Upute za izradu seminarskog rada

Autori: Ivo Mateljan, Alex Aiken, Marjan Sikora

Split, svibanj 2019.

1. Postavljanje okruženja za izradu seminarskog rada

Za izradu seminarskog rada, koji u stvari znači izradu kompajlera za jezik **Cool**, trebate najprije postaviti okruženje u kojem možete izraditi kompajler. Zbog složenosti instaliranja svih potrebnih alata, pripremljena je unaprijed konfigurirana Linux virtualna mašina (VM), koja se koristi putem programa **Oracle VirtualBox**.

1.1 Postavljanje virtualne mašine

Ideja VM-a je da pokrenete zasebno virtualno računalo sa svojim operativnim sustavom unutar programa za virtualizaciju, u ovom slučaju **VirtualBox**-a. Ovakvo virtualno računalo i operativni sustav koriste vaše stvarno računalo, ali ne izravno, nego posredstvom programa za virtualizaciju.

Za ovaj kolegij pripremljena je slika VM-a, sa obavljenim svim podešavanjima i instaliranim svim programima za izradu kompajlera za **Cool**. Kada je instalirate, VM će uzeti malo više od 512 MB RAM-a (tako će vam računalo trebati imati 2 GB da bi stvari radile glatko), plus malo više od 2 GB prostora na tvrdom disku.

1.2 Instalacija VirtualBoxa

Najprije je potrebno preuzeti i instalirati program za virtualizaciju - **Oracle VirtualBox**. Program skinite sa: https://www.virtualbox.org/wiki/Downloads i instalirajte ga.

1.3 Instalacija VM slike za izradu seminarskog rada

Preuzmite sliku VM-a sa elearning sustava. Veličina je približno 750 MB. Raspakirajte datoteku u direktorij po svom izboru, veličina će biti oko 2 GB. Nakon što ste raspakirali VM, dvaput kliknite na datoteku ***Compilers.vbox*** i VM će se otvoriti u **VirtualBox**-u.

1.4 Korištenje virtualnog računala

Ovaj VM temelji se na **Bodhi Linuxu**, koji je sam po sebi baziran na popularnoj Linux distribuciji **Ubuntu**. Koristili smo **Bodhi Linux** kako bismo veličinu preuzimanja držali podnošljivijom, budući da je potpuna **Ubuntu** instalacija prilično velika. Međutim, budući da je baziran na Ubuntuu, većina Ubuntu softverskih paketa može se instalirati i na **Bodhi Linux**.

Za pokretanje VM-a kliknite na zeleni botun *Start*. Time bi se VM trebao podići. Da biste isključili VM, kliknite na okrugli botun ***Bodhi*** u donjem lijevom kutu i kliknite ***System***. Zatim odaberite ***Power Off***.

Instalirali smo ono što vam je potrebno za izvršavanje zadataka, a ako želite instalirati druge pakete, možete koristiti upravitelj grafičkog paketa **Aptitude** (pod izbornikom ***Bodhi/ Applications/***

Preferences/ Synaptic Package Manager) ili alatom za naredbeni redak **`apt-get`**. Ako niste upoznati s načinom instaliranja paketa na Linuxu postoje mnogi online vodiči koje možete pronaći na webu.

1.5 Cool primjeri

Kada instalirate VM, možete početi raditi sa jezikom Cool. Da biste otvorili terminal, kliknite na ikonu terminala na dnu zaslona. U terminalu možete započeti sa radom u jeziku **Cool**. Korisnički račun je **`compilers`**, a lozinka je **`cool`** . Sve datoteke vezane uz ovaj seminarski zadatak nalaze se u folderu **`cool`**

Primjeri ***Cool*** programa nalaze se u direktoriju **`examples`**. Da biste kompajlirali primjer, pozovite u terminalu naredbu **`coolc`**. Kompajler će stvoriti izvršnu datoteku s nastavkom **`.s`**, koju možete pokrenuti pomoću simulatora **MIPS** procesora **`spim`**.

Na primjer, da biste kompajlirali i pokrenuli ***hello_world.cl***, u terminalu, u folderu **`cool/examples`** (gdje **$** predstavlja prompt) upišite slijedeće naredbe:

```
$ coolc hello_world.cl

$ spim hello_world.s

SPIM Version 6.5 of January 4, 2003

Copyright 1990-2003 by James R. Larus (larus@cs.wisc.edu).

All Rights Reserved.

See the file README for a full copyright notice.

Loaded: /usr/class/cs143/cool/lib/trap.handler

Hello, World.

COOL program successfully executed

Stats -- #instructions : 152

         #reads : 27  #writes 22  #branches 28  #other 75
```

2. Upoznavanje programa Flex (Lex) kroz primjere

2.1 Trivijalni slučaj primjene Flex-a

Prepoznati ulazni niz znakova i proslijediti ga na izlaz. Specifikacijska datoteka za takvu analizu izgleda ovako:

```
/* File: ex1.lex */
%%
. | \n      ECHO;
%%
int yywrap(void)
{
   return 1;
}

int main(void)
{
    yylex();
    return 0;
}
```

Da bi dobili i testirali izvršni program (na Linuxu), koriste se sljedeće komande:

```
$ flex ex1.lex
$ gcc lex.yy.c –o ex1
$ /.ex1 <test.in
```

U prethodnoj specifikaciji, u sekciji pravila, koja je omeđena s `%%...%%`, napisano je samo jedno pravilo

```
. | \n      ECHO;
```

Ovo pravilo znači da ako se prepozna bilo koji znak osim nove linije (`.`) ili znak nove linije (`\n`) izvršit će se akcija **ECHO**.

ECHO je makro naredba koja je definirana u Flex-u, tipično s

```
#define ECHO fwrite(yytext, yyleng, 1, yyout)
```

što znači da se iz stringa **yytext**, proslijedi **yyleng** znakova u izlazni tok **yyout**. Ove tri varijable su globalne varijable koje Flex generira u **lex.yy.c**. Značenje ime je dano u tablici 1.

Varijabla **yytext** sadrži prepoznati leksem, a **yyleng** sadrži duljinu prepoznatog leksema.

Isti učinak bi bio postignut i s pravilom:

```
. | \n      printf("yytext");
```

jer je **yytext** pokazivač na znakovni niz koji je uvijek zaključen s `'\0'`, dakle predstavlja string.

> **Uočite:** Pravilo se sastoji od regularnog izraza i "akcije", koja sadrži naredbu C jezika. Ako ima više naredbi, mora ih se pisati unutar vitičastih zagrada.

Nakon sekcije pravila, definirane su dvije korisničke funkcije **yywrap()** i **main()**. U ovoj sekciji se može definirati proizvoljan programski kod u C jeziku, ali obvezatno se mora definirati funkcija **yywrap()**, koja mora vratiti cjelobrojnu vrijednost 0 ili 1. Ako ova funkcija vrati vrijednost 1, to znači da s leksička analiza prekida kada Flex dostigne kraj datoteke. Ako vrati vrijednost 0, analiza se nastavlja datotekom, koju mora otvoriti korisnik (obično unutar ove funkcije).

Neke verzije Flex-a sadrže biblioteke s prethodno definiranim "default" oblikom **`yywrap()`** funkcije.

Ime	Funkcija
`int yylex(void)`	poziv leksičkog analizatora (obično se realizira da vraća vrijednost tokena)
`char *yytext`	pokazivač na prepoznati leksem
`int yyleng`	duljina prepoznatog leksema
`yylval`	vrijednost tokena (korisnik definira tip od **`yylval`**)
`int yywrap(void)`	**`wrapup`**, vraća 1 ili 0, ovisno o tome da li se želi završiti procesiranje ulaza
`FILE *yyout`	izlazna datoteka (default **`stdout`**)
`FILE *yyin`	ulazna datoteka (default **`stdin`**)
`INITIAL`	inicijaliziraj startne uvjete
`BEGIN condition`	započni od startnih uvjeta
`ECHO`	ispiši prepoznati leksem na **`yyout`**

Tablica 1 Flex - predefinirane varijable, funkcije i makroi

2.2 Analiza leksema

U sljedećem programu se ispisuje da li je neki leksem: ključna riječ, operator, separator ili broj.

```
/* File ex2.lex */

%%
[\t\r ]+          ;      /* preskoci bijele znakove */;

while |
if |
do |
for |
switch |
case |
else |
break |
continue |
static       {printf("%s: kljucna rijec\n", yytext);}
[0-9]+       {printf("%s: broj\n", yytext);}
[a-zA-Z]+    {printf("%s: identifikator \n", yytext);}
[\*+-=<>]    {printf("%s: operator\n", yytext);}
[()\[\];] {printf("%s: separator\n", yytext);}
.|\n         { ECHO; /*default */}
%%

int yywrap(void) { return 1; }

int main ()
{
yylex();
return 0;
}
```

Ako otkucamo:

```
c:>ex2

do a=3+c; while(x>7);
```

dobit ćemo ispis:

```
do: kljucna rijec
a: identifikator
=: operator
3: broj
+: operator
c: identifikator
;: operator
while: kljucna rijec
(: separator
a: identifikator
>: operator
7: broj
): separator
;: operator
```

U početnoj sekciji definicija unutar `%{ ... %}` zapisuje se proizvoljni kôd u C jeziku, koji će kasnije biti uključen u program leksičkog analizatora.

U sekciji pravila, unutar `%%....%%` , kao i u prethodnom primjeru, zapisana su pravila koja se sastoje od regularnog izraza i pridružene akcije koja je zapisana naredbom C jezika.

U ovom primjeru postoji 6 pravila:

Pravilo 1 – određuje da se preskoči bijele znakove (nema akcije)

```
[\t\r ]+            /* preskoci bijele znakove */;
```

Pravilo 2 – sadrži listu ključnih riječi (odvojeno vertikalnom crtom – što označava alternativne regularne izraze) . Ako je prepoznata ključna riječ ispisuje se poruka. Ovu listu se moglo zapisati u jednom retku:

```
(while)|(if)|(do)|(for)|(...)|(else)| {printf("%s: kljucna rijec\n", yytext);}
```

ali tada svaku izraz treba napisati unutar zagrada.

Pravilo 3 - `[0-9]+` prepoznaje cijeli broj i ispisuje poruku.

Pravilo 4 - `[\*+-=<>]` prepoznaje operatore: **`*+-=<>`** i ispisuje poruku.

Pravilo 5 - `[\(\)\[\];]` prepoznaje separatore : **`()[];`** i ispisuje poruku.

Pravilo 6 - `. | \n` prepoznaje sve znakove i znak nove linije, ali koristi ovo pravilo samo ako, prema prethodnim pravilima, nije pronađen leksem veće duljine.

Zapamti: ako više sekvenci započinje istim znakovima Flex izvršava akciju za najdulju sekvencu.

Pravilo 6 (.) se gotovu uvijek zadaje, je bez njega Flex ne bi mogao razlikovati pod-sekvence, primjerice, bez ovog *default* pravila, Flex bi prepoznao leksem **do** i u sekvencama **undo** i **done**.

U završnoj sekciji je ponovo se može pisati program u C jeziku.

2.3 Rad s proizvoljnom ulaznom datotekom

U ovom primjeru izvršena je specifikacija kojom se svakoj liniji teksta na izlazu dodaje broj linije. Broj linije se bilježi u globalnoj varijabli **yylineno**. (neke verzije Flex-a imaju ugrađenu ovu varijablu). Ulazna datoteka se mora registrirati u globalnoj varijabli Flex-a **yyin**, (tipa **FILE** *). Ukoliko se ne zada vrijednost **yyin** Flex je postavlja na **stdin**.

```
/* File ex3.lex */
%{
    int lineno=0;   /* the row counter */
%}
%%
^(.*)\n    printf("%4d\t%s", ++lineno, yytext);
%%

int yywrap(void) {   return 1; }

int main(int argc, char *argv[])
{
    yyin = fopen(argv[1], "r");
    if(yyin == NULL) exit(1);
    yylex();
    fclose(yyin);
}
```

2.4 Brojanje znakova riječi i linija

Sljedeći analizator ispisuje broj znakova, riječi i linija u nekoj datoteci (sličnu funkciju ima Unix program **wc**):

```
/* File ex4.lex */
%{
    int nchar=0, nword=0, nline=0;
%}
%%
\n          { nline++; nchar++; }
[^ \t\n]+   { nword++, nchar += yyleng; }
.           { nchar++; }
%%
int yywrap(void) { return 1; }
int main(void) {
    yylex();
    printf("%d\t%d\t%d\n", nchar, nword, nline);
    return 0;
}
```

2.5 Supstitucija regularnih izraza

Za često korištene regularne izraze mogu se definirati makroi. U sljedećem primjeru, u sekciji definicija, definirani su makroi **digit** i **letter**. U sekciji specifikacije oni se zapisuju unutar vitičastih zagrada, što označava da se na tom mjestu supstituira prethodno definirani regularni izraz.

```
/* File ex5.lex */

digit     [0-9]
letter    [A-Za-z]


%{
    int count;
%}

%%
    /* match identifier */
({letter}|{digit})*    count++;
%%

int yywrap(void) { return 1; }

int main(int argc, char *argv[])
{
    yylex();
    printf("number of identifiers = %d\n", count);
    fclose(yyin);
    return 0;
}
```

Napomena: u definiciji između imena i definicije mora biti razmak.

2.6 Decimalni brojevi

Sljedeći analizator prepoznaje decimalne brojeve u običnom i eksponencijalnom formatu, ignorira bijele znakove i novu liniju, te ispisuje na izlaz sve ostale znakove koji ne pripadaju decimalnom broju (**ECHO**).

```
/* File ex6.lex */

EXP [eE][-+]?[0-9]+
DOT \.
DIG [0-9]
%%
({DIG}*{DOT}{DIG}+{EXP}?) {printf("number\n");}
({DIG}+{DOT}?{EXP}?)      {printf("number\n");}
. ECHO;

%%
int yywrap(void) { return 1; }
int main()
{
yylex();
return 0;
}
```

2.7 Velika i mala slova

Ovom primjeru, (a) velika slova se pretvaraju u mala slova, (b) odstranjuju se znakovi razmaka na kraju linije, (c) višestruki znak razmaka se zamjenjuje jednim znakom.

```
/* File ex7.lex */

%%
[A-Z]       putchar(yytext[0]-'A'+'a');
[ ]+$       ;
[ ]+        putchar(' ');
%%
int yywrap(void) { return 1; }
int main()
{
yylex();
return 0;
}
```

2.8 Tokenizacija

U ovom primjeru je specifikacija leksičkog analizatora za komandni jezik koji prepoznaje brojeve, stringove, znak nove linije, identifikator komande, te ignorira bijele znakove i komentar koji započinje znakovima // i traje do kraja linije.

```
/* File ex8.lex */

%{
#include <stdio.h>
#define NUMBER 400
#define COMMENT 401
#define STRING 402
#define COMMAND 403
#define NEWLINE 404
%}
%%
[ \t]+                  ;
[0-9]+ |
[0-9]+\.[0-9]+ |
\.[0-9]+          { return NUMBER; }
\"[^\"\n]*\"            { return STRING; }
[a-zA-Z][a-zA-Z0-9]+    { return COMMAND; }
"//".*                  { return COMMENT; }
\n                { return NEWLINE; }
%%
int yywrap(void)  { return 1; }
int main(int argc, char *argv[]) {
      int val;
      while(val = yylex())
            printf("value is %d\n",val);
      return 0;
}
```

Domaći rad: Samostalno napravite Flex kod za analizu teksta koji će raditi slijedeće:

1. skinuti sadržaj proizvoljne Intenet stranice i snimiti ga u tekstualnu datoteku
2. odabrati pet riječi koje se često pojavljuju u tekstu, te tih pet riječi u tekstu zamijeniti TISKANIM slovima
3. napraviti jedan zadatak od slijedećeg:
   - izbrojiti broj rečenica
   - izbrojiti broj interpunkcija
   - izbrojiti broj riječi koje započinju sa velikim slovom
4. napraviti jedan zadatak od slijedećeg:
   - odstraniti tekst između dvaju zagrada
   - odstraniti sve brojeve veće od 10 iz teksta
   - odstraniti svaku drugu rečenicu

Dodatni materijal za učenje Flex-a možete pronaći na stranici:
http://dinosaur.compilertools.net/lex/index.html

3. Programski jezik COOL

COOL je kratica od *Classroom Object Oriented Language*. Ovaj programski jezik je dizajniran tako da se kompajler za COOL može napraviti u jednom semestru. Neka rješenja u jeziku COOL čine ga nepraktičnim za korištenje u stvarnom svijetu, a takva rješenja su odabrana baš zato da bi se lakše pisao kompajler za COOL. Zanimljiva je činjenica da je broj COOL kompajlera puno veći od broja COOL programa.

Najvažnije komponente jezika COOL su apstraktni tipovi podataka, statičko određivanje tipova, nasljeđivanje i upravljanje memorijom (eng. *garbage collector*). Iz jezika su izostavljene neke stvari radi lakše izrade kompajlera.

Zadatak seminarskoga rada je napraviti kompajler, koji će program pisan u jeziku COOL prevesti u strojni jezik za MIPS procesor. MIPS procesor je stvarni procesor RISC tipa, razvijen u osamdesetim godinama na Sveučilištu Stanford. Na vašim računalima program preveden u strojni jezik za MIPS će se pokretati pomoću simulatora.

Seminarski rad ima četiri faze:

1. leksički analizator
2. parser
3. semantički analizator
4. generiranje koda

Sve ove faze su međusobno povezive, te zajedno čine COOL kompajler. Seminarski rad ne obuhvaća fazu optimizacije, ali ovu fazu je moguće izraditi i uključiti po volji.

3.1 Klasa Main

Programi u COOL jeziku su tekstualne datoteke sa nastavkom ***.cl***. Svaki program mora imati klasu **`Main`**:

```
class Main {
      ...
};
```

Deklaracija klase počinje ključnom riječi **`class`,** sadržaj klase je obuhvaćen vitičastim zagradama, a klasa završava znakom **;**

COOL program je lista klasa.

3.2 Metoda main()

Klase mogu sadržavati metode. Svaka metoda mora imati ime, popis argumenata u zagradama i listu izraza koji čine tijelo metode unutar vitičastih zagrada. Svaka metoda treba biti završena sa znakom **;**

Svaki program mora obavezno imati metodu **`main()`** klase **`Main`** :

```
class Main {
    main() {...};
};
```

Metodom **`main()`** započinje izvršenje programa. Ova metoda ne smije imati argumente.

Sadržaj metode je lista izraza, navedena unutar vitičastih zagrada. U jeziku COOL nema ključne riječi **`return`** za određivanje koju vrijednost vraća metoda. Vrijednost metode je vrijednost zadnjeg izraza u tijelu metode.

Za početak ćemo staviti u metodu **`main()`** samo jedan izraz:

```
class Main {
    main() { 1 };
};
```

3.3 Prevođenje i pokretanje

Ovaj program ćemo prevesti sa **coolc**:

```
"1.cl", line 2: syntax error at or near '{`
Compilation halted due to lex and parse errors
```

Prevođenje nije uspjelo. Vidimo da postoji sintaksna greška. Razlog je šta metode moraju imati tip. Odrediti ćemo tip metode **`Main`** kao **`Int`.**

```
class Main {
    main():Int { 1 };
};
```

Ponovo ćemo prevesti program. Ovaj puta je prevođenje uspjelo, što znamo jer kompajler nije dojavio nikakve greške, a također je stvorio i datoteku sa prevedenom strojnim kodom i nastavkom **`.s`** Ovu datoteku pokrećemo pomoću simulatora **`spim`**:

```
SPIM Version 6.5 of January 4, 2003
Copyright 1990-2003 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/class/cs143/cool/lib/trap.handler
COOL program successfully executed
Stats -- #instructions : 123
         #reads : 20  #writes 17  #branches 23  #other 63
```

Simulator pokrene datoteku, te izvijesti da je COOL program uspješno izvršen. Simulator također javlja broj čitanja i pisanja u registre, broj grananja i druge informacije koje mogu biti korisne prilikom optimizacije kompajliranja.

3.4 Ispis

Naš prvi program nije ništa ispisao. Za ispis je potrebno koristiti klasu **IO**. Najprije ćemo deklarirati atribut **i** klase **Main** tipa **IO**. Zatim pomoću i možemo vršiti ispis:

```
class Main {
	i:IO;
	main():Int {
		{
			i.out_string(„Hello World!\n”);
			1;
		}
	};
};
```

Sada je sadržaj metode **main()** blok koji se sastoji od dvije naredbe:

- prva je naredba za ispis
- druga je izraz jedan

Svaka naredba bloka treba biti završena sa znakom točke-zareza. Vrijednost bloka u jeziku COOL jednaka je vrijednosti zadnje naredbe bloka. Vrijednost funkcije **main()** biti će ujedno i vrijednost bloka. Blok je potrebno staviti unutar vlastitih vitičastih zagrada.

Ovakav program će se uredno prevesti, ali prilikom pokretanja javlja grešku:

```
Dispatch to void
```

Razlog je što smo varijablu **i** samo deklarirali, ali nismo alocirali. U jeziku COOL svaku varijablu je potrebno alocirati ključnom riječi **new** uz koju pišemo klasu objekta koji alociramo.

```
class Main {
	i:IO <- new;
	main():Int {
		{
			i.out_string(„Hello World!\n”);
			1;
		}
	};
};
```

Program se sada može i prevesti i pokrenuti, te ispisuje „Hello World!“.

Ovaj program možemo pojednostavniti tako da uklonimo blok i ostavimo samo naredbu za ispis:

```
class Main {
```

```
        i:IO <- new IO;
        main():Int { i.out_string(„Hello World!\n") };
};
```

Prilikom kompajliranja dobiti ćemo slijedeću grešku:

```
Inferred return type IO of method main does not conform to
declared return type Int.
```

Razlog je šta metod**a out_string** vraća tip **IO,** a ne **Int.** Promijeniti ćemo tip metode u **IO**:

```
class Main {
        i:IO <- new IO;
        main():IO { i.out_string(„Hello World!\n") };
};
```

Sada program radi ispravno.

U jeziku COOL svaka klasa je pod-klasa od klase **Object**. Zato možemo kao povratni tip metode **main()** staviti **Object**. Zbog mehanizma nasljeđivanja kompajler neće javiti grešku.

```
class Main {
        i:IO <- new IO;
        main():Object { i.out_string(„Hello World!\n") };
};
```

Za ispis nije nužno deklarirati i alocirati zaseban atribut u klasi **Main**. U jeziku COOL možemo unutar koda privremeno alocirati objekt neke klase i pozvati metodu za njega:

```
class Main {
        main():Object { (new IO).out_string(„Hello World!\n") };
```

Također, umjesto privremene alokacije objekta klase **IO**, klasa **Main** može naslijediti klasu **IO**. Tada metodu **out_string** pozivamo pomoću ključne riječi **self**. Ona predstavlja sam objekt, kao što je **this** u C++ pokazivač na sam objekt. Nasljeđivanje klase se navodi iza imena klase, pomoću ključne riječi **inherits**, iza koje slijedi ime klase od koje se nasljeđuje.

```
class Main inherits IO {
        main():Object { self.out_string(„Hello World!\n") };
```

Nije nužno eksplicitno pisati **self** prilikom poziva metode. Ukoliko je neka metoda pozvana bez navođenja objekta, podrazumijeva se da je pozvana za **self**.

```
class Main inherits IO {
        main():Object { out_string(„Hello World!\n") };
```

3.5 Ulaz

Za čitanje stringa sa tipkovnice koristimo funkciju **in_string()** klase **IO**.

```
class Main inherits IO {
	main():Object {
		out_string(
			in_string().concat(„\n”)
		)
	};
};
```

U ovom programu nakon čitanja ispišemo string. Na učitani string nadovezali smo znak nove linije pomoću funkcije **concat()** klase **String.**

3.6 Konverzija stringa u broj

Ukoliko želimo pročitati cijeli broj, koristimo konverziju iz stringa u cijeli broj. Konverzija se radi pomoću funkcije **a2i()** klase **A2I**:

```
class Main inherits IO {
	main():Object {
		out_string(
			(new A2I).i2a(
				(new A2I).a2i( in_string() )
				+1
			).concat("\n")
		)
	};
};
```

U kodu potom na učitani broj dodamo 1. Na kraju, za ispis koristimo konverziju natrag u string.

Kada pokušamo prevesti ovaj program dobiti ćemo grešku:

```
'new' used with undefined class A2I.
```

Razlog je što se kod klase **A2I** se nalazi u drugoj datoteci - **atoi.h**. Da bi preveli više datoteka zajedno, prilikom poziva kompajlera navedemo sve potrebne datoteke navedemo odvojene razmakom:

```
coolc 2.cl atoi.cl
```

3.7 Izrada nove metode

Povećanje broja za jedan ćemo prebaciti u funkciju:

```
class Main inherits IO {
	main():Object {
		out_string(
			(new A2I).i2a(
				fact((new A2I).a2i( in_string()))
			).concat("\n")
		)
	};
	fact(i:Int):Int { i+1 };
};
```

Dodali smo funkciju **`fact()`**. Ona prima jedan cijeli broj i vraća također cijeli broj, uvećan za jedan. Ovu funkciju zovemo iz funkcije **`main()`**.

Sada ćemo napisati funkciju koja računa faktorijel cijeloga broja.

```
	fact(i:Int):Int {
		if(i=0) then
			1
		else
			i*fact(i-1)
		fi
	};
```

Funkciju za faktorijele smo napisali rekurzivno. Koristili smo **`if then else`** grananje, a kao granični uvjet provjeru je li argument jednak 1. U tom slučaju vratili smo 1, a inače ulazimo u rekurzivni poziv. U jeziku COOL **`if`** grananje s završava sa **`fi`**.

Napišimo funkciju za faktorijel iterativno:

```
	fact(i:Int):Int {
		let fact:Int <- 1 in {
			while(not(i=0)) loop {
				fact <- fact * i;
				i <- i - 1;
			} pool;
```

```
                fact;
        }
    };
```

U ovom slučaju koristimo petlju **`while loop pool`**. Kao brojač koristimo argument **`i`**. Uvjet petlje je da brojač nije jednak nuli – koristimo relacijski predikat **`not`**. Za proračun će nam trebati i lokalna varijabla u koju ćemo spremati umnožak. Lokalnu varijablu **`fact`** deklarirati ćemo pomoću naredbe let. U jeziku COOL lokalne varijable smiju biti istog imena kao i funkcije.

Važno je primijetiti da se za dodjelu vrijednosti u jeziku COOL koristi operator **`<-`**

Ako umjesto toga greškom stavimo operator **=** program će se uredno prevesti, ali će pri pokretanju uđe u beskonačnu petlju, te ćemo nekoliko puta dobiti poruku **`Increasing heap...`** i nakon toga će program prekinuti rad.

3.8 Izrada nove klase

Pokazati ćemo kako se u jeziku COOL izrađuje nova klasa na primjeru izrade samo-referentne strukture podataka – liste. Najprije ispišimo **`Hello World!`** na malo drugačiji način.

```
class Main inherits IO {
    main():Object {
        let  hello: String <- "Hello ",
             world: String <- "World!",
             newline: String <- "\n"
        in
             out_string(hello.concat(
                  world.concat(newline)))
    };
};
```

Pomoću **`let`** smo definirali tri lokalne varijable. Definicije se povezuju sa operatorom zareza. Tijelo funkcije **`main()`** povezuje sve tri lokalne varijable i ispisuje rezultat.

Napravimo sada klasu za samo-referentnu strukturu podataka – listu:

```
class List {
    item: String;
    next: List;
    init(i: String, n: List): List {
        {
             item <- i;
             next <- n;
```

```
                self;
            }
        };
};
```

Ovo je lista stringova. Klasa ima dva atributa **item** i **next**, te ima metodu **init()**, koja postavlja vrijednost čvora liste. Metoda **init()** vraća sam objekt, kako bi mogli rekurzivno nadovezivati čvorove liste.

Preradimo sada glavnu klasu i metodu, kako bi koristili listu za ispis:

```
...
class Main inherits IO {
    main():Object {
        let  hello: String <- "Hello ",
             world: String <- "World!",
             newline: String <- "\n",
             nil: List,
             list: List <-
                  (new List).init(hello,
                   (new List).init(world,
                    (new List).init(newline, nil)))
        in
             ...
    };
};
```

U glavnoj funkciji smo kao lokalne varijable dodali dva objekta: **nil** i **list**. Varijabla **nil** je ne inicijalizirani objekt, koji nam služi kao **NULL** pokazivač – naime u COOL jeziku ne postoji **NULL** pokazivač. Varijabla **list** je sama lista, koja je deklarirana, alocirana i inicijalizirana rekurzivno.

Ovdje treba istaknuti jedno svojstvo deklaracije lokalnih varijabli pomoću ključne riječi **let** – naime lokalnih varijabli može biti više, a deklariraju se odvojene zarezom. Nakon što navedemo lokalnu varijablu, nju možemo koristiti već za slijedeću lokalnu varijablu, iza zareza. Dakle njena deklaracija i alokacija se vrši odmah, ne čeka se da se uđe u blok naredbi iza ključne riječi **in**.

Listu ispisujemo pomoću funkcije **flatten()**:

```
class List {
    ...
    flatten(): String {
```

```
            if( isvoid next ) then
                  item
            else
                  item.concat(next.flatten())
            fi
      };
};
class Main inherits IO {
      ...
            out_string(list.flatten())
      };
};
```

Funkcija **`flatten()`** radi rekurzivno. Njen granični uvjet je da **`next`** član nije alociran. Alokaciju objekta provjeravamo pomoću ključne riječi **`isvoid`**.

3.9 Polimorfizam

Preraditi ćemo klasu **`List`** kako bi mogla sadržavati bilo koji objekt, a ne samo stringove.

```
class List inherits A2I {
      item: Object;
      next: List;
      init(i: Object, n: List): List {
            {
                  item <- i;
                  next <- n;
                  self;
            }
      };
      ...
};
```

Našu listu možemo generalizirati promijenivši tip varijable **`item`** u **`Object`**. Najprije ćemo sve tipove **`String`** promijeniti **u `Object`.**

Potom je potrebno je doraditi funkciju **`flatten()`**:

```
class List inherits A2I {
```

```
...
      flatten(): String {
            let string: String <-
                  case item of
                        i: Int => i2a(i);
                        s: String => s;
                        o: Object => { abort(); ""; };
                  esac
            in
                  if( isvoid next ) then
                        string
                  else
                        string.concat(next.flatten())
                  fi
      };
};
```

Ispis trebamo prilagoditi tipu objekta koji je pohranjen u varijabli **item**. Tip objekta ćemo doznati pomoću grananja **case of**. Ovo je posebna vrsta grananja, koja kao uvjet uzima ulaznu varijablu **item**, te je potom ovisno o njenom tipu vraća varijable **i** ili **s** ili **o**. Promijenjena varijabla će poprimiti sadržaj koji se nalazi sa desne strane operatora **=>**. Ako je varijabla tipa string zadržati će vrijednost koju već sadrži. Ako je tipa **int** izvršiti će se konverzija pomoću funkcije **i2a()**. Ukoliko je varijabla tipa **Object** program se prekida. Za to služi funkcija **abort()**. Iza poziva ove funkcije stavljamo prazan string, kako bi zadovoljili provjeru tipova.

U glavnoj klasi dodati ćemo jedan element liste koji nije string:

```
...
class Main inherits IO {
      main():Object {
            let   hello: String <- "Hello ",
                  world: String <- "World!",
                  i: Int <- 42,
                  newline: String <- "\n",
                  nil: List,
                  list: List <-
                        (new List).init(hello,
```

```
                    (new List).init(world,
                    (new List).init(i,
                    (new List).init(newline, nil))))
            in
                  out_string(list.flatten())
      };
};
```

Dodali smo jedan broj, pa je sada naša lista stvarno polimorfna.