Razširitev označevalnega jezika Markdown z aritmetičnimi operacijami, matematičnimi in logičnimi funkcijami v tabelah

Aleks Marinič, Tomaž Kosar

Fakulteta za elektrotehniko, računalništvo in informatiko, Univerza v Mariboru, Koroška cesta 46, 2000 Maribor
E-pošta: aleks.marinic@student.um.si, tomaz.kosar@um.si

Extending the Markdown Markup Language with Arithmetic Operations, Mathematical and Logical Functions in Tables

Markdown is a simple markup language used to format text with plain-text editors that use additional tools to display formatted text or even generate HTML. It is popular among programmers because the syntax is easy to learn. However, simple and minimalistic Markdown notation can be problematic for those with more elaborate needs.

In this article, we present how to extend the existing Markdown language. We developed an extension for mathematical and logical functions in combination with general arithmetic operators in tables. The syntax is similar to that of table functions in spreadsheet programmes but with some modifications adapted for Markdown. Other developers with similar additional needs can learn how to use Markdown language and adjust it using our extension method.

1 Uvod

Markdown spada med označevalne jezike in se uporablja za prevajanje surovega besedila v HTML ali drug podoben jezik [3]. Ta majhen domensko specifičen jezik [4, 5, 6] se uporablja v forumih za oblikovanje objav in za oblikovanje besedilnih datotek (npr. *README.md* - vodnik, ki daje uporabniku kratek opis projekta [7] v GitHub, GitLab in podobnih repozitorijih). Vse pogosteje pa ga srečamo tudi v drugačnih okoliščinah, kot je izdelava statičnih spletnih strani (npr. *Jekyll* [1]) in v izdelavi predstavitev (npr. *Slidev* [2]).

S široko paleto ukazov pokriva večino zahtev navadnih uporabnikov in z raznimi dodatki tudi naprednih v specifičnih okoliščinah. Študenti se radi naslanjajo na Markdownova programja za izdelavo zapiskov, saj z njimi hitro vizualno uredijo besedilo, videz pa celotnem dokumentu konsistenten. Predavatelji pa zaradi deljenih predstavitev v obliki strežniške storitve vse pogosteje uporabljajo orodje Slidev.

Enostavna in minimalistična notacija Markdown, pa lahko za tiste z več potrebami predstavlja težavo. Uporaba aritmetičnih operatorjev v kombinaciji z matematičnimi in logičnimi funkcijami je velik plus, ko zapisujemo določene podatke v tabelah. Funkcije v tabelah so hitre in natančne rešitve, saj je prepisovanje ter ročno računanje podatkov dolgotrajno in lahko se pojavijo površne napake.

V tem članku predstavljamo, kako se lotiti razširitve jezika Markdown. Razvili smo razširitev za matematične in logične funkcije v kombinaciji s splošnimi aritmetičnimi operatorji v tabelah, kakršnih smo uporabniki navajeni iz programov za urejanje preglednic (npr. Microsoft Excel). Sintaksa je podobna tem, vendar z nekaterimi modifikacijami prilagojenimi za Markdown.

Članek ima sedem poglavij. V drugem poglavju predstavimo motivacijo naše razširitve. Sledi tretje poglavje, kjer je predstavljen prevajalnik jezika Markdown. V četrtem poglavju je predstavitev implementacije razširitve. Peto poglavje predstavi implementirane funkcije, ki jih lahko uporabljamo v tabelah Markdown. Šesto poglavje prikazuje uporabo naše razširitve na realnih problemih. Zadnje poglavje povzame vsebino članka.

2 Motivacija za razširitev jezika Markdown

Ob pisanju različnih dokumentov velikokrat uporabljamo tabele za preglednejši prikaz določenih podatkov. Večkrat nam je iz vseh teh podatkov najpomembnejša informacija, ki pa jo treba izračunati, na primer: na računu kupca nas najbolj zanima vsota zneska, na anketi se podatki predstavijo v obliki odstotnega deleža itd.

Označevalni jezik Markdown privzeto ne podpira nobenih aritmetičnih operacij, matematičnih ali logičnih funkcij v tabelah in vse računanje ob vsaki spremembi je treba opraviti ročno in vnesti rezultat. Ob tem se pojavlja velika verjetnost računske ali tipkarske napake človeka.

Te potrebne manjkajoče funkcionalnosti smo dodali v obliki razširitve prevajalnika s paleto ukazov, oblikovanih po konceptu:

```
==[arit. operacije in mat./log. funkcije]==
```

Označba == označuje meje aritmetičnih operacij, matematičnih in logičnih funkcij v tabeli. Te so v obliki operacij, kot jih poznamo iz popularnih programov za urejanje preglednic, na primer:

```
==if(sum(A2-A5) > 100, B3, "premalo")==
==avg(B3-B16)==
```

Primer uporabe funkcije samodejne vsote je na sliki 1, kjer smo s pomočjo uporabe matematične funkcije hitro izračunali samodejno vsoto. Rezultat kode je viden na sliki 2.

```
| Študenti      | Zbrana sredstva  |
| ------------- | ---------------- |
| Sara Novak    | 105 €            |
| Manca Horvat  | 96.50 €          |
| Nik Kovačič   | 90.90 €          |
| Tobias Hribar | 121 €            |
| **skupaj**    | ==sum(b1-b4)== € |
```

Slika 1: Markdown zapis primera o zbiranju sredstev

Študenti	Zbrana sredstva
Sara Novak	105 €
Manca Horvat	96.50 €
Nik Kovačič	90.90 €
Tobias Hribar	121 €
skupaj	413.4 €

Slika 2: Izhodni izpis primera o zbiranju sredstev

3 Prevajalnik jezika Markdown

Za implementacijo rešitve je bil potreben prevajalnik jezika Markdown. Ker obstoječih ukazov označevalnega jezika ne bomo spreminjali, smo se odločili vzeti že obstoječ prevajalnik in mu dodali razširitev. Potrebovali smo odprtokodni projekt, ki ga lahko prosto spreminjamo in distribuiramo. Med široko paleto raznih prevajalnikov Markdown smo se odločili za *Marked* [8], saj ima možnost prevajanja prek ukazne vrstice ali klasične JavaScript knjižnice. Prav tako je eden izmed bolj priljubljenih prevajalnikov za Markdown, ki ga uporabniki radi nadgrajujejo, saj ima veliko vejitev. Je tudi živ projekt, saj ga redno nadgrajuje zdaj že okoli 150 razvijalcev.

Projekt je razvit v zalednem izvajalskem okolju Node.js, ki ga je začel razvijati Christopher Jeffrey [9] leta 2011 in ga najdemo pod licenco MIT. Koda projekta se prevaja, surova pa se hrani v mapi *./src*. Ob prevajanju se zgradi v zgoščeno datoteko JavaScript[1]. Posledično je datoteka manjše velikosti in se ob zahtevi hitreje prenese na računalnik odjemalca.

3.1 Struktura kode prevajalnika

Celotna koda projekta *Marked* se deli na razrede, metode in funkcije v različnih datotekah – vse znotraj mape *./src*. Začetna točka je datoteka *marked.js*. Ta vsebuje funkcijo *marked(...)*, ki jo kličemo v dokumentu HTML znotraj značke *SCRIPT* in ji kot parameter podamo besedilo Markdown za prevajanje.

Besedilo se nato pregleda s pomočjo regularnih izrazov (ki so v datoteki *rules.js*) in različnimi metodami znotraj razreda *Lexer* (v datoteki *Lexer.js*). Posamezne metode se uporabljajo za razrezanje besedila glede na ukaze in klic ustreznih metod znotraj razreda *Tokenizer* (v datoteki *Tokenizer.js*) za izdelavo tako imenovanih žetonov.

[1] Vsebuje manjšo dolžino kode, saj so iz nje izbrisani nepotrebni prazni znaki in zamenjana so imena nekaterih spremenljivk.

Žetoni so objekti, ki hranijo posamezne lastnosti ukazov (med drugim tip, surov in obdelan ukaz). Te žetone sprejmejo ustrezne metode znotraj razreda *Parser* (v datoteki *Parser.js*) in glede na njihovo vrsto pokliče ustrezno metodo v razredu *Renderer* (iz datoteke *Renderer.js*). Posamezna metoda vstavi parametre žetona v vnaprej definirano definirano HTML in jo vrne v obliki niza znakov (angl. *string*) nazaj.

Pri tem se uporabljajo še druge metode znotraj razredov:

- *TextRenderer*, ki je v datoteki *TextRenderer.js*. Njegove metode vračajo vsebino znotraj ukazov in
- *Slugger*, ki je v datoteki *Slugger.js*. Omogoča generacijo enolično določenih identifikatorjev (vrednosti *ID* atributov znotraj naslovnih značk).

Med brezrazredne funkcije štejemo še *getDefaults* in *setDefaults* iz datoteke *defaults.js* (ki skrbita za branje in nastavljanje vnaprej definiranih vrednosti) ter pomožne funkcije iz *helpers.js*, kjer so definirane funkcije za ostale namene.

3.2 Vstopna točka za razširitev prevajalnika

Nove implementirane matematične in logične funkcije kot tudi osnovne aritmetične operacije so namenjene izvajanju samo v tabelah. Posledično se morajo potencialni ukazi iskati samo znotraj celic tabele in ne potrebujemo globalnega iskanja podatkov.

Uporabili smo rekurzivno funkcijo *CellCalculator(...)*. Kot parameter prejme tabelo in posamezno celico (ki obravnava) in (neobvezno oziroma vnaprej definirano) iteracijo izvajanja funkcije. Iteracija izvajanja se uporablja za preprečevanje klica rekurzije v neskončnost ob uporabi nepravilne reference (na celico) ob klicu posameznih funkcij oziroma ob pojavljanju semantičnih napak.

Ta funkcija se prvič kliče v datoteki *Tokenizer.js* med oblikovanjem prejetega niza znakov v objekt. Tako se funkcija izračuna pred gradnjo tabele v niz znakov za izpis.

Uporabljeni koncept ločevanje razredov in funkcij po kriterijih v različne datoteke je tukaj smiselno uporabljen zaradi globalnih iskanj posameznih ukazov (na primer: naslovi, odebeljeno besedilo, povezave, ...). Mi uporabljamo ukaze za funkcije le znotraj celic tabel in posledično je bolj primerno lokalno iskanje ukazov. Ker imamo drugačne (unikatne) kriterije iskanja, je primerno, da ločimo tudi kodo v svojo datoteko. S tem drugim razvijalcem nakažemo drugačnost in je koda kot celota tudi bolj pregledna. Ločena datoteka za kodo je smiselno poimenovana *TableFunction.js* in ima glavno funkcijo – ta je *CellCalculator(...)*, pomožne funkcije in globalne konstante.

4 Implementacija razširitve

Programska implementacija se deli na dve bistveni funkciji za obdelavo, nekaj pomožnih in treh konstantnih objektov. Pomožne funkcije se uporabljajo za:

- pretvorbo klica reference celice (na primer: `BA34`) v objekt s številko stolpca in vrstice referencirane celice,
- napredno izluščenje vgnezdene funkcije, definirane z večkratnim gnezdenjem oklepajev,
- prepisovanje obstoječe funkcije *parseFloat* zaradi odprave napake zaokroževanja ob shranjevanju decimalnih vrednosti v 64-bitna števila s plavajočo vejico ipd.

Posamezne aritmetične operacije in tudi matematične in logične funkcije so predstavljene s tremi konstantnimi objekti. Te tvorijo regularni izrazi za detekcijo funkcije in izluščevanje njenih parametrov ter funkcija za obdelavo.

Dodana je možnost za onemogočenje razširitve. To se definira v klicu metode *setOptions(...)*, kjer nastavimo vrednost *tableFunctions* na *false*:

```
marked.setOptions({tableFunctions: false});
```

4.1 Funkcija CellCalculator (...)

Predstavlja glavno funkcijo, ki izlušči celotni matematični in/ali logični izraz iz vsebine celice ter ga posreduje naprej funkciji *solveFunction(...)*, ki vrne odgovor na mesto, kjer je bil prej iskani izraz. Iskanje funkcije poteka s pomočjo regularnega izraza med robnimi oznakami ==. Funkcija s pomočjo rekurzivnega števca iteracij ob semantični napaki v funkciji preprečuje klice v neskončnost. Privzeto število rekurzivnih klicev funkcije brez izpisa napak je 25 in ko se to število preseže, dobimo napako. Če hočemo uporabljati več ali manj rekurzivnih klicev, lahko to dosežemo s klicem metode *setOptions(...)*, kjer nastavimo vrednost *tableFunctionRecursionNumber* na želeno število:

```
marked.setOptions({
  tableFunctionRecursionNumber: 35
});
```

4.2 Funkcija solveFunction (...)

Je rekurzivna funkcija, ki prejme izluščeno funkcijo v obliki niza in celotno tabelo ter (neobvezno – vnaprej definirano) iteracijo.

Posamezni ukazi se v matematičnem in/ali logičnem izrazu prepoznajo s pomočjo regularnih izrazov in obdelajo po prednosti matematičnih pravil. Če je v matematičnem in/ali logičnem izrazu referenca na celico (na primer: `A1`), se ta prav tako obravnava kot ločen izraz in se obdela samostojno v novem klicu iste rekurzivne funkcije. Enako se zgodi v primeru oklepajev, saj je tudi tam večja prednost računanja. Ko se izraz izračuna in imamo v odgovoru samo eno število (ali opomin napake), se vrne v očetovsko funkcijo. Zadnjo očetovsko funkcijo predstavlja glavna funkcija, poklicana v datoteki *Tokenizer.js*.

Pri uporabi matematičnih ali logičnih funkcij lahko pride do napak, te se delijo na semantične in sintaktične. Semantične napake nastanejo ob uporabi referenc na celice brez števil ali uporabi matematično nepravilnih računskih operacij (na primer: deljenje z 0, kvadratni koren negativnih števil). Ob tem se izpiše odgovor *NaN*. Sintaktične napake pa nastanejo ob napačni uporabi matematičnih operatorjev ali neobstoječih/nepravilno podanih matematičnih/logičnih funkcij. Ob tem se izpiše odgovor *[ERROR]*.

5 Dodane funkcije za delo s tabelami

Dodani so bili osnovni aritmetični operatorji (seštevanje, odštevanje, množenje, deljenje in potenciranje) in najbolj priljubljene osnovne matematične in logične funkcije.

5.1 Matematične funkcije

Matematične funkcije se izvajajo nad podatki v obsegu. Le-ta je določen z znakom **–** (pomišljaj). Vse matematične funkcije pokličemo z krajšavo njihovega imena in zatem dodamo oklepaje, v katerih je definiran obseg.

V nadaljevanju podajamo nekaj dodanih matematičnih funkcij v naši razširitvi jezika Markdown:

- *Samodejna vsota* se uporablja za seštevanje celic v definiranem obsegu tabele. Primer klica funkcije za računanje samodejne vsote: `sum(A1-C5)`.
- *Minimalna vrednost* se uporablja za iskanje najmanjše vrednosti v definiranem obsegu tabele. Primer klica funkcije za iskanje minimalne vrednosti: `min(A1-C5)`.
- *Maksimalna vrednost* se uporablja za iskanje največje vrednosti v definiranem obsegu tabele. Primer klica funkcije za iskanje maksimalne vrednosti: `max(A1-C5)`.
- *Povprečna vrednost* (ali aritmetična sredina) se uporablja za računanje povprečne vrednosti v definiranem obsegu tabele. Primer klica funkcije za računanje povprečne vrednosti: `avg(A1-C5)`.

Kot rezultat vse matematične funkcije vrnejo število ali obvestilo o napaki.

5.2 Funkcija za logično preverjanje

Pogojni stavek se uporablja za pogojni izpis ali računanje v določeni celici. Funkcijo pokličemo z imenom, zatem dodamo oklepaje, v katerih so 3 deli, ločeni z dvema vejicama. Prvo je pogoj, nato prvi stavek in drugi stavek. Prvi stavek se uporabi, ko je pogoj resničen, in drugi, ko pogoj ni resničen.

Primer klica take funkcije je:

```
if(A1>0, "Poz. št.", "Neg. št.")
```

Kot odgovor vrne zapis *Poz. št.* (kar označuje pozitivno število večje od 0) ali *Neg. št.* (kar označuje 0 ali negativno število).

6 Uporaba razširitev označevalnega jezika Markdown

Podajmo nekaj primerov uporabe naše razširitve jezika Markdown.

6.1 Primer računanja ničel funkcije

Veliko ljudi uporablja razne programske rešitve za izdelovanje zapiskov. Ko pišemo matematične zapiske, lahko uporabljamo podatke v tabelah za hitro izračunavanje vrednosti. Na sliki 3 je primer zapiska, kjer hočemo izračunati ničli polinoma, podanega (1) prek enačb (2) in (3). Temu primeren izhod pa je na sliki 4.

$$f(x) = 4x^2 + 4x - 4 \tag{1}$$

$$x_{1,2} = \frac{-b \pm \sqrt{D}}{2a} \tag{2}$$

$$D = b^2 - 4ac \tag{3}$$

```
| Oznaka | Vrednost                  |
| :----: | :-----------------------: |
| a [x²] | 4                         |
| b [x]  | 4                         |
| c      | -4                        |
| Disk.* | ==(b2^2-4*b1*b3)==        |
| x1     | ==(0-b2-b4^(1/2))/(2*b1)== |
| x2     | ==(0-b2+b4^(1/2))/(2*b1)== |
```

Slika 3: Zapis primera Markdown o iskanju ničel polinoma

Oznaka	Vrednost
a [x²]	4
b [x]	4
c	-4
Disk.*	80
x1	-1.61803
x2	0.61803

Slika 4: Izhodni izpis primera o iskanju ničel polinoma

6.2 Primer računanja deležev

Kadar pišemo poročila anket ali eksperimentov, moramo te tudi primerno predstaviti. Na sliki 5 je primer ankete o starosti obiskovalcev neke knjižnice. Matematične funkcije lahko uporabimo neposredno za računanje deležev. Ob spremembah podatkov se deleži samodejno preračunajo in imamo takoj ustvarjeno novo poročilo. Primer kode računanja deležev iz podatkov v tabeli je prikazan na sliki 5, njen izhodni rezultat pa na sliki 6.

```
| starost | število | delež           |
| ------- | ------- | --------------- |
| 0-10    | 20      | ==b1/b7*100== % |
| 11-20   | 30      | ==b2/b7*100== % |
| 21-30   | 75      | ==b3/b7*100== % |
| 31-40   | 35      | ==b4/b7*100== % |
| 41-50   | 45      | ==b5/b7*100== % |
| nad 50  | 45      | ==b6/b7*100== % |
| vsi     | ==sum(b1-b6)== | 100 %    |
```

Slika 5: Zapis Markdown primera o računanju deležev

7 Zaključek

Markdown ni samo označevalni jezik, pač pa je celovit projekt, ki raste in se širi v nova okolja. Med uporabniki je priljubljen, saj je sintaksa lahka, kratka in preprosta za

starost	število	delež
0-10	20	8 %
11-20	30	12 %
21-30	75	30 %
31-40	35	14 %
41-50	45	18 %
nad 50	45	18 %
vsi	250	100 %

Slika 6: Izhodni zapis primera o računanju deležev

učenje. Uporabniki (ki dobro poznajo sintakso) tudi veliko hitreje napišejo dokument, saj jim pri vizualnem urejanju dokumenta ni treba delati premikov rok s tipkovnice na miško. Hkrati je tudi vizualni izgled dokumenta boljši, saj je vnaprej določen in v vseh dokumentih konsistenten.

Naša razširitev je kot projekt javna na GitHub repozitoriju [10] in je na voljo drugim za uporabo ali nadgradnjo. S tem je zasnovan koncept za uporabo aritmetičnih operacij s kombinacijo matematičnih in logičnih funkcij v tabelah za priljubljeni označevalni jezik Markdown. S tem člankom smo želeli pokazati, kako lahko drugi razvijalci s potrebami po razširitvi uporabijo jezik Markdown in ga prilagodijo svojim potrebam.

Literatura

[1] Jekyll, https://jekyllrb.com

[2] Slidev, https://sli.dev

[3] Xie, Y., Dervieux, C. Riederer, E. R markdown cookbook. (Chapman, 2020).

[4] Mernik, M., Heering, J. Sloane, A. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*. 37, 316–344 (2005).

[5] Kosar, T., Bohra, S. Mernik, M. Domain–Specific Languages: A Systematic Mapping Study. *Information And Software Technology*. 71, 77–91 (2016).

[6] Kosar, T., Gaberc, S., Carver, J. Mernik, M. Program comprehension of domain-specific and general-purpose languages: replication of a family of experiments using integrated development environments. *Empirical Software Engineering*. 23, 2734–2763 (2018).

[7] L. Richards, J. M. Morse: README FIRST for a User's Guide to Qualitative Methods, 2013.

[8] Marked, https://github.com/markedjs/marked

[9] C. Jeffrey, https://github.com/chjj/

[10] Marked with table functions, https://github.com/Linux-Alex/marked-with-table-functions