Turingov stroj: web-aplikacija

Kocijan, Mislav

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Mislav Kocijan

TURINGOV STROJ: WEB-APLIKACIJA

Diplomski rad

Voditelj rada:
doc. dr. sc. Vedran Čačić

Zagreb, Veljača, 2024.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Roditeljima i baki na potpori kroz cijelo moje obrazovanje i život.
Braći, Marti i Jakovu koje uskoro čeka slična muka s pisanjem vlastitih radova.
Fra Draganu na duhovnoj i konkretnoj pomoći.
Mentoru, doc. dr. sc. Vedranu Čačiću na stručnom vodstvu, velikom strpljenju i uloženom vremenu, te svima onima kojima će ova aplikacija biti od koristi.

Sadržaj

Uvod

Na internetu postoje brojni simulatori Turingovih strojeva. Ipak, ogromna većina njih u ime praktičnosti čini uskrate teoriji, jer nisu izgrađeni sa svrhom produbljivanja razumijevanja temelja teorije izračunljivosti, već sa svrhom što lakšeg rješavanja konkretnih praktičnih zadataka. Primjerice, većina njih ima obostrano neograničenu traku, koja je korisna kad treba na brzinu zapamtiti neki pomoćni podatak lijevo od ulaza, ali je teža za matematičko modeliranje.

Na kolegiju Izračunljivost obrađujemo Turingove strojeve, ali s osnovnom svrhom dokaza ekvivalentnosti s RAM-strojevima, pri čemu je mnogo prirodnije koristiti jednostrano neograničenu traku. Također, izvjesna ograničenja na pomak glave (točno za jednu ćeliju ulijevo ili udesno) čine modeliranje rada Turingova stroja (npr. parcijalno rekurzivnim funkcijama) jednostavnijim.

Iako im je osnovna svrha teorijska, u svrhu upoznavanja s radom takvih strojeva studenti često rješavaju zadatke, i pritom je dobro da imaju emulator koji koristi iste konvencije, kako bi lakše provjeravali svoja rješenja. U modernom svijetu web-aplikacije su portabilne i svugdje dostupne te predstavljaju idealni vid implementacije takvog emulatora. Implementacija i dokumentacija takve web-aplikacije je svrha ovog diplomskog rada.

Poglavlje 1

Izračunljivost jezičnih funkcija

Turingov stroj je jedan od (brojnih) modela izračunljivosti; konkretno, to je model koji izračunava jezične funkcije. Stoga u ovom poglavlju prije svega želimo navesti osnovne definicije i rezultate o jezicima, te jezičnim funkcijama, kao matematičkim objektima.

1.1 Abeceda i jezik

Mnogi pojmovi u ovom odjeljku, odnosno njihove definicije i notacije za njih, preuzete su iz [3].

Definicija 1.1.1. Neka je Σ konačni neprazni skup te neka je w proizvoljni konačni niz (uređena n-torka) elemenata iz Σ. Kažemo da je niz w **riječ** nad **abecedom** Σ, te da je skup Σ **abeceda riječi** w (ili samo **abeceda**). Elemente abecede zovemo **znakovima**.

Sa Σ^* označavamo skup svih riječi nad abecedom Σ. Proizvoljni podskup $L \subseteq \Sigma^*$ zovemo **jezikom** nad Σ, a cijeli Σ^* nazivamo **univerzalnim jezikom** nad abecedom Σ.

Primjer 1.1.2. Neka je $\Sigma = \{\mathtt{a}, \mathtt{d}, \mathtt{n}\}$ abeceda.
Nizovi $(\mathtt{d}, \mathtt{a}, \mathtt{n})$, $(\mathtt{n}, \mathtt{a}, \mathtt{d}, \mathtt{a})$ i $(\mathtt{d}, \mathtt{a}, \mathtt{d}, \mathtt{a})$ su riječi nad Σ.
Nizovi $(\mathtt{a})$, $(\mathtt{a}, \mathtt{a})$, $(\mathtt{a}, \mathtt{a}, \mathtt{a})$ i $(\mathtt{n}, \mathtt{a}, \mathtt{a}, \mathtt{a})$ također su riječi nad Σ.
Skup $\{(\mathtt{d}, \mathtt{a}, \mathtt{n}), (\mathtt{n}, \mathtt{a}, \mathtt{d})\}$ je jedan jezik nad abecedom Σ.
Skup $\{(\mathtt{n}, \mathtt{a}), (\mathtt{n}, \mathtt{a}, \mathtt{n}, \mathtt{a}), (\mathtt{n}, \mathtt{a}, \mathtt{n}, \mathtt{a}, \mathtt{n}, \mathtt{a}), \dots\}$ je još jedan (beskonačni) jezik nad Σ.

Neodređene znakove (znakovne varijable) pišemo malim grčkim slovima s početka alfabeta (α, β, γ), dok konkretne znakove pišemo u fontu fiksne širine ($\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{0}, \mathtt{1}$). Riječi obično označavamo slovima w, v ili u.

Riječ bez ijednog znaka (prazni niz, ili „uređenu nultorku") zovemo **praznom riječju** i označavamo je s ε.

Definicija 1.1.3. Neka je Σ abeceda i neka je $w \in \Sigma^*$ riječ nad Σ.
Broj svih znakova u riječi w zovemo **duljinom riječi** w i označavamo s $|w|$.

Definicija 1.1.4. Neka su $u = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ i $v = (\beta_1, \beta_2, \ldots, \beta_m)$ riječi nad istom abecedom. **Konkatenacija riječi** u i v je riječ $u \cdot v = uv := (\alpha_1, \alpha_2, \ldots, \alpha_n, \beta_1, \beta_2, \ldots, \beta_m)$.

Napomena 1.1.5. Slično kao kod umnoška brojeva, gdje izbjegavamo eksplicitno pisanje oznake množenja gdje god je to moguće, tako i točkicu za konkatenaciju uglavnom nećemo pisati. Riječ duljine jedan poistovjećujemo s njenim jedinim znakom (drugim riječima, svejedno je pišemo li $(\mathtt{a})$ ili $\mathtt{a}$) i analogno, jednočlani jezik poistovjećujemo s njegovim jedinim elementom (dakle, svejedno je pišemo li $\{w\}$ ili w). S time na umu, nepraznu riječ možemo pisati kao konkatenaciju znakova umjesto kao uređenu n-torku.

$$(\mathtt{l}, \mathtt{o}, \mathtt{p}, \mathtt{t}, \mathtt{a}) = (\mathtt{l})(\mathtt{o})(\mathtt{p})(\mathtt{t})(\mathtt{a}) = \mathtt{l} \cdot \mathtt{o} \cdot \mathtt{p} \cdot \mathtt{t} \cdot \mathtt{a} = \mathtt{lopta}$$

Definicija 1.1.6. Neka je $w = (\alpha_1, \alpha_2, \ldots, \alpha_n)$ riječ nad abecedom Σ.
Riječ $(\alpha_n, \ldots, \alpha_2, \alpha_1)$ nad Σ zovemo **reverzom** riječi w i označavamo je s w^{R}.

Definicija 1.1.7. Neka su L i M jezici nad istom abecedom Σ. **Konkatenacija jezika** L i M je jezik $LM := \{uv : u \in L, v \in M\}$. Induktivno definiramo i potenciranje jezika:

$$L^0 := \varepsilon,$$
$$L^{n+1} := L^n L.$$

Napomena 1.1.8. Uz poistovjećivanje jednočlanog jezika s njegovim jedinim elementom prema napomeni 1.1.5, iz potenciranja jezika lako dobivamo i potenciranje riječi, koje se ponaša onako kako bismo i očekivali:

$$\mathtt{abc}^3 = \{\mathtt{abc}\}^3 = \{\mathtt{abc}\}\{\mathtt{abc}\}\{\mathtt{abc}\} = \{\mathtt{abcabcabc}\} = \mathtt{abcabcabc},$$

ili općenitije

$$w^0 = \{w\}^0 = \{\varepsilon\} = \varepsilon,$$
$$w^{n+1} = \{w\}^{n+1} = \{w\}^n\{w\} = w^n w.$$

Definicija 1.1.9. Za proizvoljni jezik L definiramo:

- **Kleenejev plus** jezika L je jezik $L^+ := \bigcup_{n \in \mathbb{N}} L^k$;
- **Kleenejeva zvijezda** jezika L je jezik $L^* := \bigcup_{n \in \mathbb{N}_0} L^k$;
- **Kleenejev upitnik** jezika L je jezik $L^? := \bigcup_{n \in \{0,1\}} L^k = L^0 \cup L^1 = \varepsilon \cup L$.

Primjer 1.1.10. Riječ banana je riječ nad abecedom $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{n}\}$. Skupovi

$$L_1 = \{\mathtt{ba}(\mathtt{na})^k : k \in \mathbb{N}_0\} = \{\mathtt{ba}, \mathtt{bana}, \mathtt{banana}, \mathtt{bananana}, \dots\} \text{ i}$$
$$L_2 = \{\mathtt{a}^k\mathtt{b}^k : k \in \mathbb{N}_0\} = \{\varepsilon, \mathtt{ab}, \mathtt{aabb}, \mathtt{aaabbb}, \dots\}$$

su jezici nad abecedom $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{n}\}$. Primijetimo da je jezik L_1 konkatenacija jednočlanog jezika ba i Kleenejeve zvijezde jednočlanog jezika na; $L_1 = \mathtt{ba}(\mathtt{na})^*$.

Napomena 1.1.11. Samu abecedu jezika možemo također shvatiti kao jezik (znakovi su riječi duljine jedan) nad samom sobom te je u tom smislu njena Kleenejeva zvijezda Σ^* upravo skup svih riječi nad Σ, što opravdava oznaku iz definicije 1.1.1.

Napomena 1.1.12. Abecedu jezika smatramo dijelom njegovog identiteta. Jezik L_2 iz primjera 1.1.10 i jezik L_2' nad abecedom $\Sigma' = \{\mathtt{a}, \mathtt{b}\}$ definiran istim pravilom kao L_2, smatrat ćemo različitim jezicima iako su to skupovi s istim elementima, upravo jer su im abecede različite.

Definicija 1.1.13. Neka je Σ abeceda. Parcijalnu funkciju $\varphi : \Sigma^* \rightharpoonup \Sigma^*$ zovemo **jezičnom funkcijom** nad (abecedom) Σ.

Napomena 1.1.14. U definiciji 1.1.13 koristili smo pojam *parcijalne funkcije*. Najjednostavnije rečeno, parcijalna funkcija je funkcija koja ne mora biti definirana na svakoj točki svoje „domene" odnosno skupa koji bi joj trebao biti domena. Formalnije, to znači da im je domena zapravo podskup nekog većeg (univerzalnog) skupa koji promatramo.
Ako su dani skupovi A i B te funkcija $f : \mathcal{D}_f \to B$, takva da je $\mathcal{D}_f \subseteq A$, kažemo da je f parcijalna funkcija **iz** A u B što označavamo „polustrelicom" $f : A \rightharpoonup B$.
Ako je $\mathcal{D}_f = A$, tada kažemo i da je f **totalna** funkcija **sa** A u B što pišemo $f : A \to B$ s cijelom strelicom.

Napomena 1.1.15. Kao i kod jezika, za jezičnu funkciju je bitno nad kojom abecedom je definirana. Primjerice (uz oznake kao u napomeni 1.1.12), funkcije $\varphi : L_2 \to \Sigma^*$ i $\varphi' : L_2' \to \Sigma'^*$, obje definirane s $\varphi(w) := w^{\mathrm{R}}$, nećemo smatrati jednakima iako imaju isto pravilo pridruživanja (pa čak i „iste" domene).

Primjer 1.1.16. Za abecedu $\Sigma = \{\mathtt{a}, \mathtt{n}, \mathtt{s}\}$ definiramo funkciju $\varphi : \Sigma^* \to \Sigma^*$ pravilom pridruživanja

$$\varphi(\alpha_1\alpha_2 \dots \alpha_n) = \mathtt{a}\alpha_1\mathtt{a}\alpha_2 \dots \mathtt{a}\alpha_n\,.$$

Odnosno, ispred svakog znaka riječi w dodajemo znak a.
Tako je, na primjer, $\varphi(\mathtt{nns}) = \mathtt{ananas}$.

Primjer 1.1.17. (a) Neka je $\Sigma = \{\mathsf{a}\}$ abeceda. Primijetimo da je njen univerzalni jezik $\Sigma^* = \mathsf{a}^* = \{\mathsf{a}^k : k \in \mathbb{N}_0\}$ vrlo sličan (izomorfan) skupu $\mathbb{N}_0$. Za svaku funkciju $f : \mathbb{N}_0 \rightharpoonup \mathbb{N}_0$ možemo definirati jezičnu funkciju $\varphi_f : \Sigma^* \rightharpoonup \Sigma^*$ pravilom

$$\varphi_f(w) := \mathsf{a}^{f(|w|)}, \forall w \in \Sigma^* \text{ t.d. } |w| \in \mathcal{D}_f\,.$$

Obrnuto, za svaku jezičnu funkciju $\varphi : \Sigma^* \rightharpoonup \Sigma^*$ možemo definirati funkciju $f_\varphi : \mathbb{N}_0 \rightharpoonup \mathbb{N}_0$ s

$$f_\varphi(n) = |\varphi(\mathsf{a}^n)|, \forall n \in \mathbb{N}_0 \text{ t.d. } \mathsf{a}^n \in \mathcal{D}_\varphi\,.$$

Ovakve su transformacije međusobno inverzne; za svaku funkciju $g : \mathbb{N} \rightharpoonup \mathbb{N}$, te za svaku jezičnu funkciju $\psi : \mathsf{a}^* \rightharpoonup \mathsf{a}^*$ vrijedi:

$$f_{\varphi_g} = g\,, \quad \varphi_{f_\psi} = \psi.$$

(b) Pogledajmo funkciju $\varphi_{1/2}$ nad $\{\mathsf{a}\}$ koja riječi parne duljine pridružuje njezinu prvu (ili zadnju) polovicu; njezin pandan $f_{\varphi_{1/2}}$ je funkcija koja broju n pridružuje $\frac{n}{2}$, a koja nije definirana na neparnim brojevima, odnosno, domena joj je $2\mathbb{N}$.

1.2 Turingov stroj i izračunavanje

U literaturi postoje brojne varijante Turingova stroja; mi slijedimo Komputonomikon [2] te navodimo definicije odatle.

Definicija 1.2.1. Neka je Σ abeceda. **Turingov stroj** (nad Σ) je matematički (idealizirani) stroj, obično zapisan kao uređena sedmorka $\mathcal{T} = (Q, \Sigma, \Gamma, \sqcup, \delta, q_0, q_z)$, koji sadrži:

- konačnu **ulaznu abecedu** Σ i konačnu **radnu abecedu** $\Gamma \supsetneq \Sigma$, s istaknutim elementom $\sqcup \in \Gamma \setminus \Sigma$ (**praznina**);
- konačni skup **stanja** Q, s istaknutim elementima $q_0 \in Q$ (**početno stanje**) i $q_z \in Q$ (**završno stanje**);
- konačnu **funkciju prijelaza** $\delta : (Q \setminus \{q_z\}) \times \Gamma \longrightarrow Q \times \Gamma \times \{-1, 1\}$.

Definicija 1.2.2. Neka je $\mathcal{T} = (Q, \Sigma, \Gamma, \sqcup, \delta, q_0, q_z)$ Turingov stroj. **Konfiguracija** od $\mathcal{T}$ je bilo koja uređena trojka $(q, n, t) \in Q \times \mathbb{N}_0 \times \Gamma^{\mathbb{N}_0}$, takva da je niz t skoro svuda $\sqcup$ (odnosno, $t^{-1}[\Gamma \setminus \{\sqcup\}]$ je konačan skup). Komponente konfiguracije zovu se redom **stanje**, **pozicija** i **traka**. Konfiguracija je **završna** ako joj je stanje završno (q_z). **Početna konfiguracija** s ulazom $w = \alpha_0\alpha_1 \ldots \alpha_{|w|-1} \in \Sigma^*$ je trojka $(q_0, 0, w_\sqcup \ldots)$, čija je traka definirana s $(w_\sqcup \ldots)_i := (\alpha_i \text{ ako je } i < |w|, \text{ a inače } \sqcup)$.

Za konfiguracije $c = (q, n, t)$ i $d = (q', n', t')$ Turingova stroja $\mathcal{T}$ kažemo da c **prelazi** u d i pišemo $c \rightsquigarrow d$, ako je c završna i $c = d$ (pišemo $c \circlearrowleft$), ili uz oznake $\delta(q, t_n) =: (p, \beta, d)$ vrijedi $q' = p$, $n' = \max\{n + d, 0\}$, $t'_n = \beta$ te $t'_i = t_i$ za sve $i \in \mathbb{N}_0 \setminus \{n\}$.

Konfiguraciju možemo shvatiti kao „stanje" Turingovog stroja — ne stanje u smislu elementa skupa Q, nego kao stanje u kojem se cijeli stroj može naći tijekom svog rada. To „stanje" u širem smislu sadrži informaciju o stanju (u užem smislu) $q \in Q$, zatim informaciju o tome kako trenutno izgleda traka po kojoj se piše (niz t), te na kraju informaciju o tome koji znak na traci stroj trenutno gleda, odnosno iznad kojeg mjesta na traci se glava trenutno nalazi. Traku možemo zamisliti kao s jedne strane ograničen, a s druge strane neograničen, niz ćelija u kojem su od nekog mjesta nadalje samo prazne ćelije (one u kojima piše praznina).

Lema 1.2.3. *Svaka konfiguracija Turingova stroja prelazi u točno jednu konfiguraciju.*

Dokaz. Neka je $\mathcal{T}$ Turingov stroj te $c = (q, n, t)$ proizvoljna njegova konfiguracija. Ako je $q = q_z$, tada $c \rightsquigarrow c$, i ni u koju drugu konfiguraciju jer δ nije definirana u (q_z, t_n). Ako pak c nije završna, postoje jedinstveni p, β i d takvi da je $\delta(q, t_n) = (p, \beta, d)$, koji jednoznačno (zajedno s q, n i t) određuju $q' := p$, $n' := \max\{n + d, 0\}$ i niz $t' := (t'_j)_{j \in \mathbb{N}_0}$ (gdje je $t'_n = \beta$, a $t'_j = t_j$ za sve ostale $j \neq n$) takve da $c \rightsquigarrow (q', n', t')$. □

Drugim riječima, relacija „prelazi u" (oznake $\rightsquigarrow$) na skupu svih konfiguracija (nekog fiksiranog Turingova stroja) ima funkcijsko svojstvo.

Definicija 1.2.4. Neka je Σ abeceda, neka je $w \in \Sigma^*$ riječ te neka je $\mathcal{T}$ Turingov stroj nad Σ. **$\mathcal{T}$-izračunavanje** s w je niz $(c_n)_{n \in \mathbb{N}_0}$ konfiguracija od $\mathcal{T}$, takav da je c_0 početna konfiguracija s ulazom w, a za svaki $i \in \mathbb{N}_0$, $c_i \rightsquigarrow c_{i+1}$. Kažemo da izračunavanje **stane** ako postoji $n_0 \in \mathbb{N}_0$ takav da je c_{n_0} završna konfiguracija.

Definicija 1.2.5. Neka je Σ abeceda, neka φ jezična funkcija nad Σ te neka je $\mathcal{T}$ Turingov stroj nad Σ. Kažemo da $\mathcal{T}$ **računa** φ ako za sve $w \in \Sigma^*$ vrijedi:

- Ako je $w \in \mathcal{D}_\varphi$, tada $\mathcal{T}$-izračunavanje stane i završna konfiguracija mu je oblika $(q_z, n, \varphi(w)\sqcup \dots)$ za neki $n \in \mathbb{N}_0$ (pozicija nije bitna).
- Ako $w \notin \mathcal{D}_\varphi$, tada $\mathcal{T}$-izračunavanje s w ne stane.

Jezična funkcija φ je **Turing-izračunljiva** ako postoji Turingov stroj koji je računa.

Primjer 1.2.6. Neka je $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ abeceda i $\varphi : \Sigma^* \rightharpoonup \Sigma^*$ funkcija nad njom, definirana s $\varphi(\alpha_0\alpha_1 \dots \alpha_{n-1}\alpha_n) := \alpha_0\alpha_1 \dots \alpha_{n-1}$ (svakoj riječi obrišemo zadnji znak). Funkcija je definirana na svim riječima osim ε (duljina riječi mora biti barem jedan da bi postojao zadnji znak) što znači da φ nije totalna. Konkretno, $\mathcal{D}_\varphi = \Sigma^+ \subsetneq \Sigma^*$.
Definirajmo sada Turingov stroj koji računa φ. Neka je

$$\mathcal{T} := (\{q_p, q_m, q_b, q_g, q_z\}, \Sigma, \Sigma \cup \{\sqcup\}, \sqcup, \delta, q_p, q_z)$$

Turingov stroj čiju funkciju prijelaza δ definiramo po slučajevima:

$$\delta(q,\alpha) := \begin{cases} (q_m, \alpha, 1) & \text{ako je } q \in \{q_p, q_m\} \text{ i } \alpha \neq \text{␣} \\ (q_g, \text{␣}, -1) & \text{ako je } q = q_p \text{ i } \alpha = \text{␣} \\ (q_b, \text{␣}, 1) & \text{ako je } q = q_m \text{ i } \alpha = \text{␣} \\ (q_z, \text{␣}, -1) & \text{ako je } q = q_b \\ (q_g, \alpha, -1) & \text{ako je } q = q_g \end{cases}$$

Stanja možemo označiti bilo kako, ali uglavnom ih želimo označiti tako da nam pomognu razumjeti cijeli stroj (ili bar tako da nas ne zbunjuju). U ovom primjeru smo ih označili s $q_{\text{<slovo>}}$ s početnim slovima riječi „početak", „micanje", „brisanje", „greška" i „završno". Naš Turingov stroj bi jednako dobro radio da smo stanja označili brojevima, $Q' = \{1,2,3,4,5\}$ ili da smo umjesto $q_{\text{<slovo>}}$ pisali samo <slovo>, $Q'' = \{p, m, b, g, z\}$. Potonje možda nije najpametnije budući da će tada stanje b biti slično znaku b, ali taj problem možemo izbjeći primjerice koristeći velika slova.

Pokažimo sada da $\mathcal{T}$ doista računa funkciju φ. Neka je $w \in \Sigma^* = \{\mathsf{a}, \mathsf{b}\}^*$. Ako je $w = \varepsilon$, tada je početna konfiguracija $c_0 := (q_p, 0, t)$ gdje je $t := \text{␣}\ldots$, to jest $t_i := \text{␣}$ za sve $i \in \mathbb{N}_0$. Vrijedi da je $\delta(q_p, t_0) = \delta(q_p, \text{␣}) = (q_g, \text{␣}, -1)$ stoga c_0 prelazi u $c_1 := (q_g, 0, \text{␣}\ldots)$. Nadalje, jer je $\delta(q_g, \text{␣}) = (q_g, \text{␣}, -1)$, vidimo da konfiguracija c_1 prelazi u konfiguraciju $c_2 := (q_g, 0, \text{␣}\ldots) = c_1$, odnosno u samu sebe (iako nije završna). Budući da svaka konfiguracija prelazi u jedinstvenu konfiguraciju, c_2 će prijeći opet u samu sebe i tako u nedogled. Formalnije, $\mathcal{T}$-izračunavanje s ε je niz $(c_n)_{n\in\mathbb{N}_0}$, gdje je $c_0 = (q_p, 0, \text{␣}\ldots)$ i $c_i = (q_g, 0, \text{␣}\ldots)$, za $i \neq 0$. Nijedna od konfiguracija nije završna, stoga $\mathcal{T}$-izračunavanje s ε **ne** stane što smo i htjeli pokazati (jer $\varepsilon \notin \mathcal{D}_\varphi$).

Ako je $w \neq \varepsilon$ tada je $n := |w| - 1 \geq 0$ i vrijedi da je $w = \alpha_0 \ldots \alpha_n$ (za neke $\alpha_i \in \Sigma$, $i \in \{0, \ldots, n\}$). Početna konfiguracija će biti $d_0 := (q_p, 0, w\text{␣}\ldots) = (q_p, 0, \alpha_0 \ldots \alpha_n\text{␣}\ldots)$. Budući da nijedan α_i nije ␣, a posebno ni α_0 nije ␣, imamo da je $\delta(q_p, \alpha_0) = (q_m, \alpha_0, 1)$. Dakle, d_0 prelazi u $(q_m, 0+1, \alpha_0 \ldots \alpha_n\text{␣}\ldots) =: m_1$. Vrijedi i općenito:

$$m_i := (q_m, i, \alpha_0 \ldots \alpha_n\text{␣}\ldots) \rightsquigarrow (q_m, i+1, \alpha_0 \ldots \alpha_n\text{␣}\ldots) = m_{i+1}, \forall i \in \{1, \ldots, n\},$$

a jednom kad bude $i = n+1$, za traku t iz $d_{i+1} = (q_m, i, \alpha_0 \ldots \alpha_n\text{␣}\ldots) = (q_m, i, t)$ bit će $t_i = \text{␣}$, pa imamo $\delta(q_m, \text{␣}) = (q_b, \text{␣}, -1)$. Dakle, d_{n+1} prelazi u $b_n := (q_b, n, \alpha_0 \ldots \alpha_n\text{␣}\ldots)$, a jer je $\delta(q_b, \alpha) = (q_z, \text{␣}, -1)$, b_n prelazi u $z_{n-1} := (q_z, n-1, \alpha_0 \ldots \alpha_{n-1}\text{␣␣}\ldots)$. Konfiguracija z_{n-1} je završna, pa po definiciji prelazi u samu sebe ($z_{n-1} \circlearrowleft$).

Sa svime time na umu, $\mathcal{T}$-izračunavanje s $w \neq \varepsilon$ je niz $(d_i)_{i\in\mathbb{N}_0}$ definiran s:

$$d_i := \begin{cases} (q_p, 0, w\text{␣}\ldots) & \text{ako je } i = 0 \\ m_i & \text{ako je } i \in \{1, \ldots, n+1\} \\ b_n & \text{ako je } i = n+2 \\ z_{n-1} & \text{inače} \end{cases}$$

Sada, za $n_0 = n + 3 = |w| + 2$, a i za sve $j > n_0$, vrijedi da je $d_j = d_{n_0}$ završna, stoga izračunavanje stane, te vrijedi $d_{n_0} = z_{n-1} = (q_z, n-1, \alpha_0 \dots \alpha_{n-1} \sqcup \sqcup \dots)$ što je upravo $(q_z,\ n-1,\ \varphi(\alpha_0 \dots \alpha_n)\sqcup \dots) = (q_z, n-1, \varphi(w)\sqcup \dots)$, što smo i htjeli pokazati. Dakle, $\mathcal{T}$ računa φ.

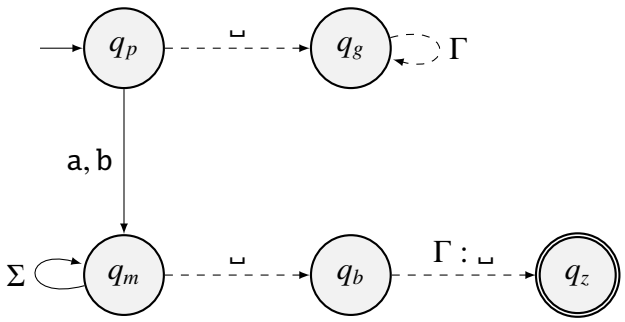
1.3 Metode prikazivanja Turingovih strojeva

Napomena 1.3.1. U praksi, rijetko koja funkcija prijelaza može biti ovako jednostavno zapisana kao što je bila zapisana funkcija δ iz primjera 1.2.6. Kada bismo htjeli formalno zapisati funkciju prijelaza nekog kompliciranijeg Turingova stroja, imali bismo mnogo više slučajeva, a svaki od slučajeva bi pokrivao vrlo mali broj vrijednosti koje funkcija poprima. Iz tog razloga, funkcije prijelaza Turingovih strojeva, budući da su konačne, najčešće se zapisuju tablicom i/ili dijagramom.

Na primjer, funkciju prijelaza iz primjera 1.2.6 mogli smo zadati tablicom:

δ	a	b	$\sqcup$
q_p	$(q_m, \mathsf{a}, +1)$	$(q_m, \mathsf{b}, +1)$	$(q_g, \sqcup, -1)$
q_m	$(q_m, \mathsf{a}, +1)$	$(q_m, \mathsf{b}, +1)$	$(q_b, \sqcup, -1)$
q_b	$(q_z, \sqcup, -1)$	$(q_z, \sqcup, -1)$	$(q_z, \sqcup, -1)$
q_g	$(q_g, \mathsf{a}, -1)$	$(q_g, \mathsf{b}, -1)$	$(q_g, \sqcup, -1)$

ili dijagramom:

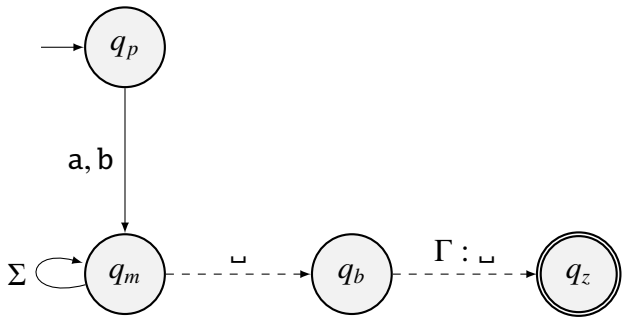


Recimo ponešto o konvencijama pri crtanju ovakvih dijagrama. Crtamo konačni usmjereni graf, čiji su vrhovi stanja, a bridovi prijelazi. Opće pravilo je da se prijelaz $\delta(p, \alpha) = (q, \beta, d)$ prikazuje kao strelica od vrha p prema vrhu q, na kojoj piše $\alpha : \beta$. Ako je $d = 1$ (pomak udesno) strelicu crtamo punom crtom, a ako je $d = -1$ (pomak ulijevo), crtamo je isprekidanom crtom.

Više prijelaza s istim p, q i d prikazujemo strelicom s više oznaka $\alpha_1 : \beta_1,\ \dots,\ \alpha_n : \beta_n$. Ako se znak ne mijenja ($\alpha = \beta$), umjesto $\alpha : \alpha$ pišemo samo α. Oznaku za skup $S \subseteq \Gamma$

možemo napisati na brid umjesto pojedinih elemenata (kao što smo na dijagramu pisali Σ i Γ). Početno stanje označavamo „strelicom niotkud" (kao što je na dijagramu označeno stanje q_p), a završno stanje označavamo dvostrukim krugom (kao što je označeno stanje q_z).

Primijetimo da nam i tablica i dijagram daju nešto više informacija nego što nam je ustvari potrebno. Ako znamo (odnosno želimo) da se nakon ulaska u neko stanje više nikad ne može doći do završnog stanja, tada takva stanja i prijelaze iz njih, odnosno u njih, nema potrebe crtati u dijagram, štoviše preporuča se izostaviti ih jer tako povećavamo preglednost dijagrama. U našem slučaju, takva stanja i prijelazi su samo stanje q_g i prijelazi prema njemu, te bez njih naš dijagram izgleda ovako:



Slična konvencija nije toliko uobičajena, ali je sasvim analogna za tablice; stanje q_g ne moramo pisati u tablicu funkcije δ (možemo izostaviti cijeli redak za to stanje) uz dogovor da se svi prijelazi oblika (q_g, α) preslikavaju u $(q_g, \alpha, -1)$ odnosno, ostajemo u stanju q_g, znak ne mijenjamo, a pomičemo se uvijek ulijevo. U ćelije s prijelazima koji vode prema q_g (iz nekog drugog stanja) možemo napisati samo q_g umjesto cijele trojke, podrazumijevajući nemijenjanje znaka i pomak u lijevo. U našem primjeru, takva ćelija je samo ona za slučaj $(q_p, \text{␣})$.

Primijetimo još jedan drugi prijelaz, onaj za slučaj $(q_b, \text{␣})$. Može se pokazati da nam taj prijelaz nikad neće trebati pri radu ovog Turinogvog stroja, odnosno za generiranje izračunavanja s bilo kojom riječi, nijednom neće biti potrebno znati vrijednost funkcije δ u točki $(q_b, \text{␣})$. Ipak funkcija prijelaza mora biti definirana na cijelom $Q \setminus \{q_z\} \times \Gamma$, pa smo ju dodefinirali s $(q_z, \text{␣}, -1)$, ali budući da tu vrijednost nikad nećemo koristiti, mogli smo ju definirati bilo kako. Radi lakšeg čitanja, ovdje i dalje u radu, takve prijelaze ćemo u tablici pisati sivom bojom. Tablica funkcije δ će tada izgledati ovako:

δ	a	b	␣
q_p	$(q_m, \text{a}, +1)$	$(q_m, \text{b}, +1)$	q_g
q_m	$(q_m, \text{a}, +1)$	$(q_m, \text{b}, +1)$	$(q_b, \text{␣}, -1)$
q_b	$(q_z, \text{␣}, -1)$	$(q_z, \text{␣}, -1)$	$(q_z, \text{␣}, -1)$

Napomena 1.3.2. Konfiguracije smo dosad pisali kao uređene trojke, po definiciji. U praksi se međutim koriste skraćeni zapisi, koji su kraći za pisanje i nešto lakši za čitanje. Konfiguraciju $c := (q, n, (t_m)_{m\in\mathbb{N}_0}) = (q, n, t_0t_1 \dots t_{n-1}t_nt_{n+1} \dots)$ skraćeno pišemo na jedan od dva načina:

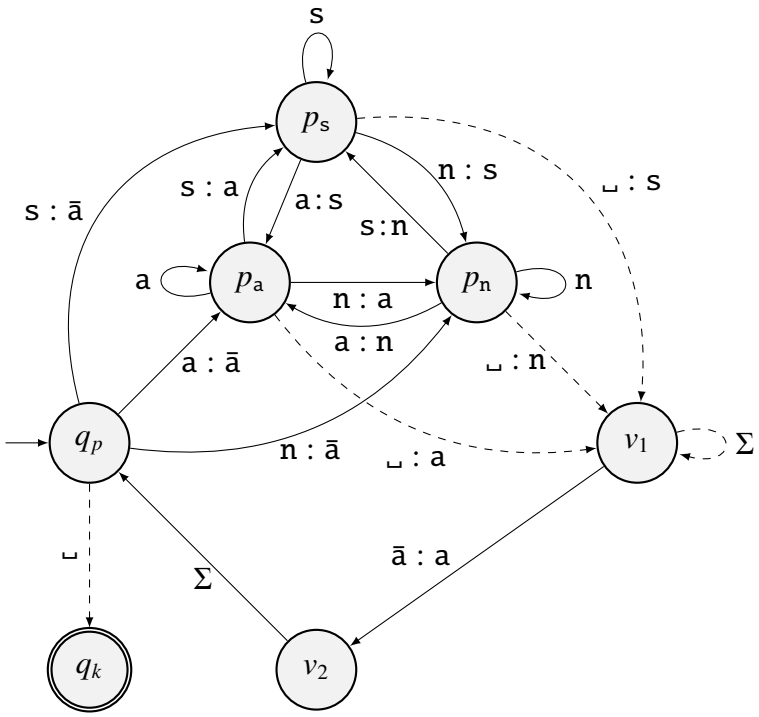
$$\underset{q}{t_0t_1 \dots t_{n-1}t_nt_{n+1} \dots} \qquad \text{ili} \qquad q : t_0t_1 \dots t_{n-1}\boxed{t_n}t_{n-1} \dots$$

odnosno, pišemo samo traku, bez zagrada, a oznaku trenutnog stanja pišemo ispod trenutno čitanog znaka ili iznad njega, ili je pišemo ispred cijele trake, a trenutno čitani znak označavamo (zaokružujemo) pravokutnikom. Oznaka za stanje ponekad može biti zapisana uz pravokutnik (netom ispred, iznad ili ispod njega), odnosno unutar pravokutnika zajedno s trenutno čitanim znakom (kao neka kombinacija prvog i drugog načina).
U takvim skraćenim zapisima ne pišemo (ali podrazumijevamo) $\sqcup \dots$ na kraju.

Primjer 1.3.3. Prisjetimo se funkcije φ iz primjera 1.1.16. To je totalna funkcija nad abecedom $\Sigma = \{\mathtt{a}, \mathtt{n}, \mathtt{s}\}$, definirana s

$$\varphi(\alpha_1\alpha_2 \dots \alpha_n) = \mathtt{a}\alpha_1\mathtt{a}\alpha_2 \dots \mathtt{a}\alpha_n\,.$$

Nađimo Turingov stroj koji je računa. Iako, dok pokušavamo definirati Turingov stroj, na prvi pogled izgleda da prvo definiramo skup stanja Q i radnu abecedu Γ, zapravo je prirodnije prvo „isprogramirati" funkciju prijelaza tako da radi ono što želimo, a tek onda ustanoviti koja stanja i pomoćni znakovi (oni iz $\Gamma \setminus \Sigma$) su nam potrebni.
Po napomeni 1.3.1, funkciju prijelaza definirat ćemo prvo dijagramom, a onda i tablicom.



δ	a	n	s	␣	ā
q_p	$(p_a, \bar{a}, +1)$	$(p_n, \bar{a}, +1)$	$(p_s, \bar{a}, +1)$	$(q_k, ␣, -1)$	$(q_p, \bar{a}, -1)$
p_a	$(p_a, a, +1)$	$(p_n, a, +1)$	$(p_s, a, +1)$	$(v_1, a, -1)$	$(q_p, \bar{a}, -1)$
p_n	$(p_a, n, +1)$	$(p_n, n, +1)$	$(p_s, n, +1)$	$(v_1, n, -1)$	$(q_p, \bar{a}, -1)$
p_s	$(p_a, s, +1)$	$(p_n, s, +1)$	$(p_s, s, +1)$	$(v_1, s, -1)$	$(q_p, \bar{a}, -1)$
v_1	$(v_1, a, -1)$	$(v_1, n, -1)$	$(v_1, s, -1)$	$(q_p, ␣, -1)$	$(v_2, a, +1)$
v_2	$(q_p, a, +1)$	$(q_p, n, +1)$	$(q_p, s, +1)$	$(q_p, ␣, -1)$	$(q_p, \bar{a}, -1)$

Dakle, uz $Q := \{q_p, p_a, p_n, p_s, v_1, v_2, q_k\}$, $\Gamma := \{a, n, s, ␣, \bar{a}\}$ i već zadanu ulaznu abecedu Σ, možemo definirati Turingov stroj $\mathcal{T} = (Q, \Sigma, \Gamma, ␣, \delta, q_p, q_k)$.

U ovom primjeru nećemo dokazivati da $\mathcal{T}$ doista računa φ, nego ćemo samo pokazati kako izgleda izračunavanje s riječi $w = nns$. Početna konfiguracija izačunavanja s nns je $c_0 := (q_p, 0, nns␣\ldots)$ ili pisano u skraćenoj oznaci iz napomene 1.3.2, početna konfigura-

cija je $c_0 = \underset{q_p}{\mathtt{nns}}$. Preostale konfiguracije slijede iz početne:

$$\underset{q_p}{\mathtt{nns}} \rightsquigarrow \underset{p_\mathtt{n}}{\mathtt{\bar{a}ns}} =: c_1 \rightsquigarrow \underset{p_\mathtt{n}}{\mathtt{\bar{a}ns}} =: c_2 \rightsquigarrow \underset{p_\mathtt{s}}{\mathtt{\bar{a}nn␣}} \rightsquigarrow \underset{v_1}{\mathtt{\bar{a}nns}} \rightsquigarrow \underset{v_1}{\mathtt{\bar{a}nns}} \rightsquigarrow \underset{v_1}{\mathtt{\bar{a}nns}} \rightsquigarrow \underset{v_2}{\mathtt{anns}} \rightsquigarrow$$
$$\underset{q_p}{\mathtt{anns}} \rightsquigarrow \underset{p_\mathtt{n}}{\mathtt{an\bar{a}s}} \rightsquigarrow \underset{p_\mathtt{s}}{\mathtt{an\bar{a}n␣}} \rightsquigarrow \underset{v_1}{\mathtt{an\bar{a}ns}} \rightsquigarrow \underset{v_1}{\mathtt{an\bar{a}ns}} \rightsquigarrow \underset{v_2}{\mathtt{anans}} \rightsquigarrow \underset{q_p}{\mathtt{anans}} \rightsquigarrow \underset{p_\mathtt{s}}{\mathtt{anan\bar{a}␣}} \rightsquigarrow$$
$$\underset{v_1}{\mathtt{anan\bar{a}s}} \rightsquigarrow \underset{v_2}{\mathtt{ananas}} \rightsquigarrow \underset{q_p}{\mathtt{ananas␣}} \rightsquigarrow \underset{q_k}{\mathtt{ananas}} =: c_{19} \circlearrowleft$$

Došli smo do završne konfiguracije, pri čemu je na traci ispisano ananas, a to je upravo $\varphi(\mathtt{nns})$.

Poglavlje 2

Ukratko o programskom jeziku JavaScript

Prije nego što opišemo implementaciju same aplikacije, želimo dati kratki uvod u *JavaScript*, programski jezik u kojem je aplikacija implementirana. Uglavnom ćemo se držati onoga što piše u [1], s time da ćemo izostaviti najosnovnije stvari, pretpostavljajući da je čitatelj zna pisati i pratiti jednostavne programe te da je upoznat s programskim jezicima C i C++ ili s nekim njima sličnim jezikom.

Programski jezik JavaScript uveden je 1995. godine s ciljem da se dodaju programi u (inače statične) web-stranice unutar internetskog preglednika „Netscape Navigator". Nakon toga su ga prihvatili i svi ostali značajni grafički web preglednici. JavaScript je omogućio stvaranje modernih web-aplikacija — aplikacija s kojima je moguće raditi direktno, bez potrebe da se, za svaku obavljenu akciju, ponovo učitava cijela stranica (kao što je slučaj s PHP-formama).

Važno je napomenuti da JavaScript, osim samog imena, nema skoro nikakve veze s programskim jezikom *Java*. Java je u to vrijeme bila snažno promovirana te je dobivala na popularnosti, pa je tvrtka Netscape svom jeziku odlučila dati slično ime ne bi li i on dobio dio te rastuće popularnosti.

Nakon širenja i prihvaćanja JavaScript-a i izvan preglednika Netscape, javila se potreba za formalnom standardizacijom jezika. Tu standardizaciju obavila je *Ecma International organisation* po kojoj je i sam standard dobio ime — ECMAScript. U praksi se „JavaScript" i „ECMAScript" mogu koristiti ravnopravno kao dva imena za isti jezik.

2.1 Osnovni tipovi podataka

Brojevni tip (`Number`)

Mogućnost reprezentacije i pohrane brojeva te izvršavanja (barem) osnovnih aritmetičkih operacija, ne samo da je prirodni zahtjev koji bismo imali za neki programski jezik, već je upravo jedan od razloga zašto su ljudi uopće počeli programirati. Stoga gotovo svi programski jezici imaju bar jedan, a često i više tipova za reprezentaciju brojeva (neki jezici imaju samo brojevne tipove), pa tako i JavaScript ima tip za brojevne vrijednosti, tip `Number`. Za razliku od nešto nižih programskih jezika, kao što je C, JavaScript nema posebni tip za cijele i realne brojeve, nego su svi brojevi istog tipa.

Osnovne aritmetičke operacije označavaju se kao u većini programskih jezika, uz standardne znakove `+` , `-` , `*` i `/` za zbrajanje, oduzimanje, množenje i dijeljenje, te znak `%` za ostatak pri dijeljenju. Redoslijed izračunavanja možemo precizirati pomoću oblih zagrada, a kada se zagrade ne pišu, računanje se provodi prema uobičajenim prioritetima operatora, s time da `%` ima viši prioritet od zbrajanja i oduzimanja, a isti prioritet kao množenje i dijeljenje.

Vrijednost ovog tipa može biti bilo koji broj unutar određenog raspona, određene preciznosti, ili jedna od tri specijalne vrijednosti: `Infinity`, `-Infinity` (pozitivna i negativna beskonačnost) i `NaN` (*Not a Number* ili *nebroj*). Vrijednost `NaN` se javlja kao rezultat operacija koje nemaju matematičkog smisla (na primjer dijeljenje nule nulom ili oduzimanje beskonačnosti od same sebe) ili kao rezultat operacija u kojima jedan od operanada već jest `NaN`, te u još nekim slučajevima koje uglavnom želimo izbjeći.

Tekstni tip (`String`)

Još jedan tip podataka s kojim bismo hjeli raditi u nekom programskom jeziku su znakovi ili simboli (kao što je tip `char` u programskim jezicima C i C++-u), a zatim i nizovi simbola koji zajedno čine dijelove teksta. JavaScript ovdje „preskače jednu stepenicu” i omogućava samo spremanje nizova simbola u tipu `String`, a ako u nekoj varijabli želimo samo jedan simbol, jednostavno u nju stavimo string duljine jedan. (Primijetimo sličnost takve reprezentacije s konvencijom iz napomene 1.1.5.)

Stringove pišemo pomoću navodnika:

```
"ovo je neki tekst"
'ovo je neki drugi tekst'
`ovo je također tekst`
```

Jednostruki i dvostruki navodnici imaju gotovo istu uporabu: oba podržavaju zapise specijalnih znakova poput `\n` za označavanje prijelaza u novi red. *Backtick* ili kosi jednostruki navodnik omogućava pisanje stringova u više redova i još neke funkcionalnosti.

JavaScript podržava konkatenaciju stringova označenu znakom plus +, a vrlo je jednostavna i konverzija (pretvaranje) iz jednog tipa u drugi. Svaki se broj može pretvoriti u string, točnije u niz simbola koji predstavljaju decimalni zapis tog broja, a string, ako je u pravilnom obliku (pa čak i ako ima višak nula na početku), može se pretvoriti u broj.

```
a = Number("12"+"000"); // u a je spremljen broj 12000
b = String(7+8); // u b je spremljen string "15"

c = Number(12+34+"56"+78); // u c je spremljen broj 465678
d = Number("12"+34+56+78); // u d je spremljen broj 12345678
```

Ovdje moramo biti oprezni, jer osim eksplicitne konverzije (`Number(izraz)` ili `String(izraz)`) moguća je i implicitna konverzija; štoviše, ona je mnogo češća i u mnogo slučajeva prikrivena. U primjeru `c = Number(12+34+"56"+"78")` samo se prvi znak + interpretira kao znak za zbrajanje (koji zbraja 12 i 34), dok se već sljedeći znak + interpretira kao znak za konkatenaciju (koji konkatenira broj 46 i string "56"). Ovo je možda kontraintuitivno, prvi operand je broj, očekivali bi da se drugi operand „prilagodi" prvome i konvertira u broj, te da se izvrši zbrajanje, a događa se obrnuto; prvi operand se konvertira u string te se izvrši konkatenacija. JavaScript uglavnom daje prednost stringovima pred brojevima. Gdje god pišemo znak +, JavaScript će ga prvo pokušati interpretirati kao konkatenaciju. Čim je bar jedan od operanada string, izvršit će se konkatenacija, a samo u slučaju kad su oba operanada brojevi će se izvršiti zbrajanje.

Logički tip (`Boolean`)

Iako sve logičke operacije možemo obaviti samo koristeći samo brojevni tip, nije rijetkost da programski jezik ima posebni tip za *logičke* varijable, koje mogu poprimiti samo dvije vrijednosti. U JavaScript-u to je tip `Boolean` koji može poprimiti vrijednosti `false` i `true`. Logički operatori označavaju se oznakama uobičajenim u jeziku C: `&&` za logičko i, `||` za logičko ili, te `!` za logičko ne.

Iako ćemo rijetko eksplicitno koristiti varijable ovoga tipa, logičke vrijednosti će se često javljati implicitno, na primjer pri usporedbi dva broja ili dva stringa. Rezultat usporedbe bit će ili `false` ili `true`.

2.2 Varijable kao pokazivači

Jedna od prvih stvari koje možemo primijetiti kada prvi put učimo programirati u JavaScriptu jest njegova velika liberalnost, u smislu oblika izraza koje dozvoljava. Veliki broj

izraza koje bismo intuitivno i razumski smatrali nedozvoljenima, neispravnima ili nepotpunima, JavaScript će prihvatiti i interpretirati, možda ne uvijek na način na koji bismo očekivali, ali izvršit će program, bez poruke o grešci. Na primjer, zaboravmo li napisati točku sa zarezom na kraju linije koda, u većini slučajeva to neće biti problem. Linija će biti interpretirana kao da je točka sa zarezom tamo. Ili, ako pokušamo pomnožiti broj i string (koji se ne može pretvoriti u broj), nećemo dobiti poruku o grešci, ali rezultat množenja bit će `NaN`. Međutim, ako ih pokušamo zbrojiti, znak + neće biti interpretiran kao znak za zbrajanje nego kao znak za konkatenaciju (kao što smo već i vidjeli), pa čak i onda kada se string lako može pretvoriti u broj.

```
a = 4 * "dva"; // u a je spremljen broj NaN
a = 4 * "2" // u a je spremljen broj 8,
            // nedostaje ; ali to ne smeta
            //  string "2" se konvertira u broj
b = 4 + "dva"; // u b je spremljen string "4dva"
b = 4 + "2"; // u b je spremljen string "42"
            // string "2" se neće konvertirati u Number!!
```

Sličnu slobodu imamo i kod definiranja novih varijabli; ne moramo ih formalno deklarirati. Barem nam se tako na prvi pogled čini. Napišemo li

```
c = "jezik C ovo ne dozvoljava!";
```

a varijabla `c` se nigdje prije toga nije pojavljivala, pri izvršenju programa, stvorit će se globalna varijabla `c` i pridružit će joj se string. Ipak, ako želimo pisati uredan kod, te ne želimo da nam sve varijable u programu budu globalne, nove varijable možemo deklarirati pomoću ključne riječi `let`

```
let d;
d = "Daruvar";
let e = "Ernestinovo";
```

Ovako deklarirana varijabla će biti lokalna, te kao i u C-u, ona je „vidljiva”, to jest, može se dohvatiti samo unutar programskog bloka (dijela koda omeđenim vitičastim zagradama) u kojem je deklarirana.

Za deklaraciju novih varijabli možemo koristiti još i ključne riječi `const` o kojoj ćemo nešto više reći kasnije i `var` o kojoj nećemo govoriti u ovom radu, budući da se u novije vrijeme sve rijeđe koristi.

Još jedna neobičnost s varijablama jest njihova neosjetljivost ili nevzanost za tipove. Varijabli kojoj smo inicijalno pridružili broj, kasnije možemo pridružiti string i obrnuto. Obično kad govorimo o tipovima mislimo na tip *vrijednosti* (onoga što se sprema) a ne na

tip varijable (onoga u što se sprema). Nadalje, varijable u JavaScript-u se zapravo ponašaju slično kao pointeri ili reference u C-u. Nijedna varijabla u JavaScript kodu ustvari ne sadrži pridruženu vrijednost nego je ta vrijednost spremljena „negdje u pozadini", a naša varijabla samo „pokazuje" na tu vrijednost. Da stvari budu još čudnije, kada govorimo o osnovnim tipovima, te vrijednosti u pozadini su nepromjenjive (*immutable*), odnosno konstantne. Kada varijabli želimo pridružiti novu vrijednost, neće se promijeniti vrijednost na adresi na koju smo imali prije, nego će se stvoriti nova vrijednost (s novom adresom), a naša varijabla-pointer će početi pokazivati na tu novu vrijednost. Knjiga *Eloquent JavaScript* [1] predlaže zamišljati varijable kao krakove radije nego kutije ili posude (krakovi ne sadržavaju pridružene vrijednosti, oni ih samo hvataju i „drže se" za nih), te ih osim varijablama naziva još i „*binding*" (povez, vez ili veza). Mi ćemo ih ipak i dalje nazivati samo varijablama, imajući na umu da se one u pozadini ponašaju kao pokazivači, a primijetit ćemo da, kada radimo s osnovnim tipovima podataka, varijable u JavaScript-u se doista ponašaju kao (malo fleksibilnije) C-ovske varijable, dok ćemo za složenije tipove ionako imati drugačije nazive.

2.3 Složeniji tipovi podataka

Sve dosad navedene tipove možemo grupirati u složenije strukture podataka. Dvije najčešće strukture u JavaScript-u su polje i objekt, čiji su koncepti i sintakse u mnogome slični poljima i strukturama, odnosno klasama u C-u i C++-u.

Polje (`Array`)

Polja su strukture podataka s uređenim, odnosno numeriranim elementima i u JavaScript-u ih definiramo pomoću uglatih zagrada:

```
let polje1 = [3, 1, 4, 1, 5, 9];
let polje2 = ["a", "b", "c"];
```

Elemente polja dohvaćamo također pomoću uglatih zagrada, a dozvoljeno je dohvaćati i mjesta s većim indeksom od duljine polja. U tom slučaju, iščitana vrijednost s takvog mjesta će biti `undefined`, odnosno, ako pridružujemo vrijednost, polje će se samo produljiti do traženog indeksa i na mjesto s tim indeksom spremiti poslanu vrijednost.

```
let slovo1 = polje2[1]; // slovo1 = "b"
let slovo2 = polje2[5]; // slovo2 = undefined
polje2[5] = "d"; // polje2 se produljuje do indeksa 5
                 // i u njega se sprema "d"
                 // na indeksima 3 i 4 su prazna mjesta
```

```
slovo2 = polje2[3]; // slovo2 = undefined
slovo2 = polje2[5]; // slovo2 = "d";
```

Elementi jednog polja čak ni ne trebaju biti istoga tipa.

```
let polje3 = [14, "srijeda", "veljača", ["polje", "u polju"] ];
```

Ovo ni ne čudi, budući da se elementi polja, točnije, mjesta u polju ponašaju kao i varijable, odnosno, nemaju tip, mogu pokazivati (ili hvatati) bilo koju vrijednost, neovisno o tipu.

Objekt (`Object`)

Objekti su strukture podataka s neuređenim (ali imenovanim) elementima. Definiramo ih pomoću vitičastih zagrada.

```
let objekt1 = { ime: "lopta", boja: "plava" };
let objekt2 = { ime: "lopata", duljina: 1.4 };
```

Elemente objekata nazivamo *članovima* ili *svojstvima*, a dohvaćamo ih pomoću točke, ali i pomoću uglatih zagrada. Razlika je u tome što nakon točke treba pisati točno ime člana koji dohvaćamo, a u uglatim zagradama treba pisati izraz koji se pretvara u string koji treba predstavljati ime člana.

```
let string1 = objekt1.boja; // string1 = "plava"
let string2 = objekt2["ime"]; // string2 = "lopata"

let visina = "duljina";
let broj1 = objekt2[visina] // broj1 = 1.4;
objekt1["bo"+"ja"] = "zelena";
```

Iako je uobičajenije koristiti točku, u nekim slučajevima moramo koristiti uglate zagrade, na primjer, ako ime svojstva započinje brojem ili sadrži razmak.

```
let objekt3 = {
    "3d": "xyz",
    "ime svojstva": "vrijednost svojstva"
};

let str;
str = objekt3.3d; // greška !!
str = objekt3["3d"]; // može
```

```
str = objekt3.ime svojstva; // greška !!
str = objekt3["ime svojstva"]; // može
```

Istaknimo da smo pri definiciji objekta `objekt3` koristili navodnike da definiramo njegove članove (`3d` i `ime svojstva`), dok smo u definiciji objekata `objekt1` i `objekt2` imena svojstava pisali bez navodnika. To nije slučajnost: iz istog razloga zbog kojeg se svojstva objekta `objekt3` ne mogu dohvatiti pomoću točke nego samo pomoću uglatih zagrada, imena tih svojstava u definiciji objekta morali smo navesti unutar navodnika. Općenito pravilo je: ako ime svojstva može biti ime neke varijable, tada se to svojstvo može dohvatiti pomoću točke, a u definiciji objekta ime tog svojstva može biti navedeno bez navodnika. U suprotnom, ako ime svojstva ne može biti ime neke varijable (na primjer, jer započinje brojem ili sadrži razmak), tada ime svojstva uvijek mora biti navedeno unutar navodnika, a to svojstvo ne može biti dohvaćeno pomoću točke.

Za razliku od programskih jezika sa statičkim tipovima kao što su C i C++, u JavaScript-u nije potrebno definirati sve članove neke vrste objekta prije stvaranja instance takve vrste objekta. Imena i vrijednosti članova smo samo naveli unutar vitičastih zagrada i objekt je bio stvoren. U JavaScript-u, svi objekti su zapravo istog tipa, ali taj tip ne određuje broj i imena članova objekata. Svakom objektu možemo definirati članove neovisno o ostalim objektima. Štoviše, čak i nakon definicije nekog objekta, možemo mu dodavati i uklanjati članove. Slično kao i s poljima, čitanje s člana koji ne postoji vraća `undefined` (ali ne ruši program), pridruživanje članu koji ne postoji stvara novi član s danim imenom i pridružuje mu danu vrijednost, a član možemo ukloniti pomoću operatora `delete`.

```
str = objekt3.d3; // str2 = undefined
objekt3.d3 = "z"; // stvara se novo svojstvo d3
                  // i u njega se sprema string "z"
objekt3["ime_svojstva"] = objekt3["ime svojstva"];
// stvara se novo svojstvo, kopira se vrijednost iz starog
str = objekt3.ime_svojstva; // str = "vrijednost svojstva"

delete objekt3.d3; // uklanjamo novododano svojstvo d3
delete objekt3["ime svojstva"];
// uklanjamo staro svojstvo iz objekta, ostaje novo
```

U definiciji objekta koristili smo ključnu riječ `let`, kao i kod varijabli osnovnih tipova vrijednosti, odnosno stvorili smo novu varijablu (varijable `objekt1`, `objekt2`, ...). Nije rijetkost takve varijable, one kojima pridružujemo objekte, također nazivati objektima, međutim, te varijable nisu po ničemu drugačije od varijabli kojima pridružujemo vrijednosti osnovnih tipova. Također, kao i s osnovnim tipovima, varijable kojima pridružujemo objekt nikada ne *sadrže* cijele objekte već samo pokazuju na njih, a čitav objekt se nalazi „negdje u pozadini". Posljedica toga je da svaka varijabla može pokazivati na svaki objekt (ili

vrijednost) neovisno o njegovu tipu. Čak i više, tip pridružene vrijednosti neke varijable ne mora biti fiksan. Kao što smo varijabli koja je pokazivala na broj mogli pridružiti string, isto tako smo joj mogli pridružiti i objekt (ili polje). Doduše, takvo što je rijetko kad korisno; mi ćemo uglavnom svakoj varijabli kojoj je bila pridružena vrijednost nekog tipa pridruživati nove vrijednosti samo toga tipa, a ako je varijabli bio pridružen objekt (ili polje), tada toj varijabli nikad nećemo pridružiti nešto što nije objekt (ili polje). Štoviše, varijabli kojoj je već pridružen neki objekt, rijetko kada ćemo pridruživati išta novo. Češće ćemo samo mijenjati objekt koji joj je već pridružen, a i onda kad ćemo joj pridruživati novi objekt, taj će biti vrlo sličan starom. Time opravdavamo način izražavanja u kojem kažemo da je varijabla određenog tipa ako obično pokazuje na podatke tog tipa.

S druge strane, članovi nekog objekta (kao i elementi polja) također se ponašaju kao varijable; možemo im pridružiti bilo koju vrijednost koju možemo pridružiti nekoj varijabli. Posebno, članu nekog objekta možemo pridružiti neku složeniju strukturu, dakle polje ili drugi objekt. Ovo omogućava korisnu praksu definiranja objektata s podobjektima.

```
let trokut1 = {
    točkaA : { x: 0, y: 0, z: 0},
    točkaB : { x: 12, y: 0, z: 0},
    točkaC : { x: 0, y: 3, z: 4},
    opseg : 30
    površina : 30
}
```

Elemente podobjekta dohvaćamo ulančanim korištenjem točke ili uglatih zagrada. Na primjer, `trokut1.točkaC.y` ili `trokut1["točkaC"]["y"]`, a moguće je pisati i `trokut1["točkaC"].y`, odnosno `trokut1.točkaC["y"]`.

U radu s objektima i poljima bit će očitija pokazivačka priroda varijabli u JavaScriptu. Naime, dok god radimo s osnovnim tipovima vrijednosti, bilo nam je ustvari svejedno sadrži li varijabla vrijednost koju smo joj dodijelili ili samo pokazuje na nju, jer smo joj i dalje mogli lako pristupiti, a budući da je ta vrijednost bila nepromjenjiva, imali smo garanciju da se ta vrijednost neće promijeniti dok god ne promijenimo samu varijablu, dodjeljujući joj novu vrijednost. Kad radimo s objektima, moramo biti pažljiviji jer onaj objekt ili polje „u pozadini" na koji naša varijabla pokazuje više nije nepromjenjiv, stoga je moguće da dvije varijable koje pokazuju na isti objekt jedna drugoj mijenjaju podatke na neočekivan (ako smo navikli na jezike s vrijednosnom semantikom) i moguće nepovoljan način.

```
let objektA = { x:1, y:2 };
let objektB = objektA; // Ne kopira se objekt nego pokazivač !!
```

```
objektB.y = 3;
let y_vani =  objektA.y; // y_vani = 3, a ne 2 !!

// - -

let brojA = 2;
let brojB = brojA;

brojB = 3; // ne mijenja se dvojka nego se stvara novi broj
           // brojB sad pokazuje na novu vrijednost
           // brojA i dalje pokazuje na istu vrijenost
broj = brojA; // broj = 2
```

Nadalje, s ovim znanjem o objektima i varijablama kao pokazivačima možemo reći nešto više o ključnoj riječi `const`. Rekli smo da uz `let` i `var` za deklaraciju novih varijabli možemo koristiti i `const`, a njeno ime nam već govori da se radi o nekoj varijabli koja se ne može promijeniti. Da budemo precizniji, varijabla deklarirana s ključnom riječi `const`, već pri deklaraciji mora biti i definirana, dakle nakon imena varijable mora pisati znak = te neki izraz s desne strane (moramo joj odmah pridružiti vrijednost), a nakon toga, pa sve do kraja postojanja takve varijable, ona se ne smije pojaviti s lijeve strane znaka = (ne možemo joj pridružiti novu vrijednost). Međutim, iako je takva varijabla sama nepromjenjiva, vrijednost koja joj je pridružena ne mora biti!

U slučaju osnovnih tipova podataka nema razlike; njihove vrijednosti (one u pozadini) su ionako konstantne. Ali u slučaju polja i objektata, vrijednosti nisu nepromjenjive. Iako konstantna varijabla do kraja svog postojanja pokazuje na isti objekt, on može biti potpuno izmijenjen.

```
const a = 3.14;
const b = { vrijednost: 3.14 };

a = 3.1415; // greška
b = { vrijednost: 3.1415 }; // greška
// a i b ne smiju biti s lijeve strane znaka =

b.vrijednost = 3.1415; // ok
```

Već smo vidjeli da ime svojstva nekog objekta može započeti brojem, no ime svojstva može i čitavo biti broj, iako će to ustvari biti string (niz znamenki) budući da sva imena svojstava moraju biti stringovi. Svojstva s takvim imenima, dakako, ne možemo dohvaćati pomoću točke, nego moramo koristiti uglate zagrade, ali ne moramo pisati navodnike

unutar zagrada. Ovo ima smisla budući da u uglate zagrade pišemo izraz koji se može konvertirati u string. Ono što se može učiniti neobičnim jest da se već pri definiciji objekta imena takvih svojstava također mogu pisati bez navodnika.

```
let objekt4 = { 3: 123};
let broj2 = objekt[3]; // u broj2 je spremljen broj 123

objekt4["3"] = 456; // iako smo koristili string
                //dohvatili smo isto svojstvo
broj2 = objekt4[3]; // u broj2 je spremljen broj 456

objekt4[3] = 789;
broj2 = objekt4["3"]; // ponovo dohvaćamo isto svojstvo
                // u broj2 je spremljen broj 789
```

2.4 Funkcije i metode

Funkcije u JavaScript-u definiramo pomoću ključne riječi `function`, nakon koje slijedi popis parametara u oblim zagradama, a zatim tijelo funkcije u vitičastim zagradama.

```
let f = function(x)
{
    if(typeof x == "number") x = 2*x;

    return x;
};

//...

let rezultat = f(8);
// rezultat = 16

let poruka = f("str");
// poruka = "str"
```

Kao što je i do sad bio slučaj s varijablama, parametri funkcija (koji su također privremeno stvorene varijable) nemaju tip. To znači da funkcije za ulazni podatak mogu primiti bilo što, bez obzira na tip. Štoviše, ne samo da tip ulaznih podataka nije eksplicitno specificiran, nego čak ni broj ulaznih podataka ne mora odgovarati broju parametara navedenih u definiciji funkcije. Stoga moramo biti pažljivi kako definiramo i kako pozivamo.

Na primjer, možemo tijelo funkcije pisati tako da se dobro ponaša za bilo koji tip ulaznih podataka, te, ako je potrebno, provjerava tip ulaznog podatka (u prethodnom primjeru, funkcija `f` koristi operator `typeof` da provjeri je li ulazni podatak `x` doista broj kako bi ga mogla pomnožiti s dva), ili možemo pri svakom pozivu funkcije paziti da tipovi (i broj) parametra odgovaraju tipovima koje funkcija može primiti.

```
rezultat = f(); // rezultat = undefined
// nema ulaznog podatka
// zadana vrijednost parametra x je undefined
// tip undefined nije "number", ne obavlja se množenje

rezultat = f(4,5,6); // rezultat = 8
// uzima se prvi ulazni podatak, x=4
// ostali ulazni podatci se zanemaruju

// oba poziva funkcije se izvršavaju, nijedan ne ruši program
```

Primijetimo da smo za definiciju funkcije, pored ključne riječi `function`, koristili i ključnu riječ `let`. Drugim riječima, stvorenu funkciju (kod između znakova = i `;`) pridružili smo nekoj varijabli. Funkcije su dakle vrijednosti poput brojeva i stringova, te je varijabla kojoj pridružimo neku funkciju i dalje obična varijabla kojoj kasnije možemo pridružiti nešto drugo. To obično ne želimo — intuitivno, kad stvaramo novu funkciju, obično razmišljamo da funkciji „dajemo ime" a ne da je pridružujemo varijabli, stoga želimo da ta varijabla (to „ime funkcije") uvjek bude vezano uz tu funkciju. Iz ovog razloga, u praksi se za deklaraciju funkcija često koristi ključna riječ `const` umjesto `let`.

Postoji i nešto kraći način da se definira funkcija, sličniji onom iz jezika C odnosno C++ u kojem prvo pišemo ključnu riječ `function`, a zatim ime funkcije i zagrade s parametrima i tijelom funkcije:

```
function g(x)
{
    if(typeof x == "string") x = x+x;

    return x;
}
```

Kada definiramo funkciju na ovaj način, ne piše se točka sa zarezom na kraju definicije. Stvara se nova *globalna* varijabla (`g`) i njoj se pridružuje funkcija. Funkcije definirane na ovaj način dostupne su u cijelom programu u kojem su definirane, pa čak i ako se pozivaju „prije" mjesta na kojem su definirane. To nije slučaj s funkcijama definiranim pomoću `let`.

```
if(true)
{

    let a = f1(1); // greška! f1 jos nije definirana
    let b = f2(2); // može, b=2
    let f1 = function(x){return x;};
    function f2(x){return x;}
}

let c = f1(3); // greška! f1 više ne postoji
let d = f2(4); // može, d=4
```

(Ovdje koristimo `if(true)` samo radi stvaranja novog bloka koda.)

Rekli smo da su funkcije vrijednosti koje pridružujemo varijablama. Ako ih ne pozivamo (pišemo ih bez oblih zagrada) možemo s njima raditi isto kao i s ostalim varijablama — na primjer, pridruživati ih drugim varijablama, slati ih kao argumente drugim funkcijama (ili čak (pokušati) zbrojiti ih, što će pozvati konverziju u string), ali nama će biti puno korisnije spremati ih kao članove polja ili objekata. Već smo vidjeli da članu nekog objekta (ili elementu nekog polja) možemo pridružiti bilo što što možemo pridružiti nekoj varijabli (pa čak i cijeli neki drugi objekt ili polje), stoga je sasvim prirodno pridružiti mu funkciju, budući da su funkcije upravo to — vrijednosti koje pridružujemo varijablama. Međutim, definiranjem funkcije kao člana nekog objekta dobivamo još jednu korisnu funkcionalnost; funkcija koja je svojstvo nekog objekta ima „pristup" ostalim svojstvima tog objekta.

```
let objektSfunkcijom = {
    stanje: 0,
    brojPoziva: 0,
    zadnjiPoziv: 0,

    funkcija: function(broj)
    {
        this.stanje += broj;
        this.brojPoziva++;
        this.zadnjiPoziv = broj;

        return this.stanje;
    }
};
```

```
let a = objektSfunkcijom.funkcija(10);
// a = 10
// promjene unutar objekta objektSfunkcijom:
// stanje = 10, brojPoziva = 1, zadnjiPoziv = 10

let b = objektSfunkcijom.funkcija(-2);
// b = 8
// promjene unutar objekta objektSfunkcijom:
// stanje = 8, brojPoziva = 2, zadnjiPoziv = -2
```

Možemo to zamisliti kao da pri svakom pozivu funkcije koja je svojstvo, uz ostale argumente koji su poslani funkciji, funkcija prima i pokazivač na cijeli objekt kao još jedan skriveni argument. Tom implicitno poslanom argumentu pristupamo pomoću ključne riječi `this`. Funkcije koje su članovi nekog objekta obično nazivamo *metodama* tog objekta.

2.5 Klase (`class`)

Od 2015. godine u JavaScript je uvedena mogućnost definicije klasa pomoću ključne riječi `class`. Međutim, to je samo „*syntactic sugar*", odnosno pokrata za nešto kompliciranije. Jezik JavaScript sam po sebi zapravo ne nudi mogućnost stvaranja klasa, u smislu definiranja novog (specifičnog) tipa objekta. ipak, moguće je simulirati ih. U ovom radu nećemo pisati o tome kako se to točno izvodi, nego ćemo ukratko opisati kako se ta mogućnost koristi.

Sintaksa je slična onoj iz jezika C++. Kad stvaramo novu klasu, prvo pišemo ključnu riječ `class` a zatim ime nove klase i na kraju vitičaste zagrade unutar kojih navodimo metode klase.

```
class klasaSfunkcijom {
    constructor(pocetnoStanje)
    {
        if(typeof pocetnoStanje == "number")
            this.stanje = pocetnoStanje;
        else
            this.stanje = 0;

        this.brojPoziva = 0;
        this.zadnjiPoziv = 0;
    }

    funkcija(broj)
```

```
        {
            this.stanje += broj;
            this.brojPoziva++;
            this.zadnjiPoziv = broj;

            return this.stanje;
        }
    }
```

Ako jednu od metoda nazovemo ključnim imenom `constructor`, ona će se interpretirati kao konstruktor te klase, odnosno funkcija koja se poziva pri svakom stvaranju nove instance klase.

Instance klasa stvaramo pomoću ključne riječi `new`:

```
let objektSfunkcijom2 = new klasaSfunkcijom(16);
let objektSfunkcijom3 = new klasaSfunkcijom(8);

objektSfunkcijom2.funkcija(-8); // stanje = 8
objektSfunkcijom3.funkcija(8); // stanje = 16
```

Treba napomenuti da su ovako definirane instance klasa po gotovo svemu jednaki općenitom objektu u JavaScript-u (štoviše, `typeof objektSfunkcijom2` je `"object"`). Možemo im dohvaćati, mijenjati, dodavati ili brisati svojstva, čak i predefinirati metodu `funkcija`. U tom smislu, klase u JavaScript-u možemo više smisla ima shvaćati kao „šablone" za stvaranje novih objekata, a ne kao „tipove" objekta. Stvoreni objekt samo pri svom nastanku ima zadanu strukturu, koja se nakon toga može mijenjati.

Poglavlje 3

Implementacija aplikacije

3.1 Objekt `TS`

Pri implementaciji aplikacije, jedan od ciljeva je da varijable unutar same aplikacije budu što sličnije matematičkim objektima koje predstavljaju. Tako će i sam Turingov stroj bili prikazan kao nešto najsličnije uređenoj sedmorci iz definicije 1.2.1 što JavaScript može prikazati. To je, dakako, tip `Object` sa sedam svojstava. Moglo bi se prigovoriti da se time donekle gubi uređenost sedmorke, ali budući da svojstva možemo nazvati intuitivnim imenima (riječima, umjesto rednim brojevima), to nam neće stvarati probleme, te iako JavaScript omogućava definiciju `Array`-a kojima nisu svi elementi istog tipa, mnogo je prirodnije Turingov stroj prikazati kao općeniti objekt a ne kao polje.

S druge strane, `Array` je dobar izbor za reprezentaciju skupova. Iako skupovi nisu uređeni sami po sebi, uređenost koju ćemo dobiti reprezentacijom skupova `Array`-em će nam biti od koristi jer ćemo funkciju prijelaza definirati kao tablicu, odnosno kao dvodimenzionalno polje, odnosno kao niz čiji su elementi nizovi.

Da bi neki objekt predstavljao Turingov stroj u kontekstu ove aplikacije, taj objekt mora imati sedam svojstava, i to:

- `skupQ`, niz (`Array`) `String`-ova; svaki string treba predstavljati jedno stanje;
- `Sigma`, niz `String`-ova; svaki string treba predstavljati jedan znak iz ulazne abecede;
- `Gama`, niz `String`-ova; analogno kao `Sigma`, svaki string treba predstavljati jedan znak iz radne abecede;
- `praznina`, `String` koji predstavlja prazninu;
- `delta`, niz nizova koji na j-tom mjestu, i-tog niza ima objekt koji predstavlja vrijednost funkcije prijelaza za i-ti član niza `skupQ` i j-ti član niza `Gama`; taj objekt mora

imati svojstva `znak`, `q` i `d`, koji su redom dva `String`-a koji predstavljaju znak i stanje, te `Number` koji predstavlja pomak;

- `pocetno`, `String` koji predstavlja početno stanje;
- `zavrsno`, `String` koji predstavlja završno stanje.

Na primjer, Turingov stroj iz primjera 1.2.6 bismo u JavaScript-u mogli definirati kao objekt:

```
let TS1 = {
  skupQ:["qp","qm","qb","qg","qz"],
  Gama:["a","b","_"],
  Sigma:["a","b"],
  praznina:"_",
  delta:[
    [ {znak: "a", q: "qm", d: 1}, {znak: "b", q: "qm", d: 1},
      {znak: "_", q: "qg", d: -1} ],
    [ {znak: "a", q: "qm", d: 1}, {znak: "b", q: "qm", d: 1},
      {znak: "_", q: "qb", d: -1} ],
    [ {znak: "_", q: "qz", d: -1}, {znak: "_", q: "qz", d: -1},
      {znak: "_", q: "qz", d: -1} ],
    [ {znak: "a", q: "qg", d: -1}, {znak: "b", q: "qg", d: -1},
      {znak: "_", q: "qg", d: -1} ]
  ],

  pocetno:"qp",
  zavrsno:"qz"
};
```

Ovako definirani objekt `TS1` ima sve „informacije" koje nam trebaju za simulacju rada Turingova stroja u programu, te njegove članove lako možemo dohvatiti. Ako, na primjer, želimo evaluirati vrijednost funkcije prijelaza za neki zadani par stanja i znaka, možemo pisati:

```
let q = "qb"; // ili neki drugi zadani "q<slovo>"
let z = "b";

trojka = TS1.delta[TS1.SkupQ.indexOf(q)][TS1.Gama.indexOf(z)];
// trojka = TS1.delta[2][1];
// trojka = {znak: "_", q: "qz", d: -1};
```

Sada objekt `trojka` reprezentira uređenu trojku $(q_z, \sqcup, -1) = \delta(q_b, \mathtt{b})$ što nam je i trebalo.

3.2 Funkcija `txtTS`

Korisniku aplikacije želimo omogućiti što jednostavniji i što intuitivniji način unosa odnosno definicije željenog Turingova stroja. Jedan od povoljnih i praktičnih načina je pisanje „tekstne tablice" odnosno višelinijskog stringa koji je napisan tako da vizualno podsjeća na tablicu. Ideja je pomoću takvog stringa napisati definiciju funkcije prijelaza željenog Turingovog stroja, odnosno njezin tablični zapis po uzoru na napomenu 1.3.1. Primjerice, prisjetimo se tablice funkcije prijelaza iz te napomene:

δ	a	b	␣
q_p	$(q_m, \mathtt{a}, +1)$	$(q_m, \mathtt{b}, +1)$	$(q_g, ␣, -1)$
q_m	$(q_m, \mathtt{a}, +1)$	$(q_m, \mathtt{b}, +1)$	$(q_b, ␣, -1)$
q_b	$(q_z, ␣, -1)$	$(q_z, ␣, -1)$	$(q_z, ␣, -1)$
q_g	$(q_g, \mathtt{a}, -1)$	$(q_g, \mathtt{b}, -1)$	$(q_g, ␣, -1)$

te je zapišimo kao tekstnu tablicu (pišući imena stanja kao stringove):

```
"     a      b      _      \n
qp    a+qm   b+qm   _-qg   \n
qm    a+qm   b+qm   _-qb   \n
qb    _-qz   _-qz   _-qz   \n
qg    a-qg   b-qg   _-qg   "
```

Preciznije, u prvi redak tekstne tablice (dijela stringa do prvog prelaska u novi red) pišemo znakove radne abecede (kao što se pišu u zaglavlje tablice funkcije prijelaza) odvojene bar jednim razmakom. Nakon prvog retka pišemo po jedan redak za svako stanje (osim završnog). Svaki redak započinjemo oznakom stanja za taj redak, a nakon nje, za svaki znak radne abecede pišemo tekst (neformalno nazvan *trojka*) koji označava vrijednost funkcije prijelaza za to stanje i taj znak, i to tako da najprije pišemo znak, zatim, ako je pomak $d = 1$ pišemo znak +, a ako je $d = -1$ pišemo znak - i nakon toga, oznaku stanja.

Pisanjem tekstne tablice jednoznačno je određena funkcija prijelaza te skup stanja (osim završnog stanja) i radna abeceda nekog Turingovog stroja. Preostale komponente (ulazna abeceda, početno stanje, ...) nisu precizirani samom tablicom. Dogovor je za svaki Turingov stroj (ako se na drugi način ne naznači drugačije) uzeti da je oznaka za prazninu znak „ _ ". Za ulaznu abecedu uzimaju se znakovi zapisani ispred znaka za prazninu. Za početno stanje uzima se ono stanje koje je prvo navedeno (u prvom retku, nakon „zaglavlja" tekstualne tablice). Završno stanje označavamo točkom, dakle, umjesto `qz` pisat ćemo samo `.`.

Primijetimo da u tablici imamo zamjetni broj „ćelija" ili trojki koji ne mijenjaju stanje ili traku. Na primjer, u retku za stanje `qm` i stupcu za znak `a`. Radi pojednostavljenja

(ubrzanja) unosa, a i preglednijeg zapisa, uvodimo još jedan dogovor, da vrijednosti koje se ne mijenjaju ne pišemo u trojku. Preciznije, ako je za neko stanje q i znak α vrijednost funkcije prijelaza oblika (q, β, d) (ne mijenja se stanje), tada se u trojci ne mora pisati oznaka za stanje. Na primjer, u retku za stanje q_m i stupcu za znak `a`, umjesto `a+qm` možemo pisati samo `a+` . Analogno, ako je vrijednost funkcije prijelaza oblika (p, α, d) (ne mijenja se znak), oznaku znaka ne moramo pisati u trojku: primjerice, u retku za stanje q_p i znak `a`, umjesto `a+qm` možemo pisati `+qm`. Na posljetku, ako je vrijednost funkcije prijelaza oblika (q, α, d) (ni znak ni stanje se ne mijenjaju), ne moramo pisati ni znak ni stanje, to jest, pišemo samo + ako je $d = 1$, odnosno - ako je $d = -1$. Imajući sve to na umu, tekstnu tablicu bismo sada napisali ovako:

```
"    a      b      _      \n
qp   +qm    +qm    -qg    \n
qm   +      +      -qb    \n
qb   _-.    _-.    -.     \n
qg   -      -      -      "
```

Nadalje, prisjetimo se još jedne pokrate iz (iste) napomene 1.3.1. Ako za neko stanje znamo da se nakon ulaska u njega više ne može doći do završnog stanja, izostavljali smo cijeli redak. Sličnu pokratu želimo uvesti i za pisanje tekstne tablice. Ako je u jednoj trojci kao oznaka stanja napisan znak `!`, tada podrazumijevamo postojanje stanja (s oznakom `!`) koje ne pišemo u tablicu, te da je vrijednost funkcije u uređenom paru tog stanja i bilo kojim znakom jednaka uređenoj trojci tog istog stanja, tog istog znaka i minus jedinice (pomaka ulijevo). Primijetimo još da pri prijelazima u takvo stanje (iz nekog drugog stanja) nije bitno koji će novi znak biti napisan na traci, ali i više, nije bitan ni pomak — oboje možemo izostaviti. Slično vrijedi i za prijelaze (iz nekog drugog stanja) u završno stanje. Razlika je u tome što je pri prijelazu u završno stanje ponekad bitan znak koji treba biti zapisan na traci, ali pomak nam je i dalje nebitan, te ga ne moramo precizirati. Uz ove dogovore, tekstna tablica sada izgleda ovako:

```
"    a      b      _      \n
qp   +qm    +qm    !      \n
qm   +      +      -qb    \n
qb   _.     _.     .      "
```

Ovako pisanim stringovima (tekstnim tablicama) korisnik može precizirati bilo koji Turingov stroj ili bar njemu ekvivalentni (do na razliku u izboru oznaka), a zbog jednoznačnosti, aplikacija iz takvog stringa može konstruirati objekt koji predstavlja traženi Turingov stroj. Tu „pretvorbu" obavlja funkcija `txtTS(txt)`. Ona prima tekstnu tablicu, odnosno `String` (`txt`) te ako je sve u redu, vraća `Object` koji predstavlja Turingov stroj, konstruiran prema navedenim pravilima, odnosno, ako nešto nije bilo u redu, vraća *neki*

`Object` koji ima član `error`, `Array` `String`-ova koji sadrže kratak opis grešaka (primjer možemo vidjeti na slici 3.4).

3.3 Klasa `TSizracunavanje`

Slično kao kod reprezentacije Turingovog stroja (matematičkog objekta) u JavaScript-u, nastavljamo s istom idejom i kod reprezentacije izračunavanja. Krenimo od konfiguracija. Po definiciji, konfiguracije su uređene trojke, dakle, kao i ranije, u našoj aplikaciji će biti reprezentirane kao općeniti objekti s tri svojstva:

- `q`, `String` koji predstavlja ime stanja
- `n`, `Number` koji predstavlja poziciju
- `t`, `Array` koji predstavlja traku, odnosno polje `String`-ova koji predstavljaju znakove na traci

Prisjetimo se da je traka neke konfiguracije bila definirana kao (beskonačni) niz čiji su članovi od jednog mjesta pa nadalje svi jednaki $\sqcup$, odnosno praznini Turingova stroja koji promatramo. Za implementiranje aplikacije to znači da na početku imamo „korisni" dio niza čije znakove spremamo, a nakon određenog mjesta imamo samo praznine koje ne trebamo spremati.

Izračunavanje je bilo definirano kao niz, stoga za reprezentaciju izračunavanja najviše smisla ima koristiti polje, odnosno `Array`. Doduše, Objekt koji u našoj aplikaciji reprezentira izračunavanje nije sam `Array` nego je općeniti `Object` kojemu je jedan od članova `Array`. Jedan od razloga za to je što izračunavanje kao matematički objekt ima na neki način više informacija nego sam niz. Prisjetimo se definicije 1.2.4. Ona *za neki* $\mathcal{T}$ i w definira „$\mathcal{T}$-izračunavanje s w". Na neki način, u imenu konkretnog izračunavanja imamo podatak o Turingovu stroju kojem to izračunavanje „pripada" te koja je riječ ulazni podatak. Pišući program, htjeli bismo zadržati tu vezu. Stoga izračunavanje ima smisla reprezentirati kao objekt s dva svojstva; jedno za Turingov stroj i jedno za polje u koje ćemo spremati konfiguracije. Ovo nije samo filozofski razlog, nego i praktični. Kada stvaramo novo izračunavanje u programu, stvorit ćemo prazan niz, odnosno niz koji ima samo nulti element (početnu konfiguraciju). Ostale elemente ćemo postepeno dodavati, a za svaku iteraciju, trebat ćemo znati nešto o Turingovu stroju o kojem se radi. Ako ništa drugo, sigurno ćemo trebati znati funkciju prijelaza. Više smisla ima taj stroj imati već spremljen („ugrađen") u objekt koji promatramo, nego za svaku iteraciju pamtiti i tražiti neki vanjski objekt za Turingov stroj, a time i riskirati i dohvaćanje nekog drugog Turingovog stroja ili objekta koji uopće ne predstavlja Turingov stroj.

Imamo i još jedan podatak, ulaznu riječ w, ali za nju ne trebamo posebno svojstvo — bit će sadržana već u nultom elementu polja. Kao što smo već rekli, nulti element tog polja predstavljat će početnu konfiguraciju. Ako se radi o $\mathcal{T}$-izračunavanju s w, to će biti početna konfiguracija s ulazom w, dakle $(q_0, 0, w\sqcup\ldots)$, koja će biti reprezentirana nekim objektom

```
{q:"q0", n:0, t:[...]}
```

Ako imamo neki objekt `izr1` koji predstavlja izračunavanje s riječi w, predstavljene s nekim stringom, možemo pisati:

```
let k0 = izr1.niz[0];

let zadnji = k0.t.length - 1;

while (k0.t[zadnji] == izr1.TS.praznina && zadnji >= 0)
{
    zadnji--;
    //trazimo zadnji element trake koji nije praznina
}

let w = k0.t.slice(0,zadnji+1);
```

Sada će u varijabli `w` biti spremljen `Array` koji predstavlja ulaznu riječ w.

Drugi razlog za enkapsulaciju polja u objekt jest taj što smo željeli da izračunavanje bude reprezentirano objektom neke klase. Naime, kao što smo već spomenuli, kada u programu stvaramo novo izračunavanje imat ćemo polje sa samo jednim elementom (početnom konfiguracijom), a svaki sljedeći član moramo izvesti iz prethodnog. To izvođenje će obavljati neka (programska) funkcija koja bi i mogla biti biti implementirana izvan samog objekta, tako da prima konfiguraciju i Turingov stroj te vraća novu konfiguraciju (onu u koju primljena konfiguracija prelazi), ili tako da prima cijelo izračunavanje i vraća isto to izračunavanje s dodanom novom konfiguracijom na kraj polja. No s obzirom na to da se ta funkcija neće koristiti za ništa drugo osim za računanje sljedeće konfiguracije, više smisla ima imati je kao funkciju članicu samog objekta, odnosno svih objekata koji reprezentiraju izračunavanje. Štoviše, takvoj funkciji nećemo trebati prosljeđivati argumente, budući da će, kao funkcija članica objekta, imati pristup svim ostalim članovima, pa tako i svim konfiguracijama izračunavanja, posebno i zadnjoj konfiguraciji, te pripadajućem Turingovom stroju ako je i on član izračunavanja, što nam potvrđuje želju, da izračunavanje ima i Turingov stroj među svojim svojstvima.

U skladu s navedenim, definirali smo klasu `TSizracunavanje` s dvije metode (funkcije članice):

- `korak()`: ako je zadnja konfiguracija u polju `niz` završna, vraća `true`, a inače računa sljedeću konfiguraciju, dodaje ju na kraj polja i ako je nova konfiguracija završna vraća `true`, a ako nije, vraća `false`;
- `koraci(broj)`: računa nove konfiguracije i dodaje ih na kraj polja `niz` sve dok se u njemu ne pojavi završna konfiguracija ili konfiguracija s indeksom `broj` (odnosno `niz.length` postane `broj+1`), vraća indeks zadnje izračunane konfiguracije;

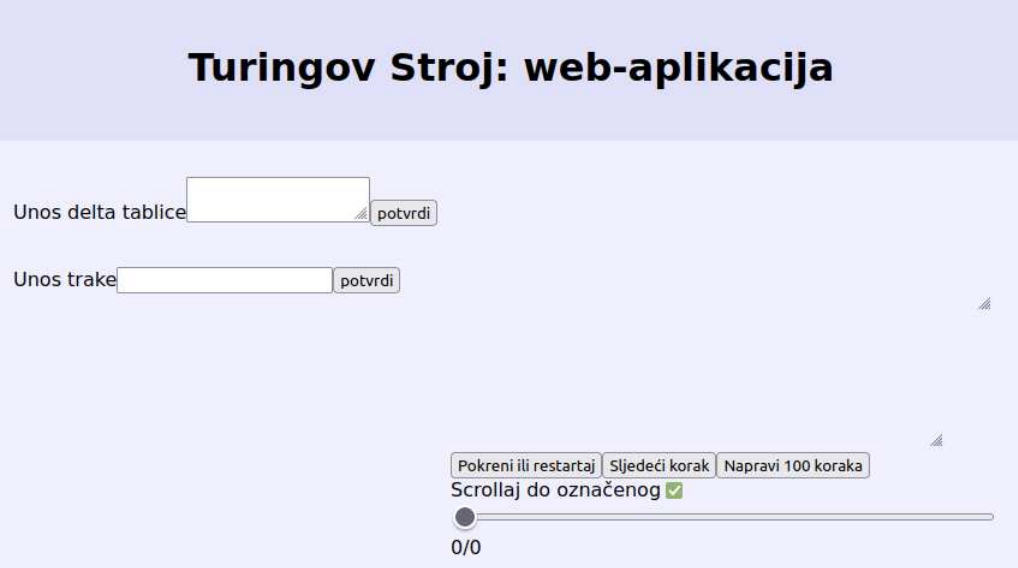
te konstruktorom koji prima jedan `Object` (Turingov stroj) i jedan `Array` (ulaznu riječ), tako da za svako novo izračunavanje i Turingov stroj i ulazna riječ moraju biti unaprijed poznati.

```
w = ["a", "a", "b"];
izr2 = new TSizracunavanje(TS1, w);
// stvara se objekt klase TSizracunavanje, te se postavlja
// izr2.TS = TS1  i  izr2.niz = [{q:"qz", n:0, t:w}]
```

Primijetimo sličnost ovakvog načina definiranja JavaScript-ova objekta s definicijom 1.2.4 (`TS1`-izračunavanje s `w`).

3.4 Primjer rada s aplikacijom

Kad otvorimo aplikaciju, na lijevoj strani vidimo dva okvira za unos teksta: jedan za unos tekstne tablice (`<textaera>`) i jedan za unos početnog stanja trake, odnosno ulazne riječi (`<input>`). Kako unosimo tekst u okvir za unos tekstne tablice, poziva se funkcija `txtTS`, odnosno uneseni tekst se parsira i u programu se stvara reprezentacija Turingovog stroja (objekt `TS`), a tablica njegove funkcije prijelaza prikazuje se desno od okvira za unos. Ovo se dešava dinamički, odnosno, pri svakoj promjeni teksta (pisanja ili brisanja) unutar okvira za unos, tekst se parsira i tablica se ažurira. Slično, za vrijeme unosu teksta u okvir za unos početnog stanja trake, svaka promjena pokreće parsiranje unesenog stringa u polje znakova iz abecede te njegovo ispisivanje ispod okvira za unos. Ovo se ponaša dobro čak i kad je neki znak abecede string duljine veće od jedan. U tom slučaju, parser pokušava prepoznati što dulji komad unesenog teksta kao znak ulazne abecede, a korisnik može precizirati kako će se znakovi prepoznavati unoseći razmak ili druge praznine (što uključuje na primjer tab `\t`, ali ne i oznaku za prazninu Turingova stroja).



Slika 3.1: Izgled aplikacije prije unosa tekstne tablice i ulazne riječi

δ	a	b	_
qp	qm, a, +1	qm, b, +1	!, _, -1
qm	qm, a, +1	qm, b, +1	qb, _, -1
qb	., _, -1	., _, -1	., _, -1
!	!, a, -1	!, b, -1	!, _, -1

Slika 3.2: Izgled aplikacije nakon unosa tekstne tablice i ulazne riječi

Kad završimo s unosom tekstne tablice i ulazne riječi, potvrdimo unos s gumbima za potvrđivanje. Pri potvrđivanju tablice funkcije prijelaza, ako nešto nije bilo uredu s unesenom tekstnom tablicom, ispisuje se popis grešaka. Ako nema poruka o greškama, možemo nastaviti. Potvrđivanje ulazne riječi omogućeno je samo kada je ulazna abeceda definirana i uneseni se tekst može prepoznati kao niz znakova iz abecede, ili ako nije uneseno ništa (ulazna riječ je ε). Tek kad potvrdimo ulaznu riječ, možemo pokrenuti izračunavanje gumbom za pokretanje (gumb „Pokreni ili restartaj"). Ako pokušamo pokrenuti izračunavanje prije nego potvrdimo ulaznu riječ, dobivamo obavijest (*alert*) da ulazna riječ nije definirana. Kad pokrenemo izračunavanje, u okviru iznad gumba za pokretanje ispisuje se skraćeni zapis početne konfiguracije (prema napomeni 1.3.2). Taj okvir je ustvari popis dosad izračunatih konfiguracija, a početna konfiguracija koja se ispisala, zapravo je samo prva (odnosno nulta) konfiguracija na popisu. Klikom na gumbe „Sljedeći korak" ili „Napravi 100 koraka" izračunat će se nove konfiguracije i dodat će se na popis.

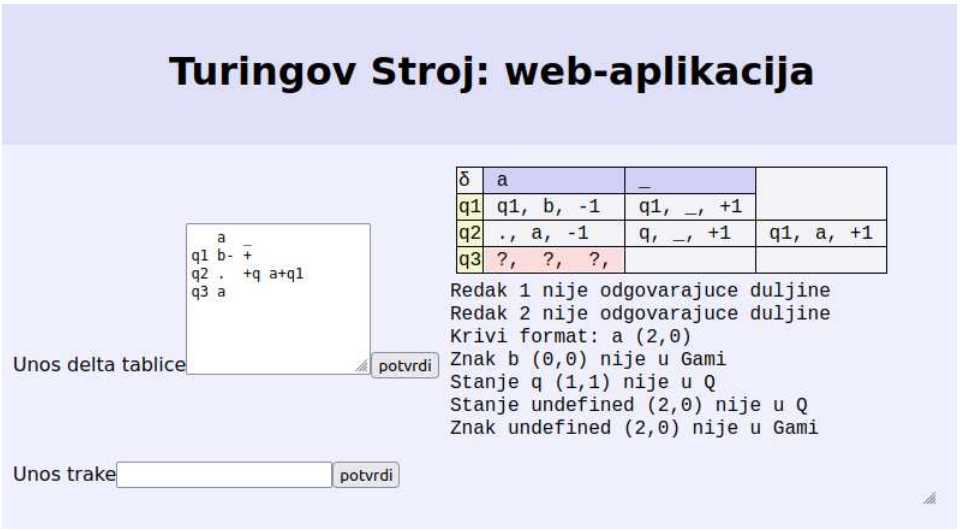
Slika 3.3: Pokrenuto izračunavanje s popisom konfiguracija

Klik na gumb „Sljedeći korak” dodat će jednu sljedeću konfiguraciju na popis, osim ako se na popisu već nalazi završna konfiguracija. Klik na gumb „Napravi 100 koraka” dodaje nove konfiguracije dok se ne pojavi završna ili dok se ne ispiše sto novih konfiguracija.

Ispod gumba za pokretanje i simulaciju izračunavanja dodan je kliznik (*slider*) koji omogućava „šetnju” po ispisanim konfiguracijama. Njegovim pomicanjem na neko mjesto, odnosno na neku vrijednost, na popisu se isticava konfiguracija s indeksom koji odgovara toj vrijednosti, a ista se konfiguracija ispisuje i iznad cijelog popisa.

Primjer poruka o greškama

Ako u okvir za unos tekstne tablice unesemo string koji se ne može parsirati kao objekt koji predstavlja Turingov stroj, pozvana funkcija `txtTS` vraća objekt koji ima član `error`. Rekli smo da je to polje stringova koje predstavlja popis grešaka, odnosno svega što nije u redu s unesenom „tekstnom tablicom”. Ako vraćeni objekt ima član koji se zove `error`, aplikacija ispod tablice funkcije prijelaza ili umjesto nje ispisati sve stringove iz tog polja jedan ispod drugog.



Slika 3.4: Krivo unesena tekstna tablica i popis grešaka

Svaka ispisana poruka sadrži i broj retka i stupca („adresu”) mjesta s greškom, brojeći od nule, počevši od prvog retka i stupca ne računajući gornje niti lijevo zaglavlje. Primjerice, ćelija u retku stanja `q1` i stupcu znaka `a` (drugi redak i drugi stupac tablice) označava se s `(0,0)`.

Bibliografija

[1] Marijn Haverbeke, *Eloquent JavaScript*, No Starch Press, 2018.

[2] Vedran Čačić, *Komputonomikon*, 2022.

[3] Vedran Čačić i Marko Horvat, *Interpretacija programa - predavanja*, 2022./2023.

Sažetak

Cilj ovog diplomskog rada je implementacija i opis web-aplikacije za emuliranje Turingova stroja.

U prvom poglavlju uvodimo osnovne pojmove i definicije vezane uz jezike i jezične funkcije. Zatim uvodimo Turingov stroj i izračunavanje, te ih objašnjavamo kroz nekoliko primjera.

Drugo poglavlje predstavlja kratki uvod u JavaScript, programski jezik u kojem je aplikacija implementirana. Ono sadrži opis osnovnih tipova podatka i dviju najčešće korištenih struktura podataka, te opis i primjer korištenja osnovnih funkcionalnosti jezika.

U trećem poglavlju govorimo o implementaciji same aplikacije. Opisujemo reprezentaciju pojmova iz prvog poglavlja objektima u JavaScript-u i zapis Turingova stroja kao tekstne tablice (stringa). Na kraju opisujemo korištenje aplikacije uz nekoliko *screenshotova*.

Summary

The aim of this paper is the implementation and description of a web-application that emulates the Turing machine.

In the first chapter we introduce basic terms and definitions related to formal languages and language functions, as well as Turing machine and its computation. We explain them with a few examples.

The second chapter is a short introduction to JavaScript, programming language the application is implemented in. It contains descriptions of simple data types and two most commonly used data structures, followed by description and demonstration of the basic functionalities of this language.

In the third chapter we talk about the implementation of our web-application. We describe the way terms from the first chapter are represented as JavaScript objects and the convention of writing down a Turing machine as a textual table (a string). Lastly, we demonstrate the application usage with a few screenshots.

Životopis

Rođen sam 29. rujna 1995. u gradu Varaždinu od oca Miroslava i majke Zdenke rođene Pavličević. Nakon osnovnoškolskog obrazovanja u II. Osnovnoj školi Varaždin, upisao sam prirodoslovno-matematičku gimnaziju na Drugoj gimnaziji Varaždin, a uz nju sam pohađao i srednju glazbenu školu na Glazbenoj školi u Varaždinu, smjer glazbena teorija.

Godine 2014. upisao sam preddiplomski sveučilišni studij Matematika, a po njegovu završetku 2019. godine sam upisao i diplomski sveučilišni studij Računarstva i matematike na Matematičkom odsjeku Prirodoslovno-matematičkog fakulteta Sveučilišta u Zagrebu. Na istoj instituciji sam također dva semestra radio kao demonstrator.