Matematički fakultet

Master rad

Pretprocesiranjem do jednostavnijeg pisanja lepog izvornog kôda

Autor:
Nenad Vasić

Mentor:
Prof. dr Filip Marić

Članovi komisije:
Prof. dr Filip Marić
Matematički fakultet, Univerzitet u Beogradu
dr Milena Vujošević Janičić
Matematički fakultet, Univerzitet u Beogradu
dr Milan Banković
Matematički fakultet, Univerzitet u Beogradu

Septembar, 2018.

Sadržaj

Glava 1

Uvod

Kodiranje je jedna od najvažnijih veština u računarstvu. Jednom napisan programski kôd može biti korišćen jako dugo. Na primer, delovi standardne biblioteke jezika C koji se i dalje koriste napisani u periodu od 1969. do 1973. godine. Svi moderni operativni sistemi su većim delom napisani u C-u i koriste C standardnu biblioteku pa je verovatno da se delovi ovog kôda izvršavaju svaki dan na svakom računaru.

Dobro napisan programski kôd mora, osim korektnosti, da zadovoljava i mnoge druge kriterijume. Čitljivost, mogućnost testiranja, mogućnost ažuriranja, mogućnost nadogradnje, pravilno formatiranje i jednostavna integracija sa ostatkom sistema samo su neke od osobina o kojima programer treba da brine. Postoje mnogobrojni alati koji doprinose efikasnosti u radu programera. U ovom radu biće opisano rešenje za rad sa fragmentima kôda prilikom kodiranja.

Alat *Homotopy* implementiran je kao pomenuto rešenje. Homotopy povećava efikasnost rada omogućavajući konstrukciju često korišćenih sintaksičkih celina na koncizan način. Prilikom kodiranja, programer može da napiše fragment kôda u jeziku Homotopy i nakon toga ga prevede u izvorni kôd u jeziku koji koristi. Na ovaj način se povećava efikasnost kodiranja jer se deo brige delegira alatu Homotopy (besplatno dostupnom na `https://github.com/Ahhhhmed/homotopy`). Alat Homotopy je dizaniran sa fokusom na opštost i proširivost. Njegova standardna biblioteka sadrži fragmente za nekoliko popularnih programskih jezika, ali je dodavanjem novih fragmenata moguće podržati i druge jezike. Biblioteka fragmentata sadrži i često korišćene obrasce što olakšava njihovo korišćenje i čini kôd čitljivijim.

1.1 Primer „Zdravo svete“

Kao motivacioni primer dajemo generisanje programa „zdravo svete“ u jeziku C pomoću alata Homotopy. Primer „zdravo svete“ često je najjednostavniji primer koji se daje u literaturi o nekom jeziku. Ipak, ovde to nije slučaj jer se za izgradnju ovog primera koriste i naprednije osobine jezika Homotopy. Čitaocu

se zbog toga preporučuje da se vrati još jednom na ovaj primer nakon glave 3.

```
stdinc$stdio.h& &[[main]]>printf("Zdravo, svete\!");& &return 0;
```

Program se sastoji od fragmenata, koji su međusobno razdvojeni karakterima `&`, `>` ili `<`. Na najvišem nivou, ovaj fragment se sastoji iz 3 potfragmenta (razdvojena karakterima `&`).

1. `stdinc$stdio.h`,
2. prazan karakter za prazan red,
3. `[[main]]>printf("Zdravo, svete\!");& &return 0;`.

Ova 3 fragmenta se implicitno nalaze u fragmentu `block` koji razdvaja podfragmente u posebne redove. Treći fragment je složen i sadrži prvo skraćenicu `[[main]]`, a zatim nakon karaktera `>` fragment koji se sastoji od tri potfragmenta (razdvojena karakterima `&`).

1. `printf("Zdravo, svete\!");`
2. prazan karakter za prazan red,
3. `return 0;`.

Karakterom `>` uspostavljen je hijerarhijski odnos između fragmenata. Prevođenje fragmenata se vrši na osnovu *definicija fragmenata* (one donekle podsećaju na pravila kontekstno-slobodnih gramatika). Definicije fragmenata su gradivne jedinice biblioteke fragmenata. Jedna definicija sadrži ime za referisanje fragmenta, jezike u kojima je fragment relevantan i vrednost fragmenta. Pomoću vrednosti fragmenata se konstruiše izlazni kôd. Vrednosti fragmenata često sadrže meta-karaktere (kao što su `$`, `#` ili `@`). Ovi karakteri označavaju određene parametre u definicijama koji se zamenjuju vrednostima navedenim u programu (u programu se oni navode jednom, a u definicijama tri puta). Više detalja o definicijama fragmenata i parametrima dato je u glavi 3. Definicije fragmenata koji se koriste u tekućem primeru mogu se videti na slici 1.1.

Fragment `stdinc` definisan je kao `#include <$$$>` i ovde dolazi do jednostavne zamene `$$$` sa `stdio.h` kako bi se dobilo `#include <stdio.h>`.

Prazni redovi (dva uzastopna karaktera za prelazak u novi red), dobijaju se tako što se u postprocesiranju red koji sadrži samo uneti prazan karakter očisti od praznih karaktera sa početka i kraja reda.

Dalje je prikazano kako se treći (složeni) fragment prevodi (na osnovu navedenih definicija fragmenata).

1. `[[main]]` se prevede u `func#int@main#int$argc#char*$argv[]`.
2. `func` se prevede u `### @@@({{params}}){\n{{inside_block}}\n}`.
3. `###` se prevede u `int`.

```
[
    {
        "name": "stdinc",
        "language": "C++",
        "snippet": "#include <$$$>"
    },
    {
        "name": "main",
        "language": "C++",
        "snippet": "func#int@main#int$argc#char*$argv[]"
    },
    {
        "name": "func",
        "language": "C++",
        "snippet": "### @@@({{params}}){\n{{inside_wblock}}\n}"
    },
    {
        "name": "params",
        "language": "C++",
        "snippet": "### $$${{opt_params}}"
    },
    {
        "name": "opt_params",
        "language": "C++",
        "snippet": ", ### $$${{opt_params}}"
    },
    {
        "name": "inside_block",
        "language": "C++",
        "snippet": "\t>>>{{opt_inside_block}}"
    },
    {
        "name": "opt_inside_wblock",
        "language": "C++",
        "snippet": "\n\t>>>{{opt_inside_block}}"
    }
]
```

Slika 1.1: Definicije fragmenata koje se koriste u primeru „zdravo svete”

4. `@@@` se prevede u `main`.

   Rezultat nakon toga je

   `int main({{params}}){\n{{inside_block}}\n}}`.

5. `###` ne postoji u trenutnom rezultatu pa se `{{params}}` razvija jer ono sadrži `###`.

   Sada je rezultat

   `int main(### $$${{opt_params}}){\n{{inside_block}}\n}}`.

6. `###` se prevede u `int`.

7. `$$$` se prevede u `argc`.

8. Slično kao u koraku 5, `{{opt_params}}` se razvija u
   `, ### $$${{opt_params}}`.

9. `###` se prevede u `char*`.

10. `$$$` se prevede u `argv[]`.

11. Slično kao u koraku 5 i 8, `{{inside_block}}` se razvija u

    `\t>>>{{opt_inside_block}}`.

12. Rekurzivno se prevodi fragment `printf("Zdravo, svete\!");`.
    U ovom slučaju, ovo prevođenje je trivijalno i rezultat je
    `printf("Zdravo, svete!");`. Karakter `!` je izbegnut (engl. escaped) dok je sve drugo ostalo nepromenjeno.

13. `>>>` se prevede u `printf("Zdravo, svete!");`.

14. Slično kao u prethodnim koracima `{{opt_inside_block}}` se razvija u
    `\n\t>>>{{opt_inside_block}}`.

15. Razmak se trivijalno prevodi.

16. `>>>` se prevede u razmak.

17. Slično kao u prethodnim koracima `{{opt_inside_block}}` se razvija u
    `\n\t>>>{{opt_inside_block}}`.

18. `return 0;` se trivijalno prevodi u `return 0;`.

19. `>>>` se prevede u `return 0;`.

20. Rezultat se čisti od nerazvijenih podfragmenata `{{opt_params}}` i `{{opt_inside_block}}`.

Kao razultat dobijamo sledeći izvorni kôd.

```c
#include <stdio.h>

int main(int argc, char* argv[]){
    printf("Zdravo, svete!");

    return 0;
}
```

U radu je dato još (manjih) primera koji ilistruju rad alata Homotopy. Većina primera je data u sledećem formatu. Prvo je dat program u jeziku Homotopy a potom rezlutat obrade tog programa (uglavnom rezultujući tekst u nekom programskom jeziku).

Pre priče o samom alatu, u glavi 2, dat je opis nekih od postojećih sistema koji doprinose poboljšanju kvaliteta kôda. Nakon toga, u glavi 3, dat je opis jezika Homotopy kao jezika za rad sa fragmentima kôda. Opis implemtiranog prevodioca za jezik Homotopy dat je u glavi 4. Predviđen način korišćenja prevodioca je od strane dodataka za editore (programe za uređivanje teksta, osnosno programskog kôda). Opis funkcionalnosti koje prevodilac Homotopy pruža za korišćenje od strane editora kao i konkretan dodatak za edoitor `Atom` dati su u glavi 5. U glavi 6 dat je pregled rada, kao i mogućnosti za dalji razvoj predloženog rešenja. Finalno, dodatak A sadrži listu primera rezultata prevodioca Homotopy.

Glava 2

Postojeći sistemi

U ovoj glavi biće opisani neki od postojećih alata koji se koriste pri programiranju kako bi ceo proces bio brži, sigurniji i efikasniji. Oni su najčešće ugrađeni u editore i integrisana razvojna okruženja. Za razliku od prevodioca i izvršnog okruženja, ovi alati nisu neophodni da bi se kreirali izvršivi programi, ali se obično koriste u svakom ozbiljnijem projektu. Ovo je dovelo i do taga da se prevodioci dizajniraju ne samo za prevođenje već i za integraciju sa drugim alatima koji se koriste pri programiranju. Na primer, `Clang/LLVM`[1] prevodilac je nastao kao alternativa prevodiocu `GCC`[2] (i dalje se održava kompatibilnost sa `GCC` prevodiocem) sa fokusom na proširivost i jednostavnu integraciju sa različitim alatima.

2.1 Sistemi za automatsko kompletiranje kôda

Sistemi za automatsko kompletiranje kôda sastavni su deo skoro svakog softverskog projekta. Prilikom kodiranja programeru pružaju listu reči (najčešće ranije definisanih identifikatora) iz koje programer može izabrati jednu i napisati je automatski. Na primer, prilikom pozivanja metoda na nekom objektu sistem za automatsko kompletiranje pruža listu svih metoda koje taj objekat može da izvrši. Na ovaj način se može ubrzati unos poznatih reči kao i omogućiti unos nepoznatih (ime metoda se ne mora znati napamet).

Sistemi za automatsko kompletiranje koriste se i pri unosu drugih vrsta tekstova kao što su kratke fraze u alatima za pretragu. Za unos prirodnih jezika (u formi elektronske pošte[3], na primer) takođe postoje sistemi za automatsko kompletiranje, ali ih strogo definisane gramatike i jasna semantička pravila čine posebno korisnim u radu sa programskim jezicima.

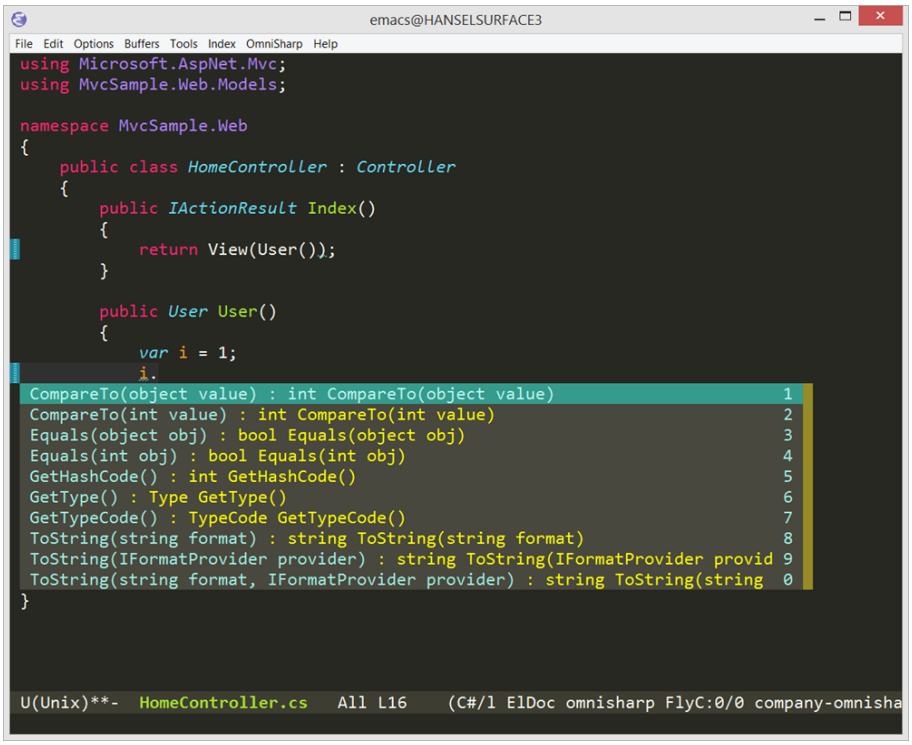
[1] `https://clang.llvm.org/`

[2] `https://gcc.gnu.org/`

[3] Kompanija Google razvija sistem za sugestije pri pisanju elektronske pošte (`https://www.blog.google/products/gmail/subject-write-emails-faster-smart-compose-gmail/`).

2.1.1 Alat OmniSharp

Alat *OmniSharp* (`http://www.omnisharp.net/`) omogućava udobniji rad na `.NET` projektima. Ovde ćemo se fokusirati na mogućnosti kompletiranja kôda koje ovaj alat omogućava. Alat `OmniSharp` omogućava kompletiranje kôda u svim jezicima koji su deo `.NET` okruženja (C#, VB, F#, ...). Kako bi se ovo omogućilo, `OmniSharp` se oslanja na prevodilac `Roslyn`[4] koji pruža informacije o sintaksi i semantičkim pravilima jezika. Prilikom korišćenja automatskog kompletiranja kôda prikazuje se lista (kao na slici 2.1) koja pruža odabir reči koje se mogu upotrebiti na datom mestu u skladu sa sintaksom i semantikom jezika u kome se piše. Alat `OmniSharp` se može koristiti u više editora (Atom, Brackets, Emacs, Sublime Text, Vim, Visual Studio Code, ...). Ovo je omogućeno odvajanjem jezgra alata u zaseban projekat, `omnisharp-roslyn`[5], koji se koristi od strane dodataka za pomenute editore koji pružaju funkcionalnosti alata krajnim korisnicima.

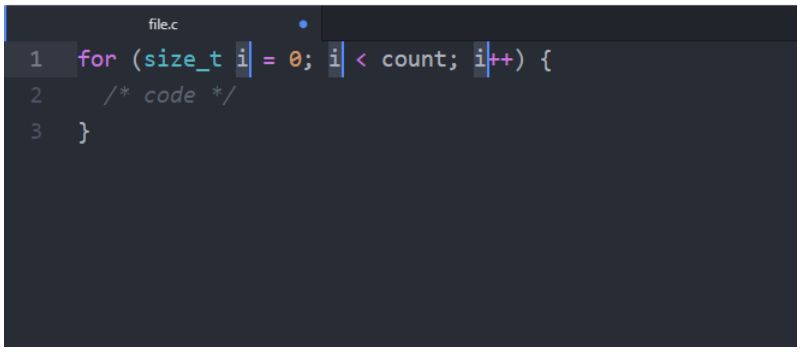


Slika 2.1: Automatsko kompletiranje kôda u Emacs-u pomoću alata OmniSharp

[4] `https://github.com/dotnet/roslyn`
[5] `https://github.com/OmniSharp/omnisharp-roslyn`

2.2 Skraćenice

Skoro svaki editor programskog kôda ima podršku za skraćenice (eng. snippets). One opisuju česte konstrukcije i omogućavaju njihov unos na brži način. Standardan način za njihov rad je sledeći: skraćenica je definisana kao tekst sa označenim mestima. Prilikom razvijanja skraćenice, skraćeni tekst se zameni tekstom definisanim za tu skraćenicu. Kursor se može pomerati (najčešće pritiskom tastera `tab`) kružno po označenim mestima. Dodatno se omogućava navođenje podrazumevane vrednosti koja se inicijalno stavlja na označenim mestima. Na ovaj način se omogućava brzo kreiranje čestih konstrukcija kao što su, na primer, petlja for. Na slici 2.2 može se videti for petlja koja je kreirana u editoru `Atom`. Odmah nakon razvijanja skraćenice moguće je menjati brojačku promenljivu (prvobitna vrednost promenljive je `i`). Pritisnom tastera `tab` označava se promenljiva koja označava gornju granicu brojanja (podrazumevano `count`) kako bi se eventualno preimenovala i narednim pritiskom tastera `tab` prelazi se na komandu za ažuriranje brojačke promenljive (podrazumevano `++`). Konačno, još jednim pritiskom `tab` tastera prelazi se u unutrašnjost for petlje (gde se na početku nalazi komentar `/* code */`) kako bi se implementirala petlja.



Slika 2.2: Petlja for kreirana skraćenicom u editoru Atom

2.2.1 Alat Emmet

Alat *Emmet* (`https://emmet.io/`) dizajniran je da olakša pisanje HTML kôda i raspoloživ je u obliku dodatka za mnoge popularne editore kôda. Emmet omogućava unos skraćenica koje se zatim prevode u HTML kôd. Na primer, kreiranje liste se može veoma efikasno dobiti na što je prikazano u primeru 2.2.1.

Pored uobičajnih skraćenica, Emmet omogućava i njihovo kombinovanje. Konkatenacija, ponavljanje više puta, ugnježdavanje i grupisanje su neki od načina na kojima se kreiraju složene strukture.

Primer 2.2.1

```
ul#nav>li.item$*4>a{Item $}
```

```
<ul id="nav">
  <li class="item1"><a href="">Item 1</a></li>
  <li class="item2"><a href="">Item 2</a></li>
  <li class="item3"><a href="">Item 3</a></li>
  <li class="item4"><a href="">Item 4</a></li>
</ul>
```

Primer 2.3.1

```
        | Fruit  | Color  |
        | -----  | -----  |
        | Apple  | Red    |
        | Banana | Yellow |
        | Kiwi   | Green  |
```

2.3 Automatsko formatiranje

Alati za automatsko formatiranje menjaju beline koje se ignorišu u većini jezika[6] u cilju bolje čitljivosti kôda. Većina programa za pisanje kôda podržava neki vid automatskog formatiranja što uključuje automatsko nazubljivanje i sređivanje belina između operatora. Na ovaj način se omogućava jednostavno uvoženje kôda koji nije u željenom formatu a koji je semantički ispravan.

2.3.1 Alat Tabular

Tabular (`https://github.com/godlygeek/tabular`) je alat za automatsko formatiranje koji omogućava centriranje kôda (ili druge vreste teksta) u odnosu na neki znak, na primer, znak jednakosti. Takođe omogućava jednostavno formatiranje teksta koji liči na tabele korišćenjem horizontalnih i uspravnih crta.

Primer 2.3.1 konstruisan je pomocu alata Tabular. Alatu se zada regularni izraz, u ovom slučaju samo jedan znak |, koji se koristi za centriranje. U interaktivnom režimu proširivanjem jedne ćelije proširuje se cela kolona kojoj ta ćelija pripada, što znatno olakšava rad sa tabelama.

[6] Kao izuzetak pominjemo programski jezik Whitespace koji je dizajniran na azbuci koja sadrži samo beline.

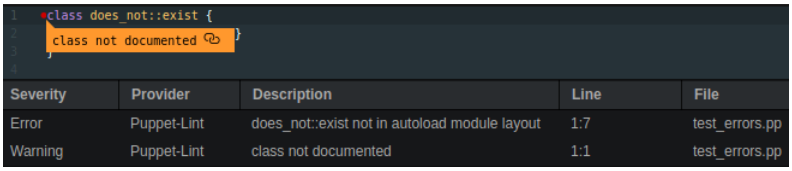
2.4 Sistemi za automatsko refaktorisanje

Kao i alati za automatsko formatiranje, sistemi za automatsko refaktorisanje [1] ne menjaju semantiku kôda, ali, za razliku od alata za automatsko formatiranje, menjaju sintaksu kôda. Koriste se kako bi se dizajn projekta poboljšao uvođenjem poznatih šablona ili za povećanje modularnosti. Na primer, kreiranje *getter* i *setter* metoda je česta praksa koja obezbeđuje enkapsulaciju podataka. Takođe se automatizuju i složeniji procesi kao što je izdvajanje metoda, zamena grananja polimorfizmom, promena imena, premeštanje metoda i mnogi drugi. Ovakvi alati često su deo integrisanih okruženja (`Visual Studio`, `Eclipse`, ...), ali postoje i samostalni alati za refaktorisanje kao što su `ReSharper`[7], `CodeRush`[8], `JS Refactor`[9] i drugi.

2.5 Sistemi za kontrolu kvaliteta kôda - linteri

Isti program je moguće napisati na mnogo načina ali se neki smatraju stilski boljim od drugih. Linteri analiziraju izvorni kôd i nastoje da ukažu na potencijalne greške (najčešće stilske, ali i semantičke), i pružaju predloge za poboljšavanje kvaliteta. Na taj način ohrabruju programere da pišu kôd koji se jednostavnije čita i održava. Naziv linteri potiče od programa *Lint* [2] koji je implementiran 1970-ih za programski jezik C. Od tada je razvijen veliki broj lintera za razne programske jezike.

Na slici 2.3 može se videti kako se uz pomoć lintera može osigurati da se ne zaboravi pisanje propratne dokumentacije. Linter prijavljuje da je izostavljena propratna dokumentacija.



Slika 2.3: Primer korišćenja lintera

2.6 Integrisana okruženja

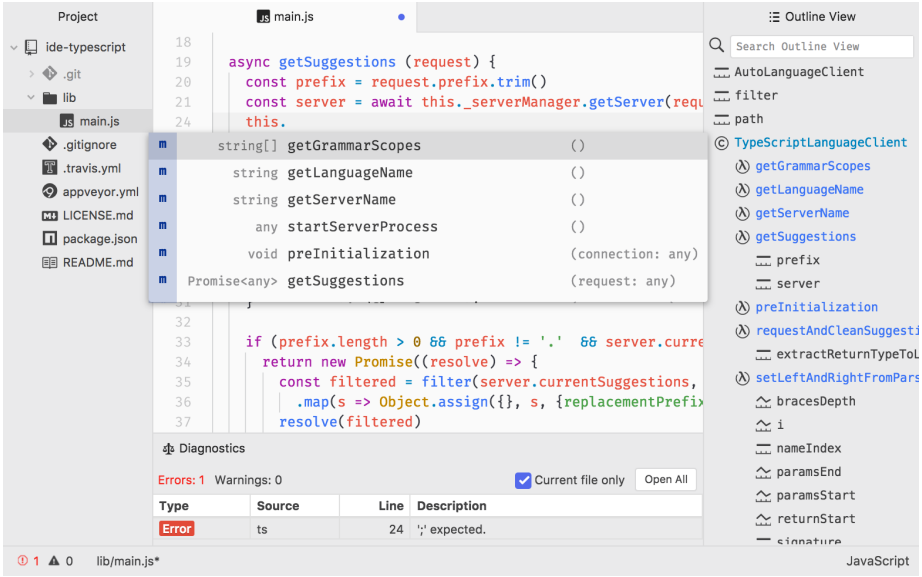
Kako bi se raznovrsni alati pri razvijanju softvera lakše koristili, razvijeni su mnogobrojni programi koji inkorporiraju nekoliko takvih alata. Veliki broj ovakvih programa nastao je zahvaljujući otvorenoj prirodi programa za obradu teksta kao što je *Emacs*[10] koji dopuštaju dodavanje novih funkcionalnosti u vidu

[7] `https://www.jetbrains.com/resharper/`
[8] `https://www.devexpress.com/products/coderush/`
[9] `https://github.com/cmstead/js-refactor`
[10] `https://www.gnu.org/software/emacs/`

dodataka. Programi kao što su `Emacs` i `Vim`[11] i dalje su vrlo popularni iako su izašli 70-ih godina. Tokom 90-ih godina nastali su i komenrcijalni programi u vidu visoko integrisanih okruženja kao što su `Visual Studio`[12], `Xcode`[13], `Intellij`[14] i drugi, koji su postali popularni jer omogućavaju celokupan razvoj u jednom programu. Ovakvi programi su poslednjih godina malo izgubili na popularnosti zbog novijih editora kao što su `Sublime Text`[15], `Atom`[16], `Notepad++`[17] i `Visual Studio Code`[18] koji su jesnostavniji i više se oslanjaju na dodatke nego na osnovne funkcionalnosti, nalik na `Emacs` i `Vim`. Ovo ih čini bržim pri radu jer nemaju veliki broj komponenti koji se ne koriste od strane tekućeg korisnika.

Zajednička osobina svih pomenutih okruženja je mogućnost dodavanja funkcionalnosti pomoću dodataka. U ovom odeljku ćemo malo detaljnije opisati editor `Atom` (slika 2.4) koji je dizajniran sa velikim fokusom na dodatke, poznate kao paketi [5] u `Atom` zajednici. Editor `Atom` pruža samo najosnovnije mogućnosti obrade teksta bez paketa. Čak i neke od funkcionalnosti koje su neophodne za rad editora su implementirana u okviru `Atom` paketa. Na primer, prikaz otvorenih datoteka, otvaranje više datoteka po ,,tabovima" i pretraga teksta su implementirani u okviru paketa koji se isporučuju sa editorom `Atom`. Kako bi se programerskoj zajednici omogućio lakši rad sa paketima, `Atom` omogućava razvoj i distribuciju paketa pomoću samog editora i alata komandne linije `apm` (koji podseća na `apt` alat na `Linux` distribucijama) kao i centralizovan repozitorijum paketa na Internetu (`https://atom.io/packages`). Aktivna zajednica čini ovakvu organizaciju mogućom jer su razni alati dostupni `Atom` korisnicima bez aktivnog angažmana od strane inženjera koji direktno razvijaju editor. Alati opisani u ovom radu, `OmniSharp`[19] i `Emmet`[20], dostupni su i kao `Atom` paketi.

[11] `https://www.vim.org/`
[12] `https://visualstudio.microsoft.com/`
[13] `https://developer.apple.com/xcode/`
[14] `https://www.jetbrains.com/idea/`
[15] `https://www.sublimetext.com/`
[16] `https://atom.io/`
[17] `https://notepad-plus-plus.org/`
[18] `https://code.visualstudio.com/`
[19] `https://atom.io/packages/omnisharp-atom`
[20] `https://atom.io/packages/emmet`



Slika 2.4: Izgled editora Atom

Glava 3

Jezik Homotopy

U ovoj glavi data je sintaksa jezika Homotopy kao i pravila po kojima se programi, u daljem tekstu fragmenti, napisani u jeziku Homotopy prevode na izlazni jezik. Jezik se sastoji iz dva dela. Prvi, jezik definicija fragmenata pomoću kojih se definiše izlazni jezik i drugi, jezik fragmenata koji se koristi za pisanje fragmenata koji se prevode u izlazni jezik. Homotopy prati i standardna biblioteka koja sadrži bogatu kolekciju unapred definisanih fragmenata (oni su definisani u jeziku definicija fragmenata). Korisnik na početku svog korišćenja alata Homotopy može da koristi unapred definisane fragmente i bitno mu je da ovlada pre svega jezikom fragmenata. U naprednijem radu, korisnik ima potrebu da definiše i svoje, nove fragmente i tada je potrebno da savlada i jezik definicija fragmenata.

3.1 Jezik definicija fragmenata

Definicija fragmenta je tekst koji u sebi sadrži obeležena mesta koja se mogu razvijati u još teksta. Svaki fragment sačinjen je od imena definicije, jezika (ili liste jezika) u kojima je ta definicija relevantna i teksta definicije fragmenta. Ovo se može videti i u tabeli 3.1. Primer definicije fragmenta može videti u primeru 3.1.1.

Deo	Opis
name	ime definicije
language	jezik ili lista jezika u kojima je definicija relevantna
snippet	tekst definicije fragmenta

Tabela 3.1: Delovi definicije fragmenta

Parametar	Opis
!	Ime klase
~	Implementacija interfejsa
:	Nasleđivanje
^	Šablon
@	Ime medoda/funkcije
#	Tip
$	Vrednost
%	Ostalo

Tabela 3.2: Konvencija pri korišćenju parametara

3.1.1 Parametri

Definicije fragmenata mogu sadržati oznake za parametre (karaktere `!`, `~`, `:`, `^`, `@`, `#`, `$` i `%`), odnosno mesta u definicijama koja se zamenjuju vrednostima parametara. Unutar definicija fragmenata parametri se označavaju ponavljanjem nekog specijalnog karaktera tri puta (isti taj karakter se koristi za navođenje vrednosti parametara). Razlog ovome je omogućavanje korišćenja vrednosti parametara unutar definicije fragmenta. Ovo se može videti na primeru 3.1.1, gde `#` ne predstavlja mesto za zamenu parametra već je deo sintakse jezika `C++`. Ovim je izbegnuta potreba za izbegavanjem unutar definicija fragmenata.

Primer 3.1.1

```
{"name": "stdinc","language": "C++","snippet": "#include <$$$>"}
```

Parametri mogu biti označeni različitim specijalnim karakterima. Ipak u standardnoj biblioteci jezika Homotopy koristi se određena konvencija kako bi se na osnovu specijalnog karaktera znalo šta paramtear označava i preporučeno je držati se te konvencije. Tabela 3.2 sadrži podržane parametre i konvenciju koju prati standardna biblioteka. Korisnicima se preporučuje da prate ovu konvenciju pri kreiranju sopstvenih definicija kako rad sa alatom bio prirodniji.

3.1.2 Spoljni parametri

Kao što će detaljno biti opisano u poglavlju 3.2.2, fragmenti se mogu organizovati hijerarhijski i unutrašnjost nekog (spoljnog) fragmenta mogu sačinjavati drugi (unutrašnji) fragmenti. Na primer, spoljni fragment može biti klasa u objektno-orijentisanom jeziku, a unutrašnji fragment može biti njen konstruktor. U jeziku Homotopy takođe je moguće i označavanje parametara koji su vezani za neki od spoljnih fragmenata (a ne samo parametara tekućeg fragmenta). Ova funkcionalnost inspirisana je činjenicom da je prilikom pisanja konstruktora u nekim programskim jezicima (C++, Java, C#, ...) potrebno koristiti ime klase.

Kako bi se označilo mesto za spoljni parametar potrebno je ispratiti sledeći format: `{{?<ime parametra>}}`. Ovo se može videti na primeru 3.1.2.

Primer 3.1.2

```
[
    {
        "name": "constr",
        "language": "C++",
        "snippet": "{{?!!!}}({{params}}){\n{{inside_block}}\n}"
    }
]
```

3.1.3 Podfragmenti

Prilikom prevođenja, svako pojavljivanje tekućeg parametra u tekućem fragmentu zamenjuje se vrednošću tog parametra. Da bi se omogućilo ponovno korišćenje istog parametra potrebno je definisati podfragment koji se razvija u slučaju da tekući fragment ne sadrži mesto za tekući paramterar.

Primer 3.1.3

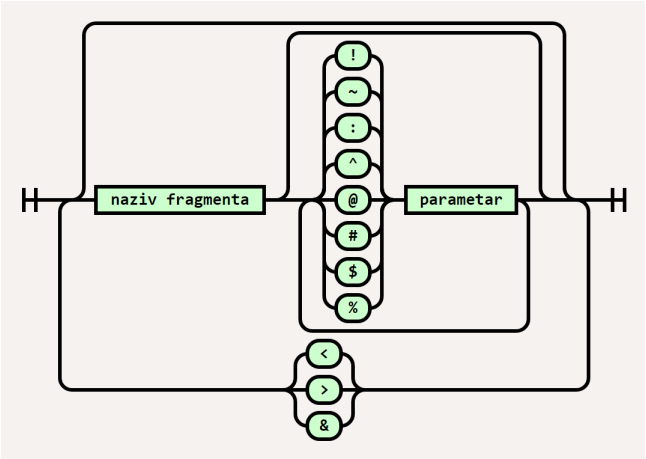
```
[
    {
        "name": "block",
        "language": "all",
        "snippet": ">>>{{opt_block}}"},
    {
        "name": "opt_block",
        "language": "all",
        "snippet": "\n>>>{{opt_block}}"
    }
]
```

U primeru 3.1.3 data je definicija za blok. Tekst `{{opt_block}}` se razvija samo ukoliko je to potrebno, osnosno ako `block` sadrzi više od jednog podfragmenta. Moguće je rekurzivno definisanje fragmenata što omogućava kreiranje fragmenata sa proizvoljnim brojem parametara, što se može videti na ovom primeru.

3.2 Jezik fragmenata

Jezik Homotopy određuje pravila po kojima se fragmenti prevode u izlazni jezik. Izlazni jezik definisan je definicijama fragmenata i to je obično neki programski jezik. Zato što se definicije fragmenata ne menjaju često, jezik fragmenata je jezik na kome se efektivno ,,programira", pa će se u daljem teksu termin jezik Homotopy koristiti umesto termina jezik fragmenata.

Osnovni sintaksički elementi jezika Homotopy su fragmenti koji su određeni imenima fragmenata i parametarima koji im se pridružuju. Dodatno je moguće kombinovati više fragmenata kako bi se konstruisale veće celine. Imena fragmenata odgovaraju fragmentima koji se nalaze u biblioteci fragmenata i pomoću kojih prevodilac konstruiše rezultat. Na slici 3.1 prikazan je sintaksički dijagram jezika Homotopy. Naziv fragmenata i parametri su zadati proizvoljnim tekstom dok su ostali karakteri vezivni operatori (ako se neki od tih karaktera navodi unutar proizvoljnog teksta, on mora biti izbegnut).



Slika 3.1: Sintaksički dijagram jezika fragmenata

3.2.1 Parametri

Najjednostavnije konstrukcije u jeziku Homotopy su parametri. Kao što je već opisano u delu 3.1.1, vrednost parametra se jednostavno zamenjuje u definiciji fragmenta na mestu označenom za taj parametar (mesto u definiciji obeleženo je trostrukim pojavljivanjem specijalnog karaktera, dok se u pozivu vrednost navodi nakon jednostrukog pojavljivanja specijalnog karaktera). Ta situacija može se videti na primeru 3.2.1 (u definiciji za `stdinc`, `#include <$$$>`, zamenjuje se tekst `$$$` datim tekstom `stdio.h`).

Primer 3.2.1

```
stdinc$stdio.h
```

```
#include <stdio.h>
```

Isti parametar se može upotrebiti i više puta. Na primer, funkcija može imati više parametara (kao u primeru 3.2.2). Ovo je omogućeno funkcionalnošću podfragmenata gde se parametri dodaju ako za to ima potrebe, slično primeru 3.1.3.

Karakter	Opis
>	Ulaz u novi fragment
<	Izlaz iz tekućeg fragmenta
&	Početak novog fragmenta

Tabela 3.3: Meta-karakteri za slaganje fragmenata

Primer 3.2.2

```
func#void@foo#int$i#int$j
```

```
void foo(int i, int j){

}
```

Parametri mogu biti označeni različitim specijalnim karakterima. U tabeli 3.2 prikazani su dopušteni parametri u jeziku Homotopy.

3.2.2 Hijerarhija fragmenata

Kombinovanjem više fragmenata moguće je kreirati složene jezičke konstrukcije. Fragmenti se mogu slagati redno, jedan uz drugi, ali moguće je da jedan (spoljni) fragment sadrži svoje (unutrašnje) fragmente čime se ostvaruje hijerarhijski odnos među fragmentima.

Fragmenti se u jeziku fragmenata razdvajaju meta-karakterima koji imaju posebno ponašanje. Karakterom `>` ,,ulazi" se u novi fragment dok se karakterom `<` ,,izlazi" iz tekućeg fragmenta. Redno spajanje, odnosno razdvajanje fragmenatra koji su sadržani unutar istog nadfragmenta, vrši se karakterom `&`. Ovo se može videti i u tabeli 3.3.

Primer 3.2.3

```
printf("first line");&printf("second line");
```

```
printf("first line");
printf("second line");
```

Primer 3.2.4

```
if$i==4>printf("unutar if naredbe");<printf("van if naredbe");
```

```
if(i==4){
    printf("unutar if naredbe");
}
printf("van if naredbe");
```

U primerima 3.2.3 i 3.2.4 može se videti ponašanje operatora `<`, `>` i `&`. U primeru 3.2.4 operator `<` koristi se kako bi se nadovezao naredni fragment nakon `if` naredbe gde se operatorom `<` efektivno izlazi iz `if` bloka dok se u primeru 3.2.3 koristi operator `&` kako bi se nadovezao naredni fragment nakon `printf` komande u istom bloku.

3.2.3 Izbegavanje

Kako bi se omogućilo korišćenje rezervisanih karaktera (a to su `<`, `>`, `&`, `!`, `~`, `:`, `^`, `@`, `#`, `$`, `%`) unutar fragmenata, alat Homotopy omogućava njihovo izbegavanje. Bilo koji karakter nakon `\` tretira se doslovno i ne pridaje mu se specijalno značenje definisano jezikom Homotopy. Tako je za navođenje uskličnika unutar tela funkcije `main` u primeru ,,zdravo svete“ bilo neophodno izbegnuti ga (u suprotnom bi bio tretiran kao parametar koji po konvenciji odgovara nazivu klase). U primeru 3.2.5 znak uzvika, `!`, izbegnut je kako se ne bi tretirao kao parametar već kao deo string u jeziku C++.

Primer 3.2.5

```
printf("Zdravo\!");
```

```
printf("Zdravo!");
```

3.2.4 Prečice

Kako bi se česte konstrukcije učinile jednostavnije za pisanje, alat Homotopy ima prečice koje se razvijaju prilikom faze pretprocesiranja u fragmente koji se prevode u izvorni kôd. Ime prečice navodi se u duplim uglastim zagradama i ono se prilikom pretprocesiranja razvija u vrednost fragmenta koji odgovara imenu prečice. U primeru 3.2.6 prikazano je korišćenje prečice za `main` funkciju. Ova prečica se koristi i u primeru ,,zdravo svete“ gde se unutar same `main` funkcije nalazi još fragmenata.

Primer 3.2.6

```
[[main]]
```

```
int main(int argc, char* argv[]){

}
```

3.3 Standardna biblioteka

Standardna biblioteka sadrži definicije fragmenata koji su neophodni za rad prevodioca kao i biblioteku jezičkih konstrukcija u nekoliko programskih jezika.

Sledi lista trenutno podržanih jezika:

- C
- C++
- Java
- Python
- JavaScript

U ovoj sekciji biće prikazani primeri fragmenata za nekoliko jezičkih konstrukcija. Više primera dato je u dodatku A.

3.3.1 Naredba switch

Switch naredba ima nekoliko potencijalnih problema. Naredba `break` se lako zaboravlja i često deluje kao da bi trebalo da bude podrazumevana. Homotopy omogućava da se na koncizan način daju informacije koje su potrebne za konstruisanje cele switch konstrukcije bez mogućnosti da se nešto izostavi. Dodatno je moguće korišćenje propadanja na jedan od retkih načina gde to ima smisla, zadavanju više vrednosti koje proizvode isto ponašanje.

Primer 3.3.1

```
switch$i>case$1$2>printf("one or two");
```

```
switch(i){
    case 1:
    case 2:
        printf("one or two");
        break;
}
```

3.3.2 Kreiranje funkcije

Kreiranje funkcije koja ima više parametara istog tipa dovodi do naizgled nepotrebnog navođenja istog tipa više puta. Homotopy omogućava da se izbegne ponovno navođenje tipa tako što se prvo navedu sve promenljive (`i` i `j` u ovom slučaju) pa se zatim navodi ime tipa samo jednom (`int` u ovom slučaju).

Primer 3.3.2

```
func#int@plus$i$j#int>return i+j;
```

```
int plus(int i, int j){
    return i+j;
}
```

3.3.3 Šabloni

Šabloni omogućavaju parametrizaciju tipova kao što funkcije omogućavaju parametrizaciju kôda. Iako je koncept isti, sintaksa za ovo je drugačija u različitim jezicima. U nekim, kao što je C++, potrebno je navesti celu ključnu reč za ovu svrhu. Alat Homotopy omogućava dodavanje šablon tipa na isti način kao dodavanje parametra.

Primer 3.3.3

```
class!A^T
```

```
template <class T>
class A {

};
```

3.3.4 Projektni obrasci

Prilikom objektno-orijentisanog programiranja često se koriste određeni projektni obrasci [3]. Neki od njih su podržani i standardnom bibliotekom jezika Homotopy. Implementacija projektnih obrazaca zna da bude naporna za kodiranje. Homotopy olakšava kodiranje nekih obrazaca. Dodatno se smanjuje šansa da se propusti neki detalj implementacije. Na primer, u jeziku C++, deklarisanje privatnog operatora dodele se lako zaboravlja a omogućava kopiranje objekta što je protivno dizajnu projektnog obrasca [3] singleton.

Primer 3.3.4

```
class!A>[[singleton]]
```

```
class A {
public:
    A& getInstance(){
        static A instance;

        return instance;
    }
private:
    A(){}
    A(A const& origin);
    void operator=(A const& origin);
};
```

3.3.5 Anonimne funkcije

Prosleđivanje anonimne funkcije kao parametra je česta praksa. Neki jezici[1] imaju mogućnost da kratke anonimne funkcije budu napisane u vrlo malo karaktera, ali to u nekim jezicima nije slučaj. Homotopy omogućava pisanje ovakvih funkcija brzo čak i kada je potrebno napisati i nekoliko ključnih reči u izlaznom jeziku.

Primer 3.3.5

```
call2@foo$param1>f$x>x
```

```
foo(
  param1,
  function (x){ return x; }
);
```

3.3.6 JSON

Najveći problem pri unosu velikih objekata su zagrade i ostali kontrolni karakteri. Homotopy omogućava unošenje ovakvih objekata na način koji podseća na obilazak stabla u *pre-order* redosledu.

Primer 3.3.6

```
d>k$item1>d>k$nested$1<<&k$item2$2
```

```
{
  "item1": {
    "nested": 1
  },
  "item2": 2
}
```

[1] JavaScript standard ECMA 6 uvodi koncizniji način zadavanja anonimnih funkcija.

Glava 4

Prevodilac Homotopy

4.1 Uvod

Homotopy je prevodilac za fragmente kôda sa jezika Homotopy na željeni izvorni jezik (definisan bibliotekom definicija fragmenata). Jezik Homotopy dizajniran je sa akcentom na lakoću unosa kôda što je sažetije moguće na uštrb čitljivosti. Za čitljivost i formatiranje kôda brine alat Homotopy pa se kao rezultat omogućava programeru da ne brine o urednosti kôda već da se koncentriše na ideju koju implementira.

4.2 Implementacija

Alat Homotopy[1] implementira jezik opisan u glavi 3. Ovo poglavlje sadrži detalje implementacije. Prvo je dat grub prikaz rada prevodioca, zatim su opisane tehnologije i metodologije koje se korišćene pri implementaciji, a nakon toga su opisani delovi alata, njihove odgovornosti i funkcionalnosti.

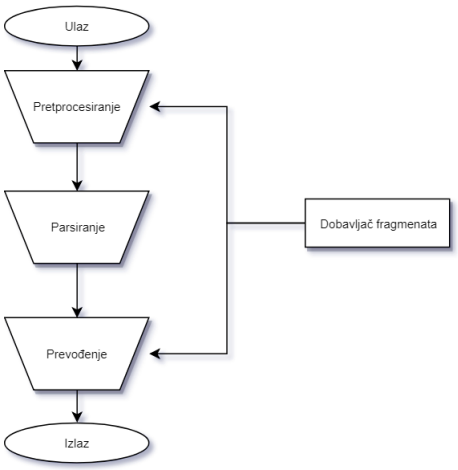
Na slici 4.1 prikazan je ugrubo algoritam rada prevodioca. Ulaz se prvo pretprocesira, što uključuje razvijanje prečica i postavljanje oznake za poziciju kursora ukoliko je to potrebno. Nakon toga se tekst parsira i kreira se sintaksičko stablo. Potom se generiše izlaz uz pomoć dobavljača skraćenica. Dobavljač skraćenica se takođe koristi i od strane pretprocesora pri procesu razvijanja prečica.

4.2.1 Alati i razvojno okruženje

Python

Alat Homotopy razvijen je kao paket u programskom jeziku Python[2]. Python je slabo tipiziran programski jezik visokog nivoa opšte namene. Sintaksa

[1] Dostupan besplatno na `https://github.com/Ahhhhmed/homotopy`
[2] `https://www.python.org`



Slika 4.1: Dijagram toka

programskog jezika Python dizajnirana je sa akcentom na preglednost izvornog kôda. Jedan je od najpopularnijih programskih jezika u svetu. Zbog toga postoji mnoštvo alata koji olakšavaju razvijanje programa u jeziku Python. Ovo je jedan od glavnih razloga zbog kojih je alat Homotopy razvijen u jeziku Python. Još jedan bitan razlog za biranje Python-a za razvijanje alata Homotopy je činjenica da se programi napisanu u Python-u mogu pokretati na svim popularnim operativnim sistemima. Iako su performanse uvek bitne, Python nije dizajniran sa glavnim fokusom na brzinu izvršavanja. Ovo dovodi to toga da algoritam napisan u Pythonu može da bude znatno sporiji od istog algoritma napisanom u nekom drugom jeziku, na primer u jeziku C++. S obzirom na to da alat Homotopy ne očekuje velike ulaze ova osobina jezika Python ne dolazi do izražaja.

Python unittest

Alat Homotopy koristi Python unittest[3] za testiranje kôda. Sav kôd je pokriven testovima. Python unittest je deo Python standardne biblioteke i služi za testiranje softvera. Omogućava jednostavno kreiranje, grupisanje i pokretanje testova.

Pypi

Alat Homotopy kreiran je kao paket koji se jednostavno može instalirati[4] pomoću pip alata. Pypi (`https://pypi.org/`) je zvaničan repozitorijum za Pyton projekte. Omogućava održavanje i distribuciju Python paketa kao i jednostavno instaliranje pomoću propratnog alata komandne linije, pip.

[3] `https://docs.python.org/3/library/unittest.html`
[4] pip install homotopy

Git

Homotopy projekat koristi Git (`https://git-scm.com/`) za kontrolu verzija kôda. Git je sistem za upravljanje izvornim kôdom koji prati promene računarskih datoteka i koordiniše rad nad tim datotekama između više ljudi. Primarno se koristi za kontrolu izvornog kôda pri razvoju softvera, ali se može koristiti i za praćenje promena na bilo kakvim datotekama.

GitHub

Izvorni kôd alata Homotopy nalazi se na GitHub-u. GitHub (`https://github.com/`) je servis za čuvanje izvornog kôda pod sistemom Git. Svojim korisnicima omogućava, osim korišćenja svih funkcionalnosti Git-a, i dodatne funkcionalnosti kao što je praćenje grešaka, praćenje predloga za nove funkcionalnosti, upravljanje projektom i održavanje dokumentacije.

CI/CD

Da bi se obezbedilo veći kvalitet kôda Homotopy, kôd je potpuno pokriven testovima. Ovi testovi se izvršavaju svaki put kada se promeni kôd na GitHub repozitorijumu. Testovi se svaki put pokreću unutar svežeg okruženja tako da ne postoji šansa da postoje zavisnosti kôda od nečega što nije u repozitorijumu ili nije eksplicitno navedeno kao zavisnost.

U slučaju da je komit (eng. commit) označen (eng. tag) dodatno se nakon pokretanja testova paket automatski postavlja na Pypi.

Ovaj princip omogućava povećanu sigurnost pri razvoju i olakšava objavljivanje novih verzija. U literaturi je poznat pod nazivom CI/CD [4] (eng. Continuous Integration/Continuous Delivery).

JSON

Biblioteka fragmenata koje koristi alat Homotopy data je u JSON datotekama. JSON (`http://json.org`) je tekstualni format za čuvanje podataka. Inspirisan objektima u jeziku JavaScript[5] JSON čuva podatke unutar nizova i objekata kao kompozitnih struktura koje omogućavaju čuvanje podataka proizvolje veličine.

Alat Homotopy čuva fragmente u datotekama koje su u JSON formatu, što uključuje i standardnu biblioteku fragmenata. Za parsiranje teksta koji je u JSON formatu Homotopy koristi json modul koji je deo standardne biblioteke jezika Python.

4.2.2 Pretprocesor

Pre parsiranja i kreiranja sintaksičkog stabla, preprocesor obrađuje ulaz kako bi omogućio dodatne funkcionalnosti.

[5] JSON je skraćenica za *JavaScript Object Notation*

Prečice

Postoje obimnije konstrukcije koje se često koriste. Iz tog razloga, Homotopy uvodi prečice koje omogučavaju brzo pisanje ovakvih konstrukcija. Prilikom pretprocesiranja imena fragmenata navedenih u duplim uglastim zagradama se razvijaju u vrednosti odgovarajućih fragmenata kako bi se omogućila ova funkcionalnost.

Preprocesor zamenjuje sva mesta u ulazu koji su prepoznati regularnim izrazom `\[\[(.*?)\]\]`. Ovo smo videli u primeru „zdravo svete“. U isečku kôda 1 može se videti implementacija razvijanja prečica. Pretprocesor se oslanja na dobavljač skracenica (o kome će biti reči u odeljku 4.2.5) za vrednosti koje treba da stoje na označenim pozicijama.

```
def expand_decorators(self, snippet_text):
    """
    Expand decorators to enable concise writing of common patterns.

    :param snippet_text: Snippet text
    :return: Expanded snippet text
    """
    return re.sub(
            r'\[\[(.*?)\]\]',
            lambda match_group:
                self.snippet_provider[match_group.group(1)],
            snippet_text)
```

Isečak kôda 1: Razvijanje dekoratora

Oznaka za kursor

Prilikom korišćenja alata u nekom od alata za obradu teksta potrebno je da se nakon razvijanja fragmenta kursor postavi na što bolju poziciju. Kako bi se ovo omogućilo, pretprocesor stavlja marker koji označava mesto na kome treba postaviti kursor.

Marker se postavlja tako što se na kraj fragmenta doda još jedan fragment, `&[{cursor_marker}]`. Ovo ima efekat da tekst dodat nakon razvijanja bude na mestu na kome bi bio isti tekst kada bi bio u narednom fragmentu. Drugim rečima, razvijanje fragmenta i dodavanje teksta komutiraju. Implementacija se može videti u isečku kôda 2. Ova funkcionalnost je opciona jer se od korisnika (koji je najčešće dodatak za editor) očekuje da dodatno obradi rezultat.

4.2.3 Sintaksičko stablo

Fragment se interno predstavlja kao sintaksičko stablo. Sintaksa jezika Homotopy nije bogata jezičkim kontstrukcijama pa je sintaksičko stablo relativno

```python
def put_cursor_marker(snippet_text):
    """
    Put cursor marker witch should be used by editor plugins.

    :param snippet_text: Snippet text
    :return: Snippet text with marker at the end
    """
    return "".join([
            snippet_text,
            parser.Parser.and_operator,
            Preprocessor.cursor_marker])
```

Isečak kôda 2: Postavljanje markera za kursor

jednostavno.

Sintaksičko stablo je kompozitna struktura[6] koja može sadržati dve vrste čvorova, jednostavne i kompozitne fragmente. Dodatno, ova strukrura podržava dodavanje funkcionalnosti (na primer, prevođenje uz pomoć biblioteke fragmenata) u skladu sa projektnim obrascem [3] posetilac.

Jednostavan fragment

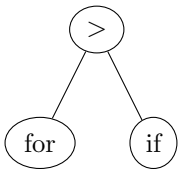
Jednostavan fragment čuva jednu vrednost i to je vrednost fragmenta. Ovo je osnovna gradivna jedinica svih fragmenata.

Kompozitni fragment

Kompozitni fragment spaja dva fragmenta operatorom. Čuva levi i desni podfragment kao i vrednost operatora kojim su oni spojeni. Primer 4.2.1 prikazuje stablo u kome su čvorovi `for` i `if` spojeni operatorom za ulaz u novi fragment.

Primer 4.2.1

`for>if`



[6] Dizajniran u skladu sa projektnim obrascem [3] kompozicije (eng. Composite)

4.2.4 Parser

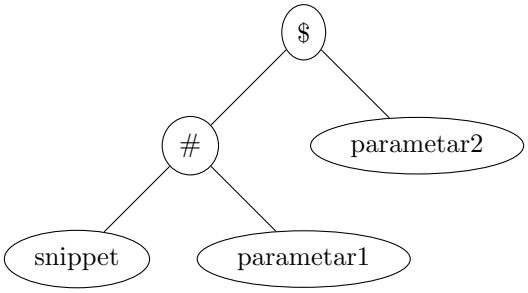
Parser je odgovoran za konstruisanje sintaksičkog stabla na osnovu teksta fragmenta. Homotopy koristi specijalizovani parser kako bi konstruisao sintaksičko stablo. Iako postoji veliki broj paketa (na primer, ply[7]) koji omogućavaju automatizovanje parsera na osnovu gramatike Homotopy koristi specijalizovani, ručno implementiran parser kako bi lakše omogućio rad sa unutrašnjim fragmentima.

Parametri

Kada parser prepozna parametar kreira se novi kompozitan čvor koji sa leve strane sadrži predhodni koren a sa desne jednostavan čvor koji sadrži parametar. Vrednost parametra se takođe čuva u novom kompozitnom čvoru koji postaje novi koren prepoznatog stabla. Lista parametara koje parser prepoznaje data je u tabeli 3.2. U primeru 4.2.2 dat je fragment i sintaksno stablo koje parser kreira za taj fragment. Prvo se prepozna jednostavan fragment `snippet`, zatim se sačuva vrednost poslednjeg parametra (`#` u ovom slučaju). Nakon prepoznavanja jednostavnog fragmenta `parameter1` dolazi do spajanja tog fragmenta i predhodno konstruisanog fragmenta (`snippet`) u kompozitni fragment. Ovaj proces se ponavlja za `parameter2` i operator `$` da bi se dobilo stablo prikazano na slici.

Primer 4.2.2

```
snippet#parametar1#parametar2
```



Unutrašnjost fragmenta

Kada bi postojao samo ulaz u novi fragment (označen operatorom `>`) parsiranje bi se moglo izvesti kao parsiranje binarnih operatora gde je ulaz u novi fragment operator desno asocijativan operator nižeg prioriteta dok su svi ostali operatori levo asocijativni i višeg prioriteta (što se može videti u isečku kôda 3). Međutim, ovo nije slučaj (zbog operatora `<` i `&`) pa je potrebno vršiti parsiranje na drugačiji način.

[7] Više informacija dostupno na zvaničnoj stranici paketa. `http://www.dabeaz.com/ply`

```
if last_operator == Parser.in_operator:
    stack.append(make_snippet(current_match))
else:
    current_snippet = stack.pop()
    stack.append(
        CompositeSnippet(
            current_snippet,
            last_operator,
            make_snippet(current_match)))
```

Isečak kôda 3: Dodavanje novoprepoznatog fragmenta na stek

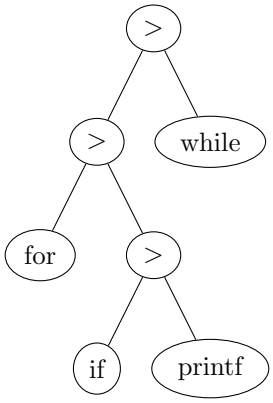
Prilikom parsiranja čuva se taj stek sa prepoznatim fragmentima. Kada se naiđe na operator za novi fragment prelazi se na novi nivo steka (inače se spajaju dva predhodno prepoznata fragmenta kao što se može videti u isečku kôda 3). Na kraju se nivoi na steku povezuju operatorom za novi fragment slično kao obični parametri. Ovo podseća na rekurzivni pristup. Zapravo, logika vezana za operator za ulaz u novi fragment i jeste rekurzivna, samo što se umesto implicitnog steka koji Python koristi za izvršavanje programa koristi eksplicitan stek. Razlog ovome je što se stek koristi u implementaciji ostala dva operatora za nove fragmente. Nailaskom na početak novog fragmenta (operator `&`) poslednja dva nivoa na steku se spajaju operatorom ulaska u novi fragment. Ovo za efekat ima da se naredni fragment nalazi u unutrašnjosti fragmenta dva nivoa od vrha steka, što je željeno ponašanje. Nakon nailaska na operator za izlaz iz tekućeg fragmenta potrebno je da naredni fragment bude unutar fragmenta koji se nalazi tri nivoa od vrha steka. Zato se u ovom slučaju operacija spajanja poslednja dva nivoa jednostavno ponovi dva puta. Isečak kôda 4 implementira ovo ponašanje. Svi operatori (`&`, `<` i `>`) se u sintaksičkom stablu čuvaju kao jedinstaven operator (proizvoljno je odabran `>`), jer se u daljem prevođenju svi ovi operatori ponašaju potpuno isto.

```
if c == Parser.and_operator:
    last_operator = Parser.in_operator
    Parser.merge_stack(stack)

if c == Parser.out_operator:
    last_operator = Parser.in_operator

    for _ in range(2):
        Parser.merge_stack(stack)
```

Isečak kôda 4: Spajanje vrha steka

Primer 4.2.3

```
for>if>printf<while
```



U primeru 4.2.3 ilustovana je `for` petlja u kojoj se nalazi `if` naredba u kojoj se nalazi `printf` komanda. Nakon toga se izlazi iz `if` naredbe kako bi se nadovezala `while` petlja koja je takođe unutar `for` petlje.

Implicitni fragment

U predhodnom odeljku opisan je rad parsera pomoću steka sa prepoznatim fragmentima. Prilikom prepoznavanja moguće je izaći i iz fragmenta na prvom nivou. Kako ovo ne bi proizvodilo grešku parser koristi `block` fragment kao fragment na nivou iznad tekućeg. Ovo efektivno znači da parser počinje sa stekom sa beskonačno mnogo implicitnih fragmenata na njemu (to naravno nije slučaj u praksi, ali bi ponašanje alata bilo identično). Fragment `block` definisan je u standardnoj biblioteci za sve jezike. On razdvaja svoje podfragmente po redovima.

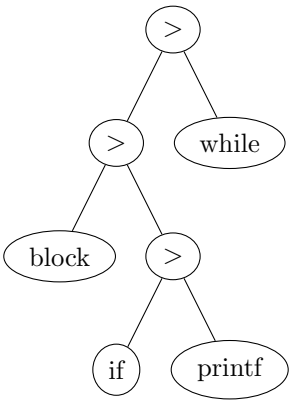
Primer 4.2.4, u kome se može videti implicitni fragmen, sličan je primeru 4.2.3. Jedina razlika je što je izostavljena `for` petlja. Sintaksno takođe izgleda slično, razlikuju se samo u tome što se u ovom primeru koristi implicitni fragment `block` umesto `for` petlje.

Izbegavanje karaktera

Kako bi se omogućilo korišćenje karaktera koji označavaju operatore jezika Homotopy moguće je izbeći njihovo regularno ponašanje. Ovo se radi tako što se ispred željenog znaka stavi karakter \. Nakon ovog karaktera, svaki karakter (pa i karakteri koji imaju specijalna značenja u jeziku Homotopy) će biti tretiran doslovno, kao običan karakter. Implementacija se može videti u isečku kôda 5. U primeru 4.2.5 znak `<` (manje) je izbegnut kako se ne bi tretirao kao operator za izlazak iz fragmenta.

Primer 4.2.4

```
if>printf<while
```



Primer 4.2.5

```
if$i \< 3>printf("i je manje od 3");
```

```
if(i < 3){
  printf("i je manje od 3");
}
```

```
if in_escape_sequence and c != "\0":
    current_match.append(c)
    in_escape_sequence = False
    continue

if c == Parser.escape_character:
    in_escape_sequence = True
    continue
```

Isečak kôda 5: Izbegavanje

4.2.5 Dobavljač skraćenica

Kako bi se rad sa bibliotekom fragmenata razdvojio od ostatka alata radi lakšeg korišćenja i veće fleksibilnosti uvodi se komponenta dobavljač skraćenica. Dobavljač skraćenica ostatku alata omogućava da koristi biblioteku koja je napisana u više datoteka jednostavnim pozivom metode koja za dati fragment vraća njegovu prevedenu vrednost. Dodatno, dobavljač skraćenica je odgovoran za dopremanje skraćenica za odgovarajući programski jezik.

Datoteke sa definicijama

Dobavljač skraćenica očekuje datoteke formatirane na sledeći način. Svaka datoteka treba da sadrži niz JSON objekata koji opisuju konkretne fragmente. Svaki od tih objekata treba da sadrži ime fragmenta, ime ili niz imena jezika u kojima se može koristiti (moguće je i označavanje fragmenata koji su validni u svim jezicima pomoću ključne reči `all` kao i pojedinačno isključivanje jezika dodavanjem prefiksa `~` ispred imena jezika) i vrednost fragmenta.

U primeru 4.2.6 dat je JSON niz u kome su definicije fragmenata za `stdinc`, `block` i pomoćni fragment `opt_block`. Fragment `stdinc` relevantan je samo u jezicima `C` i `C++` dok su fragmenti `block` i `opt_block` relevantni u svim jezicima.

Greške

Postoji nekoliko grešaka koje mogu da se dogode pri obradi definicija fragmenata koje mogu biti od značaja za korisnika.

Na primer, ukoliko datoteka ne sadrži tekst koji je u potrebnom formatu, na izlaz za standardnu grešku se upisuje informacija o grešci i ta datoteka se ignoriše. Slično, u slučaju da se među definicijama nalazi dve ili više definicija za isti fragment, na izlaz za standardnu grešku se takođe upisuje informacija o toj grešci.

Dobavljač fragmenata se trudi da nastavi rad iako postoje greške u nekim od datoteka i da o tome obaveštava korisnika umesto da obustavlja rad celog alata.

Primer 4.2.6

```
[
    {
        "name": "stdinc",
        "language": ["C++", "C"],
        "snippet": "#include <$$$>"
    },
    {
        "name": "block",
        "language": "all",
        "snippet": ">>>{{opt_block}}"
    },
    {
        "name": "opt_block",
        "language": "all"
        ,"snippet": "\n>>>{{opt_block}}"
    }
]
```

4.2.6 Generator izlaznog kôda

Generator izlaznog kôda je ključni deo alata Homotopy. On pretvara sintaksičko stablo u rezultujući tekst oslanjajući se na dobavljač skraćenica.

Prosti fragmenti

Za proste fragmente, odnosno listove sintaksičkog stabla, izlaz se generiše na sledeći način (dat je i isečak kôda 6 koji impmentria ovu funkcionalnost).

- Ukoliko list označava vrednost parametra izlaz je vrednost u listu u kojoj su razvijeni spoljni parametri.
- Inače se prevođenje delegira dobavljaču skraćenica. Dobavljač skraćenica vraća vrednost fragmenta ukoliko u biblioteci fragmenata postoji fragment sa datim imenom. Ukoliko to nije slučaj vraća prosleđenu vrednost imena fragmenta. Nakon dobijanja vrednosti od dobavljača skraćenica vrši se razvijanje spoljnih parametara.

Parametri

Zamena parametra njegovom vrednošću je osnovna transformacija koja se dešava pri prevođenju. Pre nego što se izvrši zamena rekurzivno se prevode leva strana. Nakon toga se ukoliko ne postoji adekvatno mesto za smeštanje desne strane razvijaju podfragmenti. Potom se zamenjuje rekurzivno prevedena desna

```
def visit_simple_snippet(self, simple_snippet):
    """
    Compile simple snippet.

    :param simple_snippet: Simple snippet
    :return: Text of compile snippet
    """
    snippet_text = simple_snippet.value \
            if self.inside_parameter \
            else self.snippet_provider[simple_snippet.value]

    return self.expand_variable_operators(snippet_text)
```

Isečak kôda 6: Obilazak prostoh fragmenta

strana u levoj strani na mesta označena odgovarajučim parametrom. Implementacija ove funkcionalnosti može se videti u isečku kôda 7.

Primer 4.2.7 ilustruje redosled obrade čvorova ulaznog sintaksnog stabla. Jedina operacija koja se vrši u ovom primeru je zamena oznake za parametar njegovom vrednošću. Na početku primera data je i definicija fragmenta koji se koristi pri prevođenju.

Podfragmenti

Fragment u svojoj definiciji može sadržati druge podfragmente koji se razvijaju ako postoji potreba za to, odnosno ako se razvijanjem dobija mesto na kome se stavlja vrednost tekućeg parametra. U slučaju da ne postoji mesto na kome se može staviti vrednost tekućeg parametra, generator kôda počinje pretragu za podfragmentom koji sadrži takvo mesto. Generator kôda vrši pretragu pomoću regularnog izraza `\[\[(.*?)\]\]` i za svaki od rezultata pretrage proverava da li u svojoj definiciji sadrži mesto za tekući parametar. Prvi takav podfragment se zamenjuje svojom definicijom na mestu koje je prepoznato regulatnim izrazom. Nakon toga se vrši zamena parametra kao što je opisano u predhodnom odeljku. Isečak kôda 8 prikazuje ovu funkcionalnost.

Postoji mogućnost da i nakon obrade stabla ostanu oznake za podfragmente u rezultatu. Zato se nakon obilaska stabla svi podfragmenti koji su ostali nerazvijeni uklanjaju iz rezultata.

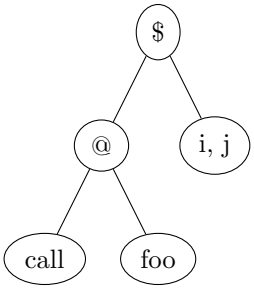
Primer 4.2.8 sličan je primeru 4.2.7 (rezultat je isti), samo što se ovde koriste podfragmenti kako bi se razdvojili parametri. U koracima 4 i 6 dolazi do razvijanja podfragmenata kako bi se dobilo mesto za zamenu parametra. Definicije fragmenata koji se koriste u ovom primeru dati su na slici 4.2.

Primer 4.2.7

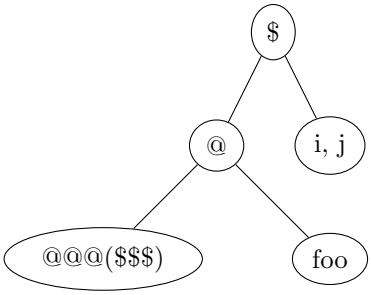
```
{
    "name": "call",
    "language": "C++",
    "snippet": "@@@($$$)"
}
```

```
call@foo$i, j
```

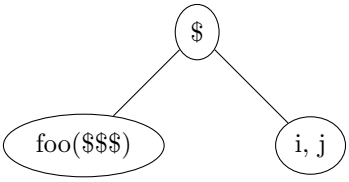
1. korak



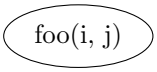
2. korak



3. korak



4. korak



```python
def visit_composite_snippet(self, composite_snippet):
    """
    Generate output for composite snippet.

    :param composite_snippet: Composite snippet
    :return: Text of left side replaced with right side
    """
    left_side = self.visit(composite_snippet.left)
    operation_text = composite_snippet.operation * 3

    if operation_text not in left_side:
        left_side = self.expand_snippet(left_side, operation_text)

    return self.substitute(
        left_side,
        composite_snippet.operation,
        composite_snippet.right,
        operation_text)
```

Isečak kôda 7: Zamena parametara

Spoljni parametri

U definicijama fragmenata moguće je koristiti parametre koji su već korišćeni u predhodnim fragmentima. Kako bi ovo podržavao, generator kôda vodi računa o promenljivama koje je menjao kao i nivoima unutrašnjih fragmenata u kojima je trenutno. Logika za ovo nalazi se u posebnoj pomoćnoj klasi i generator kôda prati sledecu proceduru pri svom radu.

- Kada se vrši zamena parametra, njegova vrednost se pamti u tekućem rečniku.
- Kada se ulazi u unutrašnji fragment, dodaje se novi rečnik na stek.
- Kada se izlazi iz unutrašnjeg fragmenta, briše se rečnik sa vrha steka.
- Kada je potrebno zameniti parametar, koristi se vrednost parametra najbliža vrhu steka, ne uključujući vrh steka (jer se tu nalaze parametri na tekućem nivou).

Glavna primena ove funkcionalnosti je sa konstruktorima koji sadrže ime klase u kojoj se nalaze što se može videti u primeru 4.2.9.

4.2.7 Korisnički interfejs

Python paket u kome je napisan prevodilac za Homotopy predviđen je da se koristi kao alat iz konzolne linije od strane dodatka za editore.

```python
def expand_snippet(self, snippet_text, operation_text):
    """
    Expend snippet to uncover possible operator definition.

    :param snippet_text: Snippet text
    :param operation_text: Operation text
    :return: Expanded snippet
    """
    match_found = False

    def expansion_function(match_object):
        nonlocal match_found

        if not match_found \
                and operation_text \
                in self.snippet_provider[match_object.group(1)]:
            match_found = True
            return self.snippet_provider[match_object.group(1)]

        return match_object.group(0)

    return re.sub(
        r'{{([^{]*?)}}',
        expansion_function,
        snippet_text)
```

Isečak kôda 8: Razvijanje podfragmenata

Homotopy kao biblioteka

Alat Homotopy moguće koristiti i kao Python biblioteku. Paket Homotopy izvozi jednu klasu pod imenom `Homotopy`. Ova klasa implementira projektni obrazac [3] fasadu i služi za interakciju sa alatom. Program komandne linije `homotopy` parsira argumente komandne linije i prosleđuje ih ovoj klasi. Primer 4.2.10 prikazuje Python program koji konstruiše instancu klase `Homotopy` koja služi sa prevođenje fragmenata na jezik `C++` i ispisuje prevod fragmenta `for#int$i%5>printf("hello");`.

Korišćenje iz komandne linije

Program komandne linije prikuplja parametre komandne linije, parsira ih u odgovarajući format i prosleđuje ostatku alata na prevođenje. Nakon prevođenja ispisuje rezultat na standardni izlaz.

Za rad sa argumentima komandne linije koristi se paket iz standardne biblioteke pod nazivom `argparse` [6]. Ovaj paket omogućava navođenje potrebnih i

```python
def generate_code(self, snippet):
    """
    Generate code for a snippet. Visit and then perform a clean.

    :param snippet: Snippet
    :return: Text of compiled snippet
    """
    compiled_snippet = self.visit(snippet)

    return re.sub(r'({{[^{]*?}})', "", compiled_snippet)
```

Isečak kôda 9: Generisanje kôda

opcionih parametara i proverava da li je korisnik ispravno uneo parametre i ispisuje informacije o grešci ukoliko do nje dođe. Takođe olakšava pisanje komandi za ispisivanje pomoći. Na slici 4.3 prikazano je kratko upuststvo za korišćenje koje se generiše pomoću biblioteke `argparse`.

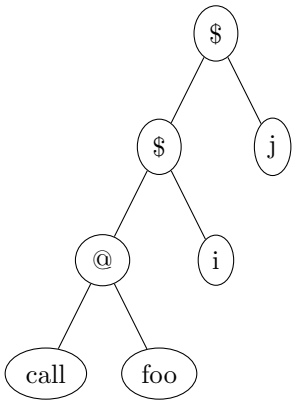
```
[
    {
        "name": "call",
        "language": "C++",
        "snippet": "@@@({{params}})"
    },
    {
        "name": "params",
        "language": "C++",
        "snippet": "$$${{opt_params}}"
    },
    {
        "name": "opt_params",
        "language": "C++",
        "snippet": ", $$${{opt_params}}"
    }
]
```

Slika 4.2: Definicije fragmenata korišćene u primeru 4.2.8

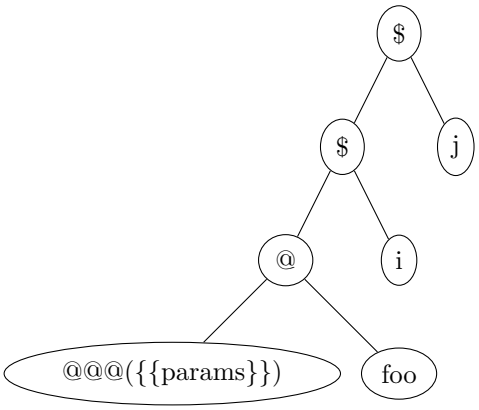
Primer 4.2.8

```
call@foo$i$j
```

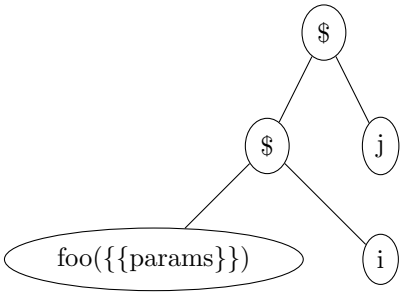
1. korak



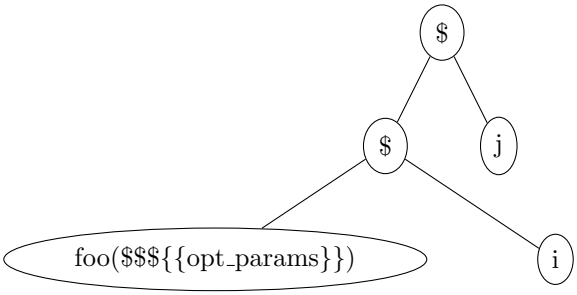
2. korak



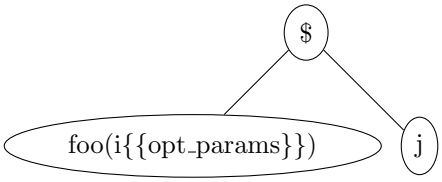
3. korak



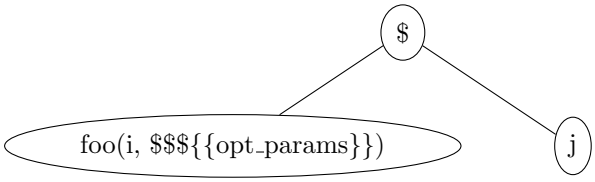
4. korak



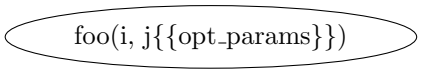
5. korak



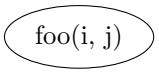
6. korak



7. korak



8. korak



Primer 4.2.9

```
{
    "name": "constructor",
    "language": "java",
    "snippet": "public {{?!!!}}({{params}}){}"
}
```

Primer 4.2.10

```
from homotopy import Homotopy

cpp_snippets = Homotopy("c++")
print(cpp_snippets.compile('for#int$i%5>printf("hello");'))
```

Izlaz:

```
for(int i=0; i<5; i++){
  printf("hello");
}
```

```
C:\dev\homotopy>homotopy -h
usage: homotopy [-h] [-t N] [-c] [-p PATH] language snippet

Compile a snippet.

positional arguments:
  language              Language for the snippet to be compiled to
  snippet               A snippet to be compiled

optional arguments:
  -h, --help            show this help message and exit
  -t N, --tabsize N     Number of spaces in one tab. Tabs remain
                        tabs if absent
  -c, --cursor          Indicate cursor marker in compiled snippet
  -p PATH, --path PATH  Path to snippet library folders separated
                        by ::
```

Slika 4.3: Upustvo za korišćenje alata komandne linije homotopy

Glava 5

Dodaci za editore teksta

Predviđen način korišćenja alata Homotopy je unutar editora. Kako bi se omogućila jednostavnija podrška za različite programe prevodilac za jezik Homotopy nalazi se u Python paketu koji se može koristiti kao alat komandne linije od strane različitih dodataka za različite editore. U ovoj sekciji biće reči o funkcionalnostima koje prevodilac pruža kako bi se jednostavnije integrisao sa drugim programima kao i o stvarima koje treba imati na umu kako bi se omogućilo udobno korišćenje. Nakon toga biće reči o konkretnoj implementaciji dodatka za progam Atom.

5.1 Kako napraviti dodatke

Homotopy paket sadrži program pod istim nazivom, `homotopy`, koji se koristi za prevođenje fragmenata. Uzima nekoliko parametara i ispisuje rezultujući fragment na standardni izlaz. Poruke o upozorenjima ili greškama se ispisuju na standardni izlaz za grešku.

Na visokom nivou, rad dodatka treba da bude sledeći:

1. Dodatak određuje argumente na osnovu trenunog stanja teksta u editoru.
2. Dodatak poziva prevodilac `homotopy`.
3. Dodatak zamenjuje tekst fragmenta u editoru sa rezultatom dobijenim od prevodioca `homotopy`.

5.1.1 Parametri komandne linije

Dodatak poziva prevodilac `homotopy` kao program komandne linije i prilikom poziva prosleđuje mu argumente. Program komandne linije `homotopy` prihvata dva obavezna poziciona parametra. To su jezik na koji se fragment prevodi i vrednost samog fragmenta (ovo se može videti na primeru 5.1.1). Pored toga, postoji nekoliko opcionih imenovanih parametara.

Primer 5.1.1

```
homotopy c++ if$true
```

```
if(true){

}
```

Ako se navede opcija `-c` izlazni tekst sadržaće označeno mesto na kome treba postaviti kursor. Primer 5.1.2 ilustruje ovu funkcionalnost.

Primer 5.1.2

```
homotopy -c c++ "if$true>"
```

```
if(true){
    [{cursor_marker}]
}
```

Dodatne putanje na kojima se nalaze definicije fragmenata navode se pomoću argumenta `-p` (primer 5.1.3). Ako se navodi više putanja, one treba da budu razdvojene sa `::`.

Primer 5.1.3

```
homotopy -p folder1::folder2 c++ if$true
```

```
if(true){

}
```

Homotopy interno koristi karakter `tab` kako bi označio nazubljivanje. Česta praksa je da se koriste obične beline pri nazubljivanju. Kako bi se omogućio nesmetan rad u takvom okruženju `homotopy` prihvata argument `-t` kojim se navodi broj belina koji odgovara jednom `tab` znaku (primer 5.1.4).

Primer 5.1.4

```
homotopy -t 4 c++ if$true
```

```
if(true){

}
```

5.1.2 Pozicija kursora

Kako bi se rad korisnika učinio prirodnijim, `homotopy` može da označi mesto na koje treba postaviti kursor. Dodatak treba da zameni tekst koji označava poziciju kursora sa stvarnim kursorom. Ako, iz bilo kog razloga, oznaka za kursor ne postoji u rezultatu, kursor treba postaviti posle kraja razvijenog teksta. Ako je to moguće, treba učiniti da cela operacija razvijanja fragmenta bude atomična u smislu *undo/redo* logike.

5.1.3 Podešavanja

Svaki dodatak trebalo bi da podržava dodavanje liste putanja do dodatnih definicija fragmenata. Ovako se korisniku omogućava da pored standardne biblioteke koristi i sopstvenu biblioteku sa fragmentima.

Pored toga, poželjno je i omogućiti podešavanje putanje do izvršne datoteke `homotopy` kako bi se omogućio rad sistema i u slučaju da se ova datoteka ne nalazi u promenljivoj `PATH`.

5.1.4 Obrada grešaka

Ukoliko dođe do greške pri pokretanju alata, potrebno je ispisati korisniku odgovarajuću grešku. Takođe, ukoliko na standardnom izlazu za greške postoji neka greška, treba je prikazati korisniku kako bi mogao da reaguje na odgovarajući način.

5.2 Dodatak za Atom

U ovoj sekciji biće reči o konkretnom dodatku za program Atom. Atom paket Homotopy implementira funkcionalnosti opisane u prethodnoj sekciji i omogućava korisnicima Atom-a korišćenje alata Homotopy.

5.2.1 Instalacija i korišćenje

Atom paket Homotopy[1] može se naći u zvaničnoj biblioteci Atom paketa. Instalacija je moguća preko grafičkog korisničkog okruženja unutar Atom-a kao i putem komandne linije[2]. Nakon instalacije paket registruje svoje komande i prečice. Homotopy paket dodeljuje akciju za komandu `homotopy:expand` koja se može pozvati od strane korisnika prečicom na tastaturi, iz padajućeg menija ili iz liste svih dostupnih komandi. Isečak kôda 10 prikazuje kôd u kome se registruje komanda `homotopy:expand`.

[1] Dostupan besplatno na `https://atom.io/packages/homotopy`
[2] apm install homotopy

```
this.subscriptions.add(atom.commands.add('atom-workspace', {
  'homotopy:expand': () => {
    let editor
    if (editor = atom.workspace.getActiveTextEditor()) {
      new SnippetExpansion(editor).expand()
    }
  }
}));
```

Isečak kôda 10: Registrovenje Atom komande od strane paketa Homotopy

5.2.2 Argumenti komandne linije

Nakon pozivanja komande za razvijanje fragmenta koju definiše dodatak Homotopy određuju se argumenti koje je potrebno proslediti konzolnom alatu Homotopy. Ovo uključuje određivanje jezika koji se trenutno koristi, teksta koji je potrebno razviti kao i dodatnih argumenata. Ukoliko je Atom podešen da koristi obične beline umesto `tab` znakova prosleđuje se veličina `tab` znaka u blanko znakovima. Takođe se prosleđuje i lista putanja, koje korisnik može da navede, do datoteka sa dodatnim fragmentima kao i oznaka da u rezltat treba uključiti oznaku za kursor. Ovo se može videti u isečku kôda 11.

5.2.3 Pozivanje alata komandne linije

Atom je napisan u Node.js okruženju. Za pozivanje alata komandne linije koristi se funkcija `execFile` koja se nalazi u standatdnoj biblioteci Node.js-a (isečak kôda 12). Nakon pozivanja ove funkcije, standardni izlaz i standardna greška nalaze se u string promenljivama i koriste se za razvijanje fragmenta.

5.2.4 Razvijanje fragmenta

Prilikom razvijanja fragmenta treba imati u vidu da se u rezultatu nalazi marker koji označava poziciju kursora. Rezultat se zbog toga razdvaja na dva dela, na deo pre i deo posle markera. Zatim se tekst fragmenta zamenjuje prvim delom rezultata tako da bude pre kursora. Nakon toga se dodaje drugi deo rezltata tako da bude posle kursora. Ovim postupkom se izvršava više promena nad tekstom tekuće datoteke i to se manifestuje kao više koraka u undo/redo logici Atom programa. Kako bi se ovo izbeglo, Atom omogućava grupisanje promena u transakcije koje se vrše nad tekstom, što se može videti u isečku kôda 13.

```
prepareArguments(language_name, snippet_text){
  var args = [];

  let userPath = atom.config.get('homotopy.User lib path')
  if(userPath.length){
    args.push("-p");
    args.push(userPath.join('::'))
  }

  if(this.editor.getSoftTabs()){
    args.push("-t");
    args.push(this.editor.getTabLength())
  }

  args.push("-c");
  args.push(language_name);
  args.push(snippet_text);

  return args;
}
```

Isečak kôda 11: Pripremanje argumanata

```
this.execFile(
  this.homotopy_command,
  this.prepareArguments(language_name, snippet_text),
  (error, stdout, stderr) =>{
    // Obrada rezultata...
  })
```

Isečak kôda 12: Pozivanje prevodioca pomoću funkcije execFile

```
this.editor.transact(()=>{
  this.editor.setTextInBufferRange(range, snippet_split[0])
  var cursorPosition = this.editor.getCursorBufferPosition()
  this.editor.setTextInBufferRange(
    [cursorPosition, cursorPosition],
    snippet_split[1])
  this.editor.setCursorBufferPosition(cursorPosition)
})
```

Isečak kôda 13: Razvijanje fragmenta

Glava 6

Zaključci i dalji rad

U ovom radu opisano je korišćenje alata koji pomažu pri kodiranju. Dat je pregled postojećih rešenja ako što su sistemi za automatsko kompletiranje koda, skraćenice, sistemi za automatsko formatiranje koda, sistemi za refaktorisanje koda i linteri. Dat je i pregled softverskih okruženja u kojima se ta rešenja koriste. Predloženo je novo rešenje koje omogućava programeru da se fokusira na ideju koju implementira, dok je briga o urednosti kôda prebačena na softver. Dat je jezik Homotopy, implementacija prevodioca jezika Homotopy kao i dodatak za program Atom kojim programer interaguje sa alatom. Kako bi se funkcionalnosti alata Homotopy omogućile što većoj publici, alat je dizajniran na modularan način koji omogućava lako dodavanje novih komponenti. Nalik alatu `OmniSharp` (koji je opisan u glavi 2), alat homotopy izdvaja deo funkcionalnosti (prevodilac) u poseban projekat koji je moguće koristiti u različitim editorima. Dodatno, izlazni jezik prevodioca Homotopy jako je fleksibilan i može se definisati i od strane korisnika dodavanjem novih definicija fragmnata. Ovo omogućava dodavanje jezika koji nisu podržani u standardnoj biblioteci.

Rad na kôdu je vrlo kompleksan pa su i alati koji pomažu u tome vrlo složeni i isprepletani. U ovom radu se ne govori o vezi između različitih alata. Na primer, prilikom unosa fragmenta ne dobijaju se sugestije za narednu reč kao što je slučaj u mnogim okruženjima. Ovakva integracija je moguća i zahtevala bi izmene i na sistemima za sugestije. Različiti projekti u istom jeziku prate različite konvencije. Alati kao što su linteri podržavaju podešavanje ovih konvencija (na primer, otvorena zagrada za početak novog bloka se nakada stavlja u poseban red, a nekada ne). Homotopy trenutno ne podržava više od jedne konvencije po jeziku. Dodavanje ove funkcionalnosti ne bi obavezno dovelo do promene jezgra prevodioca jer se izlazni jezik definiše definicijama fragmenata koje mogu pratiti različite konvencije.

Dodavanjem podrške za nove jezike i editore, kao i bogatije biblioteke fragmenata bi svakako učinile ovaj alat fleksibilnijim za korišćenje. Fragmenti u standardnoj biblioteci fokusirani su na opšte konstrukcije u programskim jezicima. Dodavanjem fragmenata koji su specifični za uže domene ili određene biblioteke omogućio bi kreiranje još kompleksnijih konstrukcija relativno krat-

kim zapisom. U ovom radu opisan je rad dodatka za editor `Atom`. Na ličan način moguće je kreirati dodatak za neki drugi editor i na raj način doprineti široj primenljivosti alata Homotopy.

Jedna od najvećih prepreka za korišćenje jezika Homotopy je (kao i kod većine programskih jezika) dug period prilagođavanja korisnika na funkcionalnosti jezika. Zbog toga su resursi za učenje jezika koji uvode jezičke konstrukcije postepeno vrlo bitni za korišćenje celog alata. Homotopy dokumentacija (`https://homotopy.readthedocs.io`) sarži, između ostalog, materijale za upoznavanje sa funkcionalnostima jezika Homotopy.

Bibliografija

[1] Martin Fowler (with Kent Beck, John Brant, William Opdyke, and Don Roberts) *Refactoring - Improving the Design of Existing Code*

[2] S. C. Johnson *Lint, a C Program Checker*

[3] Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm *Design Patterns: Elements of Reusable Object-Oriented Software*

[4] Jez Humble, David Farley *Continuous Delivery - Reliable Software Releases through Build, Test, and Deployment Automation*

[5] *Atom Flight Manual* `https://flight-manual.atom.io/`

[6] *Python documentation* `https://docs.python.org`

Dodatak A

Primeri konstrukcija u alatu Homotopy

A.1 Bazične skraćenice

Primer 1 block

```
block>line1&line2
```

```
line1
line2
```

Primer 2 wblock (širok blok)

```
wblock>line1&line2
```

```
line1

line2
```

A.2 Kontrola toka

Primer 3 for

```
for#int$i%0%n
```

```
for(int i=0; i<n; i++){

}
```

Primer 4 forr

```
forr#int$i%n%0
```

```
for(int i=n; i>=0; i--){

}
```

Primer 5 forin

```
forin#int$i%array
```

```
for(int i: array){

}
```

Primer 6 if

```
if$true>printf("Always");
```

```
if(true){
    printf("Always");
}
```

Primer 7 else

```
if$i==2>return 4;<else>return 3;
```

```
if(i==2){
    return 4;
}
else {
    return 3;
}
```

Primer 8 while

```
while$true>printf("Forever and always");
```

```
while(true){
    printf("Forever and always");
}
```

Primer 9 switch

```
switch$i>case$1>printf("one");<case$2>printf("two");
```

```
switch(i){
    case 1:
        printf("one");
        break;

    case 2:
        printf("two");
        break;
}
```

Primer 10 switch (sa propadanjem)

```
switch$i>case$1$2>printf("one or two");
```

```
switch(i){
    case 1:
    case 2:
        printf("one or two");
        break;
}
```

A.3 Objekti

Primer 11 struct

```
struct!pair>int first, second;
```

```
struct pair {
    int first, second;
};
```

Primer 12 class

```
class!A:B%public>private>int a;<public>int b;
```

```
class A: public B {
private:
    int a;
public:
    int b;
};
```

Primer 13 enum

```
enum!Colors>red&green&blue
```

```
enum Colors {
    red,
    green,
    blue
};
```

Primer 14 enum1 (enum u jednoj liniji)

```
enum1!Colors>red&green&blue
```

```
enum Colors { red, green, blue };
```

A.4 Funkcije

Primer 15 func

```
func#int@five>return 5;
```

```
int five(){
    return 5;
}
```

Primer 16 func (sa više parametara)

```
func#int@plus#int$i#int$j>return i+j;
```

```
int plus(int i, int j){
    return i+j;
}
```

Primer 17 func (bez ponovnog navođenja int tipa)

```
func#int@plus$i$j#int>return i+j;
```

```
int plus(int i, int j){
    return i+j;
}
```

A.5 Metodi

Primer 18 method

```
class!A>public>method#int@five>return 5;
```

```
class A {
public:
    int five(){
        return 5;
    }
};
```

Za različite vrste metoda u C++-u postoje različiti fragmenti.

Fragment	Opis
method	običan metod
nimethod	neimplementiran metod
amethod	abstraktan metod
dmethod	obrisan metod

Primer 19 methodi1 (implementacija jednog metoda)

```
methodi1!A#int@five>return 5;
```

```
int A::five(){
    return 5;
}
```

Primer 20 methodi (implementacija metoda)

```
wblock!A>methodi#int@five>return 5;<methodi#int@six>return 6;
```

```
int A::five(){
    return 5;
}

int A::six(){
    return 6;
}
```

Primetimo da se u predhodnom primeru koristi fragment `wblock` kako bi se navelo ime klase.

Primer 21 constr (konstruktor)

```
class!A>public>constr#int$i
```

```
class A {
public:
    A(int i){

    }
};
```

A.6 Šabloni

Primer 22 class (sa šablonom)

```
class!A^T
```

```
template <class T>
class A {

};
```

Primer 23 func (sa šablonom)

```
func@nothing#void^T
```

```
template <class T>
void nothing(){

}
```

Primer 24 method (sa šablonom)

```
class!A>public>method#void@nothing^T
```

```
class A {
public:
 template <class T>
 void nothing(){

 }
};
```

A.7 Obrasci

Primer 25 singleton

```
class!A>[[singleton]]
```

```
class A {
public:
    A& getInstance(){
        static A instance;

        return instance;
    }
private:
    A(){}
    A(A const& origin);
    void operator=(A const& origin);
};
```

Primer 26 composite

```
class!Composite:Component%public>[[compositeclass]]
   &public>method#void@traverse>[[compositemethod]]
```

```
class Composite: public Component {
public:
    void add(Component *item){
        children.puch_back(item);
    }
private:
    std::vector<Component*> children;
public:
    void traverse(){
        for(int i=0; i<children.size(); i++){
            children[i]->traverse();
        }
    }
};
```

A.8 Komande

Primer 27 call (poziv funkcije)

```
call@foo$param1>param2
```

```
foo(param1, param2);
```

Za parametre je moguće koristiti i $ i >. Na ovaj način se omogućava prosleđivanje parametara koji su takođe fragmenti.

Primer 28 call2 (poziv funkcije u više redova)

```
call2@foo$param1>param2
```

```
foo(
  param1,
  param2
);
```

Primer 29 stdinc

```
stdinc$stdio.h
```

```
#include <stdio.h>
```

Primer 30 inc

```
inc$homotopy.h
```

```
#include "homotopy.h"
```

A.9 JSON

Primer 31 dict (rečnik)

```
dict>key$item1$1&key$item2$2
```

```
{
  item1: 1,
  item2: 2
}
```

Primer 32 d (rečnik, kraća verzija):

```
d>k$item1$1&k$item2$2
```

```
{
  "item1": 1,
  "item2": 2
}
```

Primer 33 d (rečnik, ugnježdavanje):

```
d>k$item1>d>k$nested$1<<&k$item2$2
```

```
{
  "item1": {
    "nested": 1
  },
  "item2": 2
}
```

Primer 34 dict1 (rečnik u jednom redu):

```
dict1>key$"item1"$1&k$item2$2
```

```
{"item1": 1, "item2": 2}
```

Primetimo da se u svim predhodnim primerima koristi fragment `key` za doslovne vrednosti ključeva i `k` za ključeve u kojima je vrednost ključa string.

Primer 35 array

```
array>item1&item2
```

```
[
  item1,
  item2
]
```

Primer 36 array1 (niz u jednom redu)

```
array1>item1&item2
```

```
[item1, item2]
```

A.10 Anonimne funkcije

Primeri anonimnih funkcija dati su u jeziku JavaScript.

Primer 37 func

```
func$i$j>return i+j;
```

```
function (i, j){
  return i+j;
}
```

Primer 38 func1 (funkcija u jednom redu)

```
func1$i$j>return i+j;
```

```
function (i, j){ return i+j; }
```

Primer 39 f (funkcija sa jednom komandom)

```
f$i$j>i+j
```

```
function (i, j){ return i+j; }
```

Primer 40 arrow (funkcija strela)

```
arrow$i$j>return i+j;
```

```
(i, j) => {
  return i+j;
}
```

Primer 41 arrow1 (funkcija strela u jednom redu)

```
arrow1$i$j>return i+j;
```

```
(i, j) => { return i+j; }
```

Primer 42 a (funkcija strela sa jednom komandom)

```
a$i$j>i+j
```

```
(i, j) => i+j
```

Primer 43 call2 (poziv funkcije sa funkcijom kao parametrom)

```
call2@foo$param1>f>3
```

```
foo(
  param1,
  function (){ return 3; }
);
```