

Tema 3

Generare de cod

CPL

Cuprins

1	Obiective	1
2	Descriere	1
3	Cerințe	6
3.1	Definiții de clase	6
3.2	Valori predefinite	7
3.3	Literali	7
3.4	Definiții și apeluri de metode	8
3.5	Construcția <code>let</code>	8
3.6	Construcția <code>new</code>	8
3.7	Verificarea egalității obiectelor	9
3.8	Construcția <code>case</code>	9
3.9	Metode și rutine predefinite	9
4	Testare	10
5	Precizări	11
6	Referințe	11

1 Obiective

Obiectivele temei sunt următoarele:

- **Generarea de cod** MIPS pornind de la programe Cool.
- Utilizarea opțională a unui **instrument** dedicat pentru generarea de cod, *StringTemplate*.

2 Descriere

Tema abordează ultima etapă a construcției compilatorului pentru limbajul Cool, generarea de cod. Aceasta pornește de la reprezentarea intermediară elaborată în etapa anterioară, de analiză semantică, în forma unui arbore de sintaxă abstractă, adnotat cu simboluri și tipuri. Rezultatul acestei etape îl

constituie un **program MIPS** echivalent cu programului Cool inițial. Acesta va putea fi rulat în simulatoarele *QtSpim* sau *spim*.

Tema de față va primi ca parametri în linia de comandă numele unuia sau mai multor fișiere conținând programe Cool, și va tipări la *standard output* **programul MIPS** dorit. Programele de test vor fi **corecte lexical, sintactic și semantic!**

Punctul de plecare îl constituie **sistemul de execuție** Cool, sub forma unui fișier *exception handler*, `trap.handler.nogc`, încărcat în simulator. Printre altele, acesta conține mecanisme de **alocare dinamică** a memoriei, precum și implementarea **metodelor predefinite**, din clasele `Object`, `IO` și `String`, pe care le veți invoca din codul generat de voi.

Mai jos, funcționalitatea este exemplificată pornind de la cel mai simplu program Cool:

```
1 class Main {  
2     main() : Object { 0 };  
3 };
```

Un posibil program MIPS echivalent este ilustrat începând cu pagina următoare. În continuare, este prezentată **structura de ansamblu** a acestuia, urmând ca detaliile să fie furnizate în secțiunea 3.

- Ca în orice program MIPS, se observă cele două **zone de memorie**, data (linia 1), respectiv text (linia 186).
- Zona data debutează cu o serie de definiții **globale** (liniile 3–11), utilizând directiva `.globl`, care sunt vizibile și în alte programe MIPS încărcate concomitent. În particular, este vorba despre **sistemul de execuție** menționat mai sus, care utilizează anumite etichete generate de voi.
- Cele trei etichete, `_int_tag`, `_string_tag` și `_bool_tag` (liniile 12–17) precizează **etichetele** celor trei clase predefinite, fiind utilizate de sistemul de execuție pentru implementarea testului de **egalitate** între valori predefinite.
- În liniile 18–66, sunt definiți toți **literalii șir de caractere** din program, incluzând **numele claselor**, și eventual ale **fișierelor**, în anumite cazuri. Reprezentarea acestor literalii este discutată ulterior.
- În liniile 67–91, sunt definiți toți **literalii întregi** din program. Reprezentarea acestora este discutată ulterior.
- În liniile 92–101, sunt definiți cei doi **literalii booleeni**. Reprezentarea acestora este discutată ulterior.
- Liniile 102–108 ilustrează **tabelul numelor de clase** (`class_nameTab`), necesar pentru afișarea anumitor mesaje de eroare de către sistemul de execuție. **Atenție!** Tabelul este indexat de **eticheta** clasei, reprezentată printr-un număr întreg.

```

1      .data
2      .align 2
3      .globl class_nameTab
4      .globl Int_protObj
5      .globl String_protObj
6      .globl bool_const0
7      .globl bool_const1
8      .globl Main_protObj
9      .globl _int_tag
10     .globl _string_tag
11     .globl _bool_tag
12     _int_tag:
13         .word 2
14     _string_tag:
15         .word 3
16     _bool_tag:
17         .word 4
18     str_const0:
19         .word 3
20         .word 5
21         .word String_dispTab
22         .word int_const0
23         .asciiiz ""
24         .align 2
25     str_const1:
26         .word 3
27         .word 6
28         .word String_dispTab
29         .word int_const1
30         .asciiiz "Object"
31         .align 2
32     str_const2:
33         .word 3
34         .word 5
35         .word String_dispTab
36         .word int_const2
37         .asciiiz "IO"
38         .align 2
39     str_const3:
40         .word 3
41         .word 5
42         .word String_dispTab
43         .word int_const3
44         .asciiiz "Int"
45         .align 2
46     str_const4:
47         .word 3
48         .word 6
49         .word String_dispTab
50         .word int_const1
51         .asciiiz "String"
52         .align 2
53     str_const5:
54         .word 3
55         .word 6
56         .word String_dispTab
57         .word int_const4
58         .asciiiz "Bool"
59         .align 2
60     str_const6:
61         .word 3
62         .word 6
63         .word String_dispTab
64         .word int_const4
65         .asciiiz "Main"
66         .align 2
67     int_const0:
68         .word 2
69         .word 4
70         .word Int_dispTab
71         .word 0
72     int_const1:
73         .word 2
74         .word 4
75         .word Int_dispTab
76         .word 6
77     int_const2:
78         .word 2
79         .word 4
80         .word Int_dispTab
81         .word 2
82     int_const3:
83         .word 2
84         .word 4
85         .word Int_dispTab
86         .word 3
87     int_const4:
88         .word 2
89         .word 4
90         .word Int_dispTab
91         .word 4
92     bool_const0:
93         .word 4
94         .word 4
95         .word Bool_dispTab
96         .word 0
97     bool_const1:
98         .word 4
99         .word 4
100        .word Bool_dispTab
101        .word 1
102     class_nameTab:
103         .word str_const1
104         .word str_const2
105         .word str_const3
106         .word str_const4
107         .word str_const5
108         .word str_const6

```

```

109 class_objTab:
110     .word Object_protObj
111     .word Object_init
112     .word IO_protObj
113     .word IO_init
114     .word Int_protObj
115     .word Int_init
116     .word String_protObj
117     .word String_init
118     .word Bool_protObj
119     .word Bool_init
120     .word Main_protObj
121     .word Main_init
122 Object_protObj:
123     .word 0
124     .word 3
125     .word Object_dispTab
126 IO_protObj:
127     .word 1
128     .word 3
129     .word IO_dispTab
130 Int_protObj:
131     .word 2
132     .word 4
133     .word Int_dispTab
134     .word 0
135 String_protObj:
136     .word 3
137     .word 5
138     .word String_dispTab
139     .word int_const0
140     .asciiiz ""
141     .align 2
142 Bool_protObj:
143     .word 4
144     .word 4
145     .word Bool_dispTab
146     .word 0
147 Main_protObj:
148     .word 5
149     .word 3
150     .word Main_dispTab
151 Object_dispTab:
152     .word Object.abort
153     .word Object.type_name
154     .word Object.copy
155 IO_dispTab:
156     .word Object.abort
157     .word Object.type_name
158     .word Object.copy
159     .word IO.out_string
160     .word IO.out_int
161     .word IO.in_string
162     .word IO.in_int
163 Int_dispTab:
164     .word Object.abort
165     .word Object.type_name
166     .word Object.copy
167 String_dispTab:
168     .word Object.abort
169     .word Object.type_name
170     .word Object.copy
171     .word String.length
172     .word String.concat
173     .word String.substr
174 Bool_dispTab:
175     .word Object.abort
176     .word Object.type_name
177     .word Object.copy
178 Main_dispTab:
179     .word Object.abort
180     .word Object.type_name
181     .word Object.copy
182     .word Main.main
183     .globl heap_start
184 heap_start:
185     .word 0
186     .text
187     .globl Int_init
188     .globl String_init
189     .globl Bool_init
190     .globl Main_init
191     .globl Main.main
192 Object_init:
193     addiu $sp $sp -12
194     sw $fp 12($sp)
195     sw $s0 8($sp)
196     sw $ra 4($sp)
197     addiu $fp $sp 4
198     move $s0 $a0
199     move $a0 $s0
200     lw $fp 12($sp)
201     lw $s0 8($sp)
202     lw $ra 4($sp)
203     addiu $sp $sp 12
204     jr $ra
205 IO_init:
206     addiu $sp $sp -12
207     sw $fp 12($sp)
208     sw $s0 8($sp)
209     sw $ra 4($sp)
210     addiu $fp $sp 4
211     move $s0 $a0
212     jal Object_init
213     move $a0 $s0
214     lw $fp 12($sp)
215     lw $s0 8($sp)
216     lw $ra 4($sp)

```

217	addiu	\$sp \$sp 12	271	lw	\$s0 8(\$sp)
218	jr	\$ra	272	lw	\$ra 4(\$sp)
219	Int_init:		273	addiu	\$sp \$sp 12
220	addiu	\$sp \$sp -12	274	jr	\$ra
221	sw	\$fp 12(\$sp)	275	Main.main:	
222	sw	\$s0 8(\$sp)	276	addiu	\$sp \$sp -12
223	sw	\$ra 4(\$sp)	277	sw	\$fp 12(\$sp)
224	addiu	\$fp \$sp 4	278	sw	\$s0 8(\$sp)
225	move	\$s0 \$a0	279	sw	\$ra 4(\$sp)
226	jal	Object_init	280	addiu	\$fp \$sp 4
227	move	\$a0 \$s0	281	move	\$s0 \$a0
228	lw	\$fp 12(\$sp)	282	la	\$a0 int_const0
229	lw	\$s0 8(\$sp)	283	lw	\$fp 12(\$sp)
230	lw	\$ra 4(\$sp)	284	lw	\$s0 8(\$sp)
231	addiu	\$sp \$sp 12	285	lw	\$ra 4(\$sp)
232	jr	\$ra	286	addiu	\$sp \$sp 12
233	String_init:		287	jr	\$ra
234	addiu	\$sp \$sp -12			
235	sw	\$fp 12(\$sp)			
236	sw	\$s0 8(\$sp)			
237	sw	\$ra 4(\$sp)			
238	addiu	\$fp \$sp 4			
239	move	\$s0 \$a0			
240	jal	Object_init			
241	move	\$a0 \$s0			
242	lw	\$fp 12(\$sp)			
243	lw	\$s0 8(\$sp)			
244	lw	\$ra 4(\$sp)			
245	addiu	\$sp \$sp 12			
246	jr	\$ra			
247	Bool_init:				
248	addiu	\$sp \$sp -12			
249	sw	\$fp 12(\$sp)			
250	sw	\$s0 8(\$sp)			
251	sw	\$ra 4(\$sp)			
252	addiu	\$fp \$sp 4			
253	move	\$s0 \$a0			
254	jal	Object_init			
255	move	\$a0 \$s0			
256	lw	\$fp 12(\$sp)			
257	lw	\$s0 8(\$sp)			
258	lw	\$ra 4(\$sp)			
259	addiu	\$sp \$sp 12			
260	jr	\$ra			
261	Main_init:				
262	addiu	\$sp \$sp -12			
263	sw	\$fp 12(\$sp)			
264	sw	\$s0 8(\$sp)			
265	sw	\$ra 4(\$sp)			
266	addiu	\$fp \$sp 4			
267	move	\$s0 \$a0			
268	jal	Object_init			
269	move	\$a0 \$s0			
270	lw	\$fp 12(\$sp)			

- Liniile 109–121 ilustrează **tabelul prototipurilor și al rutinelor de inițializare** ale instanțelor claselor (`class_objTab`), discutate mai jos. Tabelul este necesar pentru construcția `new SELF_TYPE`, care alocă un nou obiect cu tipul dinamic al lui `self`. **Atenție!** Tabelul este indexat în funcție de **eticheta** clasei, reprezentată printr-un număr întreg.
- Liniile 122–150 conțin **prototipurile instanțelor** fiecărei clase din program (`<class>_protObj`), atât predefinite, cât și definite de programator. Prototipurile respectă structura discutată la curs.
- Liniile 151–182 surprind **tabelele de metode** aferente fiecărei clase din program (*dispatch table*, `<class>_dispTab`), ca secvență de etichete din zona `text`, unde se găsesc implementările acestor metode.
- Liniile 184–185 marchează finalul zonei statice și începutul zonei de *heap* (`heap_start`), în conformitate cu cerințele sistemului de execuție.
- Zona `text` debutează în linia 186 cu câteva definiții **globale**, solicitate de sistemul de execuție.
- Începând cu linia 192, sunt definite **rutinele de inițializare** a instanțelor fiecărei clase (`<class>_init`), și **metodele** din fiecare clasă, cu numele `<class>.<method>`.

Sistemul de execuție realizează următorii pași la **lansarea** unui program:

- Este instanțiată **clasa** `Main`. Acest lucru se realizează copiind pe *heap* obiectul prototip `Main_protObj`.
- Este apelată **rutina de inițializare** `Main_init` pentru această nouă instanță.
- Este apelată **metoda** `Main.main`, adresa instanței (`self`) fiind în registrul `$a0`, iar adresa de revenire, în registrul `$ra`.

3 Cerințe

În continuare, sunt prezentate particularitățile de reprezentare și implementare pentru anumite construcții de limbaj. Utilizați **semantica operațională** a limbajului Cool (secțiunea 13 a manualului) pentru a înțelege ce presupune evaluarea fiecărei expresii. Pentru cazuri concrete, consultați testele (vezi secțiunea 4).

Atenție! NU este necesar să generați cod identic cu cel din teste. Acesta va fi verificat în raport cu funcționalitatea îndeplinită.

3.1 Definiții de clase

Pentru fiecare definiție de clasă, predefinită sau definită de programator, este necesară definirea unui **obiect prototip** (`<class>_protObj`) și a unui **tabel de metode** (`<class>_dispTab`) în zona `data`, precum și a unei **rutine de inițializare** (`<class>_init`) în zona `text`.

Câmp	Offset
Etichetă clasă	0
Dimensiune în cuvinte	4
<i>Dispatch pointer</i>	8
Atribut 1	12
Atribut 2	16
⋮	⋮

Figura 1: Reprezentarea obiectelor în memorie

Reprezentarea în memorie a **obiectelor prototip** respectă convențiile discutate în curs, pe slide-urile 304–307, și este reluată în fig. 1. **Atenție!** Atributele din reprezentarea unui obiect le includ atât pe cele definite în clasa curentă, cât și pe toate cele de pe întregul lanț de **moștenire**! Atributele vor fi inițializate la **valorile implicite** în funcție de tip (vezi slide-ul 333).

Tabelele de metode respectă convențiile de pe slide-urile 308–309, și cuprind toate metodele accesibile unui obiect, de pe întregul lanț de **moștenire**. Spre exemplu, `IO_dispTab` include atât metodele specifice clasei `IO`, cât și pe cele ale clasei `Object`.

Rutinele de inițializare vizează **atributele** instanței curente. De asemenea, este necesară inițializarea atributelor moștenite, printr-un apel la rutina de inițializare a **superclasei**. Spre exemplu, `IO_init` invocă `Object_init`. Acestea se comportă ca niște metode, care sunt discutate pe larg în secțiunea 3.4. Ele primesc adresa obiectului curent în registrul `$a0`, unde trebuie să o și întoarcă.

3.2 Valori predefinite

Instanțele clasei **Int** conțin drept unic atribut valoarea întreagă MIPS. Spre exemplu, `int_const2` are valoarea întreagă 2.

Instanțele clasei **String** conțin două atribute: adresa unui obiect `Int` cu lungimea șirului, urmat de șirul propriu-zis. **Atenție! Dimensiunea** în cuvinte a unui obiect `String` va depinde de **lungimea** șirului! Spre exemplu `str_const2`, redând șirul `"IO"`, are lungimea descrisă de `int_const2`, iar câmpul de dimensiune este 5: 3 cuvinte pentru antet, 1 cuvânt pentru lungime, și 1 cuvânt pentru șirul `"IO"` plus caracterul terminator de șir ($2 + 1 = 3$ octeți, deci 1 cuvânt). Având în vedere că un șir se poate încheia în mijlocul unui cuvânt, definițiile următoare trebuie **aliniate**, utilizând instrucțiunea `.align 2`. De asemenea, este important să procesați șirurile înainte să le emiteți, deoarece pot conține **caractere speciale**, ca `'\n'`.

Instanțele clasei **Bool**, sunt similare celor ale clasei `Int`, unicul atribut fiind 0 sau 1, pentru `false`, respectiv `true`.

void este reprezentat prin întregul MIPS 0.

3.3 Literali

Pentru fiecare literal întreg, șir de caractere și boolean din program, există o definiție aferentă în zona `data`, respectând convențiile din secțiunea 3.2. În acest

Conținut	Adresă
Parametru n	
\vdots	\vdots
Parametru 2	$\$fp + 16$
Parametru 1	$\$fp + 12$
$\$fp$	
$\$s0$	
$\$ra$	$\$fp$
	$\$sp$

Figura 2: Înregistrarea de activare

scop, este util să vă definiți **tabele de literali** indexate prin numele literalilor, pentru a le putea **refolosi** pe cele deja definite.

3.4 Definiții și apeluri de metode

Protocolul de apel este similar celui din curs, slide-urile 288–295, cu câteva modificări, datorate (1) prezenței suplimentare a obiectului pentru care realizăm apelul, în afara parametrilor, și (2) faptului că sistemul de execuție Cool consideră că registrul $\$fp$ (*frame pointer*) este gestionat exclusiv de către **apelat**, nu și de apelant! Prin urmare, salvarea registrului $\$fp$ se face tot de către apelat, la fel ca și reîncărcarea lui la final. Această strategie este **mai eficientă**, întrucât, în codul generat, salvarea este menționată acum o singură dată pentru fiecare definiție de metodă, și nu la fiecare apel al acesteia, ca în curs.

În ceea ce privește obiectul **self**, adresa acestuia este depusă în registrul $\$a0$. Având în vedere că acesta va fi **modificat** în cursul execuției metodei, fie pentru a invoca alte metode, fie pentru a stoca rezultatul evaluării unei expresii, se recomandă **salvarea** acestui registru în $\$s0$, pentru acces facil. În consecință, vechea valoare a acestui registru trebuie ea însăși **salvată** pe stivă, alături de $\$fp$ și $\$ra$. Noua **înregistrare de activare** este ilustrată în fig. 2. Exemplul de cod din secțiunea 2 reflectă acest protocol.

În vederea implementării **apelurilor dinamice** de metodă (*dynamic dispatch*), este necesară accesarea intrării corecte din tabelul de metode aferent tipului dinamic al obiectului. Odată ce încărcăm într-un registru adresa metodei, o puteți invoca utilizând instrucțiunea *jalr* (*jump and link register*), care, similar instrucțiunii *jal*, depune în registrul $\$ra$ adresa de revenire.

În cazul în care apelul se realizează pentru un obiect **void**, este necesar să invocați rutina predefinită `_dispatch_abort` (vezi secțiunea 3.9).

3.5 Construcția **let**

Variabilele de **let** sunt alocate pe **stivă**. Puteți utiliza orice schemă de stocare a acestora. Modelele din teste utilizează convenția din fig. 3.

3.6 Construcția **new**

Instrucțiunea **new** presupune alocarea dinamică a unui nou obiect pe *heap*, pornind de la obiectele prototip din zona statică. În cazul în care **new** este utili-

Conținut	Adresă
Parametru n	
\vdots	\vdots
Parametru 2	$\$fp + 16$
Parametru 1	$\$fp + 12$
$\$fp$	
$\$s0$	
$\$ra$	$\$fp$
Variabilă let 1	$\$fp - 4$
Variabilă let 2	$\$fp - 8$
\vdots	\vdots
Variabilă let m	
	$\$sp$

Figura 3: Posibilă înregistrare de activare cu variabile de let

zat cu un **nume de clasă**, e.g. `new C`, se aplică mai întâi metoda predefinită `Object.copy` asupra obiectului prototip `C_protObj` (vezi secțiunea 3.9), iar apoi, asupra noului obiect, rutina de inițializare `C_init`.

În cazul expresiei **new SELF_TYPE**, este utilizat tabelul `class_objTab`, indexat pornind de la eticheta clasei reprezentând tipul dinamic al lui `self`. Astfel, dacă eticheta este *tag*, obiectul prototip se găsește în tabel la *offset*-ul $8 \cdot tag$, iar rutina de inițializare, la $8 \cdot tag + 4$.

3.7 Verificarea egalității obiectelor

Pentru verificarea egalității obiectelor, este necesară mai întâi testarea egalității **referințelor**. Dacă acestea sunt diferite, trebuie invocată rutina predefinită `equality_test` (vezi secțiunea 3.9), care verifică egalitatea de **conținut** în cazul valorilor predefinite, de tip `Int`, `String` și `Bool`.

3.8 Construcția case

Construcția `case` solicită prezența la execuție a anumitor informații despre ierarhia de clase. O observație utilă este că, asociind etichete claselor în ordinea dată de **parcurea în adâncime** a arborelui de moștenire, fiecare clasă și toate subclasele ei vor ocupa un interval de etichete consecutive, e.g. 5, 6 și 7, astfel încât extremele acestui interval pot fi utilizate pentru a surprinde orice tip dinamic aferent unei anumite ramuri a construcției `case`.

În cazul în care nicio ramură **nu se potrivește**, este necesară invocarea rutinei predefinite `_case_abort`. În cazul în care obiectul analizat este **void**, este necesară invocarea rutinei predefinite `_case_abort2` (vezi secțiunea 3.9).

3.9 Metode și rutine predefinite

Secțiunea prezintă metodele și rutinele predefinite, și modalitatea lor de invocare:

Object.copy Copiază pe *heap* obiectul cu adresa în registrul \$a0 și depune adresa noului obiect tot în \$a0.

Object.abort Afișează numele tipului dinamic al obiectului cu adresa în \$a0 și încheie execuția programului. Tabelul `class_nameTab` este accesat în acest scop.

Object.type_name Întoarce în \$a0 numele tipului dinamic al obiectului cu adresa în \$a0.

IO.out_string Afișează obiectul șir de caractere primit ca parametru.

IO.out_int Afișează obiectul număr întreg primit ca parametru.

IO.in_string Întoarce în \$a0 adresa obiectului șir de caractere citit de la tastatură.

IO.in_int Întoarce în \$a0 adresa obiectului număr întreg citit de la tastatură.

String.length Întoarce în \$a0 adresa obiectului număr întreg care reprezintă lungimea obiectului șir de caractere cu adresa în \$a0.

String.concat Întoarce în \$a0 adresa unui **nou** obiect șir de caractere, rezultat din concatenarea obiectului cu adresa în \$a0 cu obiectul primit ca parametru.

String.substr Întoarce în \$a0 adresa unui nou obiect șir de caractere, reprezentând subșirul obiectului cu adresa în \$a0, determinat de index-ul și lungimea date ca parametri.

equality_test Verifică dacă obiectele cu adresele în \$t1 și \$t2 au același tip dinamic predefinit și aceeași valoare. Dacă da, este întoarsă valoarea din \$a0; altfel, din \$a1.

_dispatch_abort Este utilizată în cazul unui apel de metodă pe un obiect **void**. Afișează un mesaj de eroare, ce include numele fișierului, ca obiect șir de caractere, cu adresa în \$a0, și numărul liniei din fișier, ca întreg MIPS (utilizați instrucțiunea `li`) în \$t1. Încheie execuția programului.

_case_abort Este utilizată când **nu** se realizează potrivire cu nicio ramură a instrucțiunii `case`. Afișează un mesaj de eroare ce include numele tipului dinamic al obiectului cu adresa în \$a0 și încheie execuția programului.

_case_abort2 Este utilizată când instrucțiunea `case` analizează un obiect **void**. Similar cu `_dispatch_abort`, afișează un mesaj de eroare cu numele fișierului (\$a0) și linia (\$t1) aferente obiectului problematic, și încheie execuția programului.

4 Testare

Testele pot fi **runate** pe Linux, executând scriptul `tester.sh`, în cadrul căruia puteți actualiza la nevoie variabila `CLASSPATH` pentru a indica jar-ul de ANTLR. Pentru rularea manuală a unui program MIPS în simulatorul *spim*, în linia de comandă, utilizați:

```
1 spim -exception_file trap.handler.nogc -file <file.s>
```

Pentru rularea în *QtSpim*, selectați Simulator -> Settings -> MIPS -> Load exception handler și alegeți fișierul trap.handler.nogc. Apoi, încărcați programul dorit utilizând butonul Reinitialize and Load File.

Testele se află în directorul tests/tema3 din rădăcina arhivei de pornire. Fișierele .cl conțin programe Cool de compilat. Modele de programe MIPS echivalente sunt furnizate în fișierele .s-model. Fișierele .ref conțin ieșirile de referință ale simulatorului. Pentru fiecare test, sistemul de testare redirecționează ieșirea standard a compilatorului către un fișier .s, pe care îl execută în simulator, și stochează ieșirea acestuia într-un fișier .out, pe care îl compară apoi cu cel de referință.

Având în vedere că testele verifică **incremental** funcționalitatea compilatorului, le puteți folosi pentru a vă ghida **dezvoltarea** temei!

Primele patru teste nu afișează nimic. Scopul lor este de a vă ajuta să organizați codul generat pentru ierarhia de clase. Ultimul test, **32**, este singurul care citește intrarea de la consolă, folosind metoda IO.in_int. Sistemul de testare fixează intrarea acestuia la 5, astfel încât să nu fie necesară introducerea manuală a numărului de fiecare dată când rulați testele.

5 Precizări

- Clasele, variabilele și metodele trebuie **indexate**, pentru a le putea referi corect în codul generat. Spre exemplu, clasele necesită etichete, atributele trebuie referite în funcție de *offset*-ul în cadrul reprezentării obiectului din care fac parte, metodele sunt caracterizate de *offset*-ul în cadrul tabelelor de metode, parametrii formali, de *offset*-ul pe stivă etc. Puteți reține aceste informații în **simbolurile** asociate.
- **StringTemplate** posedă anumite mecanisme mai avansate, pe care le puteți exploata în rezolvarea temei. Spre exemplu, apelarea metodei add pe un *template*, utilizând un obiect List ca valoare, este echivalentă cu secvența de apeluri add pentru fiecare element al listei. *Template*-urile pot fi aplicate asupra fiecărei valori a unui atribut (*mapping*), utilizând sintaxa <attr:transform()>, unde attr este un atribut cu mai multe valori, iar transform este un *template* cu un singur parametru. De asemenea, există și *template*-ul special <if>, prin care puteți cosmetiza formatarea. Găsiți mai multe detalii în documentația de la adresa <https://github.com/antlr/stringtemplate4/blob/master/doc/index.md>.

6 Referințe

1. Manualul limbajului Cool.
<https://curs.upb.ro/2022/mod/resource/view.php?id=25343>
2. Documentația sistemului de execuție al limbajului Cool.
<https://curs.upb.ro/2022/mod/resource/view.php?id=25344>
3. Documentația SPIM.
<https://curs.upb.ro/2022/mod/resource/view.php?id=25350>

4. Documentația *StringTemplate*.

<https://github.com/antlr/stringtemplate4/blob/master/doc/index.md>