UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Milan Metelko

Simulator nedeterminističnega Turingovega stroja

DIPLOMSKO DELO
VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Ljubljana, 2011

UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Milan Metelko

Simulator nedeterminističnega Turingovega stroja

DIPLOMSKO DELO
VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: pred. dr. Boštjan Slivnik

Ljubljana, 2011

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00096/2011
Datum: 04.04.2011

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **MILAN METELKO**

Naslov: **SIMULATOR NEDETERMINISTIČNEGA TURINGOVEGA STROJA**
A SIMULATOR OF THE NONDETERMINISTIC TURING MACHINE

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija prve stopnje

Tematika naloge:

Turingov stroj je eden izmed temeljnih računskih modelov. Izdelajte program, ki omogoča simuliranje poljubnega nedeterminističnega Turingovega stroja. Pri tem pa natančno opišite, katere razširitve Turingovega stroja vaš simulator podpira, in kakšne omejitve ima vaš simulator glede velikosti simuliranega Turingovega stroja in prostora na traku.

Mentor:
B. Slivnik
pred. dr. Boštjan Slivnik

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:
prof. dr. Nikolaj Zimic

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani Milan Metelko,

z vpisno številko 63060562,

sem avtor diplomskega dela z naslovom:

Simulator nedeterminističnega Turingovega stroja.

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom pred. dr. Boštjana Slivnika,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«.

V Ljubljani, dne 13.10.2011 Podpis avtorja:

ZAHVALA

Ob zaključku diplomskega študija bi se rad zahvalil svojemu mentorju pred. dr. Boštjanu Slivniku za pomoč pri diplomskem delu.

Posebna zahvala gre mojim staršem, ki so me podpirali in mi omogočili študij.

Zahvalil bi se tudi vsem ostalim, ki so mi skozi študijsko leto stali ob strani in mi ob morebitnih vprašanjih vedno bili pripravljeni pomagati.

KAZALO

SEZNAM UPORABLJENIH KRATIC IN SIMBOLOV

API – Application Programming Interface

DTM – Deterministic Turing Machine

DTS – deterministični Turingov stroj

GUI – Graphical User Interface

GUV – grafični uporabniški vmesnik

JDK – Java Development Kit

JRE – Java Runtime Envirorment

NTM – Nondeterministic Turing Machine

NTS – nedeterministični Turingov stroj

R/W – Read/Write

TM – Turing Machine

TS – Turingov stroj

POVZETEK

Na začetku smo predstavili Turingov stroj, njegovo uporabo in funkcionalnost. Ker Turingov stroj pravzaprav ni fizični stroj, smo dodali nekaj primerov njegove uporabe v praksi. Bolj kot je bil zapleten primer, več časa smo potrebovali za njegovo rešitev. Če bi imeli računalniški program, bi lahko čas računanja drastično zmanjšali. In ravno to smo tudi storili. Ustvarili smo grafični uporabniški vmesnik (GUV), ki simulira dejanja Turingovega stroja. V vhodno datoteko napišemo navodila za stroj in simulacija jim sledi. Na koncu nam aplikacija vrne potek simulacije korak za korakom. Tako nam ni treba izračunavati na roko. V nadaljevanju smo pojasnili strukturo aplikacije in globlje metode, ki se skrivajo v ozadju programa. Program simulira šest vrst Turingovih strojev, vključno z deterministčnim (DTS) in nedeterminističnim (NTS) Turingovim strojem, kjer je vsak simuliran na svoj poseben način.

Ključne besede:

GUV, Turingov stroj, Java, neskončen trak, nedeterminističen, simulacija

ABSTRACT

The diploma work presents the Turing machine, its use and functionality. The Turing machine is not actually a physical machine, that's why we added a few examples of its use in practice. The more complex the case, the more time we needed to solve it. If we had a computer programme, the calculating time would be reduced drastically. And that is exactly what we did. We have created a graphical user interface (GUI) which simulates the actions of the Turing machine. The instructions for the machine are written in the input file, they are followed by the simulation of the machine. At the end the application displays the simulation sequence step by step. Thus we do not need to calculate by hand. Further on the structure of the application and the deeper methods that are hidden in the background of the programme are explained. The programme simulates six types of Turing machines, including the deterministic (DTM) and nondeterministic (NTM) Turing machines, all simulated in their own special ways.

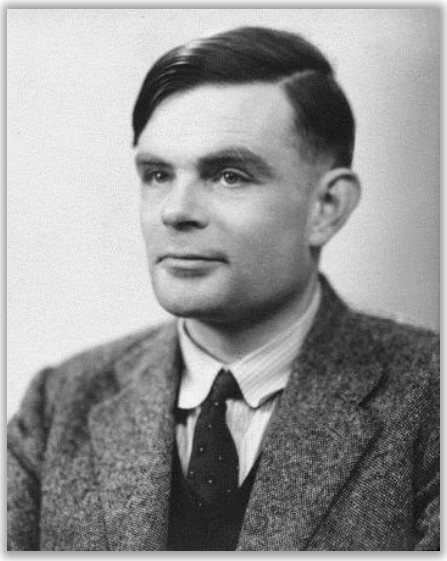
Keywords:

GUI, the Turing machine, Java, infinite tape, nondeterministic, simulation

1 UVOD

1.1 Zgodovina Turingovega stroja

Alan Mathison Turing [1] se je rodil 23. 6. 1912 v Londonu (slika 1). Že zelo mlad je s preprostimi kemijskimi poskusi pokazal zanimanje za znanost. Pozneje, ko se je udejanjal kot raziskovalec, se je uveljavil tudi kot pionir računalniške znanosti. Leta 1928 je začel študirati relativnost. Vprašanje »Kako je človeški um vključen v materijo?« ga je vodilo na študij fizike dvajsetega stoletja, kjer ga je zanimala povezava kvantne mehanike in njegovega vprašanja o materiji in misli. Leta 1931 je bil sprejet na kraljevo univerzo v Cambridgeu. Tu je tudi spoznal von Neumannovo delo. Zelo ga je zanimalo vprašanje izračunljivosti, zato je raziskoval napravo, ki bi bila zmožna rešiti vse rešljive probleme. Ta naprava se danes imenuje Turingov stroj [2].



Slika 1: Alan Mathison Turing (1912–1954)

Med drugo svetovno vojno je svoje znanje uveljavljal na oddelku za komunikacije Velike Britanije in poskušal razbiti nemške kode, ki so jih Nemci uporabljali za komunikacijo. To je bila zelo zahtevna naloga, saj je kodiranje opravljal računalnik, ki so ga Nemci zasnovali prav za ta namen. Imenoval se je Enigma. Skupina, v kateri je delal tudi Turing, je odgovorila s Colossusom, enoto, ki je hitro in učinkovito razbila Enigmino kodo. Colossus je bil prvi korak k današnjim digitalnim računalnikom.

Po vojni je Turing služboval v državnemu fizikalnemu laboratoriju, da bi nadaljeval z razvojem računalnika. Junija leta 1954 je umrl. V današnjem času obravnavamo Turinga kot enega od najpomembnejših začetnikov računalništva.

Svoj matematični model, danes tako imenovan Turingov stroj, je Alan Turing objavil leta 1936. Glavni ideji Turingovega razmišljanja se glasita:

- Matematična funkcija je izračunljiva, če jo je mogoče v končnem številu korakov izračunati na enem od Turingovih strojev.
- Funkcij je več kot Turingovih strojev oziroma obstajajo tudi neizračunljive funkcije.

Vsak Turingov stroj je mogoče na preprost način opisati z uporabo mehaničnih delov [3], kot so trak, bralno-pisalna glava in mehanizem za pomik po traku. Vsak Turingov stroj vsebuje naslednje dele:

- Trak, ki je neskončen po velikosti. Je enodimenzionalen in razdeljen v zaporedje enakih kvadratkov. Vsak kvadrat je sposoben vsebovati samo en simbol iz končne množice ali pa je prazen. Čeprav je trak neskončen v dolžino, imamo na njem zapisano končno število simbolov oziroma znakov. Za vse kvadratke, ki ostanejo nezapisani, se predvideva, da so prazni. Trak se uporablja samo za branje in pisanje simbolov oziroma znakov.
- Program, ki je zaporedno končno število navodil oziroma ukazov. Program ukaže glavi, kaj naj piše in kam naj se premakne, glede na podatek na traku ter trenutno stanje programa. Turingov stroj upošteva navodila v istem vrstnem redu, v katerem se pojavljajo. Ko ni več pravila za kombinacijo stanje-simbol, ki jo Turingov stroj sreča, se bo stroj enostavno ustavil in ne bo izvršil nobenega ukaza več.
- Bralno-pisalna glava, ki v vsakem trenutku bere določen kvadratek na traku in opravlja ukaze programa v določenem koraku. Lahko bere simbol s traka in glede na ta simbol in trenutno stanje bo napisala drug simbol čezenj. Bralno-pisalna glava se lahko premakne na desno ali pa levo vzdolž po neskončnem traku.

Elementi Turingovega stroja so predstavljeni povsem mehansko, da lažje razumemo, za kaj gre. V praksi so Turingovi stroji največkrat predstavljeni kot računalniški programi, Turingov stroj pa lahko, podobno kot von Neumannovo arhitekturo, pojmujemo kot model računalnika, ki zna vse, kar je mogoče vedeti oziroma znati.

Vedeti pa moramo, da mehanična ali sploh kakršnakoli fizična osnova stroja ni bistvena in je samo sredstvo za lažje razumevanje Turingove ideje, ki je v bistvu postopek za matematično dokazovanje. Rečemo lahko, da je Turingov stroj matematični pojem. Mehanična metafora je bila v pomoč Turingu samemu, ki je bil med drugim tudi eden od pionirjev pri razvoju digitalnih računalnikov.

1.2 Matematična definicija Turingovega stroja

Obstaja veliko različic definicije Turingovega stroja [4], saj obstajajo različne razširitve Turingovega stroja, ki imajo drugačne lastnosti in temu primerno različne matematične definicije [5]. Omenili bomo tisto, ki je najbolj pogosto uporabljena [6] in najbolj razširjena oblika zapisa formulacije, ki se glasi $M = \langle Q, \Gamma, b, \Sigma, \delta, q_0, F \rangle$, kjer je pomen simbolov sledeč:

- Q je končna množica stanj,
- Γ je končna abeceda (to so znaki, ki se lahko pojavijo na traku),
- $b \in \Gamma \setminus \Sigma$ ($b \equiv$ blank; oznaka za prazno polje),
- $\Sigma \subseteq \Gamma \setminus \{b\}$ je množica veljavnih vhodnih znakov, ki jih lahko zapišemo na trak,
- $\delta: Q \setminus F \times \Gamma \to Q \times \Gamma \times \{L, R\}$ je prehodna funkcija,
- $q_0 \in Q$ je začetno stanje,
- $F \subseteq Q$ je množica končnih stanj.

Prehodna funkcija je potem definirana kot $\delta(q, s) = (q', s', \Delta)$, kjer je q trenutno stanje stroja, s trenutni znak na traku, q' novo stanje stroja, s' nov znak, Δ pa smer premika po traku [7].

Vzemimo primer Turingovega stroja $M = \langle Q, \Gamma, b, \Sigma, \delta, q_0, F \rangle$, pri katerem imamo lastnosti: $Q = \{q_0, q_1\}$, $\Gamma = \{0,1\}$, $b = B$, $\Sigma = \{0,1\}$, $q_0 = q_0$, $F = \{q_1\}$. Njegove prehodne funkcije pa so: $\delta(q_0, 0) = (q_0, 1, R)$, $\delta(q_0, 1) = (q_0, 0, R)$ in $\delta(q_0, B) = (q_1, 0, R)$. Če poenostavimo delovanje tega Turingovega stroja, lahko rečemo, da negira vsa števila na traku.

Predpostavimo, da imamo na traku zaporedje znakov `00110`. Delovanje Turingovega stroja bi potekalo na sledeči način. Začetno stanje stroja je q_0 in na traku je zapisan znak `0`, zato stroj izbere prvo prehodno funkcijo. Stroj gre v novo stanje q_0, na trak zapiše znak `1` in se premakne v desno stran. Naslednji korak je enak prvemu. Stroj je v stanju q_0 in na traku je spet zapisan znak `0`, zato stroj spet izbere prvo prehodno funkcijo. Stroj gre v novo stanje q_0, na trak zapiše znak `1` in se premakne v desno stran. Naslednji korak pa je malo drugačen kot prejšnji. Stroj je še vedno v stanju q_0, ampak tokrat je na traku zapisan znak `1`, zato stroj izbere drugo prehodno funkcijo. Spet gre v novo stanje q_0, ampak na trak zapiše znak `0` in se premakne v desno stran. Naslednji korak je enak prejšnjemu. Stroj je v stanju q_0 in na traku je spet zapisan znak `1`, zato stroj spet izbere drugo prehodno funkcijo. Stroj gre v novo stanje q_0, na trak zapiše znak `0` in se premakne v desno stran. Naslednji korak pa je spet tak, kot je bil prvi. Stroj je še vedno v stanju q_0, na traku je zapisan znak `0`, zato stroj izbere prvo prehodno funkcijo. Še zadnjič gre v novo stanje q_0, na trak zapiše znak `1` in se premakne v

desno stran. Zdaj pa mu je zmanjkalo zapisanih znakov, saj je prišel na polje, ki je prazno, kar označimo z znakom B. Stroj je še vedno v stanju q_0, ampak zaradi praznega polja tokrat izbere tretjo prehodno funkcijo. Zdaj se mu tudi spremeni stanje, saj gre v novo stanje q_1. Na trak zapiše znak `0` in nato zaključi z delovanjem, saj je stroj prišel v stanje q_1, ki pa je definirano kot njegovo končno stanje.

Čeprav smo imeli zelo preproste definicije Turingovega stroja, smo vseeno porabili kar nekaj časa, da smo rešili naveden primer. Hkrati pa tudi nismo imeli veliko znakov na traku. Če bi imeli še veliko več različnih prehodnih funkcij in zapisanih znakov na traku, bi porabili zelo veliko časa in še verjetnost napake bi bila večja. Zato smo napisali program, ki simulira delovanje Turingovega stroja, in s tem uporabniku zelo olajšali delo.

Definicije Turingovega stroja zapišemo v vhodno datoteko, ki jo program sprejme in vrne rezultat simulacije. Vsebina vhodne datoteke za pravkar razloženi primer prikazuje slika 2.

```
q0 0 -> q0 1 R
q0 1 -> q0 0 R
q0 B -> q1 0 R
start q0
stop q1
```

Slika 2: Vsebina vhodne datoteke

Zapis v vhodni datoteki je precej enak formalnemu zapisu prehodnih funkcij. Poleg teh funkcij pa zraven dodamo še začetno in končno stanje Turingovega stroja.

2 SIMULATOR TURINGOVEGA STROJA

2.1 Delovanje Turingovega stroja

Za lažjo predstavo bomo predpostavili, da imamo nek stroj po imenu Turingov stroj, skozi katerega je speljan neskončno dolg trak [8]. Poleg traku pa stroj vsebuje tudi bralno-pisalno glavo, s katero se pomika po traku in bere ali piše podatke (slika 3). Trak je razdeljen na majhna polja in v vsakem polju je lahko zapisan le en znak. Ta znak z drugimi besedami imenujemo tudi tračni simbol. Polje pa je lahko tudi prazno, kar pomeni, da na traku ni nič napisanega. Stroj lahko v danem trenutku bere le iz enega polja in ravno tako zapisuje le v eno polje, ki pa je hkrati isto polje, ki ga je ravnokar prebral.

Slika 3: Simbolični model Turingovega stroja

Da pa stroj ve, kateri znak mora zapisati v polje, potrebujemo nekakšna navodila, ki so shranjena v stroju. Ta navodila povedo, pod kakšnimi pogoji naj bo zapisan določen znak v danem trenutku oziroma položaju Turingovega stroja. Ker pa imamo neskončen trak, žal ne moremo številsko določiti trenutnega položaja na traku, zato stroj vsebuje lastnost, ki se imenuje stanje. Skozi potek branja in pisanja po traku se ta stanja spreminjajo, saj med delovanjem prehajamo iz enega stanja v drugo stanje. Včasih pa se tudi zgodi, da gremo iz določenega stanja znova v isto stanje. Tako imamo v danem trenutku v stroju shranjeno trenutno stanje in tračni simbol. Ko pridobimo omenjena podatka, sledimo nadaljnjim navodilom, ki so zapisana v stroju. Nekakšne vrste obrazec za navodila bi se lahko glasil tako: če je stroj v določenem stanju in če smo prebrali določen tračni simbol, potem naj gre stroj v neko novo stanje, hkrati pa naj na trak zapiše nek nov tračni simbol in nazadnje naj nadaljuje po traku v določeno smer. Za lažje razumevanje lahko podamo konkreten primer: če je stroj v stanju `q0` in če smo prebrali `0`, potem naj gre stroj v stanje `q1`, hkrati pa naj na trak zapiše `1`

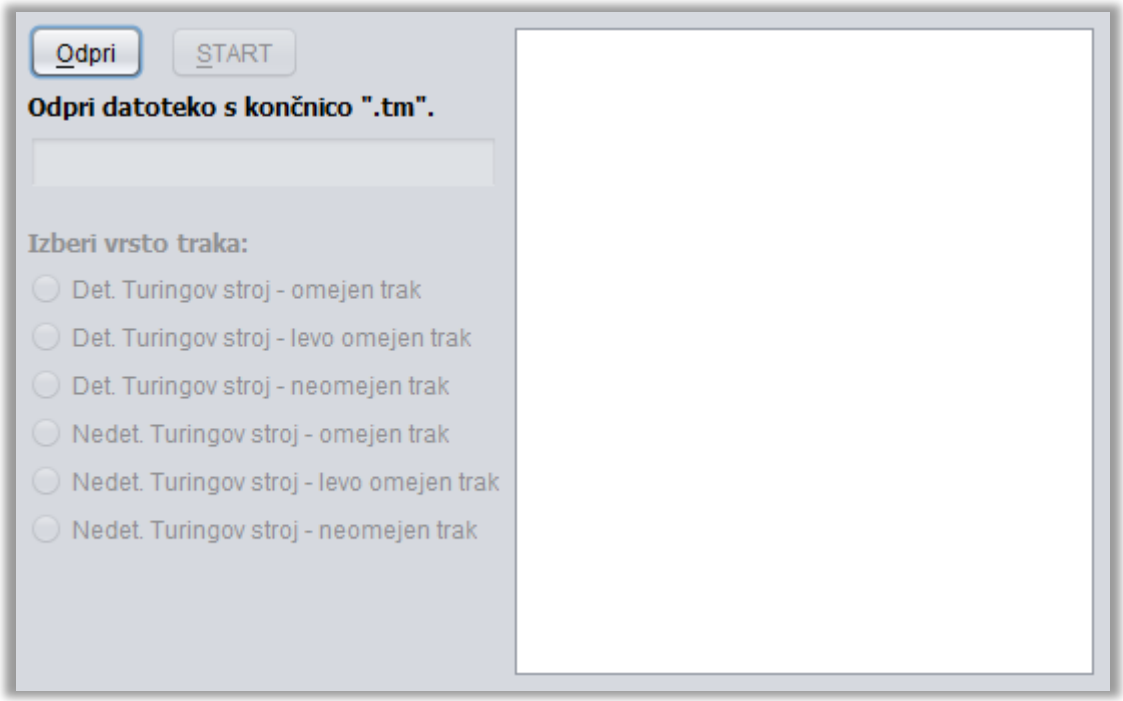
in nazadnje naj nadaljuje po traku v desno smer. Takšen postopek delovanja bi se v navodilih glasil tako: `q0 0 -> q1 1 R`. Več o oblikovanju navodil pa pozneje v poglavju 2.2.2. Omenjen postopek je le eden izmed mnogih možnih navodil, ki so lahko shranjena v stroju. Za podanim navodilom bi nato sledilo neko novo navodilo in iz novega spet neko drugo navodilo. Tako bi se nadaljevalo, vse dokler stroj ne bi prišel do nekega točno določenega stanja, ki bi povzročilo, da se stroj ustavi.

2.2 Opis uporabniškega vmesnika

Uporabniški vmesnik [9] smo zasnovali tako, da je uporabniku prijazen in je enostaven za uporabo, saj lahko vsak uporabnik z zelo malo truda kmalu ugotovi, kakšen je naslednji korak, ki ga mora storiti. K povečanju uporabnosti oziroma k zmanjšanju verjetnosti napak smo pripomogli tudi s tem, da smo ob določenih korakih zasenčili del programa in s tem uporabniku onemogočili dostop do nekaterih stvari, ki jih trenutno še ne potrebuje. Ob morebitni napaki ali nepravilni izbiri pa uporabnika obvestimo z različnimi napisi ali predlogi in ga s tem usmerjamo k pravilni uporabi aplikacije.

2.2.1 Začetni prikaz uporabniškega vmesnika

Zgoraj omenjeni onemogočen dostop je zelo opazen že pri prvi uporabi aplikacije. V začetnem oknu se lahko opazijo možnosti izbire in prazna polja, ampak edina stvar, ki jo lahko naredimo, je ta, da kliknemo na gumb `Odpri`. Vse ostale možnosti so zasenčene in onemogočene, kar tudi prikazuje slika 4. Za začetno pomoč uporabniku smo tudi dodali besedilo `Odpri datoteko s končnico ».tm«.`, da uporabnika napotimo v pravo smer.

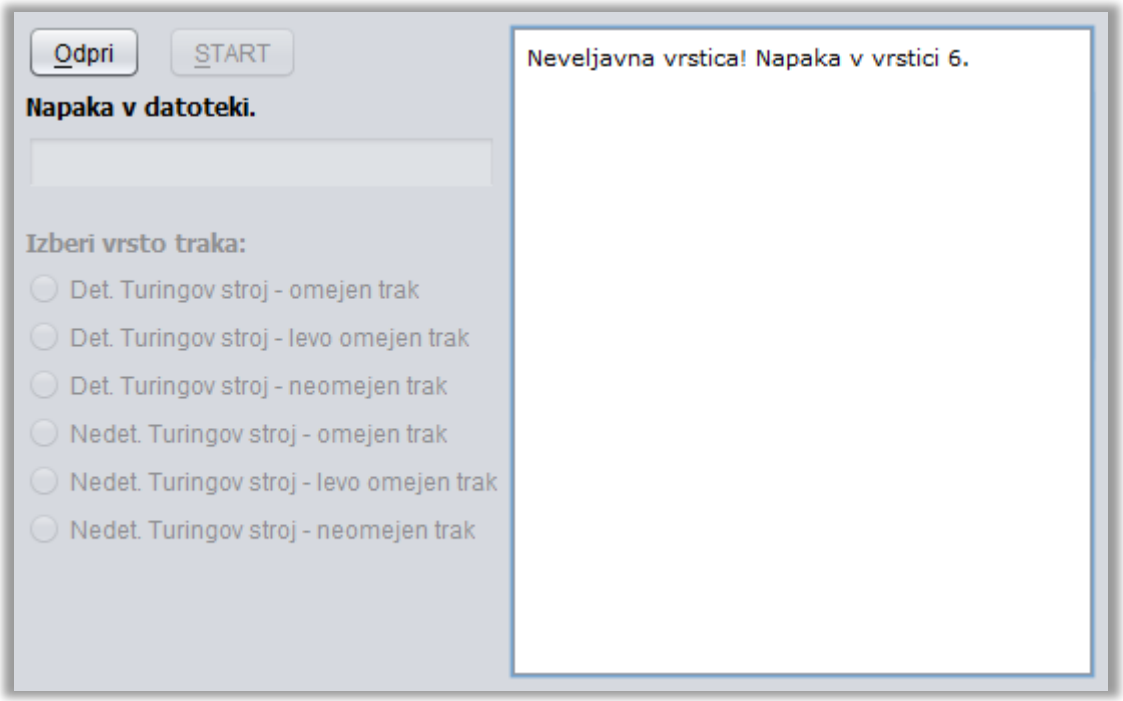


Slika 4: Začetni prikaz uporabniškega vmesnika

Kot lahko opazimo, program za svoj vhod potrebuje datoteko s točno določeno končnico, to je `.tm`. Končnico smo izpeljali iz angleškega izraza *Turing Machine*, kjer smo vzeli začetnice omenjenih besed. Kljub začetnemu opozorilu pa se lahko zgodi, da uporabnik vseeno izbere datoteko z drugačno končnico. V tem primeru se namesto napisa `Odpri datoteko s končnico ».tm«.` pojavi napis `Napačna datoteka.`, s čimer uporabnika opozorimo, da naj znova poskusi izbrati datoteko s pravilno končnico.

2.2.2 Vsebina vhodne datoteke

Pomembna pa ni zgolj končnica, ampak tudi vsebina izbrane datoteke. Napisana mora biti po točno določenih pravilih, ki jih je treba upoštevati. V nasprotnem primeru datoteka spet ni sprejemljiva in program nas na to tudi opozori. Namesto napisa `Odpri datoteko s končnico ».tm«.` se pojavi napis `Napaka v datoteki.`. Hkrati pa se v desnem polju izpiše opozorilo o nepravilno napisani datoteki. Poleg tega izpisa pa zraven tudi podamo številko vrstice, v kateri je napaka, da jo lahko uporabnik prej odpravi. Primer takšnega izpisa bi lahko bil `Neveljavna vrstica! Napaka v vrstici 6.` (slika 5). Napaka se izpiše tudi v primeru nepopolne vsebine, zato je nujno, da napišemo vse potrebne informacije, ki jih Turingov stroj potrebuje.



Slika 5: Izpis napake

Ena izmed pomembnih informacij je ta, da stroj ve, kje mora začeti. Zato potrebuje začetni ukaz `start`, za tem ukazom pa še napisano stanje, v katerem stroj začne izvajati svoj postopek, se pravi neko začetno stanje stroja. Vsako stanje ima točno določeno obliko. Vedno se začne s črko `q` in takoj za njo napisano nenegativno število, katerega velikost ni pomembna. Tako bi lahko za neko stanje izbrali `q1232`, ampak za lepšo preglednost bomo za primer izbrali začetno stanje `q0`. S tem smo sestavili vrstico `start q0`, v kateri povemo, da naj Turingov stroj začne v stanju `q0`. Poleg začetnega stanja moramo podati tudi končno stanje, ki pa se od začetnega razlikuje le v tem, da je namesto besede `start` napisana beseda `stop` in iz tega dobimo primer končnega stanja `stop q2`.

Za delovanje programa pa ni dovolj, da le napišemo začetno in končno stanje, ampak potrebujemo tudi vmesna stanja. Primer enega vmesnega stanja smo že zasledili v poglavju 2.1. Vsako vmesno stanje ima poleg imena stanja dopisane tudi druge lastnosti. Takoj za stanjem navedemo vsaj en presledek in za njim dopišemo tračni simbol, ki ga trenutno preberemo s traku. Tračni simbol pa ima malo drugačno sestavo kot stanja. Sestavljen mora biti iz le enega znaka in ta znak je lahko nenegativno število od 0 do 9 ali mala tiskana črka od a do z po angleški abecedi ali pa znak `B`, s katerim označimo, če je polje prazno – na traku ni zapisanega simbola, zato tu navedemo `B` (*Blank*). Za tračnim simbolom navedemo presledek ali več presledkov, lahko pa tudi izpustimo presledek. Pomembno je, da nato navedemo puščico `->`, ki jo dobimo iz kombinacij znakov `-` in `>`, med katerima pa ne smemo

napisati presledkov. Za napisano puščico pa lahko znova zapišemo poljubno število presledkov, ampak zaradi lepše preglednosti običajno tu napišemo le enega. Zapis nadaljujemo z novim stanjem, v katerega naj gre stroj. Način zapisa tega stanja je isti kot pri zapisu začetnega stanja, saj imajo vsa stanja iste lastnosti zapisa, se pravi črka `q`, ki ji sledi poljubno nenegativno število. Takoj za stanjem navedemo vsaj en presledek in za njim dopišemo nov tračni simbol, ki ga želimo napisati na trak. Lastnosti napisanega tračnega simbola so enake lastnostim prebranega tračnega simbola. Nato spet navedemo vsaj en presledek in nazadnje za njim še napišemo smer, kamor naj se premaknemo po traku. Za smer sta sprejemljivi črka `L` za pomik v levo smer (*Left*) in črka `R` za pomik v desno smer (*Right*). Ni pa vedno nujno, da se pomaknemo v kakšno smer, namesto tega lahko tudi ostanemo na istem mestu, kot smo, kar pa povemo s tem, da namesto črk `L` in `R` napišemo `-`.

Ob upoštevanju navedenih navodil smo sestavili eno veljavno vrstico, ki jo lahko zapišemo v našo vhodno datoteko. Za primer takšne vrstice lahko vzamemo `q0 0 -> q1 1 R`. Kot smo omenili že prej, pa gre stroj lahko iz trenutnega stanja znova v isto stanje, zato je veljaven tudi zapis `q0 0 -> q0 0 R`. V tem zapisu imamo stanje stroja `q0` in na traku zapisan znak `0`. Nato gre stroj iz stanja `q0` znova v stanje `q0`, namesto znaka `0` na trak zapiše `0` in se pomakne v desno stran.

Do zdaj smo omenili in opisali le primera, kjer gre Turingov stroj iz enega stanja v drugo točno določeno stanje, na primer pri primeru `q0 0 -> q1 1 R` lahko preidemo iz stanja `q0` le v točno določeno stanje, to je stanje `q1`. V primeru, da so vse vrstice napisane na takšen način, potem rečemo, da je ta Turingov stroj deterministično.

Poznamo pa tudi nedeterministično Turingov stroj, kjer prehod stanj ni točno določen oziroma obstaja izbira med različnimi stanji. Če vzamemo primer `q0 0 -> q1 1 R`, ki je deterministično, ga lahko nadgradimo in spremenimo v nedeterminističnega s tem, da mu dodamo novo stanje, zapis na trak in smer, na primer: `q2 0 L`. To naredimo tako, da obstoječi vrstici dodamo vejico in za njo dopišemo novo izbiro. Pred in za vejico je lahko poljubno število presledkov, najbolje pa je, če je eden na vsaki strani vejice. Tako bi bila vrstica zapisana kot `q0 0 -> q1 1 R , q2 0 L`. V takšnem primeru imamo na voljo dve različni izbiri. Iz stanja `q0` gremo lahko v stanje `q1` ali pa v stanje `q2`. Če bi se odločili, da gremo v stanje `q1`, potem je rezultat isti kot pri deterministično zapisu, saj bi na trak spet zapisali `1` in šli desno. V primeru, da bi se odločili za stanje `q2`, pa bi bil rezultat malo drugačen, saj bi na trak zapisali `0` in nato bi se pomaknili levo. Poleg ene dodatne izbire bi jih lahko po istem postopku dodali še več in s tem še bolj povečali raznovrstnost izbire. Primer z različnimi izbirami bi lahko bil `q0 0 -> q1 1 R , q2 0 L , q3 0 R , q4 1 L`.

Vrstni red podajanja različnih izbir ni pomemben. Bolj moramo biti pazljivi, da ne združimo različnih trenutnih stanj stroja oziroma da ne združimo različnih pravil. Kadar navedemo novo pravilo, ga moramo zapisati v novo vrstico. Vrstni red vrstic oziroma pravil ni pomemben, prav tako ni pomembno, v kateri vrstici imamo zapisan `start` in `stop`. Pomembno je le, da je vsaka stvar v svoji vrstici. Čeprav vrstni red ni pomemben, zaradi boljše preglednosti vseeno upoštevamo sledeča navodila. Sprva napišemo vse različne prehode stanj drugega pod drugim, vsakega v svojo vrstico. Zatem v novi vrstici navedemo `start` in začetno stanje ter čisto na koncu v novi vrstici navedemo še `stop` in končno stanje. Poleg tega moramo še upoštevati, da pred začetkom in ob koncu vrstice ne dajemo odvečnih presledkov, saj v nasprotnem primeru program ne bo deloval. Če pa imamo željo, lahko vrstici `start` in `stop` zapišemo na začetku dokumenta in ne na koncu. Slika 6 prikazuje primer pravilnega in napačnega zapisa vhodne datoteke.

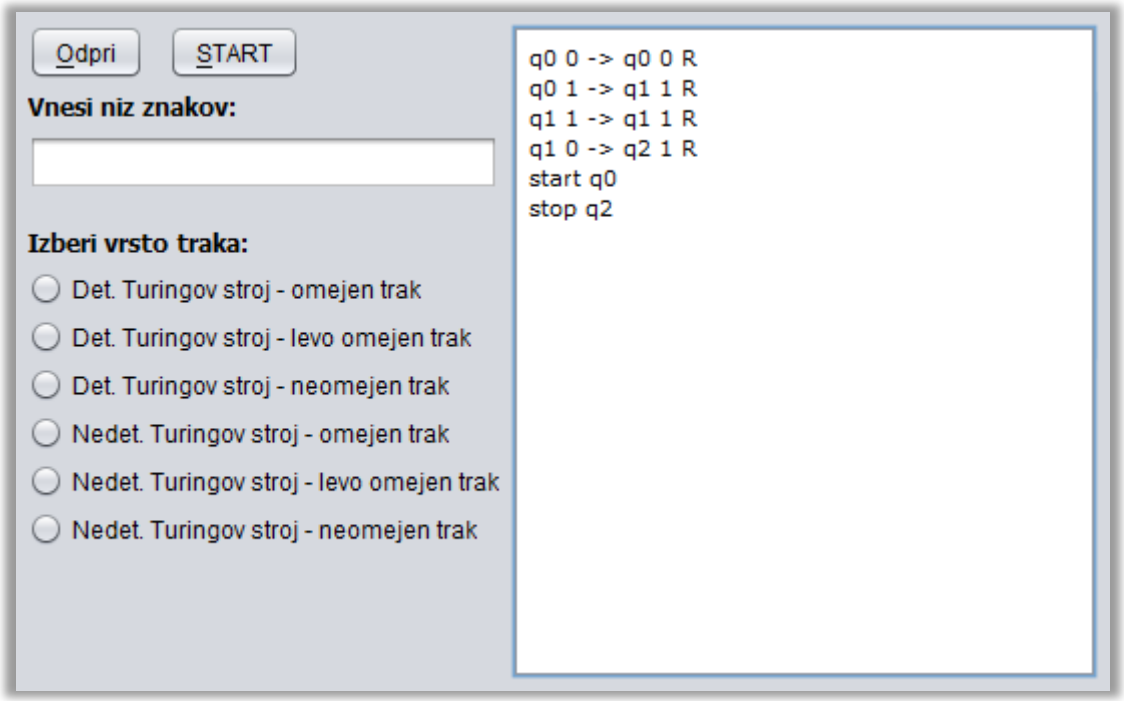
```
q0 0 -> q0 0 R
q0 1 -> q1 1 R
q1 1 -> q1 1 R
q1 0 -> q2 1 R
start q0
stop q2
```

```
q0 0 <- q0 0 R
q0 1 -> q1 1R
q1 1 -> q1 1 R
q1 0 -> q2 R
start q0
stop
```

Slika 6: Pravilen (levo) in napačen (desno) zapis vhodne datoteke

Pri desnem napačnem zapisu opazimo kar nekaj napak, zaradi katerih program ne bi sprejel takšne vhodne datoteke. V prvi vrstici je puščica obrnjena v napačno smer, kar ni dovoljeno. V drugi vrstici manjka presledek med novim stanjem in znakom za pomik po traku. Naslednja vrstica je napisana pravilno, nato pa je spet nesprejemljiva vrstica, saj manjka zapis novega stanja. Tudi naslednja vrstica je napačna, čeprav na prvi pogled ni videti tako. Napaka je v nevidnih presledkih, ki so zapisani desno od zapisa končnega stanja. V zadnji vrstici pa manjka zapis končnega stanja. Navedli smo le nekaj napak, na katere moramo biti pozorni, ko pišemo vsebino vhodne datoteke.

Ko napišemo želeno vsebino, datoteko shranimo s končnico `.tm` in jo poskusimo odpreti v aplikaciji. Ob morebitni napaki smo opozorjeni in jo poskusimo odpraviti. V primeru, da smo pravilno napisali datoteko, pa se nam vsebina datoteke izpiše v polju na desni strani (slika 7).



Slika 7: Izpis vsebine odprte datoteke

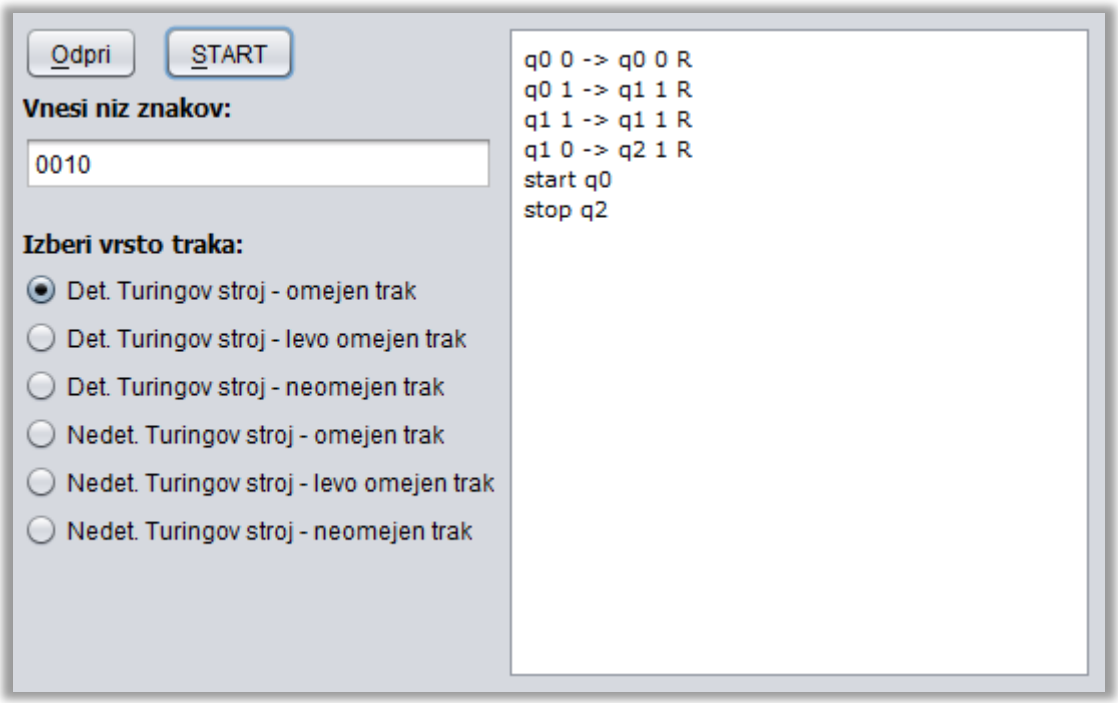
Poleg izpisa pa se nam tudi omogočijo nove možnosti, ki jih lahko zdaj izberemo, uporabimo ali v polje vnesemo besedilo. Dejanje, ko smo odprli datoteko, bi lahko enačili z besedami, da smo v Turingov stroj vnesli navodila, kaj naj stroj stori, ko se pomika po traku. Z drugimi besedami: ustvarili smo svoj edinstven Turingov stroj, ki opravlja stvari tako, kot smo mu mi ukazali. Kar pomeni, da se premika po traku tako, kot mi želimo, in da na trak zapisuje znake, ki smo mu jih mi podali.

2.2.3 Branje in pisanje po traku

Zaradi pravilne strukture vhodne datoteke se nam je pod gumbom `Open` odprlo prazno polje, v katerega lahko vpišemo niz znakov. Nad poljem se je prav tako pojavilo nadaljnje navodilo `Vnesi niz znakov:`.

Lahko bi predpostavili, da na traku nimamo napisanega ničesar, in nanj le pisali določene znake, ampak s tem ne bi popolnoma izkoristili zmožnosti Turingovega stroja, saj bi uporabili le pisanje, branje bi pa zanemarili. Zato bi stroju radi nekako povedali, katere stvari so že zapisane na traku. In ravno to napišemo v polje, ki nam je zdaj na voljo. Vanj vpišemo niz znakov, za katere predpostavimo, da so že zapisani na traku, na primer `0010`. Pod tem nizom pa imamo na voljo izbiro med različnimi vrstami Turingovih strojev. Ker imamo precej enostaven primer, se odločimo za osnovni deterministični Turingov stroj. Zdaj smo opravili

vse, kar je bilo treba storiti iz uporabnikove strani. Preostane nam le še, da kliknemo na gumb `START` in počakamo na rezultat (slika 8).

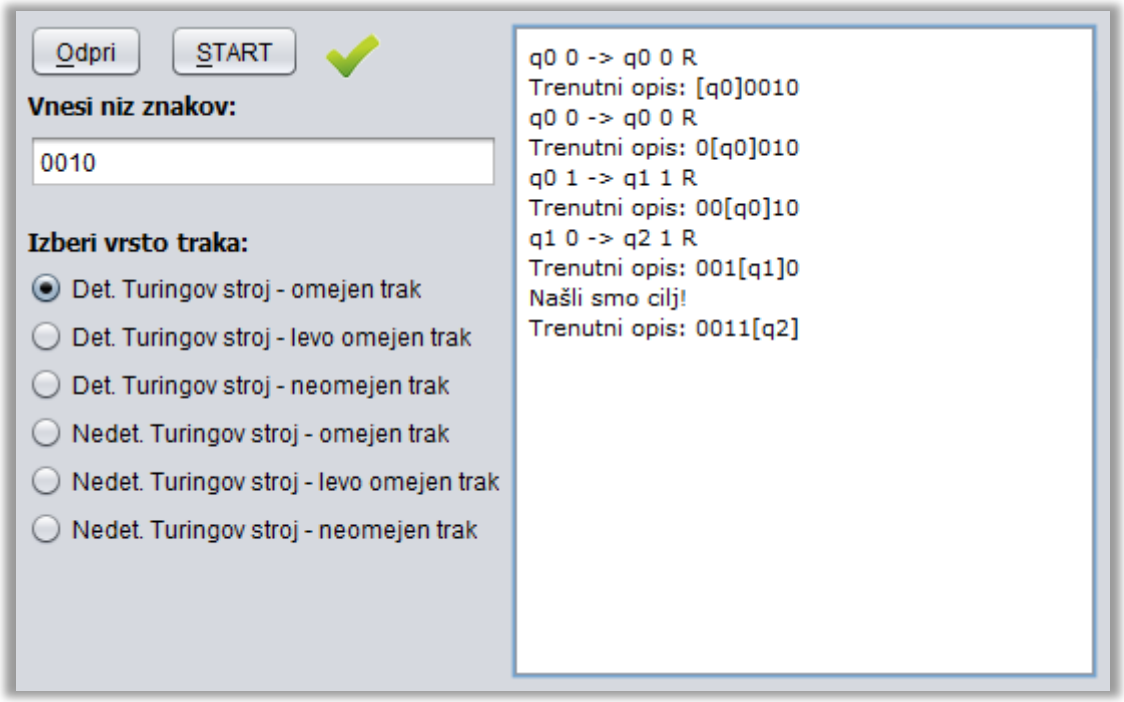


Slika 8: Prikaz izbire pred začetkom simulacije

Program začne simulirati delovanje Turingovega stroja, katerega ukaze smo podali z vhodno datoteko, ki smo jo na začetku odprli. Vsak korak, ki bi ga stroj naredil, se izpiše v desnem polju, da lažje spremljamo delovanje stroja in da hitreje ugotovimo, kako je stroj prišel do določenih rezultatov. V primeru, da imamo vhodne podatke podane tako, kot prikazuje slika 8, bi bil postopek delovanja sledeč.

Sprva najdemo ukaz `start` in poleg njega preberemo začetno stanje `q0`. Nato preberemo prvi znak, ki je zapisan na traku, to je znak `0`. Pridobljena podatka združimo, dobimo `q0 0`, kar tudi poiščemo v eni izmed vrstic naše vhodne datoteke. Ker imamo podano trenutno stanje, gledamo zapis, ki je na levi strani puščice. Najdemo ga že na začetku v prvi vrstici `q0 0 -> q0 0 R`. Iz te vrstice razberemo, da gre stroj iz prejšnjega stanja `q0` v novo stanje `q0`, na trak namesto `0` napišemo `0` in nato se pomaknemo po traku desno. V bistvu nismo naredili nobene opazne razlike, razen tega, da smo se prestavili po traku za eno polje v desno. Na traku preberemo nov znak, ki je spet `0`. Skupaj sestavimo trenutno stanje stroja `q0` in prebran znak `0` ter dobimo `q0 0`, kar poiščemo v vhodni datoteki, in spet uporabimo prvo vrstico `q0 0 -> q0 0 R`. Tudi tokrat razberemo, da gre stroj iz stanja `q0` v stanje `q0`, na trak namesto `0` napišemo `0` in nato se pomaknemo po traku desno. Spet nismo naredili nobene opazne razlike, razen tega, da smo se prestavili po traku za eno polje v desno. Tokrat pa je v

nasprotju s prejšnjim stanjem na traku zapisan znak `1`. Skupaj sestavimo trenutno stanje stroja `q0` in na novo prebran znak `1` ter dobimo `q0 1`, kar tudi poiščemo v vhodni datoteki. Tu zdaj uporabimo drugo vrstico `q0 1 -> q1 1 R`. Tokrat razberemo, da gre stroj iz stanja `q0` v novo drugačno stanje `q1`, na trak pa namesto `1` napišemo `1` in nato se pomaknemo po traku desno. Preberemo znak na traku, ki je spet `0`. Tudi tokrat sestavimo trenutno stanje stroja, ki je zdaj `q1` in na novo prebran znak `0` ter dobimo `q1 0`, kar tudi poiščemo v vhodni datoteki. Tu zdaj uporabimo četrto vrstico `q1 0 -> q2 1 R`. Tokrat razberemo, da gre stroj iz stanja `q1` v novo stanje `q2`, na trak pa namesto `0` napišemo `1`. V nasprotju s prejšnjim stanjem tokrat na trak zapišemo drugačen znak, kot smo ga prebrali. Namesto predhodnega zapisa na traku `0010` imamo zdaj nov zapis `0011`. Poleg tega pa je zdaj stroj prišel v stanje `q2`, ki pa je pravzaprav tudi končno stanje, kar lahko razberemo iz vrstice `stop q2`. Zaradi prehoda v končno stanje se zdaj stroj ustavi. Ker je stroj brez kakršnihkoli napak prišel v končno stanje, temu rečemo, da je bil postopek uspešen, in v polje dopišemo `Našli smo cilj!`. V samem programu pa ni priporočljivo takšno razlaganje, zato so zapisani le bistveni podatki za razumevanje delovanja stroja (slika 9).



Slika 9: Izpis po uspešno končani simulaciji

Sprva iz vhodne datoteke izpišemo vrstico, ki jo stroj trenutno potrebuje oziroma s pomočjo katere se izvaja trenutni ukaz. Na primer iz vrstice `q0 0 -> q0 0 R` lahko razberemo vse, kar potrebujemo: trenutno stanje stroja, trenuten zapis na traku, novo stanje stroja, nov zapis na traku in nazadnje še smer premika po traku. Nato je izpisana vrstica, ki se začne z

besedilom `Trenutni opis:` in nadaljuje z izpisom trenutnega zapisa na traku. V omenjeni izpis pa vmes vrinemo oglati oklepaj in zaklepaj ter med njiju napišemo trenutno stanje stroja. Oglata oklepaja in trenutno stanje se skozi postopek simulacije vriva v različne pozicije zapisa in s tem ponazarja premikanje bralno-pisalne glave po traku v Turingovem stroju. Primer takšnega izpisa je `Trenutni opis: 00[q0]10`, iz katerega lahko razberemo, da je stroj trenutno v stanju `q0` in da je bralno-pisalna glava nad poljem z znakom `1`, se pravi znakom, ki je na desni strani oglatega zaklepaja.

2.2.4 Različice Turingovega stroja

Glavna lastnost Turingovega stroja je ta, da imamo trak, po katerem se pomikamo in iz katerega beremo znake in nato nanj tudi zapišemo nov znak. Ampak ta lastnost se lahko nadgradi z različnimi novimi lastnostmi, ki dodatno definirajo vedenje Turingovega stroja. Te dodatne lastnosti pa se od stroja do stroja razlikujejo, zato dobimo več različic Turingovega stroja. V simulaciji Turingovega stroja smo uporabniku ponudili šest različnih možnosti, med katerimi lahko izbira. Za predhodni primer smo izbrali prvega, najbolj osnovnega. Vseh šest različic, med katerimi lahko izbiramo, prikazuje slika 9. Vsaka izmed izbir ima svoje lastnosti, zaradi katerih se razlikuje od drugih različic:

- Prva izbira `Det. Turingov stroj - omejen trak` določi stroju lastnost deterministočnosti, kar pomeni, da gre lahko stroj iz enega določenega stanja v le eno drugo točno določeno stanje. V primeru `q0 1 -> q1 1 R` gre lahko stroj iz stanja `q0` le v stanje `q1`, kar mu določi deterministočnost. Poleg te lastnosti pa imamo podano še lastnost traka, po katerem se stroj sprehaja. Lastnost `omejen trak` pomeni, da imamo na traku končno število polj. Z drugimi besedami lahko rečemo, da je trak omejen na levo in desno stran. Traku dejansko določimo njegovo dolžino, ko v polje vnesemo niz znakov, kot je na primer `0010`. Pri takšnem nizu znakov povemo, da ima trak le štiri polja, kjer je v prvem polju napisan znak `0`, v drugem znak `0`, v tretjem znak `1` in v četrtem znak `0`. Če bi se s strojem hoteli pomakniti dlje, kot nam trak dopušča, na primer v desno stran, bi dobili opozorilo `Šli smo preveč desno!`. V tem primeru bi se stroj ustavil in izpisal napako.

- Pri drugi izbiri `Det. Turingov stroj - levo omejen trak` so lastnosti zelo podobne prejšnjemu primeru. Tudi tokrat imamo deterministočen Turingov stroj. V nasprotju s predhodnikom pa imamo tokrat trak omejen le v levo stran. Takšnemu traku bi lahko rekli, da ima začetek, nima pa konca. Ob vnosu zapisa na traku `0010` se ta upošteva kot začetek traku, kjer prvo polje na traku vsebuje znak `0`, drugo znak `0`, tretje znak `1`, četrto znak `0`, vsa nadaljnja polja pa so prazna in dostopna. Kljub v desno neskončno dolgem traku, pa lahko ob nepazljivem premikanju po traku vseeno

dobimo opozorilo `Šli smo preveč levo!`, saj imamo trak, ki je omejen na levo stran.

- Omejitvi traku pa se izognemo pri tretji izbiri `Det. Turingov stroj - neomejen trak`, kjer imamo še vedno deterministični Turingov stroj, ampak tokrat nimamo nikakršne omejitve glede dolžine traku. Po traku se lahko pomikamo levo ali desno, kolikor hočemo. Ob vnosu zapisa na traku `0010` se ta upošteva, kot da imamo nekje na traku zapisan ta niz znakov in da se bralno-pisalna glava trenutno nahaja nad prvim znakom v tem nizu. Zatem pa se lahko prestavi v desno ali levo smer.

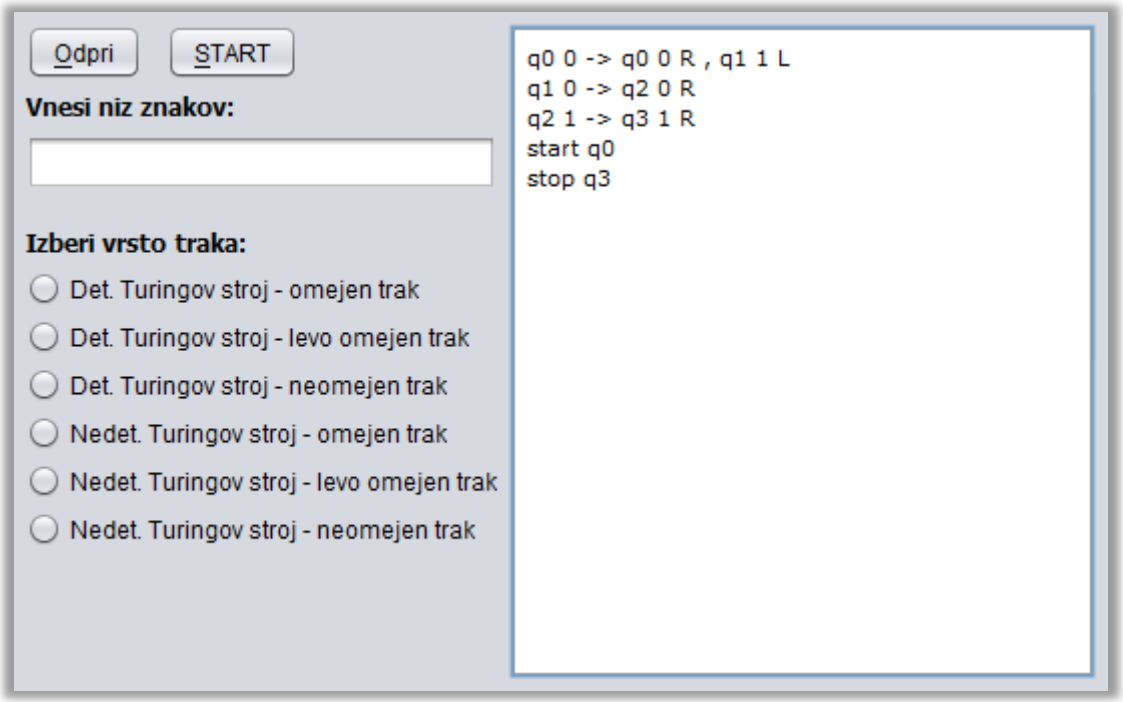
- Včasih pa želimo, da bi imel Turingov stroj malo več svobode oziroma več različnih možnosti, med katerimi bi lahko izbiral. To željo izpolnimo s tem, da naredimo Turingov stroj nedeterminističen. Takšno možnost pa podpira naslednja izbira `Nedet. Turingov stroj - omejen trak`. Kot lahko opazimo, imamo zdaj spet omejen oziroma končno dolg trak. Ampak zdaj je Turingov stroj nedeterminističen, kar pomeni, da gre stroj lahko iz enega določenega stanja v enega izmed več različnih stanj, ki jih ima na voljo. V primeru `q0 1 -> q1 1 R , q2 0 L` gre stroj lahko iz stanja `q0` v stanje `q1` in s tem na trak zapiše znak `1` ter nadaljuje desno, lahko pa se odloči za drugačno izbiro in gre iz stanja `q0` v stanje `q2`. S tem pa izbere tudi drugačna nadaljnja navodila, saj se tokrat na trak zapiše znak `0` in pomaknemo se po traku v levo smer. Pri vsakem stanju lahko imamo na voljo eno, dve, tri ali več izbir, med katerimi lahko stroj izbira.

- Tudi tokrat bomo najprej stroj nadgradili z neskončno dolgim trakom v desno smer in na voljo dali izbiro `Nedet. Turingov stroj - levo omejen trak`. Turingov stroj ostaja nedeterminističen, kar mu spet omogoča več izbire. Poleg tega pa smo iz popolnoma omejenega traku naredili trak, ki je omejen le na levi strani, katerega lastnosti so iste kot pri drugi izbiri različic Turingovega stroja.

- Nazadnje pa imamo še izbiro `Nedet. Turingov stroj - neomejen trak`, kjer imamo združene vse lastnosti, ki nam omogočajo največjo svobodo pri uporabi Turingovega stroja. Kot prej imamo tudi tokrat nedeterminističen stroj, hkrati pa imamo še neskončno dolg trak, ki je levo ali desno neomejen.

Katera izmed omenjenih izbir je najbolj primerna, je zelo odvisno od tega, kako pričakujemo, da se bo stroj vedel. V primeru, da hočemo imeti omejen trak, izberemo temu primerno možnost. Za kakšen drugačen primer bi nam mogoče prišla prav izbira z neomejenim trakom. Vse je odvisno od tega, kaj dejansko hočemo, da stroj naredi, in kako si želimo, da se stroj odzove na navodila, ki mu jih podamo z vhodno datoteko.

Pri določenih primerih pa nimamo ravno veliko izbire in smo primorani izbrati točno določeno možnost. Slika 10 nam prikazuje primer, kjer si s prvim tremi možnostmi ne moremo prav veliko pomagati, saj potrebujemo nedeterminističen Turingov stroj. Če vseeno izberemo katero izmed prvih treh možnosti, kmalu dobimo opozorilo `Imamo več možnih poti pri ključu 'q0 0'!`, kar nam pove, da trenutna izbira ne bo sprejemljiva in da moramo izbrati nekaj kontekstu bolj primernega.

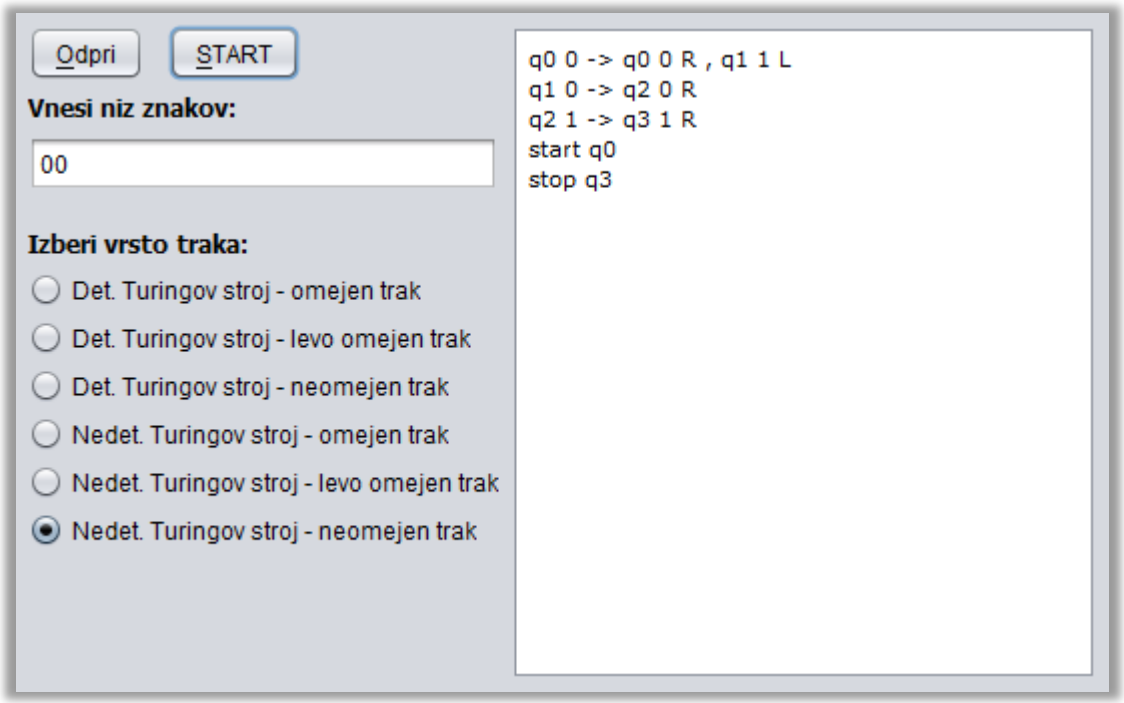


Slika 10: Primer nedeterminističnega Turingovega stroja

Predpostavimo, da smo se odločili za zadnjo možnost `Nedet. Turingov stroj - neomejen trak`. Poleg te izbire pa se moramo tudi odločiti, kateri znaki bodo že zapisani na traku. Da bomo prej prišli do končnega stanja, izberemo kratek niz znakov `00`. Poglejmo si postopek delovanja Turingovega stroja ob takšnih vhodnih podatkih, ki jih prikazuje slika 11.

Sprva najdemo ukaz `start` in poleg njega preberemo začetno stanje `q0`. Nato preberemo prvi znak, ki je zapisan na traku, to je znak `0`. Pridobljena podatka združimo, dobimo `q0 0`, in pogledamo prvo vrstico vhodne datoteke `q0 0 -> q0 0 R , q1 1 L`. Opazimo, da imamo dve različni možnosti. Odločimo se za prvo, kar pomeni, da gre stroj iz prejšnjega stanja `q0` v novo stanje `q0`, na trak namesto `0` napišemo `0` in nato se pomaknemo po traku desno. Dejanske vsebine na traku nismo nič spreminjali, saj so na njem še vedno isti znaki `00`. Nato na traku preberemo nov znak, ki je spet `0`. Skupaj sestavimo trenutno stanje stroja `q0` in prebran znak `0` ter dobimo `q0 0`, kar poiščemo v vhodni datoteki in spet preberemo prvo vrstico `q0 0 -> q0 0 R , q1 1 L`. Tudi tokrat izberemo prvo možnost in gremo

iz stanja `q0` v stanje `q0`, na trak namesto `0` napišemo `0` in nato se pomaknemo po traku desno. S tem premikom prispemo na prazno polje. Če bi na začetku izbrali možnost `Nedet. Turingov stroj - omejen trak`, bi tu prejeli opozorilo `Šli smo preveč desno!`. Ampak pred takšnim opozorilom smo se obvarovali, saj smo pri izbiri `Nedet. Turingov stroj - neomejen trak` izbrali neskončno dolg trak. Zato gremo lahko na prazno polje in ga preberemo.

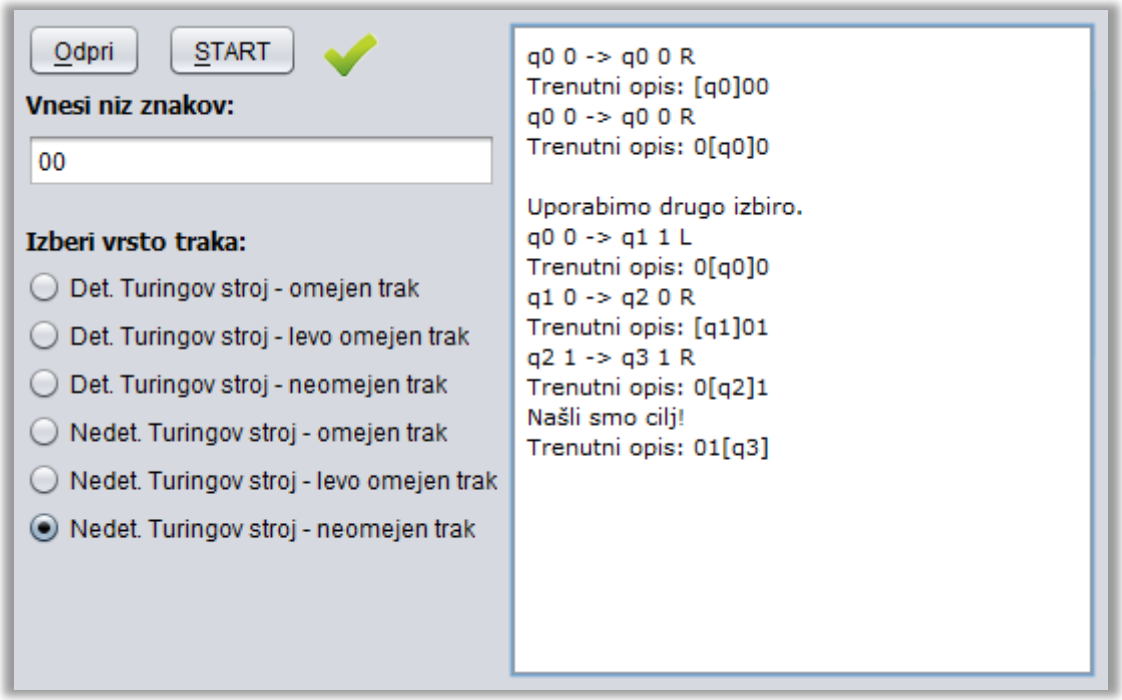


Slika 11: Vhodni podatki pred izvajanjem simulacije

Ker je polje prazno, trenutni zapis na traku `00` spremenimo v `00B`, kjer smo na desni strani dodali `B`, ki ponazarja prazno polje. Zdaj imamo stanje stroja `q0` in prebran znak `B` ter ob združitvi dobimo `q0 B`. Pregledamo vhodno datoteko in kmalu ugotovimo, da nimamo vrstice s takšnim začetkom. Stroj zdaj ne ve, kako bi nadaljeval, saj nima nadaljnjih navodil. Če bi imeli deterministični stroj, bi se tu ustavil in javil napako `Vhodni niz ni sprejemljiv!`, kar pomeni, da s podanimi vhodnimi znaki in podano vhodno datoteko stroj ne pride do svojega končnega stanja, zato javi napako, da naj spremenimo zapis na traku tako, da bo ta zapis sprejemljiv oziroma da bo lahko Turingov stroj prišel do svojega končnega stanja. Lahko pa tudi spremenimo vsebino samega stroja in dodamo vrstico, v kateri napišemo, kako naj se stroj odzove, če naleti na `q0 B`.

Pri trenutno izbranem nedeterminističnem stroju prav tako opazimo, da na takšen način ne moremo nadaljevati. Zato stroj pri vrstici `q0 0 -> q0 0 R , q1 1 L` spremeni svojo

odločitev, v polje izpiše `Uporabimo drugo izbiro.` in se tokrat odloči, da gre po drugi poti, oziroma izbere prehod v drugačno stanje. Nazadnje, ko je imel takšno izbiro, je bilo, ko je bil stroj na poziciji drugega znaka v nizu, se pravi na drugem znaku `0`. Prej smo šli v stanje `q0`, tokrat pa se odločimo za drugačno pot in gremo v stanje `q1`. Poleg tega pa zdaj namesto `0` na trak zapišemo `1` in nazadnje se še pomaknemo po traku v levo smer. Po tem koraku imamo na traku zapis `01`, saj smo drugi znak prepisali z novim znakom `1`. Zaradi premika v levo pa smo trenutno spet na položaju, kjer je na traku zapisan znak `0`, in stroj je trenutno v stanju `q1`. V vhodni datoteki poiščemo vrstico, ki se začne z `q1 0`, in vidimo, da je to druga po vrsti, ter se glasi `q1 0 -> q2 0 R`. Iz nje razberemo, da gre zdaj stroj iz trenutnega stanja `q1` v novo stanje `q2`, na traku pa namesto `0` zapišemo znak `0`, iz česar sledi, da na traku ostane isti zapis `01`, kot je bil že prej. Nazadnje pa se še pomaknemo po traku v desno smer. Tokrat je stroj v stanju `q2`, hkrati pa s traku prebere znak `1`, kar sestavi skupaj v `q2 1` in v tretji vrstici poišče `q2 1 -> q3 1 R`. Tokrat razberemo, da gre stroj iz stanja `q2` v novo stanje `q3`, na traku pa se namesto znaka `1` zapiše znak `1`. Na traku imamo spet isti zapis `01`, kot smo ga imeli že prej. Ampak tokrat imamo trenutno stanje stroja `q3`, kar pa je hkrati tudi končno stanje stroja, ki ga lahko razberemo iz zadnje vrstice `stop q3`. S tem je stroj prišel do končnega stanja, zato se stroj uspešno ustavi. V polju se pojavi napis `Našli smo cilj!` in na desni strani gumba `START` se pojavi zelena kljukica, ki označuje uspeh.



Slika 12: Celoten izpis postopka po uspešno končani simulaciji

Slika 12 nam prikazuje izpis poteka izvajanja Turingovega stroja, katerega primer smo naredili in kjer lahko opazimo, kdaj se je stroj odločil, da gre po drugačni poti, po kateri je lahko prišel do svojega končnega stanja. Prav tako pa lahko s pomočjo oglatih oklepajev opazujemo potek premikanja bralno-pisalne glave in izpis trenutnega stanja stroja. Na koncu vidimo, da smo uspešno prišli do rešitve, saj je stroj prišel v svoje končno stanje.

Postopek delovanja pri predzadnji izbiri `Nedet. Turingov stroj - levo omejen trak` bi bil popolnoma enak, saj smo izkoristili le desni del neomejenosti traku. V primeru, da bi potrebovali neomejen trak še v levo stran, pa nam ne preostane nič drugega, kot da izberemo možnost `Nedet. Turingov stroj - neomejen trak`.

Zdaj vemo, kako Turingovemu stroju podati navodila delovanja, kako mu podati vsebino, ki je že prej zapisana na traku, in kako izbirati med nekaterimi različicami Turingovega stroja. Preostane nam le še to, da si izmislimo različne primere, s katerimi bomo simulirali delovanje Turingovega stroja.

3 RAZVOJ APLIKACIJE

3.1 Razvojno orodje

Preden smo dejansko začeli izdelovati aplikacijo, smo si najprej morali odgovoriti na nekatera vprašanja v zvezi z razvojnim okoljem in orodjem, ki ga bomo uporabili. Eno izmed pomembnejših vprašanj je bilo, komu bo aplikacija na voljo oziroma kdo bo uporabnik aplikacije. V mislih nismo imeli samo enega uporabnika, ampak več različnih uporabnikov. Več kot imamo uporabnikov, večja je verjetnost, da imajo uporabniki različne operacijske sisteme. Ker si ne želimo, da bi bil to problem, ki bi onemogočil uporabo aplikacije, smo se odločili za enega izmed bolj znanih programskih jezikov.

3.1.1 Java

Trenutno je Java [10] eden izmed najbolj razširjenih in priljubljenih programskih jezikov, saj lahko deluje na različnih operacijskih sistemih. Poleg tega pa je Java objektno usmerjen programski jezik, ki ga je razvil James Gosling s sodelavci v podjetju Sun Microsystems [11], ki je od leta 2010 podružnica podjetja Oracle Corporation [12]. Projekt, ki se je v začetku (leta 1991) imenoval Oak (hrast), je bil razvit kot zamenjava za programski jezik C++ [13]. Različica Java 1.0 je bila objavljena leta 1996, do danes pa je bilo na voljo veliko novejših, vse do trenutno zadnje različice 7. Slika 13 prikazuje različne logotipe, po katerih lahko prepoznamo Javo.

Slika 13: Prepoznavnost Jave

V primeru, da hočemo zagnati program, napisan v Javi, moramo imeti nameščen JRE (*Java Runtime Environment*). Za izdelavo programov pa ni dovolj, da imamo le JRE, ampak potrebujemo tudi JDK (*Java Development Kit*), ki je namenjen razvijanju javanskih programov. Pri nekaterih zahtevnejših ali bolj specifičnih primerih pa tudi JDK ni dovolj. Včasih potrebujemo še določene dodatke, ki vsebujejo dodatne programerske ukaze, ki niso na voljo v osnovni različici JDK-ja. Tem dodatkom najpogosteje rečemo kar API-ji

(*Application Programming Interface*). V našem primeru nismo potrebovali dodatnih ukazov, saj jih že osnovni paket JDK vsebuje dovolj.

3.1.2 NetBeans

Zelo posplošeno bi lahko rekli, da je Java neke vrste nabor ukazov, ki jih pišemo v predpisani obliki, ki ji rečemo program oziroma programska koda. Kam pa pišemo program, je odvisno od nas samih. Lahko bi pisali v najbolj osnovni beležnici (*Notepad*) in bi program popolnoma v redu deloval. Pri manjši količini programske kode je to še najbolj enostaven način, ki ga lahko izberemo. Za večje projekte pa kmalu ugotovimo, da beležnica ni najbolj primerna odločitev. Celotno besedilo je v isti pisavi in tudi besedilu ne moremo dodati kakšne lastnosti, kot je na primer krepko odebeljeno besedilo. Zaradi tega koda precej kmalu postane zelo nepregledna in posledično postane programiranje veliko bolj oteženo. Za več svobode pri oblikovanju besedila bi lahko uporabili nadgradnjo beležnice [14], ki se imenuje Notepad++ (slika 14). Pri tej nadgradnji pa imamo več svobode pri oblikovanju besedila in tudi program nam samodejno malo pomaga pri obliki besedila. Določene besede oziroma ukazi spremenijo barvo in se s tem ločijo od ostalega besedila, kar nam omogoči, da jih veliko hitreje opazimo in s tem pripomoremo k bolj učinkovitemu programiranju.

Slika 14: Logotip besedilnega urejevalnika Notepad++

Oblikovano besedilo in nekateri samodejno dodani zamiki so lepa in uporabna pridobitev pri programiranju, ampak pri veliko bolj zahtevnih projektih tudi to ni dovolj. V tem primeru pa uporabljamo programe, katerih glavna namembnost je programiranje v nekem določenem programskem okolju. Med najbolj znanimi sta programerski okolji Eclipse [15] in NetBeans [16], zato bi bilo najbolje, da se odločimo za enega izmed njiju in si s tem zagotovimo dobro podlago za programiranje. Odločili smo se za NetBeans (slika 15), saj je uporabniku veliko bolj prijazen in med samim programiranjem ponuja pomoč in nasvete pri izdelovanju programa.

Slika 15: Odpiranje programskega okolja NetBeans

Poleg programiranja v Javi pa ima podporo tudi za drugačne načine programiranja, kot so JavaScript, PHP, Python, Groovy, C, C++ in še mnogi drugi. Izkaže se, da je to zelo vsestransko okolje, ki nam lahko prav pride tudi pri različnih drugih projektih, ki niso javanskega izvora.

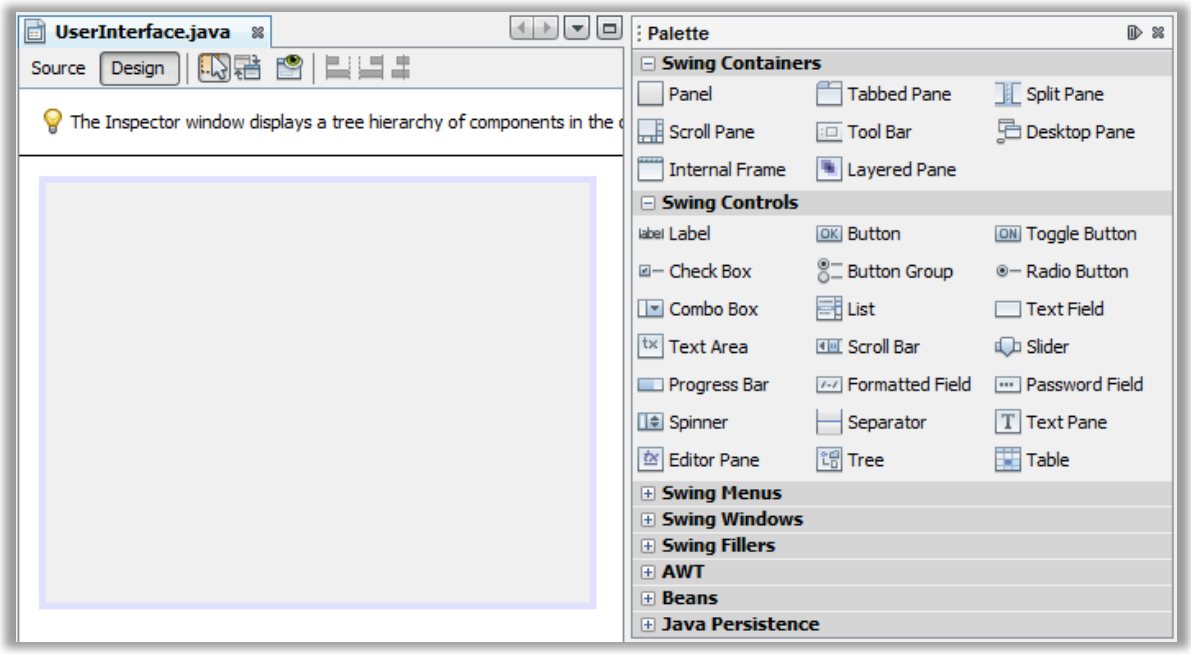
3.2 Uporaba razvojnega okolja NetBeans IDE

Več različnih možnosti programiranja v enem samem razvojnem okolju je zelo velik plus pri razvojnem okolju NetBeans IDE. Tudi sami smo to izkusili, saj smo v istem razvojnem okolju lahko napisali različne metode, ki so med seboj povezane, poleg tega pa smo lahko tudi ustvarili grafični uporabniški vmesnik oziroma GUI (*Graphical User Interface*), prek katerega se prav tako sklicujemo na različne metode, ki niso neposredno povezane z grafičnim vmesnikom.

3.2.1 Izdelava grafičnega uporabniškega vmesnika

Kot vsak javanski program bi lahko tudi GUI napisali z različnimi programerskimi ukazi, ki jih Java daje na voljo. Ampak s tem ne bi izkoristili zelo koristne zmožnosti, ki jo ponuja NetBeans IDE. Omenjeno razvojno okolje nam daje na voljo zelo veliko pomoč pri izdelavi golega ogrodja za uporabniški vmesnik. Preden ga lahko dejansko začnemo izdelovati, pa potrebujemo nekaj predpriprav, ki jih moramo storiti. Sprva moramo računalniku povedati, da bi radi programirali v Javi, zato si na računalnik namestimo JDK. Zatem je potrebna še namestitev razvojnega okolja NetBeans IDE, v katerem bomo ustvarjali, ki ga po namestitvi

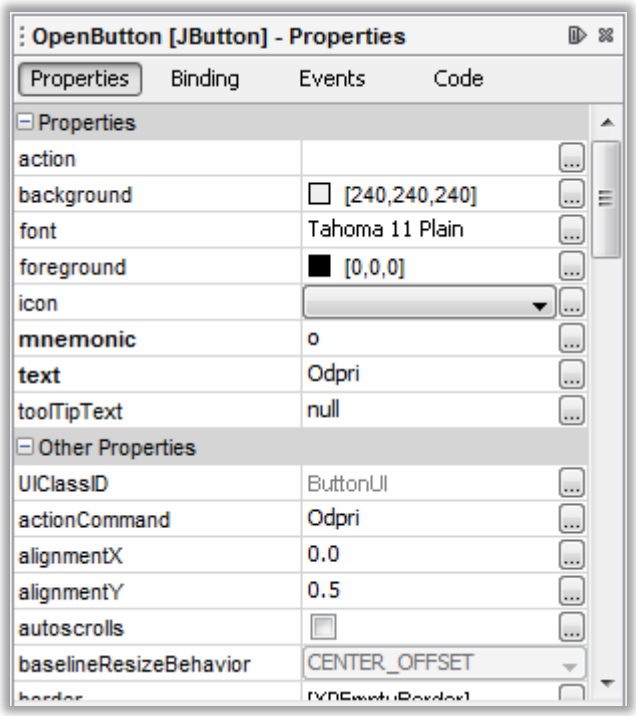
tudi odpremo. Za začetek moramo najprej ustvariti nov projekt, v katerem bomo programirali oziroma ustvarjali uporabniški vmesnik, zato izberemo `File` in zatem še `New Project`, kar odpre novo okno. Na izbiro dobimo vsa različna okolja, v katerih lahko programiramo. Ker smo se odločili, da bomo programirali v Javi, izberemo mapo `Java` in zatem še `Java Application`, nato nadaljujemo do naslednjega koraka. V označeno polje napišemo ime našega projekta in pritisnemo gumb `Finish`. Zdaj imamo nov projekt, v katerem lahko programiramo. Trenutno pa ni najbolj pomembno dejansko pisanje kode, ampak izdelava uporabniškega vmesnika. Zato z desno miškino tipko kliknemo na naš projekt, izberemo `New` in nato izberemo `JFrame Form`. Pojavi se novo okno, v katerega vpišemo ime uporabniškega vmesnika, ki ga želimo ustvariti, in pritisnemo gumb `Finish`. Zdaj smo pripravljeni na ustvarjanje uporabniškega vmesnika. Sredi razvojnega okolja imamo prazno polje, ki ponazarja naš GUI, in poleg njega imamo na desni strani skupek različnih ikon, ki jih lahko uporabimo. Slika 16 prikazuje le tisti del razvojnega okolja, ki ga trenutno potrebujemo.



Slika 16: Razvojno okolje za GUI

Dejansko programiranje je zelo poenostavljeno, saj nam ni treba pisati programske kode, ampak lahko le izberemo želene gradnike in jih prenesemo v naše delovno polje oziroma okvirček na levi strani. Programska koda se v ozadju samodejno ustvarja in če jo želimo videti, kliknemo na izbiro `Source`. Ker pa trenutno urejamo postavitev različnih gumbov in polj, izberemo `Design`. V našem uporabniškem vmesniku imamo levo zgoraj gumb `Odpri`. Za postavitev tega gumba ni bilo treba narediti nobene zahtevne stvari. Na desni strani smo pod `Swing Controls` izbrali gradnik `Button` in ga prenesli na delovno polje na levi

strani. Programska koda, ki dejansko ustvari gumb, pa se je ustvarjala samodejno. Nam le še preostane, da določimo točno pozicijo in velikost gumba. Vsaka sprememba, ki jo naredimo, se hkrati samodejno spremeni v programski kodi. Tudi ko dodajamo ali spreminjamo lastnosti, se koda prilagaja našim spremembam. Na isti način smo dodali še gumb `START`. Pri obeh dodanih gumbih pa je trenutno drugačen napis, kot si ga želimo imeti, saj pri prvem piše `jButton1` in pri drugem `jButton2`. Ampak precej kmalu opazimo, da ima vsak dodan gradnik tudi svoje posebne lastnosti, ki so napisane desno spodaj. Slika 17 prikazuje nekatere lastnosti gumba `Odpri`, kjer lahko opazimo, da smo pri lastnosti `text` spremenili vsebino iz `jButton1` v `Odpri`. Podobno smo naredili pri gumbu `START`, kjer pa smo poleg napisa spremenili tudi lastnost `enabled`, saj smo odstranili kljukico zraven napisa, kar pomeni, da je zdaj gumb onemogočen.



Slika 17: Lastnosti gumba `Odpri`

Po podobnem postopku dodajanja, pa smo v levi okvirček prenesli še vse ostale elemente. Pod gumboma imamo prostor za izpis oziroma `Label`, ki je poimenovan `jLabel`, kateremu smo za lastnost `text` vnesli vsebino, ki naj jo na začetku izpiše, to je `Odpri datoteko s končnico ».tm«.`. Naslednji gradnik, ki smo ga dodali, je bilo polje z možnim enovrstičnim vnosom, ki je navedeno kot `Text Field`. Pri lastnosti `text` pa smo zbrisali ime, saj hočemo, da je polje prazno. Spet smo odstranili izbiro pri lastnosti `enabled`, saj hočemo, da je sprva polje onemogočeno. Nato spodaj znova dodamo izpis v obliki `Label`, ki mu določimo vsebino `Izberi vrsto traka:`, ki jo trenutno tudi onemogočimo. Zatem

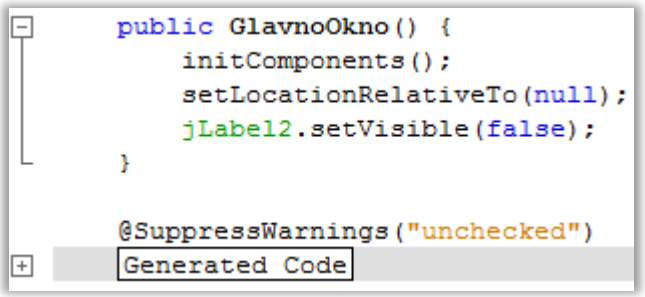
pa šestkrat dodamo `Radio Button`, ki jih prav tako onemogočimo in vsakemu določimo svoje ime, odvisno od lastnosti, ki jo ima. Dejanje, pri katerem smo dodali možnost izbire, pa ni dovolj, saj izbire niso med seboj povezane in odvisne druga od druge. Trenutno bi lahko hkrati izbrali vse možnosti, česar pa nočemo. Zato hočemo te možnosti združiti v eno skupino. Na desni strani najdemo izbiro `Button Group` in jo prenesemo na naše delovno polje. Na prvi pogled ne bomo opazili nobene spremembe, saj ni nastala nobena nova ikona in tudi če testiramo program, bodo izbire še vedno nepravilno delovale. Lahko smo brez skrbi. Skupina se je uspešno ustvarila, ampak mi še nismo povedali, katere izbire naj bodo v tej skupini. Naša ustvarjena skupina se imenuje `buttonGroup1`. Izberemo prvo možnost izbire `Det. Turingov stroj - omejen trak` in na spodnji desni strani pogledamo njene lastnosti. Opazimo lastnost `buttonGroup`, zraven pa prazen okvirček, na katerega kliknemo, in izpišejo se nam vse skupine, ki smo jih ustvarili. Glede na to, da smo ustvarili le eno, izberemo edino možno, to je `buttonGroup1`. Isti postopek naredimo še pri ostalih izbirah, da vse dodamo k isti skupini `buttonGroup1`. Zdaj bodo izbire odvisne med seboj.

Slika 18: Postavitev vseh gradnikov na delovno polje

Ko bomo kliknili na eno izbiro, bodo vse ostale izbire postale neizbrane, in ko bomo kliknili na neko drugo izbiro, bo prejšnja izbira postala neizbrana in nova izbrana. Za pravilno vedenje poskrbi skupina `buttonGroup1`, za katero se v ozadju programska koda prav tako samodejno ustvarja. Naslednja večja opazna stvar je polje, ki je na desni strani uporabniškega vmesnika. Tudi to smo dodali na precej preprost način, saj smo na desni strani našli gradnik

`Text Area` in ga prenesli na naše delovno polje. Ker hočemo, da je polje prazno, spet izbrišemo besedilo pri lastnosti `text` in nato še onemogočimo polje. Hkrati pa že vnaprej vemo, da bo to polje namenjeno le izpisu delovanja aplikacije, zato za to polje odstranimo možnost urejanja besedila. Sprva izberemo polje, nato desno spodaj pogledamo njegove lastnosti, kjer najdemo lastnost `editable`, ki ji odstranimo kljukico, s čimer naredimo polje uporabno le za branje. Nato imamo še zadnjo stvar, ki jo dodamo, to je ikona, ki spreminja svoje stanje glede na rezultat Turingovega stroja. V primeru, da Turingov stroj uspešno doseže svoje končno stanje, se prikaže ikona kljukice, ki je poimenovana `Yes_mini.png`. Če pa je bilo izvajanje neuspešno, se prikaže ikona križca, ki ga najdemo pod imenom `No_mini.png`. Obe ikoni moramo imeti shranjeni v isti mapi, kjer imamo napisan program. Ob pravilni poziciji ikon se obe samodejno dodata k projektu in ju lahko uporabimo. Hitri pogled na vse gradnike, ki jih imamo na voljo, nam kmalu sporoči, da nimamo gradnika, s katerim bi lahko vstavili sliko ali ikono. Zato se moramo znajti na drugačen način. Na svoje delovno polje dodamo nov `Label`, ki ga postavimo na desno stran gumba `START`. Pogledamo lastnosti na novo ustvarjenega `jLabel2` in pri lastnosti `text` zbrišemo besedilo, ker ga ne bomo izpisovali. Nato pa najdemo lastnost `icon`, kjer izberemo `Yes_mini.png`. Zdaj imamo ikono kljukice namesto besedila. Ob dodanem še zadnjem gradniku pa imamo na našem delovnem polju vse, kar potrebujemo (slika 18).

Istočasno z našim postavljanjem gradnikov na delovno polje se je v ozadju samodejno ustvarjala programska koda, ki opisuje pozicijo gradnikov in vse druge njihove lastnosti. Skozi čas ta koda postane zelo velika, saj lastnosti gradnikov ni malo. Poleg tega pa je celotna koda zapisana po točno določenih pravilih. Zaradi tega razloga nam razvojno okolje celo onemogoči spreminjanje samodejno ustvarjene kode. Hkrati pa celotno kodo strne v eno vrstico, v kateri piše `Generated Code`, da nam ne vzame preveč prostora in da ohrani preglednost ostale programske kode (slika 19).



Slika 19: Samodejno ustvarjena programska koda

V primeru, da si vseeno želimo pogledati programsko kodo, pa imamo znak plus v okvirčku na levi strani, na katerega kliknemo in prikaže se nam skrita koda.

Vrnimo se nazaj h gradnikom in njegovim lastnostim. Prej smo spreminjali lastnost `text`, ponekod pa tudi izbire `enabled`, `editable` in `buttonGroup`. Poleg naštetih lastnosti pa imamo na voljo še veliko drugih, s katerimi lahko spreminjamo lastnosti, navedene v sklopu `Properties`. Na voljo pa imamo še več lastnosti, ki jih najdemo v dodatnih sklopih, kot je na primer sklop `Code`. Tu lahko spreminjamo lastnosti, ki ne bodo vplivale na obliko gradnikov, ampak bolj na zapis programske kode. Največkrat smo spremenili lastnosti `Variable name`, s čimer smo spremenili ime spremenljivke, ki se uporablja v programski kodi. Nekatere gradnike smo preimenovali zaradi boljše preglednosti in manjše zmede pri programiranju. Spremembe spremenljivk pri gradnikih so bile sledeče: `jButton1` smo spremenili v `OpenButton`, `jButton2` v `StartButton`, `jLabel1` v `stanjeDatoteke` in nazadnje še `jTextField1` v `inputStringField`. Ostale spremenljivke smo pustili v prvotnem zapisu. Lahko bi tudi spreminjali druge lastnosti, ampak smo se temu raje izogibali. Nujna je bila le ena sprememba pri gradniku `jTextArea1`. Njegovo lastnost `Variable Modifiers` smo iz `private` spremenili v `public static`. Sprememba je bila potrebna, saj smo s tem omogočili, da lahko tudi druge metode spreminjajo vsebino, ki je zapisana v tem polju. Zakaj je to potrebno, bomo videli pozneje v poglavju 3.2.2.

Ob koncu spreminjanja vseh lastnosti pa zdaj zaženemo ustvarjeno aplikacijo. Odpre se nam okno, kjer je omogočen le gumb `Odpri`. Kliknemo ga in kmalu ugotovimo, da se nič ne zgodi, zato nam ne preostane nič drugega, kot da zapremo aplikacijo. Do zdaj smo naredili le to, da imamo prikazane gradnike, nismo pa še omenili, kaj se zgodi, ko na določen gradnik kliknemo oziroma ko ga izberemo. Zato gremo pogledat, kaj piše v programski kodi, kjer ugotovimo, da so funkcije dejansko prazne. Ker nimamo ničesar napisanega, se tudi nič ne zgodi (slika 20).

```
private void OpenButtonActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
}
```

Slika 20: Prazna funkcija v programu

Zdaj je treba dopolniti programsko kodo z dogodki, ki se zgodijo ob morebitni izbiri določenega gradnika oziroma v našem primeru ob kliku na gumb `Odpri`. Slika 21 prikazuje del programske kode za gumb `Odpri`, ki se izvede ob kliku nanj.

Opazimo, da je poleg odpiranja datoteke treba nastaviti še nekatere druge lastnosti, ki se nam na prvi pogled zdijo odvečne, ampak pozneje ugotovimo, da jih je treba nastaviti. Takšna lastnost je prva, ki jo navedemo, `jTextArea1.setText(null)`, kjer povemo, da

izberemo polje na desni strani, ki ima ime `jTextArea1`, in mu z ukazom `setText` nastavimo izpisano vsebino v polju. V tej vrstici bi lahko napisali stavek `jTextArea1.setText("Hello!");` in s tem bi se besedilo v polju nastavilo na `Hello!`. Ker pa smo namesto izpisane vsebine podali `null`, s tem v bistvu izbrišemo vsebino, ki je trenutno v polju. Nato se vprašamo, zakaj bi to sploh naredili, saj je že na začetku polje prazno. Vzemimo primer, da odpremo datoteko, ki je pravilno zapisana, zato se vsebina te datoteke izpiše v polje na desni strani. Ampak zatem ugotovimo, da nismo odprli datoteke, ki smo jo želeli odpreti, zato znova odpremo neko drugo datoteko, katere vsebina se prav tako izpiše v polje. Če ne bi imeli ukaza, ki izprazni polje, bi zdaj imeli v polju izpis obeh datotek. Zatem pa odpremo še tretjo datoteko, ki pa tokrat ni pravilna, zato se nam v polje izpiše napaka. Če ne bi predhodno izpraznjevali polja, bi zdaj imeli v polju napisano vsebino od obeh prej odprtih datotek in hkrati še izpis napake ob odprtju nepravilne datoteke. Takšne ali podobne situacije pa si ne želimo, zato vedno izpraznimo polje.

```
private void OpenButtonActionPerformed(java.awt.event.ActionEvent evt) {
try { //imamo try ker lahko pride do napake
    jTextArea1.setText(null); //pobrisemo tekst v polju
    jLabel2.setVisible(false); //skrijemo znakec
    JFileChooser odpriDat = new FileChooser(); //izberemo vhodno datoteko
    odpriDat.showOpenDialog(null); //prikaz okna, na sredini ekrana
    File fileName = odpriDat.getSelectedFile(); //pot do izbrane datoteke
    //preverimo ali je vhodna datoteka pravilna
    if (fileName.getName().endsWith(".tm") ||
            fileName.getName().endsWith(".TM") ||
            fileName.getName().endsWith(".Tm") ||
            fileName.getName().endsWith(".tM")) {
        if (VhodnaDatoteka.preveriVsebino(fileName.getAbsolutePath())) {
            //Datoteka je pravilno napisana, zato:
            stanjeDatoteke.setText("Vnesi niz znakov:");
            inputStringField.setEnabled(true);
            jLabel1.setEnabled(true);
            jRadioButton1.setEnabled(true);
            jRadioButton2.setEnabled(true);
            jRadioButton3.setEnabled(true);
            jRadioButton4.setEnabled(true);
            jRadioButton5.setEnabled(true);
            jRadioButton6.setEnabled(true);
            StartButton.setEnabled(true);
        } else {
            stanjeDatoteke.setText("Napaka v datoteki.");
            inputStringField.setEnabled(false);
            jLabel1.setEnabled(false);
```

Slika 21: Del programske kode za gumb `Odpri`

Drugi vneseni ukaz `jLabel2.setVisible(false)` je zapisan tudi zaradi podobnega razloga. Kot vemo, je `jLabel2` v bistvu ikona, ki prikazuje naš uspeh ali neuspeh simuliranja Turingovega stroja. Ampak ta ikona ne sme biti prikazana že na začetku, saj nismo še ničesar simulirali. Iz precej razumljivega ukaza `setVisible` in besede `false` lahko razberemo, da smo ikono naredili nevidno.

Šele zdaj pa nastopijo ukazi, zaradi katerih se pojavi okno za izbiranje datoteke. Če bi imeli zelo veliko željo, bi lahko sami izdelali novo okno, v katerem bi imeli možnost izbiranja datoteke. Ampak izdelava takšnega okna je zelo zamudna in zahtevna. Zato uporabimo okno, ki nam ga daje na voljo že samo razvojno okolje in je dovolj, da se nanj le sklicujemo s stavkom `JFileChooser odpriDat = new FileChooser();`, kjer smo si sami izmislili ime okna oziroma novo spremenljivko `odpriDat`, ostalemu delu vrstice pa ne smemo spreminjati zapisa. V novi vrstici pa še določimo položaj na novo odprtega okna in ga z vnosom `null` postavimo na sredino zaslona `odpriDat.showOpenDialog(null);`. Spet se moramo zavedati, da smo do zdaj naredili le to, da se pojavi neko novo okno. Potrebujemo še ukaze, s katerimi pridobimo datoteko, ki jo je uporabnik izbral in odprl, kar dosežemo s stavkom `File fileName = odpriDat.getSelectedFile();`, kjer smo si spet izmislili le ime spremenljivke `fileName`, drugega pa ne smemo spreminjati. Zdaj smo pridobili informacije o datoteki, ki jo uporabnik želi odpreti, ki pa je trenutno lahko poljubnega tipa.

Zaradi tega pa moramo programu dopovedati, da ne sme začeti izvajanja simulacije, če vhodna datoteka ni pravilnega tipa. Vhodna datoteka mora imeti končnico `.tm`, kar preverimo z ukazom `fileName.getName().endsWith(".tm")`. Če ta sestavljen pogoj malo razčlenimo, bi lahko rekli, da pri naši vhodni datoteki `fileName` z ukazom `getName()` pogledamo njeno ime, ki pa se mora končati z znaki `.tm`, kar preverimo z ukazom `endsWith(".tm")`. Ker pa vemo, da se pri končnicah lahko zatipkamo, smo omogočili še tri dodatne oblike zapisa končnice. Tako je datoteka vseeno sprejemljiva, čeprav imamo končnico `.TM` ali `.Tm` ali `.tM`, ki jih imamo zapisane v istem pogojnem stavku `if`, ki jih ločuje znak `||`, ki pomeni pogoj `ali`.

Takoj za tem zapisom, pa imamo že nov pogoj, ki mora biti sprejemljiv, saj ni dovolj, da ima datoteka pravilno končnico. Pomembna je tudi vsebina datoteke. Zato je v pogoju zapisano `VhodnaDatoteka.preveriVsebino(fileName.getAbsolutePath())`. Tu v bistvu kličemo eno drugo metodo, ki smo jo napisali, ki preveri vsebino napisane datoteke. Podrobnejši opis te metode bo v poglavju 3.2.2. Trenutno bomo omenili le, da klic metode vrne `true` ali pa `false`, odvisno od sprejemljivosti datoteke.

V primeru, da smo vhodno datoteko napisali pravilno, lahko nadaljujemo z ukazi, ki so zajeti znotraj stavka `if`. Najprej pri `stanjeDatoteke` nastavimo napis `Vnesi niz znakov:`, zatem pa še omogočimo vse ostale gradnike z ukazom `setEnabled(true)`, da jih zdaj lahko uporabimo. Če pa ima vhodna datoteka kakšno napako, se določeni ukazi preskočijo in nadaljujemo po končanem stavku `if`, kjer imamo znotraj območja `else` ukaze, ki se upoštevajo v primeru napake v vhodni datoteki, kar ponazarja že prva vrstica `stanjeDatoteke.setText("Napaka v datoteki.");`, v kateri imamo napisano, da spremenimo vsebino gradnika `jLabel`, ki je pod gumbom `Odpri`. Poleg spremenjenega izpisa pa z ukazom `setEnabled(false)` tudi onemogočimo vse gradnike, razen gumba `Odpri`, ki ga potrebujemo, da lahko izberemo drugo datoteko.

Kot lahko opazimo, imamo veliko ukazov, ki se izvedejo v ozadju, medtem ko uporabnik le klikne na gumb `Odpri` in izbere datoteko. Ob odprtju pravilne datoteke imamo zdaj omogočene vse gradnike. V enovrstično polje na levi strani lahko zdaj vnesemo zapis znakov, ki se pojavijo na traku, in takoj zatem lahko izberemo vrsto Turingovega stroja, zdaj lahko pritisnemo tudi gumb `START`. Ampak enako kot prej pri gumbu `Odpri`, se tudi pri kliku na gumb `START` ne bo zgodilo nič, saj še nismo napisali odziva programa na omenjeno dejanje.

Poleg tega pa moramo še podati ukaze, ki se izvedejo ob izbiri določenega Turingovega stroja. Sprva smo na začetku programa ustvarili spremenljivko `izbira`, ki v sebi hrani eno število. Čisto na začetku je to število `0`, nato pa se spreminja glede na izbrani Turingov stroj. Slika 22 prikazuje, kako se vrednost spremenljivke `izbira` spremeni, ko kliknemo na katerega izmed podanih gradnikov `jRadioButton`. Trenutna vrednost spremenljivke je pomembna, saj jo potrebujemo pri izvajanju ukazov pri gumbu `START`. Začetnih nekaj vrstic pri gumbu `START` prikazuje slika 23.

```
private void jRadioButton1ActionPerformed(java.awt.event.ActionEvent evt)
    izbira = 1;
}

private void jRadioButton2ActionPerformed(java.awt.event.ActionEvent evt)
    izbira = 2;
}

private void jRadioButton3ActionPerformed(java.awt.event.ActionEvent evt)
    izbira = 3;
}
```

Slika 22: Določitev vrednosti spremenljivke `izbira`

Tako kot pri gumbu `Odpri` moramo tudi pri gumbu `START` uporabiti nekaj ukazov, preden dejansko začnemo simulirati delovanje Turingovega stroja. Sprva moramo z ukazom `getText()` prebrati morebitne znake, ki so zapisani na traku, oziroma znake, ki so zapisani v polju, katerega spremenljivko smo poimenovali `inputStringField`. Dobljen niz znakov shranimo v novo spremenljivko `inputString`, ki jo precej kmalu tudi uporabimo.

```
private void StartButtonActionPerformed(java.awt.event.ActionEvent evt) {
    inputString = inputStringField.getText(); //preberemo vhodni niz
    jTextArea1.setText(null); //pobrisemo tekst v polju
    //klicemo razlicne funkcije, glede na izbiro pri "radio buttons"
    switch (izbira) {
        case 0:
        jTextArea1.setText("Izberi vrsto traka!"); break;
        case 1:
        KoncenTrakEnaEna.testBesede(vhodniEl, inputString, start, stop);
```

Slika 23: Del programske kode za gumb `START`

Na tem mestu z ukazom `setText(null)` tudi odstranimo besedilo, ki je izpisano v polju na desni strani, saj potrebujemo prazno polje, v katerega bomo pozneje izpisali potek simulacije Turingovega stroja. Vsi nadaljnji ukazi so zapisani znotraj stavka `switch`, ki za vhod potrebuje spremenljivko `izbira`. Glede na vrednost spremenljivke `izbira` se upošteva različna programska koda. V primeru, da smo pozabili izbrati vrsto Turingovega stroja in smo kar pritisnili gumb `START`, ima spremenljivka `izbira` vrednost `0`. To pa povzroči, da se izvršijo ukazi, ki so zapisani pod pogojem `case 0:`, oziroma ukaza `setText("Izberi vrsto traka!")` in `break`. S tem uporabnika opozorimo, da ni izbral nobene izmed možnosti, ki smo mu jih dali na voljo. Predpostavimo, da nato izberemo možnost `Det. Turingov stroj - omejen trak`, ki je prva po vrsti, s čimer povzročimo, da spremenljivka `izbira` dobi vrednost `1`, in nato se izvedejo stavki, ki so zapisani pod pogojem `case 1:`. Omenjena izbira povzroči, da se v programu izvede ukazna vrstica `KoncenTrakEnaEna.testBesede(vhodniEl, inputString, start, stop);`, kar pa je klic neke druge metode, ki smo jo napisali in bomo več o njej povedali v poglavju 3.2.3.

Trenutno je dovolj, da omenimo, da se ob različni izbiri Turingovega stroja spremenljivki `izbira` določi neka številska vrednost, ki nato vpliva na izbiro klicane metode, napisane znotraj stavka `switch`.

3.2.2 Preverjanje vsebine vhodne datoteke

Enega izmed prvih klicev neke druge metode smo zasledili v stavku `if`, kjer smo preverjali vsebino vhodne datoteke. Program izpolni določene ukaze le pod uresničenim pogojem `if(VhodnaDatoteka.preveriVsebino(fileName.getAbsolutePath()))`. Če omenjeni klic vrne `true`, potem je vhodna datoteka sprejemljiva in posledično tudi stavek `if` dobi dovoljenje za izvajanje ukazov, ki sledijo pogoju. V primeru, da klic vrne `false`, se izvedejo drugačni ukazi, saj pogoj ni bil izpolnjen. Za lažje razumevanje bomo pogoj razdelili na več manjših delov in jih opisali.

Za preverjanje vsebine vhodne datoteke potrebujemo kar veliko ukazov, zato jih nismo pisali v glavni program, saj bi s tem naredili veliko zmedo in drastično povečali nepreglednost programske kode. Zato smo te ukaze napisali v neko drugo metodo `preveriVsebino`, na katero se tudi sklicujemo. Lahko bi jo napisali znotraj razreda `GlavnoOkno`, v katerem imamo zapisano kodo za uporabniški vmesnik, ampak s tem ne bi pridobili na kompaktnosti kode, saj bi ji le prestavili pozicijo, njena dolžina pa bi še vedno ostala moteče velika. Zato smo to metodo napisali v povsem nov razred `VhodnaDatoteka`. Ker kličemo neko metodo, ki ni v razredu, kjer se trenutno nahajamo, moramo povedati, kje jo lahko najdemo. Zaradi tega moramo sprva napisati ime razreda in nato še ime metode, ki jo kličemo. Da pa omenjena metoda lahko pravilno deluje, ji moramo podati nekatere vhodne podatke, saj jih metoda zahteva. V našem primeru za vhodni podatek podamo `fileName.getAbsolutePath()`. Z omenjenim ukazom pridobimo pot in ime datoteke, ki jo hočemo odpreti in preveriti njeno vsebino.

Zdaj, ko imamo shranjeno pot in ime datoteke, jo lahko preberemo. Sprva datoteko shranimo v spremenljivko `dat`, nato pa navedemo ukaz `dat.readLine()`, s katerim preberemo eno vrstico datoteke. Vsaka vrstica ima točno določen začetek oziroma lahko so le trije različni začetki vrstice, saj se lahko začne le s stanjem, kot je na primer `q0`, ali se začne z besedo `start` ali pa z besedo `stop`. Če ima vrstica drugačen začetek, to pomeni, da ni pravilno zapisana, in program javi napako. Zaradi istega razloga v programu sledijo trije stavki `if`, ki se med seboj izključujejo. Prvi stavek `if` preveri, ali se vrstica začne z znakom `q`. Če nima takšnega začetka, potem z naslednjim stavkom `if` preverimo, ali se začetni znaki glasijo `start`. Če tudi ta pogoj ni sprejet, potem preverimo z znaki `stop`, in če tudi ta niz ni sprejet, potem javimo napako o nesprejemljivi vsebini datoteke. Še preden pa navedemo vse tri pogojne stavke, izvedemo kratek test. Preverimo število znakov v vrstici in če jih je manj kot sedem, potem že takoj na začetku javimo napako. Vedno mora biti v vrstici vsaj sedem znakov. Primer najkrajšega zapisa bi lahko bil `stop q2`. Poleg kratkega testa pa imamo na začetku še ukaz `GlavnoOkno.jTextArea1.append(vrstica + "\n")`. Z ukazom `append` dopišemo v oklepaju podano besedilo. Da pa vemo, kam dopisati besedilo, imamo

pred ukazom `append` dopisano `GlavnoOkno`, kar pomeni, da se sklicujemo na razred `GlavnoOkno`, v katerem imamo napisano kodo za uporabniški vmesnik, in nato imamo še dopisano `jTextArea1`, kar je ime polja, ki je na desni strani aplikacije.

V poglavju 3.2.1 smo omenili, da smo polju `jTextArea1` spremenili lastnost `Variable Modifiers` iz predhodno nastavljene `private` v novo `public static`. Če te spremembe prej ne bi naredili, zdaj ne bi mogli dostopati do tega polja, saj se nanj sklicujemo iz metode, ki je v nekem drugem razredu. Če bi pustili lastnost `private`, ne bi imeli pravice dostopati do tega polja in posledično ne bi mogli v polje izpisati vsebine trenutno odprte datoteke. Sprva to mogoče ne predstavlja problema, saj bi lahko odstranili ukaz o izpisovanju in pač ne bi izpisali vsebine datoteke. Večji problem se pojavi pozneje, ko iz drugih metod izpisujemo potek simuliranja Turingovega stroja, kjer pa nujno potrebujemo dostop do polja, saj vanj izpisujemo pomembne rezultate.

Kot smo že prej omenili, v prvem večjem stavku `if` preverjamo začetek vrstice, ki se mora začeti z znakom `q`. Hkrati pa v istem pogojnem stavku z dodanim ukazom `Character.isDigit()` preverimo še naslednji znak, ki mora biti število. Ob sprejetju začetnih dveh pogojev nato nadaljujemo z izvajanjem nadaljnjih ukazov oziroma z nadaljnjim preverjanjem znakov v vhodni datoteki. Prvi znaki, ki jih preberemo, so namenjeni stanju in se začnejo s črko `q` in nadaljujejo s številom. Lahko pa ima stanje več kot le eno število, saj je stanje `q123` isto veljavno kot stanje `q0`. Zaradi tega moramo ugotoviti, do katerega znaka so zapisana števila. Slika 24 prikazuje zanko `while`, ki se izvaja, dokler ne pridemo do znaka, ki je presledek, saj mora med stanjem in tračnim simbolom nujno biti presledek. Medtem ko čakamo na presledek, pa štejemo, koliko znakov je številski znak. Lahko bi se nam med števila vrinila kakšna črka, kar pa ne bi bilo v redu, zato izpišemo napako v primeru, če prebrani znak ni število.

```
//lahko je vec kot le eno stevilo, za katerimi sledi presledek
while (!Character.isSpaceChar(znaki[znakiSt])) {
    if (Character.isDigit(znaki[znakiSt])) {
        znakiSt++;
    } else {
        napaka(vrsticaSt);
        return false;
    }
}
//lahko je vec presledkov
while (Character.isSpaceChar(znaki[znakiSt])) {
    znakiSt++;
}
```

Slika 24: Začetna vsebina prvega večjega stavka `if` v metodi `preveriVsebino`

Za stanjem vedno sledi presledek. Poleg tega pa moramo še upoštevati, da je lahko zapisanih več presledkov, kar naredimo z naslednjo zanko `while`, kjer prebiramo znake, vse dokler ne pridemo do znaka, ki ni več presledek. Zatem sledi znak, ki naj bi bil zapisan na traku, za katerega veljajo posebna pravila. Znak je lahko mala tiskana črka od a do z po angleški abecedi ali število od 0 do vključno 9 ali pa je znak `B`, kar v istem vrstnem redu tudi upoštevamo pri pogojnem stavku `if(Character.isLowerCase(znaki[znakiSt]) || Character.isDigit(znaki[znakiSt]) || znaki[znakiSt] == 'B')`. Temu znaku nato sledi presledek, lahko pa tudi več presledkov, kar spet upoštevamo z zanko `while`, ki se izvaja, dokler je znak enak presledku. Zatem sledi puščica, kar pomeni, da imamo znak − in nato še znak >, med katerima ne sme biti nobenega drugega znaka. Tudi presledka ne sme biti, kar spet preverimo s pogojnim stavkom `if`. Puščici nato sledi eden ali več presledkov ali pa nadaljujemo kar brez presledka, kar spet upoštevamo s stavkom `if` in zanko `while`. Zatem mora biti znak `q`, saj navajamo novo stanje, v katerega gre Turingov stroj, kateremu nato sledi še na novo zapisan znak na traku. Postopek samega programa je zelo podoben postopku, ki je že zapisan za preverjanje začetka vrstice, saj do puščice veljajo ista pravila. Razlika je le v tem, da imamo zdaj na koncu še dodano smer premika po traku, kjer upoštevamo le znake `L`, `R` in −, za kar poskrbi nov stavek `if`.

Do zdaj nam je uspelo preveriti vsebino prve vrstice v vhodni datoteki. Ob vseh sprejetih pogojih smo si pridobili pravico, da lahko uporabimo podatke, ki so zapisani v prvi vrstici. Te podatke pa moramo nekam shraniti, da lahko pozneje do njih dostopamo, zato smo uporabili `Hashtable`. Lastnost takšnega načina shranjevanja je v obliki shranjevanja podatkov. Podatki se shranjujejo na primeru kombinacij `Key-Value` oziroma `Ključ-Vrednost`. Vsak ključ ima lahko le eno točno določeno vrednost. V tabeli kombinacij pa se tudi ključi ne smejo ponavljati. V primeru, da bi dodali neko novo vrednost, ki bi imela ključ, ki že obstaja, bi ključ ostal isti in edini, ampak vrednost tega ključa bi se spremenila v novo dodano vrednost, s čimer bi izgubili podatek o prej dodani vrednosti. Vzemimo primer, da v prvi vrstici preberemo `q0 0 -> q1 1 R`. Iz takšne vrstice bi vzeli prve štiri znake `q0 0` in jih shranili kot ključ, ki bi imel vrednost `q1 1 R`. Povedano drugače, znaki na levi strani puščice bi bili podani kot ključ, znaki na desni strani puščice pa bi bili podani kot vrednost ključa. Takšen način shranjevanja podatkov je zelo primeren za deterministične Turingov stroj, saj gre lahko stroj iz točno določenega stanje v neko drugo točno določeno stanje, kjer trenutno stanje shranimo kot ključ, novo stanje pa kot vrednost ključa.

Sprememba v shranjevanju pa nastane, ko imamo nedeterministične Turingov stroj, saj gre stroj lahko iz trenutnega stanja v različna nova stanja, kar pomeni, da ima en ključ več različnih vrednosti. Ampak po definiciji ima lahko vsak ključ le eno vrednost. Zato smo za to eno vrednost v bistvu podali eno tabelo, ki pa potem lahko vsebuje več elementov. S tem ohranimo lastnost, da lahko za en ključ navedemo le eno vrednost, in hkrati omogočimo

shranjevanje dodatnih vrednosti za isti ključ. Takšen nestandarden zapis shranjevanja pa moramo definirati sami oziroma moramo posebej navesti način zapisa shranjenih podatkov. Standardno bi bilo `Hashtable<String, String>`, ampak mi smo lastnosti malo spremenili in imamo zapis definiran kot `Hashtable<String, ArrayList>`, saj `ArrayList` predstavlja tabelo, v kateri so zapisana nova stanja Turingovega stroja. Zapis celotne vrstice v programu se glasi `Hashtable<String, ArrayList> vhodniEl = new Hashtable<String, ArrayList>();`, kjer smo definirali spremenljivko `vhodniEl` z lastnostmi, ki jih ponuja naš malo spremenjen `Hashtable`.

Zdaj ko smo definirali način shranjevanja podatkov, pa lahko dodajamo elemente, kar naredimo z ukazom `vhodniEl.put()`. Omenjeni ukaz pa kot vhod sprejme ključ in vrednost, kar tudi napišemo v obliki `vhodniEl.put(Key, Value)`. Kot vhod v programu ne pišemo dejanski besedi `Key` ali `Value`, ampak znake, ki so na levi ali desni strani puščice. Da pa vemo, katere znake lahko uporabimo, kličemo eno drugo metodo `sestaviBesedo()`, ki smo jo napisali. Kot je že iz imena razvidno, ta metoda sestavi besedo, ki jo potem podamo kot ključ ali vrednost. Za pravilno delovanje pa metoda potrebuje nekaj vhodnih elementov. Sprva moramo podati znake, ki so zapisani v vrstici, saj jih potrebujemo, da lahko sestavimo besedo. Nato še podamo število, pri katerem znaku naj se sestavljanje besede začne, in nazadnje še število, pri katerem znaku naj se sestavljanje besede konča. Če vzamemo primer vrstice `q0 0 -> q1 1 R`, kjer hočemo dobiti ključ, bi klicali metodo na način `sestaviBesedo(vrstica, 0, 3)`, iz česar bi dobili besedo `q0 0`. Za vrednost tega ključa, se pravi za pridobitev besede `q1 1 R`, pa bi metodo klicali na način `sestaviBesedo(vrstica, 8, 13)`. Tu smo podali kar konkretna števila, ampak v programu ta števila nadomestijo spremenljivke, katerih vrednosti se skozi program spreminjajo. Ena izmed takšnih spremenljivk je `besedaZacetekValue`, ki označuje pozicijo znaka, kjer se začne vrednost oziroma `Value`.

Pri sestavljanju besed pa moramo biti pozorni tudi na presledke, zaradi katerih imamo pozneje lahko probleme. Za konkreten primer bi problem lahko opisali s tem, da imamo prvo vrstico z začetkom `q0 0` in zatem pozneje še enkrat vrstico z začetkom `q0  0`. Razlika med njima je le v tem, da je pri prvem primeru napisan en presledek, v drugem primeru pa sta zapisana dva. Program bi lahko primera vzel kot dva različna ključa, ampak dejansko opisujemo isti ključ. Da bi se izognili nastali situaciji, že med sestavljanjem besede odstranimo odvečne presledke, tako da imamo samo en presledek. S tem se obvarujemo pred morebitnim napakam.

V postopku vnašanja ključev in vrednosti v `Hashtable` pa imamo še nekatere ukaze, ki nastopijo le ob določenih primerih, ko se uporabnik zmoti ali namenoma napiše nesprejemljive vhodne datoteke. Prvi takšen primer bi lahko bil, da uporabnik v različnih vrsticah napiše isti ključ. V konkretnem primeru se to zgodi, če bi uporabnik v prvi vrstici

zapisal ključ `q1 1` in nato pozneje v deseti vrstici spet zapisal ključ `q1 1`. Če bi to napako spregledali, bi v bistvu povzročili, da bi vrednosti iz desete vrstice nadomestile vrednosti, zapisane v prvi vrstici. V primeru, da bi v prvi vrstici imeli zapisano končno stanje stroja, ne bi nikoli mogli priti do tega stanja, saj pozneje ne bi obstajal več, ker bi ga nadomestilo stanje iz desete vrstice. Da pa se temu izognemo, pa vrednosti ključa iz desete vrstice dodamo vrednostim ključa iz prve vrstice. Tako ima zdaj prva vrstica tudi tiste vrednosti, ki so zapisane v deseti vrstici. Deseta vrstica pa se pri dodajanju ključev ne upošteva. Da pa to lažje dosežemo, si pomagamo z ukazom `containsKey(Key)`, s katerim preverimo, ali določen ključ `Key` že obstaja. Če še ne obstaja, potem ga dodamo. Če pa že obstaja, potem se izvrši malo drugačna koda. Že obstoječemu ključu najprej pogledamo njegove trenutne vrednosti in jih primerjamo s tistimi, ki mu jih hočemo dodati. Če takšna vrednost še ne obstaja, potem jo dodamo. Če pa takšna vrednost že obstaja, potem ne dodajamo ničesar, saj nima smisla dodajati vrednosti, ki že obstaja. Slika 25 prikazuje del kode, kjer je napisano, kako preverimo vsebovanost določenega elementa in ga po potrebi dodamo. Opis pogoja bi se lahko glasil: samo v primeru, če element še ne obstaja, potem ga dodaj.

```
if(!vhodniEl.get(sestaviBesedo(znaki,0,besedaKonecKey)).contains(vhodniElDod.get(i)))
{
    vhodniEl.get(sestaviBesedo(znaki,0,besedaKonecKey)).add(vhodniElDod.get(i));
}
```

Slika 25: Izognitev dodajanja istih elementov

Hkrati s tem, ko smo preverili morebitne dvojnike ključev, smo tudi preprečili morebitno podvajanje vrednosti ključa. Lahko pa tudi zasledimo primer, kjer se moramo podvajanju vrednosti izogniti že prej, saj lahko podvajanje nastopi v isti vrstici. Vzemimo primer, kjer je zapis vrstice v obliki `q0 0 -> q0 0 R , q1 1 L , q0 0 R`. Opazimo, da imamo dvakrat napisano isto vrednost `q0 0 R`, česar pa ne potrebujemo. Zaradi tega preverimo morebitno podvojenost elementov, še preden izvedemo ukaze za dodajanje ključev.

Po omenjenih postopkih v `Hashtable` dodajamo ključe in njihove vrednosti. Drugačni ukazi pa se začnejo izvajati, kadar se vrstica vhodne datoteke ne začne z znakom `q`, ampak z znakom `s`, saj imamo v tem primeru po vsej verjetnosti zapisano besedo `start` ali pa besedo `stop`, katerima sledi neko začetno ali končno stanje.

V primeru, da se vrstica začne z znaki `start`, potem se izvede drugi večji pogojni stavek `if`, ki ga imamo v programu. Spet preverjamo veljavnost znakov, ki jih preberemo. Takoj za besedo `start` mora biti zapisan vsaj en presledek, kar preverimo s stavkom `if`, v katerem imamo zanko `while`, ki se izvaja, dokler imamo presledek, saj ko zmanjka presledkov, takrat nastopi znak `q`, ki označuje začetno stanje Turingovega stroja. Kot za vsa vmesna

stanja tudi za začetno stanje velja pravilo, da je sestavljeno iz znaka `q` in poljubnega števila. Ker obstaja možnost, da imamo več kot le enomestno število, moramo prilagoditi program, da upošteva vsa števila, kar tudi prikazuje slika 26. Znaku `q` mora slediti znak, ki je število. V nasprotnem primeru javimo napako. Zatem preverjamo vsak naslednji znak, ki mora prav tako biti število. Preverjamo, dokler ne pridemo do zadnjega znaka v vrstici, saj so zapisana števila zadnji dovoljeni znaki v vrstici. Če zasledimo kaj drugega, program javi napako.

```
if (Character.isDigit(znaki[++znakiSt])) {
    while (znakiSt < znaki.length) {
        if (!Character.isDigit(znaki[znakiSt])) {
            napaka(vrsticaSt);
            return false;
        } else {
            znakiSt++;
        }
    }
} else {
    napaka(vrsticaSt);
    return false;
}
```

Slika 26: Upoštevanje vseh števil pri začetnem stanju

Iz pridobljenih informacij in s pomočjo metode `sestaviBesedo()` sestavimo začetno stanje Turingovega stroja. Dobljeno besedo shranimo v spremenljivko `start`, ki jo bomo potrebovali pozneje pri simuliranju Turingovega stroja.

Kot smo že omenili, pa obstaja še tretja in zadnja različica začetnih znakov v vrstici, to je v primeru, da se vrstica začne z znaki `stop`. Potem se izvede še tretji večji pogojni stavek `if`, ki ga imamo v programu, ki pa se od prejšnjega ne razlikuje veliko. Za spremembo na začetku namesto znakov `start` zahtevamo znake `stop`. Nadaljnji postopek programa pa je enak kot pri prejšnjem stavku `if`, saj tudi tu preberemo neko stanje, ki ima iste lastnosti kot vsa stanja, le da se tokrat imenuje končno stanje. Vseeno pa velja, da je sestavljeno iz znaka `q` in poljubnega števila. Prav tako pa tudi tu z metodo `sestaviBesedo()` sestavimo končno stanje Turingovega stroja in dobljeno besedo shranimo v spremenljivko `stop`, ki jo bomo potrebovali pozneje pri simuliranju Turingovega stroja.

S tem smo pregledali čisto vse tri možne vrstice v vhodni datoteki. V primeru, da ima kakšna vrstica nek drugačen zapis, pa program javi napako o neveljavni vrstici. Ampak do zdaj smo preverili vsako vrstico posebej in jo sprejeli ali zavrnili glede na njeno vsebino, kar pa še ni dovolj. Lahko imamo primer, kjer so vse vrstice napisane pravilno, ampak na koncu uporabnik pozabi dodati začetno in končno stanje. Če bi program pustili na tej točki, potem bi

vhodna datoteka bila sprejeta, čeprav ne bi imeli začetnega in končnega stanja, saj bi bile vse vrstice napisane pravilno. Zato moramo uporabnika opozoriti ob morebitni napaki.

Vhodno datoteko prebiramo vsako vrstico posebej. Že na začetku programa pa smo se odločili, da bomo imeli eno spremenljivko `vseVrstice`, ki ji bomo ob vsakem prebiranju nove vrstice dodali vsebino trenutno prebrane vrstice, ki jo hranimo v spremenljivki `vrstica`. V program pa smo napisali `vseVrstice += vrstica;`. S tem smo v bistvu naredili spremenljivko, ki v enem zelo dolgem nizu znakov hrani vsebino celotne vhodne datoteke (slika 27).

```
//preberemo celotno datoteko, vsako vrstico posebej
while ((vrstica = dat.readLine()) != null) {
    //vsebino celotne datoteke napisemo v eno vrstico
    vseVrstice += vrstica;
```

Slika 27: Dodajanje vsebine datoteke v eno spremenljivko

Ta na novo pridobljen podatek pa lahko zdaj izkoristimo na koncu programa. Z ukazom `contains()` preverimo, ali v spremenljivki `vseVrstice` obstajata oba niza znakov `start` in `stop`. Celoten pogoj se glasi `if(vseVrstice.contains("start") && vseVrstice.contains("stop"))`. Če je pogoj sprejet, potem je vsebina celotne vhodne datoteke sprejeta in metoda vrne `true`.

```
//dokument mora imeti "start" in "stop"
if(vseVrstice.contains("start") && vseVrstice.contains("stop")){
    GlavnoOkno.vhodniEl = vhodniEl; //spremenimo globalni Hashtable
    return true; //vrnemo true, ker je datoteka brez napak
} else {
    napaka(vrsticaSt+1);
    return false;
}
```

Slika 28: Zadnji pogoj v metodi `preveriVsebino`

V primeru, da pogoj ni sprejet, pa metoda izpiše napako in vrne `false`, saj s tem pove, da vhodna datoteka ni sprejeta. Slika 28 prikazuje ravno omenjeni zadnji pogoj pred sprejetjem ali zavrnitvijo vhodne datoteke.

3.2.3 Metode za simuliranje Turingovega stroja

Že čisto na začetku smo omenili, da lahko naš program simulira šest različnih Turingovih strojev. Katerega želimo simulirati, pa si izberemo v uporabniškem vmesniku. Z določeno

izbiro se prav tako določi tudi vrednost spremenljivke `izbira`, ki ima dodeljeno neko število, ki je odvisno od izbire. Vrednost tega števila pa tudi določi, katera metoda se bo klicala za izvajanje simulacije. Če ima spremenljivka `izbira` vrednost `1`, pomeni, da smo izbrali prvo možnost `Det. Turingov stroj - omejen trak`, zaradi tega se kliče metoda `KoncenTrakEnaEna.testBesede()`. Metoda ima takšno ime, saj hočemo simulirati Turingov stroj, ki ima omejen oziroma končen trak, hkrati pa je naš stroj deterministični, kar pomeni, da gremo lahko iz enega stanja v le eno novo stanje, iz česar izvira `EnaEna`. Ampak to je ime razreda, mi pa se moramo sklicevati na metodo, ki je znotraj tega razreda in je poimenovana `testBesede`. Za pravilno delovanje pa moramo metodi podati nekaj vhodnih podatkov, s pomočjo katerih bo program lahko simuliral delovanje stroja: `vhodniEl, inputString, start, stop`. Prvi podatek `vhodniEl` je v bistvu `Hashtable`, ki smo ga naredili že ob preverjanju vsebine vhodne datoteke. V njem so shranjeni vsi prehodi stanj. Drugi podatek `inputString` je niz znakov, ki smo ga določili kot trenutni zapis na traku in je zapisan v polju `inputStringField`. Tretji in četrti podatek smo prav tako pridobili pri preverjanju vhodne datoteke, kjer `start` hrani začetno stanje, `stop` pa končno stanje stroja. Ob pridobitvi vseh podatkov pa lahko začnemo simulirati delovanje Turingovega stroja.

Ker imamo trenutno izbran deterministični Turingov stroj, lahko gremo iz trenutnega stanja le v eno točno določeno stanje, nimamo pa izbire med različnimi stanji. Slika 29 prikazuje kodo, ki je zapisana na začetku programa in nas obvaruje pred morebitnim nadaljevanjem, če bi imeli izbiro med več stanji.

```
//pregledamo ce ima kaksen Key vec razlicnih Value
Object[] seznamKeys = vhodniEl.keySet().toArray();
for (int i = 0; i < seznamKeys.length; i++) {
    klic = seznamKeys[i].toString();
    if (vhodniEl.get(klic).toString().contains(",")) {
        GlavnoOkno.jTextArea1.setText(
                "Imamo več možnih poti pri ključu '"
                + klic + "'!\n");
        GlavnoOkno.ikona = "No";
        vhodnaBesedaOK = false;
    }
}
```

Slika 29: Preverjanje determinističnosti Turingovega stroja

Če bi na grobo opisali te začetne vrstice, bi lahko rekli, da pregledamo vse prehode stanj vsakega posebej in preverjamo, ali kateri izmed njih vsebuje vejico. V primeru, da jo

najdemo, to pomeni, da ima neko stanje na izbiro več prehodnih elementov, kar naredi vhodno datoteko nesprejemljivo.

Slika 30 prikazuje kodo, ki nas prav tako obvaruje pred napačnim izvajanjem programa. Sprva preverimo, ali je uporabnik sploh zapisal niz znakov, ki naj bi bil zapisan na traku. Če je `inputString` prazen, potem program javi napako in prekine z nadaljnjim izvajanjem programa.

```
//v primeru, da ni vhodnega niza (iskalnega niza)
if ("".equals(inputString)) {
    GlavnoOkno.jTextArea1.setText("Ni iskane besede!\n");
    GlavnoOkno.ikona = "No";
    vhodnaBesedaOK = false;
    klic = null;
} else {
    vhodnaBeseda = inputString.toCharArray();
    klic = start + " " + vhodnaBeseda[vhodnaBesedaSt];
}
```

Slika 30: Prekinitev izvajanja programa ob morebitnem praznem zapisu na traku

Če pa je dodeljen nek zapis na traku, sestavimo vsebino spremenljivke `klic`, ki vsebuje začetno stanje `start` in prvi znak, ki je zapisan na traku. Nato nadaljujemo z izvajanjem programa. Spremenljivko `klic` potrebujemo, da se lahko sklicujemo na elemente, shranjene v `Hashtable` kot ključ oziroma `Key`. Predpostavimo, da smo za `klic` sestavili niz `q0 0`, prek katerega se sklicujemo na vrednost oziroma `Value` ključa `q0 0`, ki pa je `q0 0 R`. Iz pridobljene vrednosti pa moramo razbrati novo stanje, nov tračni simbol in smer premika po traku. Slika 31 prikazuje, kako določenim spremenljivkam dodelimo nekatere vrednosti, ki jih pridobimo iz vrednosti ključa.

```
stanje = "q"; //vedno se zacne s 'q'
//sestavimo: q + stevila
for (int i = 1; i < znakiSt; i++) {
    stanje += value[i];
}
tracniSimbolNovi = value[++znakiSt];
znakiSt++; //da preskocimo presledek
smer = value[++znakiSt]; //smer premikanja po traku
```

Slika 31: Pridobivanje elementov iz vrednosti ključa

Zdaj imamo vse, kar potrebujemo, da lahko izpišemo dejanje, ki ga bomo izvedli na traku, in trenutno stanje na traku, ki ga v oglatih oklepajih dopolnjuje trenutno stanje stroja. V našem

primeru bi na tej točki izpisali dejanje oziroma prehodno stanje `q0 0 -> q0 0 R`, nato pa v novi vrstici še stanje na traku `Trenutni opis: [q0]0010`. Za izpisom pa nadaljujemo s simuliranjem stroja. Naslednji korak, ki ga moramo narediti, je, da na trak zapišemo nov tračni simbol: `vhodnaBeseda[vhodnaBesedaSt] = tracniSimbolNovi`, ki mu sledi premik po traku v določeno smer, če sploh pride do izvedbe premika, saj lahko stroj ostane na isti poziciji. Slika 32 prikazuje, kako se vrednost spremenljivke `vhodnaBesedaSt` spremeni glede na podani znak za premik po traku. `vhodnaBesedaSt` simulira premik, saj njena vrednost označuje trenutno pozicijo na traku, ki jo lahko povečujemo ali pa zmanjšujemo, odvisno od premika.

```
if (smer == 'L' || smer == 'R' || smer == '-') {
    if (smer == 'R') {
        vhodnaBesedaSt++;
    } else if (smer == 'L') {
        vhodnaBesedaSt--;
    }
} else {
    if (smer != '-') {
        GlavnoOkno.jTextArea1.setText("Napaka!");
        GlavnoOkno.ikona = "No";
        konecPrograma = true;
    }
}
```

Slika 32: Premik po traku

Na trak smo zapisali nov znak, bralno-pisalno glavo smo prestavili v določeno smer, edino, kar nam še preostane, je, da upoštevamo spremembo stanja stroja iz trenutnega v neko novo stanje. Preveriti moramo, ali je stroj ne le v novem stanju, ampak tudi v končnem stanju, kar naredimo s preprostim stavkom `if`. Slika 33 prikazuje kodo, ki se izvede ob prehodu stroja v končno stanje. Stroj se ustavi, program pa le še izpiše trenutni zapis na traku in zaključi s simulacijo.

V primeru, da nismo prišli do končnega stanja, pa moramo preveriti trenutno pozicijo bralno-pisalne glave. Trenutno imamo omejen trak, kar pomeni, da ne smemo zaiti zunaj polj na traku, ki smo jih definirali ob začetku izvajanja simulacije. Če vrednost spremenljivke `vhodnaBesedaSt` prekorači število znakov na traku, potem izpišemo napako, saj smo se premaknili preveč v desno stran. Tudi na drugi strani traku imamo omejitev, kar spet preverjamo s spremenljivko `vhodnaBesedaSt`, ki pa ne sme postati negativa, saj bi to pomenilo, da smo se premaknili preveč v levo stran oziroma smo šli zunaj pozicije traku. Ni pa nujno, da se sploh izvede kateri izmed naštetih pogojev. Lahko se zgodi, da še nismo prišli v končno stanje in hkrati še nismo zašli zunaj pozicije traku, ampak smo le v enem izmed

prehodnih stanj. V tem primeru pa se na novo formira spremenljivka `klic`, iz katere nato dostopamo do nekega novega ključa, od katerega dobimo nove vrednosti, ki jih upoštevamo na isti način, kot smo te, ki smo jih pravkar omenili.

```
if (stanje.equals(stop)) {
    vhodnaBesedaOK = true;
    konecPrograma = true;
    GlavnoOkno.jTextArea1.append("Našli smo cilj!\n");
    /*
     * Trenutni opis - izpis
     */
    GlavnoOkno.ikona = "Yes";
} else if (vhodnaBesedaSt >= vhodnaBeseda.length) {
    GlavnoOkno.jTextArea1.append("Šli smo preveč desno!\n");
    GlavnoOkno.ikona = "No";
    konecPrograma = true;
} else if (vhodnaBesedaSt < 0) {
    GlavnoOkno.jTextArea1.append("Šli smo preveč levo!\n");
    GlavnoOkno.ikona = "No";
    konecPrograma = true;
} else {
    klic = stanje + " " + vhodnaBeseda[vhodnaBesedaSt];
}
```

Slika 33: Izpis ob prihodu v končno stanje ali ob morebitni napaki

Postopke branja stanj in različnih reakcij, ki sledijo, pa ponavljamo, vse dokler se stroj ne ustavi ob morebitni napaki oziroma dokler se program ne zaključi, za kar pa poskrbi zanka `while (vhodniEl.containsKey(klic) && konecPrograma == false)`, ki se izvaja, vse dokler obstaja ključ, ki ga želimo prebrati, in dokler ni zahtevan konec programa. Zaključek programa pa ni vedno negativna stvar. Lahko se zgodi, da je stroj prišel v končno stanje in se je zaradi tega tudi ustavil ter sporočil, da se izvajanje programa zaključi. Takšne prekinitve programa smo najbolj veseli.

Druga možna izbira načina simulacije Turingovega stroja z nazivom izbire `Det. Turingov stroj - levo omejen trak` po delovanju ne razlikuje veliko od prejšnje izbire. Ime razreda se prav tako ne razlikuje veliko od prejšnjega primera, saj je poimenovan kar `NeskoncenTrakDesnoEnaEna`, iz česar lahko razberemo edino razliko, to je v dolžini traku. Razlika je le v tem, da imamo tokrat omejen trak le na levo stran, na desno stran pa nismo omejeni, saj ima trak neskončno mnogo praznih polj. Zaradi tega, v grobem povedano, tudi ni velike razlike pri delovanju programa, ki simulira takšen način vedenja stroja. Večina programske kode je enaka, razlika je le pri pogojnih stavkih, kjer preverjamo, ali smo prišli do konca traku ali ne. Slika 34 prikazuje to razliko v kodi.

```
} else if (vhodnaBesedaSt >= vhodnaBeseda.length) {
    //na desni strani besede dodamo 'B'
    vhodnaBeseda = (new String(vhodnaBeseda) + 'B').toCharArray();
    klic = stanje + " " + vhodnaBeseda[vhodnaBesedaSt];
} else if (vhodnaBesedaSt < 0) {
    GlavnoOkno.jTextArea1.append("Šli smo preveč levo!\n");
    GlavnoOkno.ikona = "No";
    konecPrograma = true;
} else {
    klic = stanje + " " + vhodnaBeseda[vhodnaBesedaSt];
}
```

Slika 34: Det. Turingov stroj – levo omejen trak

Tokrat ne moremo priti do konca traku, vsaj ne na desni strani, zato smo to napako odstranili in jo nadomestili z novimi ukazi. Ko trak pride do zadnjega desnega znaka na traku in se zatem hoče premakniti še za eno polje v desno, mu spremenimo vsebino na traku tako, da na desno stran dodamo znak `B`, ki ponazarja prazno polje. V primeru, da bi imeli na traku zapis znakov `0010` in bi hoteli iti še dlje v desno stran, dodamo znak `B` in na traku dobimo nov niz znakov `0010B`. Zatem ustvarimo nov `klic` in program nadaljuje izvajanje, vse dokler ne javi napake ali pa stroj preide v končno stanje.

Tudi tretja izbira `Det. Turingov stroj - neomejen trak` se od prve ne razlikuje prav veliko, še manj pa od druge izbire. Še vedno obstaja velika podobnost tudi pri imenu razreda `NeskoncenTrakLevoDesnoEnaEna`, saj smo le dodali besedo `Levo`, s čimer smo povedali, da imamo zdaj neskončen trak tudi v levo smer. Razlika v kodi je spet pri pogojnih stavkih, kjer preverjamo, ali smo prišli do konca traku ali ne, kjer tokrat nikoli ne pridemo do konca traku, ampak le dodajamo znak `B` na tisto stran, kot je potrebno (slika 35).

```
} else if (vhodnaBesedaSt >= vhodnaBeseda.length) {
    //na desni strani besede dodamo 'B'
    vhodnaBeseda = (new String(vhodnaBeseda) + 'B').toCharArray();
    klic = stanje + " " + vhodnaBeseda[vhodnaBesedaSt];
} else if (vhodnaBesedaSt < 0) {
    //na levi strani besede dodamo 'B'
    vhodnaBeseda = ('B' + new String(vhodnaBeseda)).toCharArray();
    klic = stanje + " " + vhodnaBeseda[++vhodnaBesedaSt];
} else {
    klic = stanje + " " + vhodnaBeseda[vhodnaBesedaSt];
}
```

Slika 35: Det. Turingov stroj – neomejen trak

Večja razlika v kodi programa nastane pri primerjavi z nedeterminističnim Turingovim strojem. Zaradi usklajenosti imen je tokrat razred poimenovan `KoncenTrakEnaVec`, iz česar je razvidno, da se od prve izbire loči le po besedi `Vec`, ki je nadomestila besedo `Ena`, kar ponazarja, da imamo zdaj iz enega stanja na voljo več različnih prehodov stanj, v katere lahko gre stroj. Prva opazna razlika v programu je že na začetku, saj smo odstranili preverjanje števil prehodnih stanj, saj jih zdaj lahko imamo več. Preostali del kode pa se je nadgradil z novimi spremenljivkami in malo drugačnim pristopom do reševanja problema. Pri prejšnjih simulacijah je Turingov stroj izvajal svoje prehode in ukaze, vse dokler se ni ustavil. Ko je prišel do konca, ga ni niti malo zanimalo, katere korake je naredil in kaj je zapisal na trak ali zakaj se je ustavil z napako. Ko je prevzel nove lastnosti stanja, je na prejšnje pozabil oziroma jih je nadomestil z novimi. Tokrat pa moramo imeti možnost, da se stroj vrne nazaj do določene točke, kjer naredi drugačno izbiro, kot jo je naredil prvič, in poskusi uspešno priti do svojega končnega stanja. Da pa se lahko vrne na neko točko, mora vedeti, kaj je počel, oziroma zapomniti si mora prehode stanj in spremembe na traku, ki jih je naredil. Slika 36 prikazuje, kako ob vsakem koraku v `seznamOperacij` shranimo trenutni klic, število vseh različnih vrednosti pri trenutnem klicu ključa, vrstno število trenutno izbrane vrednosti, niz znakov na traku, preden smo ga spremenili, in nazadnje še smer pomika po traku.

```
seznamOperacijPodrobno = new ArrayList();
seznamOperacijPodrobno.add(klic);
seznamOperacijPodrobno.add(listOfValues.size());
seznamOperacijPodrobno.add(valueSt);
seznamOperacijPodrobno.add(new String(vhodnaBeseda));
seznamOperacijPodrobno.add(smer);
seznamOperacij.add(seznamOperacijPodrobno);
```

Slika 36: Shranjevanje korakov simuliranja

Vse naštete lastnosti potrebujemo pozneje, ko hočemo povrniti določeno stanje in ubrati drugačno pot, kot smo jo prej. Vzemimo primer, kjer imamo prvo vrstico vhodne datoteke zapisano kot `q0 0 -> q0 0 R , q1 1 L`. Stroj začne z delovanjem, kjer je začetno stanje stroja `q0` in na traku zapis znakov `00`. Že ob prvem koraku stroja si zabeležimo njegovo delovanje, ki bi se glasilo `[q0 0,2,0,00,R]`. Po shranitvi program nadaljuje s simulacijo Turingovega stroja, kjer ob vsakem novem koraku znova shrani kombinacijo lastnosti. Tako si počasi gradi seznam vseh korakov, ki jih je naredil. V primeru, da Turingov stroj že v prvih poskusih pride v svoje končno stanje, potem je bilo beleženje korakov popolnoma odveč. Lahko pa se zgodi, da prva možnost ni tista, ki bi jo stroj moral izbrati. Zaradi napačne izbire stroj pride v stanje, ki ni njegovo končno stanje in iz katerega ne more več nadaljevati. V tem primeru pa lahko uporabimo beleženje korakov. Stroj pogleda v seznam korakov in prebere svoje zadnje dejanje, ki ga je naredil, in izvede obratne operacije,

da povrne predhodno stanje. Prebere predhodni zapis na traku in ga povrne. Števec, ki označuje pozicijo bralno-pisalne glave, spremeni svojo vrednost oziroma pozicijo glede na premik, ki se je izvedel. Če smo prej šli za en korak v levo, gremo zdaj za en korak v desno, in če smo prej šli za en korak v desno, gremo zdaj za en korak v levo (slika 37).

```
seznamOperacijPodrobno = new ArrayList();
//zadnji dodan element
seznamOperacijPodrobno=(ArrayList)seznamOperacij.get(seznamOperacij.size()-1);
//povrnemo vhodno besedo - zapis na traku
vhodnaBeseda=(char[])seznamOperacijPodrobno.get(3).toString().toCharArray();
if (seznamOperacijPodrobno.get(4).toString().contains("L")){//povrnemo stevec
    vhodnaBesedaSt++;
} else if (seznamOperacijPodrobno.get(4).toString().contains("R")) {
    vhodnaBesedaSt--;
}
```

Slika 37: Povrnitev predhodnega stanja

Nato preveri, ali bi pri zadnjem koraku lahko izbral kakšno drugo možnost, kot jo je izbral prej. Če v zadnjem koraku ni imel izbire, potem ta korak izbriše iz seznama korakov in se osredotoči na predzadnji korak, kjer spet najprej povrne stanje do tega koraka. Vzemimo, da tudi v predzadnjem koraku ni imel več kot le eno izbiro, zato spet izbrišemo ta zapis koraka in se prestavimo še za en korak nazaj, kjer spet povrnemo stanje, kot je bilo pred izvedbo tega koraka. Tu pa vidimo, da je imel stroj dve možni izbiri in se je odločil za prvo. Zato tokrat izbere drugo izbiro in nadaljuje s simulacijo, kjer spet shranjuje vsak svoj korak in tokrat pride do svojega končnega stanja. Slika 38 prikazuje del kode, kjer stroj preveri, ali ima trenutno na voljo še kakšno drugo izbiro, in če jo ima, potem jo uporabi, drugače pa nadaljuje s povratkom še enega koraka.

```
//ce obstaja druga pot, jo uporabimo
if((Integer)seznamOperacijPodrobno.get(1)>(Integer)seznamOperacijPodrobno.get(2)+1){
    GlavnoOkno.jTextArea1.append("\nUporabimo drugo izbiro.\n");
    seznamOperacijPodrobno.set(2, (Integer) seznamOperacijPodrobno.get(2) + 1);
    valueSt = (Integer) seznamOperacijPodrobno.get(2);
    novaPot = true;
    konecPrograma = false; //ker imamo druge poti
    klic = seznamOperacijPodrobno.get(0).toString();
} else {
    novaPot = false;
}
```

Slika 38: Preverjanje možnosti druge izbire

Takšno vračanje po korakih in izbiranje drugih vrednosti oziroma možnosti pa poteka, vse dokler Turingov stroj ne pride do svojega končnega stanja ali ko je stroj že pregledal vse

možne izbire in ni prišel do končnega stanja. To ponavljanje smo zapisali v zanko `while`, ki se glasi `while ((!seznamOperacij.isEmpty()) && novaPot == false)`. Iskanje novih izbir poteka, vse dokler nismo pregledali vseh možnih izbir ali našli neko novo pot oziroma izbiro, ki je še nismo uporabili. Ker pa shranjujemo korake, bi lahko rekli, da se ne ustavimo, dokler imamo še kakšen neuporabljen korak na voljo. Ko pa nam še teh zmanjka, se stroj ustavi.

Predzadnja izbira, ki jo imamo na voljo, je `Nedet. Turingov stroj - levo omejen trak`, katere razred je poimenovan `NeskoncenTrakDesnoEnaVec`, ki se od prejšnje ne razlikuje skoraj nič, razlika je le pri pogojnih stavkih, kjer preverjamo, ali smo prišli do konca traku ali ne. Tudi tokrat dovolimo neskončen trak le v desno smer (slika 39).

```
} else if (vhodnaBesedaSt >= vhodnaBeseda.length) {
    //na desni strani besede dodamo 'B'
    vhodnaBeseda = (new String(vhodnaBeseda) + 'B').toCharArray();
    klic = stanje + " " + vhodnaBeseda[vhodnaBesedaSt];
    valueSt = 0; //da zopet zacnemo s prvim elementom
} else if (vhodnaBesedaSt < 0) {
    GlavnoOkno.jTextArea1.append("Šli smo preveč levo!\n");
    GlavnoOkno.ikona = "No";
    konecPrograma = true;
```

Slika 39: Nedet. Turingov stroj – levo omejen trak

Zadnja izbira, ki se glasi `Nedet. Turingov stroj - neomejen trak`, z razredom `NeskoncenTrakLevoDesnoEnaVec`, se spet le malenkostno razlikuje v pogojnih stavkih za preverjanje končnosti traku (slika 40).

```
} else if (vhodnaBesedaSt >= vhodnaBeseda.length) {
    //na desni strani besede dodamo 'B'
    vhodnaBeseda = (new String(vhodnaBeseda) + 'B').toCharArray();
    klic = stanje + " " + vhodnaBeseda[vhodnaBesedaSt];
    valueSt = 0; //da zopet zacnemo s prvim elementom
} else if (vhodnaBesedaSt < 0) {
    //na levi strani besede dodamo 'B'
    vhodnaBeseda = ('B' + new String(vhodnaBeseda)).toCharArray();
    klic = stanje + " " + vhodnaBeseda[++vhodnaBesedaSt];
    valueSt = 0; //da zopet zacnemo s prvim elementom
```

Slika 40: Nedet. Turingov stroj – neomejen trak

4 ZAKLJUČEK

V diplomskem delu smo sprva predstavili uporabnost Turingovih strojev oziroma formulacije, s katerimi si lahko pomagamo pri reševanju različnih matematičnih problemov, kajti vsako matematično funkcijo, ki je izračunljiva, je mogoče predstaviti v obliki zapisa, ki velja za Turingov stroj. Zaradi posebne oblike zapisa pa bi velikokrat porabili veliko število različnih prehodnih funkcij, katerih količina bi precej kmalu postala nepregledna, zato se večina ljudi izogiba takšnemu zapisu.

Da bi uporabniku olajšali delo, smo izdelali uporabniški vmesnik, ki simulira delovanje Turingovega stroja. S tem uporabniku prihranimo veliko časa in tudi verjetnost napake je veliko manjša, kot če bi računali sami, brez pomoči računalnika. Opisali smo potek izdelave uporabniškega vmesnika, hkrati pa še malo opisali delovno okolje, v katerem smo po delih skupaj sestavili vse gradnike, ki so prisotni na uporabniškem vmesniku. Vsem gradnikom smo tudi določili različne lastnosti, ki so vplivale na delovanje in odzivanje programa. Poleg tega pa smo opisali tudi ostale metode, ki se skrivajo in delujejo v ozadju uporabniškega vmesnika, ampak smo jih vseeno morali napisati, da program deluje tako, kot je treba.

S tem smo omogočili, da uporabnik hitreje in učinkoviteje pride do želenih rezultatov. V simulaciji Turingovega stroja smo simulirali šest različnih Turingovih strojev, kjer ima vsak svoje edinstvene lastnosti. Čeprav se nekatere lastnosti med seboj ne razlikujejo prav veliko, je vseeno definiran nov Turingov stroj, saj je njegov odziv malo drugačen. Program bi lahko nadgradili s tem, da bi dodali še več različnih Turingovih strojev in s tem še bolj povečali možnost uporabe takšnega programa.

KAZALO SLIK

LITERATURA IN VIRI

[1] (2011) Alan Turing, dostopno na: http://en.wikipedia.org/wiki/Alan_Turing.

[2] (2011) Turing machine, dostopno na: http://en.wikipedia.org/wiki/Turing_machine.

[3] (2011) Turingov stroj, dostopno na: http://www.s-sers.mb.edus.si/gradiva/w3/sistemi/turing.html.

[4] (2011) Turingov stroj, dostopno na: http://sl.wikipedia.org/wiki/Turingov_stroj.

[5] (2011) Turing machines, dostopno na: http://plato.stanford.edu/entries/turing-machine/.

[6] J. E. Hopcroft in J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, USA: Addison-Wesley, 1979.

[7] (2011) Turingov stroj in Turingov generator, dostopno na: http://www.e-studij.si/Turingov_stroj_in_Turingov_generator.

[8] (2011) A Turing machine, dostopno na: http://www.aturingmachine.com/.

[9] (2011) Grafični uporabniški vmesnik, dostopno na: http://sl.wikipedia.org/wiki/Grafi%C4%8Dni_uporabni%C5%A1ki_vmesnik.

[10] (2011) Java, dostopno na: http://en.wikipedia.org/wiki/Java_(programming_language).

[11] (2011) Sun Microsystems, dostopno na: http://en.wikipedia.org/wiki/Sun_Microsystems.

[12] (2011) Oracle Corporation, dostopno na: http://en.wikipedia.org/wiki/Oracle_Corporation.

[13] (2011) C++, dostopno na: http://en.wikipedia.org/wiki/C%2B%2B.

[14] (2011) Notepad++, dostopno na: http://www.notepad-plus-plus.org/.

[15] (2011) Eclipse, dostopno na: http://www.eclipse.org/.

[16] (2011) NetBeans, dostopno na: http://en.wikipedia.org/wiki/NetBeans_IDE.