}

5  /* main method */

Method public static void main (java.lang.String[])
max_stack 9
{
  /* initialize the class */
  invokestatic void JScmUser.init()

6  /* expressions */

  /* EXPR 1 */
  /* CONST NUMBER */
  getstatic JScmObject JScmUser.const_0
  invokevirtual void JScmObject.print()
  getstatic java.io.PrintStream java.lang.System.out
  invokevirtual void java.io.PrintStream.println()
  ...

  return

```

```

7   /* User defined function declarations */

   Object ret_addr

   /* expecting 'JScmLambda' + 'ret_addr' on the stack */
call_funct:
   astore ret_addr
   /* expecting 'JScmLambda' on the stack */
goto_funct:
   invokevirtual int JScmLambda.keyValue()
   goto goto_funct_cps

   /* expecting 'funct_key' + 'ret_addr' on the stack */
call_funct_cps:
   astore ret_addr /* save the return address */
   /* expecting 'funct_key' on the stack */
goto_funct_cps:
   lookupswitch default nowhere
   {
       0:  f_lambda_0 /* id(x) --> x function
       1:  f_lambda_1 /* user function 1 */
       2:  f_lambda_2 /* user function 2 */
       ...
   }
nowhere:
   ret ret_addr

/* identity function (continuation) */
f_lambda_0:
   ret ret_addr /* return to the saved return address */

/* user function 1 */
f_lambda_1:
   /* save actual parameters */
   ...
   goto call_funct_cps
}
}

```

(Î) Toate constantele care apar în programul **Scheme** sînt create ca variabile (cîmpuri) ale clasei `JScmUser` generate. Dacă o constantă apare de mai multe ori în cadrul programului, este creat un singur cîmp pentru clasă care este inițializat la valoarea constantei. Astfel, dacă de exemplu în program apare șirul de caractere `"Test"` de mai multe ori în diferite expresii, atunci este creat un singur cîmp în cadrul clasei: `JScmObject const_x`. Tipul cîmpului va fi `JScmString`, iar valoarea sa va fi `"Test"`:

```

...
Field public static JScmObject const_5
...
Method static void init ()
{
    ...
    /* constant no. 5, type STRING */
    new JScmString
    dup
    ldc "Test"
    invokevirtual void JScmString.<init> (java.lang.String)
    putstatic JScmObject JScmUser.const_5
    ...
}

```

Deoarece pe parcursul execuției programului, valorile acestor constante nu pot fi modificate, ele vor avea în permanență aceleași valori. Din acest motiv, precum și pentru a avea *byte-code*-ul generat mai simplu, câmpurile clasei pentru constante sînt de tipul static. *Byte-code*-ul devine mai simplu deoarece în cazul în care avem câmpuri statice, accesul la ele se face printr-o simplă instrucțiune:

```
getstatic <tip_cîmp> <nume_cîmp>
```

fără a avea nevoie de nimic pe stivă, spre deosebire de cazul câmpurilor nestatice, în care instrucțiunea de acces la câmp are aceeași formă:

```
getfield <tip_cîmp> <nume_cîmp>
```

în schimb însă, pe stivă trebuie să avem o referință la o instanță a unei astfel de clase (JScmUser).

(2) `<init>()` este metoda constructor a clasei. Singurul lui rol este de a apela constructorul clasei părinte `java.lang.Object`.

(3)(4) Metoda `init()` este metoda de inițializare a clasei, avînd rolul de a inițializa în mod corespunzător câmpurile constante ale clasei. De exemplu, pentru o constantă de tip întreg vom avea următoarea secvență de inițializare:

```

/* constant no. 2, type NUMBER */
new JScmNumber
dup
ldc 133    /* constant value */
invokevirtual void JScmNumber.<init> (int)
putstatic JScmObject JScmUser.const_2

```

(5) Metoda principală a clasei este `main(String[])`. În această metodă se află codul generat corespunzător programului. Pentru fiecare expresie **Scheme** top-level este generată o secvență de byte-code-uri echivalentă (6). Prin execuția secvenței respective (în momentul rulării programului pe mașina virtuală) se obține rezultatul evaluării expresiei.

(6) Tot în metoda `main(String[])` se află și codul generat corespunzător funcțiilor noi definite de programator. Începutul codului fiecărei funcții este etichetat, *apelul* fiecărei funcții fiind făcut pe baza acestei etichete folosind o tabelă de salt indexată.

## 4.5 Implementarea funcțiilor noi

În urma transformării **CPS**, fiecare funcție definită de utilizator va primi un parametru suplimentar, reprezentat de o continuare. Această continuare este folosită pentru a-i pasa rezultatul funcției.

Fiecare funcție compilată va fi considerată ca avînd un al doilea parametru suplimentar, pe lângă continuare. Acesta este reprezentat de un context. Acest context va cuprinde toate legările de variabile care au fost definite local, pe parcursul evaluării unei expresii, pînă în momentul apelului funcției. Acest context nu va conține nici una din definițiile top-level de variabile, acestea aflîndu-se într-un alt obiect, `JScmTopLevel`. Contextul primit de o funcție va fi folosit pentru a afla valoarea unei variabile libere folosite în cadrul funcției. O funcție care primește un astfel de context, va adăuga la acesta toate definițiile locale de variabile, efectuate în cadrul funcției. Noul context va fi folosit de celelalte funcții apelate în mod asemănător.

O clasă context va conține perechi de obiecte simbol-valoare:

`JScmSymbol` - `JScmObject`.

Definiția clasei `JScmContext` va conține două metode publice pentru căutarea, respectiv adăugarea unei noi perechi:

```
class JScmContext extends Object {
    JScmSymbol ctx_syms[]; /* the symbols from context */
    JScmObject ctx_vals[]; /* the values from context */

    public void addSymbol (JScmSymbol sym, JScmObject val) {
    public JScmObject getSymbol (JScmSymbol sym) {
}
}
```

Toate apelurile de funcții, în urma transformării **CPS**, sînt în sub forma *tail*. Asta înseamnă că ele nu mai întorc controlul în funcția apelantă (imediat superioară). Din acest motiv clasa **JScmContext** nu necesită metode pentru ștergerea unei perechi dintr-n context.

Forma *tail* permite implementarea funcțiilor recursive eliminînd folosirea ineficientă a stivei. Cu alte cuvinte apelul unei funcții din cadrul altei funcții nu se mai face cu o instrucțiune de apel de subrutina, ci cu una de salt, de tip *goto*.

Astfel, de exemplu, următorul cod recursiv, care respectă forma *tail*:

```
f:      /* stack=[arg1, arg2, ...] */
...
if <...>
    push arg1'
    push arg2'
    ...
    call f
endif
...
return /* reface stiva, stergind argumentele primite */
```

poate fi rescris, mai eficient în felul următor:

```
f:      /* stack=[arg1, arg2, ...] */
...
pop varg2
pop varg1
...
if <...>
    push arg1'
    push arg2'
    ...
    goto f
endif
...
return' /* face doar intoarcerea din apelul functiei */
```

În setul de instrucțiuni pentru mașina virtuală Java există următoarele instrucțiuni pentru transferul controlului în cadrul programului:

- `invokestatic`, `invokevirtual`, `invokenonvirtual`, `invokeinterface` pentru apelul metodelor unei clase Java;
- `jsr`, `ret` pentru apelul unor *minisubrutine* aflate în *byte-code*-ul unei aceleiași metode;

- `goto` pentru salt necondiționat doar în cadrul *byte-code*-ului unei metode.

O metodă pentru implementarea apelurilor de funcții **Scheme** definite de programator ar fi ca fiecare astfel de funcție să reprezinte o metodă distinctă într-o clasă Java. În acest fel, pentru fiecare apel de funcție trebuie generat codul corespunzător unui apel de metodă. Parametrii necesari unui apel de metodă trebuie puși înaintea apelului pe stivă, urmînd care rezultatul apelului să fie deasemenea pus pe stivă în locul parametrilor apelului. Problema apare în momentul în care încercăm să eliminăm apelurile recursive. Stiva este refăcută doar după terminarea execuției metodei din care s-a făcut apelul, în momentul execuției unei instrucțiuni de tip *return*. Să considerăm următorul exemplu **Scheme**:

```
(define f
  (lambda (x)
    (if (> x 0) (f (- x 1)) 0)))
(f 1000000)
```

Codul corespunzător scris în limbaj de asamblare ar fi:

```
class Exemplu_1 {
  ...
  Method public static int f (int)
  max_stack 3
  {
    iload_0
    ifle f_end
    iload_0
    iconst_1
    isub
    invokestatic int Exemplu_1.f (int)
    ireturn
  f_end:
    iconst_0
    ireturn
  }

  Method public static main (...)
  max_stack 3
  {
    ...
    /* apel (f 1000000) */
    ldc 1000000
```

```

        invokestatic int Exemplu_1.f (int)
        ...
    }
    ...
}

```

Funcția *f* se va apela în mod recursiv, decrementînd de fiecare dată argumentul primit. Dacă apelăm funcția cu un parametru de valoare foarte mare, atunci vom avea cu siguranță o depășire de stivă, mașina virtuală semnalînd o excepție de tipul: `java.lang.StackOverflowError`.

Soluția ar fi să aducem expresia în formă *tail*, și să implementăm funcția *f* în aceeași metodă ca și funcția apelantă. În cazul exemplului nostru simplu, expresia **Scheme** este deja în formă *tail*. Codul generat scris în limbaj de asamblare va avea forma:

```

class Exemplu_1 {
    ...
    Method public static void main (...)
    max_stack 3
    {
        ...
        /* apel (f 1000000) */
        ldc 1000000
        jsr f_start
        ...
        return

    f_start:      /* functia f */
        int n
        Object ret_addr

        /* pop the return address and the argument from the stack */
        store ret_addr
    f_goto:
        store n
        load n      /* push n onto the stack */
        iload_0     /* push constant 0 onto the stack */
        ifle f_end
        load n
        iconst_1
        isub        /* decrement n */
        goto f_goto
    f_end:
        iconst_0   /* return 0 */
        ret ret_addr
    }
}

```

```
}  
...  
}
```

Se observă că în acest caz nu mai avem depășire de stivă, apelul recursiv fiind înlocuit cu salturi în cadrul metodei.

Deci soluția pentru implementarea recursivității este folosirea combinată a instrucțiunilor `jsr`, `ret`, `goto`. Acest lucru ne obligă ca fiecare funcție Scheme (lambda definiție) să reprezinte o *minisubrutina*, apelabilă fie printr-o instrucțiune de tip `jsr`, în cazul primului apel, fie printr-o instrucțiune de tip `goto`, în cazul apelurilor recursive din cadrul funcției. Toate aceste definiții de funcții vor trebui implementate în cadrul aceleiași metode. Apelul unei astfel de funcții va fi deci posibil doar din cadrul aceleiași metode. Deci un *top-level Scheme* va fi implementat ca o clasă cu o metoda în care se definesc toate funcțiile.

Pe linga funcțiile pe care le definește utilizatorul avem și funcțiile predefinite Scheme ('R4RS essential procedure'). Acestea pot fi considerate 'primop'-uri în transformarea CPS, ceea ce înseamnă că apelul lor nu va folosi nici o funcție definită de utilizator, deci nu apare problema recursivității. În acest caz însă, acestea nu pot fi redefinite în așa fel încât să poată fi eliminată recursivitatea. Pentru aceasta și aceste funcții predefinite trebuie considerate ca 'apply'-uri în transformarea CPS. În acest caz, și funcțiile redefinite sunt considerate ca funcții definite de utilizator, beneficiind astfel de eliminarea recursivității. Funcțiile predefinite Scheme sunt implementate într-o clasă separată Java numită `JScmFunc`, care apelează metode specifice fiecărui tip, aflate în clasele de definire a tipurilor (`JScmChar`, ...). Implementarea unui program Scheme se face folosind o clasă Java cu o singură metodă. Aceasta va conține (printre altele) secvențe de cod etichetate, corespunzătoare fiecărei expresii lambda. Apelul, respectiv saltul, unei astfel de expresii lambda se face folosind o instrucțiune byte-code 'lookup-switch', saltul la următoarea instrucțiune făcându-se pe baza unei key de selecție, cheia aflată pe stivă:

```
switch_jsr:  
  astore_x  
  getstatic TopLevel.local_func  
  getfield Lambda.lookup_key
```

```

new Lambda
dup
invokenonstatic Lambda.<init> ()
swap
switch_goto:
lookupswitch {
    k0:  lambda_0
    k1:  lambda_1
    k2:  lambda_2
    \ldots
}

```

unde Lambda va fi o clasa Java care va contine cheia de salt al functiei, iar dupa caz si variabilele libere ale functiei:

```

class Lambda extend java.lang.Object {
    public int lookup_key;
    Object free_vars[];

    public Lambda () {
        lookup_key = 0;
    }

    public void PutFreeVar (int i, Object x) {
        free_vars[i] = x;
    }

    public Object GetFreeVar (int i) {
        return free_vars[i];
    }
}

```

Prima eticheta `switch_jsr` va fi folosita pentru apelul unei lambda expresii din cadrul unor expresii top-level, iar cea de-a doua `switch_goto` pentru apelul unei astfel de expresii din cadrul unei expresii in forma tail-form.

## Capitolul 5

# Asamblarea în byte-code

Asamblorul va realiza transformarea unui fișier sursă scris în limbaj de asamblare Java într-un fișier `.class` valid, conținând *byte-code*-ul corespunzător unei clase Java. Acest fișier rezultat va trebui să poată fi executat de către mașina virtuală Java, deci va trebui să respecte formatul cerut de acesta. Asamblorul realizează crearea automată a tabelii de constante (*constant-pool*) din fișierul `.class`, calculează automat offset-urile pentru salturile din cadrul metodelor.

### 5.1 Sintaxa asamblorului

Sintaxa limbajului de asamblare este asemănătoare ieșirii generate de către utilitarul **javap** din JDK. **javap** este un dezasamblor de fișiere `.class`, care afișează conținutul unei clase Java (câmpuri, metode, byte-code) într-un format mai ușor de înțeles pentru programator.

Un fișier cu codul de asamblare pentru o clasă Java trebuie să aibă următorul format:

```
[abstract] [final] [public] [interface] class nume_clasa
[extends nume_super_clasa]
[implements nume_interfata_1 [nume_interfata_2 [ ... ] ] ]
{
  [declaratii_cimpuri]
  [declaratii_metode]
  [sourcefile nume_fisier_sursa]
}
```

- **nume\_clasa** este numele valid al clasei avînd următorul format:  
part1[.part2[.part3[...]]]

part1, part2, ... sînt identificatori reprezentînd diferite părți ale numelui clasei;  
de exemplu `java.lang.Object`;

- **nume\_super\_clasa** este numele clasei din care clasa curentă este descendentă;  
în cazul în care lipsește se consideră implicit ca fiind clasa `java.lang.Object`;
- **nume\_interfata\_i** este numele unei clase care reprezintă o interfață și pe care  
clasa curentă o implementează;
- **nume\_fisier\_sursa** este numele fișierului sursă din care este generat fișierul `.class`;
- declarațiile de cîmpuri din cadrul clasei trebuie să aibă următorul format:

```
field [specificator_acces] [static] [final] [transient]
      [volatile] nume_cimp [= valoare_constanta]
```

- **specificator\_acces** specifică tipul accesului la cîmpul respectiv, și poate  
avea următoarele valori: *private*, *private protected*, *protected*, *public*;
- **nume\_cîmp** este un identificator și reprezintă numele cîmpului;
- **valoare\_constantă** este o constantă avînd tipul cîmpului respectiv (unul  
din tipurile de bază Java), și care reprezintă valoarea constantă a cîmpului;

- declarațiile de metode din cadrul clasei trebuie să aibă următorul format:

```
method [specificator_acces] [static] [abstract]
      [final] [native] [synchronized]
tip_rezultat nume_metoda ( [arg1 [, arg2 [, ...] ] ] )
[throws nume_exceptie_1 [nume_exceptie_2 [...] ] ]
max_stack valoare1
[max_locals valoare2]
{
    [cod_metoda]
    [tabela_exceptiilor]
    [tabela_numerotare_linii]
    [tabela_variabibile_locale]
}
```

- **tip\_rezultat** reprezintă tipul rezultatului întors de metodă și poate fi `void` sau un tip valid: nume de clasă, tablou (*array*) sau unul din tipurile de bază (*byte, char, double, float, int, long, short, boolean*);
- **nume\_metodă** este un identificator reprezentînd numele metodei;
- **arg1, arg2, ...** sînt tipuri valide reprezentînd tipurile parametrilor metodei; pot fi urmate opțional de numele parametrilor;
- **nume\_excepție\_i** reprezintă numele unei clase care este o excepție pe care această metodă o poate genera;
- **valoare1** reprezintă dimensiunea maximă a stivei de care metoda poate avea nevoie;
- **valoare2** reprezintă numărul maxim de variabile locale de care metoda respectivă are nevoie; dacă nu este specificat, atunci asamblorul determină automat numărul de variabile locale pe baza numărului definițiilor lor din cadrul metodei;
- **cod\_metodă** reprezintă codul efectiv al metodei și constă în linii de forma a) sau b); prima formă reprezintă apelul unei operații de tip *byte-code*, iar cea de-a doua reprezintă o definiție de variabilă locală în cadrul metodei
  - a) [etichetă] operație
  - b) [etichetă] tip\_variabilă nume\_variabilă\_locală

**etichetă** este numele unei etichete folosită pentru salturi în cadrul metodei.
- **tabelă\_excepții** are următoarea formă și reprezintă delimitarea în cadrul metodei a subprocedurilor de tratare a excepțiilor:

```

exceptions
{
    start_pc1 end_pc1 subr_pc1 catch_tip1
    start_pc2 end_pc2 subr_pc2 catch_tip2
    ...
}

```

- \* **start\_pc** - etichetă reprezentînd începutul zonei din metodă pentru care se face tratarea excepțiilor;

- \* **end\_pc** - etichetă reprezentînd sfîrșitul zonei din metodă pentru care se face tratarea excepțiilor;
  - \* **subr\_pc** - etichetă reprezentînd adresa subrutinei de tratare a excepțiilor pentru zona specificată;
  - \* **catch\_tip** - reprezintă numele unei clase de tip excepție sau valoarea 0;
- **tabelă\_numerotare\_linii** specifică numerotarea liniilor sursă echivalente fișierului compilat și are următoarea formă; această tabelă este folosită pentru depanarea unui fișier .class.

```
linenumbertable
{
    start_pc1 numar_linie_1
    start_pc2 numar_linie_2
    ...
}
```

- \* **start\_pc** - etichetă reprezentînd adresa liniei numerotate;
  - \* **numă\_linie** - reprezintă numărul efectiv în fișierul sursă al codului compilat începînd de la adresa codului specificată;
- **tabelă\_variabile\_locale** specifică zonele de definiție a variabilelor locale

```
localvariabletable
{
    start_pc1 end_pc1 tip1 local_var1 slot_num1
    start_pc2 end_pc2 tip2 local_var2 slot_num2
    ...
}
```

- \* **start\_pc**, **end\_pc** - etichete care delimitează zona în care variabila locală **local\_var** va avea o valoare de tipul **tip**, avînd slotul **slot\_num** în înregistrarea de activare a metodei curente.

## 5.2 Facilități oferite de asamblor

Asamblorul oferă o mulțime de facilități pentru ușurarea programării direct în *byte-code* pentru mașina virtuală Java.

- generarea automată a fișierului binar `.class`; asamblorul generează automat toate structurile de date folosite într-un astfel de fișier pe baza informațiilor din fișirul scris în limbaj de asamblare: tabela de constante (*constant-pool*), tabela de câmpuri a clasei, tabela de metode, tabela de excepții pentru fiecare metodă, calculează signaturile;
- generarea automată a *constant-pool*-ului: unele instrucțiuni din limbajul mașinii virtuale Java primesc ca parametru un offset în *constant-pool*; limbajul de asamblare nu permite lucrul direct cu offset-uri în *constant-pool*-ul, în schimb însă permite lucrul direct cu valorile din *constant-pool*. Astfel, de exemplu, instrucțiunea: `new` primește ca parametru un offset în *constant-pool* către numele unei clase, instrucțiunea avînd astfel o lungime de trei octeți: primul este codul operației (187 pentru `new`), restul reprezentînd offsetul pe 16 biți:

```
new <35>
```

În limbajul de asamblare se folosește direct numele clasei:

```
new java.lang.String,
```

urmînd ca asamblorul, în momentul generării fișierului binar să creeze intrarea corespunzătoare în tabela de constante și să genereze codul instrucțiunii folosind offset-ul rezultat;

- limbajul de asamblare permite programatorului să folosească variabile locale în cadrul unei metode, în locul folosirii directe a unui index în *frame*-ul Java curent. În felul acesta este mult mai simplu pentru programator, acesta nefiind nevoit să țină minte indecși pentru variabile, ci doar numele lor, programul devenind și mult mai simplu de urmărit. Astfel, în loc să scriem `iload 5`, adică încarcă valoarea din variabila a 5-a pe stivă, putem folosi următoarea secvență:

```
int variabilă_contor
...
iload variabilă_contor
```

- există în limbajul de asamblare există două noi instrucțiuni: `load` și `store`, care pot fi folosite avînd ca parametru o variabilă locală definită ca mai sus. Cele două instrucțiuni sînt echivalente cu `iload`, `lload`, `fload`, `dload`, `aload`, respectiv `istore`, `lstore`, `fstore`, `dstore`, `astore`, în funcție de tipul variabilei lo-

cale. Asamblorul, cunoscînd tipul variabilei folosite într-o astfel de instrucțiune, generează automat instrucțiunea corespunzătoare tipului variabilei. De exemplu instrucțiunea

```
load variabilă_contor
```

din exemplul precedent va genera în *byte-code* instrucțiunea `iload`. În felul acesta programatorul nu trebuie să mai scrie în program instrucțiuni diferite pentru tipuri diferite, lăsînd această sarcină în seama asamblorului.

- limbajul de asamblare permite folosirea etichetelor pentru salturi în cadrul unei metode. În felul acesta, programatorul nu trebuie să mai calculeze manual un offset pentru o instrucțiune de salt, fiind sarcina asamblorului. De exemplu, în locul următorului cod (salt după instrucțiunea `iload` care ocupă 2 octeți):

```
goto +2
iload 20
...
```

se folosește următoarea secvență:

```
goto eticheta_4
iload 20
eticheta_4:
....
```

Mai mult chiar, în cazul în care offset-ul saltului depășește dimensiunea memorabilă pe un octet (-128 ... +127), ar trebui folosită instrucțiunea `goto_w`; asamblorul își dă singur seama de acest lucru, și generează automat *byte-code*-ul pentru instrucțiunea `goto_w`, chiar dacă a fost folosit `goto`.

- etichetele pot fi folosite și pentru cele două instrucțiuni de salt bazate pe tabele: `tableswitch` și `lookupswitch`, ca în exemplul:

```
ldc 7
lookupswitch default eticheta_0
{
  1 : eticheta_1
  5 : eticheta_2
  9 : eticheta_3
}
```

```

    eticheta_0:
    ldc 86
    tableswitch 85 to 87 default eticheta_10
    {
        eticheta_11
        eticheta_12
        eticheta_13
    }
    ...

```

### 5.3 Exemplu

Un exemplu de program scris în limbaj de asamblare pentru mașina virtuală Java este următorul:

```

class Hello2
extends java.lang.Object
{
    Method public static void main (java.lang.String[])
    max_stack 2
    max_locals 2
    {
        getstatic java.io.PrintStream java.lang.System.out
        ldc "Hello World!"
        invokevirtual void java.io.PrintStream.println(java.lang.String)
        ldc 250
        invokevirtual void java.io.PrintStream.println(int)
        return
    }

    Method void <init> ()
    max_stack 2
    max_locals 1
    {
        aload_0
        invokenonvirtual void java.lang.Object.<init>()
        return
    }
}

```

Programul afișează pe ecranu șirul de caractere ”**Hello World!**” urmat de numărul întreg **250**. Fișierul .class generat de asamblor și dezasamblat cu ajutorul utilitarului **javap** este redat mai jos.

```

class Hello2 extends java.lang.Object {
    public static void main(java.lang.String []);
        /* Stack=2, Locals=2, Args_size=1 */
    Hello2();
        /* Stack=2, Locals=1, Args_size=1 */

Method void main(java.lang.String [])
    0 getstatic #8 <Field java.lang.System.out Ljava/io/PrintStream;>
    3 ldc #14 <String "Hello World!">
    5 invokevirtual #16 <Method java.io.PrintStream.println(Ljava/lang/String;)V>
    8 ldc #22 <Integer 250>
    10 invokevirtual #23 <Method java.io.PrintStream.println(I)V>
    13 return

Method Hello2()
    0 aload_0
    1 invokenonvirtual #24 <Method java.lang.Object.<init>()V>
    4 return
}

```

## Capitolul 6

# Concluzii

Lucrarea de față tratează aspectele teoretice și problemele de implementare apărute în proiectarea unui compilator pentru limbajul **Scheme**. Cea mai mare parte a transformării din limbajul intermediar CPS în *byte-code* pentru mașina virtuală Java este implementată.

Limbajul **Scheme** implementat este un subset al standardului definit în raportul [?]. Nu sînt implementate decît numerele întregi, acestea fiind echivalente cu numerele întregi din Java, pe 32 biți. Lipssește partea de aritmetică cu numere raționale și cea cu numere reale, lucrarea fiind orientată mai mult spre aspecte privind posibilitatea compilării limbajului **Scheme**, și mai puțin pe realizarea unei implementări complete a limbajului.

Dezvoltări ulterioare pot fi făcute în mai multe direcții. Printre acestea ar fi posibilitate definirii și folosirii direct din limbajul **Scheme**, utilizînd o sintaxă extinsă a limbajului, a unor mecanisme de programare orientate obiect, respectiv definirea de obiecte și instanțe ale lor. În acest sens este posibilă modificarea compilatorului, astfel încît să poată genera clase Java noi, folosind o sintaxă corespunzătoare.

O altă posibilitate de dezvoltare ar fi posibilitatea instanțierii de obiecte și apelului de metode ale claselor Java deja existente într-un program —bf Scheme. În mod asemănător și pentru aceasta este necesară extinderea sintaxei limbajului **Scheme** cu elemente care să permită aceste operații.

Anexa A

Exemple programe

Anexa B

Listing programe

# Bibliografie

- [1] Friedman, Daniel P.; Wand, Mitchell; Haynes, Christopher T. *Essentials of Programming Languages*. MIT Press, 1992.
- [2] Appel, Andrew W. *Compiling with Continuations*. Cambridge University Press, 1992.
- [3] Clinger, William and Rees, Jonathan (Editors). *Revised<sup>4</sup> Report on the Algorithmic Language Scheme*. Available by anonymous ftp from `altdorf.ai.mit.edu`. 1991.
- [4] Dybvig, R. K. *Three Implementation Models for Scheme*. University of North Carolina Computer Science Technical Report 87-011 [Ph.D. Dissertation], April 1987.
- [5] *The Java Virtual Machine Specification*. Release 1.0, Sun Microsystems, August 1995.
- [6] *The Java<sup>TM</sup> Language Specification*. Release 1.0, Sun Microsystems, October 1996.