Универзитет у Београду
Математички факултет

Никола Нинков

Унапређење подсистема за тестирање јединица кода програмског језика *Wafl* и тестирање основне библиотеке

Мастер рад

Београд
2021.

Универзитет у Београду – Математички факултет
Мастер рад

Аутор:	Никола Нинков
Наслов:	Унапређење подсистема за тестирање јединица кода програмског језика *Wafl* и тестирање основне библиотеке
Ментор:	проф. др Саша Малков
Чланови комисије:	проф. др Милена Вујошевић-Јаничић доц. др Иван Чукић
Датум:	28.9.2021.

Садржај

1. Увод 5
1.1. Преглед историје функционалних програмских језика 5
1.2. Програмски језик *Wafl* 8
1.3. Тестирање софтвера 8
1.3.1. Контрола квалитета 8
1.3.2. Основни појмови 9
1.3.3. Значај тестирања јединица кода 10
1.4. Циљ 11
1.5. Мотивација 11
2. Тестирање основне библиотеке функција програмског језика *Wafl* 12
2.1. Организација тестирања 12
2.2. Синтаксни литерали 14
2.3. Приступ члановима колекције по индексу 16
2.4. Намерно неуспешни тестови 18
2.5. Листе и низови 20
2.6. Функције вишег реда 21
2.7. Ниске 23
2.8. Регуларни изрази 24
2.9. Рад са фајл-системом 25
2.10. Конверзија типова 27
2.11. Функције за рад са целобројним типом 28
2.12. Функције за рад са реалним типом 28
2.13. Оператори 29
2.14. Мапе 30
2.15. Остале функције 30
2.16. Преглед резултата тестирања 31
3. Представљање извештаја о обављеном тестирању у формату *HTML* 32
3.1. Текстуални извештај о комплетном тестирању 32
3.2. Извештај о појединачном тестирању 34
3.3. Извештај *test.html* 35
3.4. Имплементација *HTML*-извештаја 38

3.4.1. Изглед извештаја 38
3.4.2. Садржај извештаја 39
4. Закључак 43
Литература 44

1. Увод

1.1. Преглед историје функционалних програмских језика[1]

Алонзо Черч[2] је током тридесетих година прошлог века представио ламбда рачун, формални систем у математичкој логици, помоћу којег се рачунање представља функцијама. У својој тези, Черч је изнео тврдњу да су функције над природним бројевима израчунљиве само ако се могу дефинисати ламбда рачуном. Његов ученик, Стивен Клини[3], је 1936. године доказао да је ламбда-дефинабилност функције еквивалентна рекурзивности у Геделовом[4] и Ербрановом[5] смислу[6]. За то време, Алан Тјуринг[7] је развијао данас широко познату Тјурингову машину, а 1937. године доказао да је израчунљивост на Тјуринговој машини еквивалентна ламбда-дефинабилности. Ови резултати су дали огроман допринос развоју рачунара, а ламбда рачун се често сматра првим функционалним програмским језиком. Мојсеј Шонфинкел[8] је 1924. показао да се функције са два или више аргумената могу заменити еквивалентним функцијама са једним аргументом. Роберт Фејс[9] и Хаскел Кари[10] су 1958. године, ослањајући се на Шонфинкелов рад, представили запис *(f x y)* као еквивалент за *((f x) y)* и замену за, до тада коришћен, *f(x,y)*. Тако је настао појам Каријева функција, једна од истакнутих синтаксних особина данашњих функционалних језика.

Џон Макарти[11] је током касних педесетих година радио на развоју програмског језика који би служио за истраживање на пољу вештачке интелигенције. Тај језик је назван *LISP*, а његова синтакса за представљање функција је концептуално слична ламбда рачуну. Рекурзија се изражава употребом условних израза, које је Макарти измислио, а који се и данас користе у многим програмским језицима. Основни тип податка над којим се врше операције је листа, а сакупљање отпадака, техника управљања некоришћеном меморијом, је још један Макартијев изум. Ови концепти су фундаментални за имплементацију функционалних програмских језика.

Средином шездесетих година, Питер Лендин[12] је представио *ISWIM*, осмишљен као унапређење *LISP*-а. Карактеришу га синтаксне новине – прелазак на инфиксни

---

[1] Изводи из историје функционалних програмских језика до 1989. године су преузети из [1].
[2] *Alonzo Church* (1903-1995), амерички математичар и логичар.
[3] *Stephen Cole Kleene* (1909-1994), амерички математичар.
[4] *Kurt Friedrich Gödel* (1906-1978), логичар, математичар и филозоф.
[5] *Jacques Herbrand* (1908-1931), француски математичар.
[6] То је функција над природним бројевима која је „израчунљива“ у интуитивном смислу.
[7] *Alan Mathison Turing* (1912-1954), енглески математичар, логичар, криптограф и информатичар.
[8] *Moisei Isai'evich Sheinfinkel'* (1888-1942), руски логичар и математичар.
[9] *Robert Feys* (1889-1961), белгијски логичар и филозоф.
[10] *Haskell Brooks Curry* (1900-1982), амерички математичар и логичар.
[11] *John McCarthy* (1927-2011), амерички информатичар.
[12] *Peter John Landin* (1930-2009), британски информатичар.

запис функција, *let* и *where* изрази, као и коришћење увлачења програмског кода уместо симбола који раздвајају наредбе.

Кенет Ајверсон[13] је 1962. представио *APL*, који је, упркос томе што није чисто функционалан, утицао на развој *FP*-а, који је направио Џон Бекус[14]. Бекусово излагање поводом добијања Тјурингове награде 1978. године покренуло је велика истраживања и улагања у функционални стил програмирања. Такође током седамдесетих, док је Бекус развијао *FP* у америчкој компанији *IBM*, развијао се и *ML*, као резултат неколико различитих пројеката у Великој Британији. Касније, током осамдесетих, направљена је и његова надоградња *SML*, језик који се истиче својим системом типова. По први пут се јавио строго и статички типизиран језик који допушта полиморфизам, а истовремено користи и закључивање типова (енг. *type inference*).

За време настајања *FP*-а и *ML*-а, Дејвид Тарнер[15] је развијао нове функционалне језике, желећи да синтаксу функционалних језика учини привлачнијом за програмере (енг. *syntactic sugaring*). Тако је 1976. настао *SASL*, 1981. *KRC*, а 1985. *Miranda*. Гардови (енг. *guards*) и Каријеве функције красе синтаксу *SASL*-а. *KRC* додаје ЗФ-изразе, познате данас као препознавање листи (енг. *list comprehension*) и разне друге записе за листе формата од-до (нпр. [a..b]). *Miranda* се истиче по својој строгој типизираности и по томе што подржава корисничке конкретне и апстрактне типове података. Тарнер је био један од гласноговорника лењег израчунавања и функција вишег реда, који су свеприсутни данас.

Касних седамдесетих и раних осамдесетих година, појавило се много нових функционалних језика. *Val* (касније и *SISAL*), *Id*, *FGL*, *DDN* – тзв. *dataflow* језици, као и *Hope*, *FEL*, *Lazy ML*, *ALFL* и други.

Џо Армстронг, Роберт Вирдинг и Мајк Вилијамс су 1986. године направили прву верзију програмског језика *Erlang* [9], за компанију *Ericsson*. Дванаест година касније, 1998. године, *Erlang* постаје софтвер отвореног кода. Карактеристике овог језика, намењеног развоју робусног дистрибуираног софтвера, описао је један од твораца, Армстронг, у својој докторској тези [10]. Неке од најважнијих су:

- Све је процес
- Процеси су изоловани
- Прављење и уништавање процеса не троши пуно ресурса
- Поруке су једине додирне тачке процеса, не деле ресурсе
- Процеси имају јединствена имена
- Ако је познато име процеса, може му се послати порука
- Процеси се извршавају конкурентно
- Грешке морају бити идентификоване и енкапсулиране
- Код се мења у току самог извршавања, како се систем не би заустављао
- Подаци се чувају тако да не буду изгубљени у случају пада система

[13] *Kenneth Eugene Iverson* (1920-2004), канадски информатичар.
[14] *John Warner Backus* (1924-2007), амерички информатичар.
[15] *David A. Turner* (1946), британски информатичар.

Прва званична (1.0) верзија језика *Haskell* објављена је 1990. године. Овај чисто функционални језик опште примене красе многе од наведених иновација – функције вишег реда, лење израчунавање, статична типизираност са подршком за полиморфизам, кориснички типови, подударање шаблона (енг. *pattern matching*) и препознавање листи.

Мартин Одерски је аутор програмског језика *Scala* [11], чија је прва верзија објављена 2004. године. Ово је статички типизиран језик опште намене, са системом закључивања типова. Карактерише га интероперабилност са програмским језиком *Java*, што омогућава корисницима да користе *Java* библиотеке у развоју.

Програмски језик *F#* („еф шарп“) [12] је објављен 2005. године, а 2010. постао софтвер отвореног кода. Дизајнирао га је Дон Сајм, истраживач у компанији *Microsoft Research*. Имплементација је заснована на програмском језику *OCaml*, а језик је компатибилан са осталим *.NET* језицима и подржан на свим масовно коришћеним оперативним системима. Овај језик одликује сведена синтакса, једноставна паралелизација, подршка за асинхроно програмирање и *just-in-time* превођење.

Прва верзија програмског језика *Clojure* [13] објављена је 2007. године. Овај језик је, према речима аутора, Рича Хикија, функционални дијалект програмског језика *Lisp*, на *Java* платформи. Развој језика подржава компанија *Cognitect*, као и његови активни корисници.

Програмски језик *Kotlin* [14] развила је компанија *JetBrains* и објавила његову прву верзију 2011. године. Годину дана касније, постао је софтвер отвореног кода, а 2016. је објављена прва стабилна верзија, 1.0. Ово је статички типизиран језик са закључивањем типова. Осмишљен је тако да се ослања на *Java* виртуелну машину, а касније верзије омогућиле су директно превођење у *JavaScript*. *Kotlin* је један од најпопуларнијих језика за развој апликација на *Android* платформи.

Хозе Валим је направио језик *Elixir* [15], објављен 2012. године, са циљем да буде погодан за писање поузданог софтвера који се лако одржава. Језик се због тога ослања на *Erlang* виртуелну машину. Подржано је конкурентно програмирање помоћу изолованих процеса који не користе пуно ресурса, док супервизори обезбеђују опоравак од грешке враћањем у познато, исправно стање.

Програмски језик *Swift* [16] објављен је 2014. и годину дана касније, 2015. постао софтвер отвореног кода. Ово је програмски језик опште намене, направљен са циљем да омогући писање безбедног софтвера високих перформанси. Користи се за развој софтвера на оперативним системима компаније *Apple*.

Међу популарним савременим функционалним језицима (према [17]) су и *Elm*, *Racket*, *Rust*, *PureScript* и *ReasonML*.

1.2. Програмски језик *Wafl*

Саша Малков је 2002. године, у Београду, на Математичком факултету, као свој магистарски рад представио *Wafl*, модеран функционални програмски језик намењен за развој веб апликација. Према спецификацији [18], *Wafl* је строго типизиран, вредан, функционалан и објектно-оријентисан програмски језик, са имплицитним старањем о меморији и нестриктном семантиком. Строгу типизираност обезбеђује аутоматско закључивање типова приликом превођења, без њиховог експлицитног навођења у коду. Операције над меморијом се изводе имплицитно – при прављењу објеката, меморија се аутоматски алоцира и аутоматски ослобађа када објекат постане сувишан. Нестриктна семантика подразумева да редослед израчунавања углавном није гарантован.

Програм на језику *Wafl* се дефинише једним изразом, чији је резултат истовремено и резултат извршавања програма. Сви програми који ће бити разматрани у оквиру овог рада дефинисани су *where* изразима, који су облика:

```
<izraz_where> ::= <izraz> [<where_podizraz>]
<where_podizraz> ::= where { <definicija> {<definicija>} }
```

При томе су дефиниције облика:

```
<definicija> ::= <ime> [<formalni_parametri>] = <telo_definicije>;
<formalni_parametri> ::= (<ime> {,<ime>} )
```

У наредном поглављу, особине језика ће бити представљене уз одговарајућу групу тестираних функција. Тестирањем синтаксних литерала биће обухваћени подржани типови типови података. Тестирање понашања функција за рад са колекцијским типовима – листама, низовима и нискама карактера – демонстрираће њихову употребу. Прости типови података, као и аритметички, Булови, логички и оператори поређења такође су део језика и тестирани су.

У програмском језику *Wafl*, функције су „грађани првог реда“, што значи да аргумент или резултат израчунавања функције може бити функцијског типа. Када је то случај, таква функција назива се функцијом вишег реда. Ова група функција основне библиотеке посебно је дискутована. Подржан је рад са регуларним изразима и фајл-системом.

1.3. Тестирање софтвера

1.3.1. Контрола квалитета

У цивилизацији која се технолошки развија страховитом брзином, потребе људи за новим, бољим, чудеснијим алатима и играчкама расту утолико брже. Контрола квалитета је, због тога, све важнија у свакој области. Упркос томе, чести су примери из света технологија где нови производи бивају пуштени на тржиште у стању које

захтева од њихових корисника да их сами тестирају. Софтвер је, наравно, најчешће погођен.

Тестирање је један од начина вршења контроле квалитета софтвера. Предмет овог рада, тестирање јединица кода, је један вид тестирања софтвера.

1.3.2. Основни појмови

У наставку је преглед појмова кључних за поставку проблема и његово решење. Дефиниције коришћене у овом раду преузете су из [3] и [4].

- Грешка (енг. *error*) је људски поступак који доводи до нежељених резултата односно пропуст који је начинио програмер приликом писања кода. На пример, при имплементацији одређене библиотечке функције, није размотрен неки од могућих улаза, попут *null* вредности;
- Квар (енг. *fault*) је резултат грешке, мана или недостатак у коначном производу, односно насталом програму. Уколико има грешака у некој функцији, то доводи до квара на резултујућем програму;
- Неуспех (енг. *failure*) је манифестација односно симптом грешке, незадовољавајуће понашање програма узроковано извршавањем кода који је кваран. Примери су „пуцање“ програма и израчунавање неисправног резултата;
- Инцидент (енг. *incident*) је једна манифестација неуспеха, документована извештајем;
- Извештај о инциденту (енг. *incident report*) је скуп информација о инциденту који се десио. Извештај о тестирању је један пример извештаја о инциденту, или инцидентима, ако их има више;
- Тест (енг. *test*) је скуп једног или више тест-случајева;
- Тестирање (енг. *testing*) је чин извршавања тестова;
- Тест-случај (енг. *test case*) је скуп предуслова, улаза, додатних акција, излаза и постуслова, настао на основу услова тестирања, за сврху тестирања. Када се ради о тестирању јединица кода, сваки запис позивања функције за проверу тачности представља један тест-случај;
- Функција за проверу тачности (енг. *assert function*, колоквијално само *assert*) је функција која враћа или тачно или нетачно. Користи се током тестирања за упоређивање добијеног и очекиваног резултата;
- Тестирање јединица кода (енг. *unit testing*) је вид тестирања заснован на тестирању појединачних целина софтвера. Пошто је предмет овог рада тестирање стандардне библиотеке и самог интерпретатора, појединачне јединице које се тестирају су појединачне функције стандардне библиотеке и појединачне синтаксне конструкције програмског језика;
- Колекција тестова (енг. *test suite*) је скуп тест-случајева које се извршавају при тестирању;

- Тестирање засновано на спецификацији (енг. *specification-based testing*) или функционално тестирање (енг. *functional testing*) је тестирање ради провере да ли софтвер задовољава функционалне захтеве. У контексту тестирања библиотечких функција, циљ функционалног тестирања је проверити да се свака функција понаша у складу са спецификацијом. Предмет тестирања посматра се као црна кутија, јер ономе ко спроводи тестирање није позната његова имплементација. Насупрот функционалном тестирању, постоји и тестирање засновано на коду (енг. *code-based testing*), апстраховано белом или прозирном кутијом, јер су детаљи имплементације познати ономе ко спроводи тестирање. Како је тема рада тестирање основне библиотеке функција програмског језика *Wafl*, тестирање је засновано искључиво на спецификацији језика.

Извршавање тест-случајева коришћењем алата за тестирање може довести до инцидената. Резултат извршавања тестова је текстуални извештај о тестирању у којем се, у случају инцидента, налазе информације о томе. Извештај садржи назив функције, улазне вредности, очекивану излазну вредност и добијену излазну вредност, ако она постоји. У супротном, извештај садржи само назив функције уз напомену да је дошло до неуспеха. Након уочених инцидената, уз претпоставку да су тестови ваљани, уобичајен закључак је да је дошло до грешке у имплементацији.

1.3.3. Значај тестирања јединица кода

У својој књизи „Чист код“ [5], Роберт Мартин[16] је написао следеће: „Тестови јединица кода чине код флексибилним, лаким за одржавање и поново употребљивим. Ако имате тестове, не плашите се да промените код“. У наставку тог текста, изречена је једна сурова истина програмерског заната, а то је да без доброг и свеобухватног скупа тестова увек постоји страх од промене. Свака промена, без тестова који покривају код, може довести до неочекиваних проблема. Тестови су лек за овај страх, а самим тим што омогућавају програмерима да рефакторишу код, отварају бројне могућности за побољшања. Непрекидно рефакторисање и усавршавање кода је управо оно што га чини и одржава „чистим“.

Роберт Мартин је такође навео и пет пожељних особина једног теста. Треба напоменути да се у његовој књизи говори о тестирању подразумевајући развој вођен тестовима, што овде није случај, али наведене смернице су добре и у општем случају. Исте су (углавном) поштоване при изради овог рада:

1. Брз – како би могао често да се извршава;
2. Независан – није пожељно да тестови зависе једни од других;
3. Поновљив – како би могао да се изврши било где и било када;
4. Има јасан резултат – или пролази или не пролази и
5. Писање теста не захтева пуно времена.

[16] *Robert Cecil Martin* (1952), амерички софтвер инжењер, познат и као „Ујка Боб“.

1.4. Циљ

Данас, у 2021. години, проф. др Саша Малков је ментор при изради овог мастер рада, чији је циљ унапређење тестирања јединица кода за *Wafl*. Прецизније, први задатак је израда исцрпне колекције тестова за проверу исправности интерпретатора и имплементације читаве основне библиотеке језика. Други задатак је побољшање постојећег система за тестирање јединица кода, како би приказ резултата тестирања био прегледнији.

Реализација радних задатака реализована је кроз интензивну комуникацију са ментором, уз коришћење добијених алата – интерпретатор језика *clwafl*, библиотеку функција за извршавање тестова и екстензије за рад са програмским језиком *Wafl* у текстуалном едитору *Notepad++*, а све то на оперативном систему *Windows 10*. Тестиране су функционалности језика по целинама, након чега је следила дискусија резултата. Посебно је занимљиво то што је у неким случајевима детаљно тестирање имало за последицу унапређење синтаксе језика, као и дефиниције и имплементације неких функција из библиотеке.

1.5. Мотивација

Разлог за одабир баш ове теме је лична жеља да за мастер рад изаберем област која је мање уобичајена. Такође, желео сам да искористим ретку прилику за учешће у развоју и подршци развоја једног програмског језика.

Израда овог мастер рада трајала је, са дужим прекидима, више од годину дана. Комуникација путем електронске поште послужила је као евиденција свега што је урађено на пољу тестирања као и свих измена интерпретатора и основне библиотеке програмског језика *Wafl*. Белешке о откривеним проблемима и њиховим решењима омогућиле су да овај рад обухвати разматрање читавог процеса тестирања, а не само дискусију крајњег резултата. Другим речима, документовање измена је добра пракса у развоју софтвера.

Захвалан сам на пруженој прилици, стрпљењу и указаном поверењу.

2. Тестирање основне библиотеке функција програмског језика *Wafl*

2.1. Организација тестирања

Целокупан процес тестирања подељен је у целине, према скуповима функција основне библиотеке које су биле тестиране. Колекција тестова, садржана у директоријуму *testCollection*, је структурирана у складу са тиме, при чему директоријуми унутар ње (*core_library*, *operators* и *syntax_contstructs*) раздвајају фајлове са тестовима на оне који се тичу основних библиотечких функција, оператора и синтаксних конструкција, редом.

У директоријуму *core_library* налазе се следећи фајлови са тестовима, са наведеним предметима тестирања:

- *array_functions_test.wafl*: функције за рад са низовима;
- *conditional_expressions_test.wafl*: условни изрази;
- *conversion_functions_test.wafl*: функције за конверзију типова;
- *debug_functions_test.wafl*: функције за дебаговање;
- *filesystem_functions_test.wafl*: функције за рад са фајл-системом;
- *float_functions_test.wafl*: функције над реалним типом;
- *int_functions_test.wafl*: функције над целобројним типом;
- *list_functions_test.wafl*: функције за рад са листама;
- *map_functions_test.wafl*: функције за рад са мапама;
- *mime_functions_test.wafl*: функције за рад са *MIME* ресурсима;
- *regex_functions_test.wafl*: функције за рад са регуларним изразима;
- *str_encode_test.wafl*: функције за кодирање ниски;
- *string_functions_test.wafl*: функције за рад са нискама.

Директоријум *operators* садржи:

- *arithmetic_operators_test.wafl*: аритметички оператори;
- *boolean_operators_test.wafl*: логички оператори;
- *comparison_operators_test.wafl*: оператори поређења;
- *integer_operators_test.wafl*: оператори над целобројним типом;
- *list_operators_test.wafl*: оператори над листама.

Садржај директоријума *syntax_constructs* је следећи:

- *get_element_test.wafl*: приступ елементима колекције;
- *intentional_error_test.wafl*: тест који намерно доводи до програмске грешке;
- *intentional_failure_test.wafl*: тест који намерно враћа погрешан резултат;
- *slice_prefix_test.wafl*, *slice_segment_test.wafl* и *slice_suffix_test.wafl*: оператори за одсецање;
- *syntax_literals_test.wafl*: записивање константи.

Извори информација о коришћењу функционалности језика су његова јавно доступна интернет презентација *Wafl tutorial* [6], као и документација доступна из командне линије следећим позивом интерпретатору:

```
clwafl -listlib[:<ime_ili_deo_imena_funkcije>] [-verbose]
```

Резултат овакве команде је преглед функција чија имена (или део имена) одговарају унетом улазном аргументу, а опциони параметар на крају додаје проширен опис. У случају позива без наведеног имена функције, тражене информације ће бити исписане за све функције стандардне библиотеке. Ова функционалност је била корисна при сагледавању обима тестирања основне библиотеке и праћења напретка у току рада.

Раније поменута библиотека за тестирање састоји се из два фајла, *TestRunner.wlib* и *Test.wlib*, који обезбеђују преко потребне функције за проверу тачности резултата. Уз њих је испоручен и *TestRunner.wafl*, скрипт који проласком кроз читаву колекцију покреће тестове, ослањајући се на претходно поменуте библиотеке за тестирање и при томе генерише текстуални извештај о тестирању, *waflTestReport.txt*.

У пракси, извршавање тестова подразумева следећи низ корака:

- Упознавање са типовима и функцијама које се тестирају, кроз доступну литературу и понекад писањем једноставних програма, извршаваним директно из командне линије позивима интерпретатору: `clwafl -code ”sadržaj_programa”`
- Писање тестова у виду *Wafl* програма, поштујући сва правила језика и ослањајући се на функције за проверу тачности резултата из библиотеке за тестирање; сваки тест-фајл има екстензију *.wafl* и организован је у секције (енг. *section*) које одговарају једној функцији или једном кораку сложенијег тест процеса
- Извршавање тестова коришћењем скрипта *TestRunner.wafl* или позивом из командне линије
- Проучавање резултата у *waflTestReport.txt*; у случају појаве инцидената, урађена је детаљна анализа тест-случаја, што је водило до разјашњења неспоразума око коришћења функција или до уклањања грешака у имплементацији

Током реализације мастер рада било је неколико ажурирања интерпретатора. Када се у раду разматра тренутак откривања проблема, подразумева се да се то десило са верзијом интерпретатора која је тада била коришћена, а када се дискутују резултати (да ли је нешто промењено као резултат тестирања), мисли се на понашање са последњом верзијом интерпретатора доступном у тренутку када је цео процес завршен.

Следи дискусија колекције тестова, у којој ће укратко бити представљена основна библиотека језика, део по део, редом како је тестирана. Посебно ће бити размотрени неки значајнији тест-случајеви, као и неисправности које су уочене и исправљене захваљујући спроведеном тестирању.

2.2. Синтаксни литерали

Тестирање записивања константи је размотрено на самом почетку, са циљем упознавања са типовима података које подржава програмски језик *Wafl*. Посебна пажња посвећена је тест-случајевима везаним за проверавање понашања целобројног и реалног типа на границама опсега вредности које се могу представити у оквиру постојећих ограничења. Садржај фајла *syntax_literals_test.wafl* је следећи:

```
[{
	sectionName: "int",
	tests: [
		wt::testEq( 0 , 0 ),
		wt::testEq( 0 , -0 ),
		wt::testEq( 1 , 1 ),
		wt::testEq( 9223231299366420480 , 9223231299366420480 ),
		wt::testEq( 09223231299366420480 , 9223231299366420480 ),
		wt::testEq( 0000000000092232312993664204800 , 9223231299366420480 ),
		wt::testFalse( 9223231299366420480 == -
0000000000000000000000000092232312993664204800 ),
		wt::testFalse( 0 == 1 ),
		wt::testFalse( -1 == 1 ),
		wt::testFalse( -01 == 1 ),
		wt::testFalse( -0000000000000000000000000000000001 == 1 )
	]
},{
	sectionName: "float",
	tests: [
		wt::testEq( 0.0 , 0.0 ),
		wt::testEq( 0.0 , 0.00000 ),
		wt::testEqEps( 0.0 , 0.00001 , 0.0001 ),
		wt::testFalse( 0.0 == 0.0000000000000000001 ),
		wt::testFalse( 0.0 == -0.0 ),
		wt::testEqEps( 0.0 , -0.0 , 0.000000000000000001 ),
		wt::testEq( 1.234567 , 1.234567 ),
		wt::testFalse( 1.234567 = -1.234567 ),
		wt::testFalse( 33.0 / 0.33 == 100.0 ),
		wt::testEqEps( 33.0 / 0.33 , 100.0 , 0.0000000000001 )
	]
},{
	sectionName: "string",
	tests: [
		wt::testEq( "" , "" ),
		wt::testEq( "aBc" , "aBc" ),
		wt::testFalse( "abc" == "Abc" ),
		wt::testFalse( "abcc" == "abc" )
	]
},{
	sectionName: "bool",
	tests: [
		wt::testEq( false , false ),
		wt::testEq( true , true ),
		wt::testFalse( false == true ),
		wt::testFalse( true == false )
	]
},{
	sectionName: "list",
	tests: [
```

```
                wt::test( [1,2,3] == [1,2,3] ),
                wt::test( [1,2,4] != [1,2,3] ),
                wt::test( [1,2,3] != [1,2,3,4] ),
                wt::test( [2,3,4] != [1,2,3,4] ),
                wt::test( [2,2,3,4] != [1,2,3,4] ),
        ]
},{
        sectionName: "array",
        tests: [
                wt::test( [#1,2,3#] == [#1,2,3#] ),
                wt::test( [#1,2,4#] != [#1,2,3#] ),
                wt::test( [#1,2,3#] != [#1,2,3,4#] ),
                wt::test( [#2,3,4#] != [#1,2,3,4#] ),
                wt::test( [#2,2,3,4#] != [#1,2,3,4#] )
        ]
},{
        sectionName: "tuple",
        tests: [
                wt::test( {#1, 2.0, 'a'#} == {#1, 2.0, 'a'#} ),
                wt::test( {#1, 2.0, 'a'#} != {#2, 2.0, 'a' #} ),
                wt::test( {#1, 2.0, 'a'#} != {#1, 2.5, 'a' #} ),
                wt::test( {#1, 2.0, 'a'#} != {#2, 2.0, 'b' #} )
        ]
},{
        sectionName: "record",
        tests: [
                wt::test( {a:0, b:1} == {a:0, b:1} ),
                wt::test( {a:0, b:1} == {b:1, a:0} ),
                wt::testDiff( {a:0, b:1} , {a:1, b:1} ),
                wt::testDiff ( {a:0, b:1} , {b:1, a:1} ),
                wt::testDiff( {a:0, b:1} , {a:0, b:0} ),
                wt::testDiff( {a:0, b:1} , {b:0, a:0} ),
                wt::testDiff( {a:0, b:1} , {a:1, b:0} ),
                wt::testDiff( {a:0, b:1} , {b:0, a:1} )
        ]
}]
->wt::reportResults()

where {
        wt = library file 'Test.wlib';
}
```

Раније поменута подела фајлова са тестовима у секције је овде приказана – за проверу записивања константи сваког типа постоји по једна секција, дефинисана резервисаном речју `sectionName`, иза које следи листа тестова, означена кључном речју `tests`. Називи секција одговарају називима типова:

- `int` – целобројни тип;
- `float` – реални бројеви;
- `string` – ниске карактера;
- `bool` – Булов тип;
- `list` – листа;
- `array` – низ;
- `tuple` – торка и
- `record` – слог.

Постоји и тип података `map` (мапа), али се не може записати као литерал, па се не јавља у овом тест-фајлу.

На самом крају програма, у блоку `where`, дефинишу се имена која се користе у програму. У овом примеру је дефинисано само локално име `wt` за реферисање библиотеке за тестирање.

Библиотека *Test.wlib* садржи следеће дефиниције:

- `test` – провера да ли је аргумент `true`, односно да ли је аргумент тачан исказ;
- `testFalse` – провера да ли је аргумент `false`, односно нетачан исказ;
- `testEq` – провера да ли су аргументи једнаки по вредности;
- `testEqEps` – провера да ли је апсолутна вредност разлике аргумената мања од задате вредности;
- `testDiff` – провера да ли су аргументи различити;
- `testLT` – провера да ли је први аргумент мањи од другог;
- `testLE` – провера да ли је први аргумент мањи или једнак другом;
- `testGT` – провера да ли је први аргумент већи од другог;
- `testGE` – провера да ли је први аргумент већи или једнак другом.

Извођењем тестирања настаје извештај *waflTestReport.txt*, који се налази у текућем директоријуму, где је интерпретатор позван. Извештај укључује информације о спроведеном тестирању:

```
File testCollection\syntax_constructs\syntax_literals_te OK!    Pass  8 / 8
       Section int ..................................... OK!    Pass  11 / 11
       Section float ................................... OK!    Pass  10 / 10
       Section string .................................. OK!    Pass  4 / 4
       Section bool .................................... OK!    Pass  4 / 4
       Section list .................................... OK!    Pass  5 / 5
       Section array ................................... OK!    Pass  5 / 5
       Section tuple ................................... OK!    Pass  4 / 4
       Section record .................................. OK!    Pass  8 / 8
```

Тестирање записивања константи је прошло без инцидената, односно, сви тестови успешно пролазе.

2.3. Приступ члановима колекције по индексу

Циљ овог скупа тестова је био испитати понашање синтаксних конструкција за приступање појединачним елементима ниске, листе, низа, торке, слога и мапе. У тест-фајлу *get_element_test.wafl*, називи секција одговарају називима типова чије се понашање тестира у тој секцији. Ево дела тог фајла, при чему је са три тачке (…) у запису кода назначено да су неке линије прескочене:

```
[{
       sectionName: "array",
       tests: [
              wt::testEq( [#2,4,6,8,10,12,14,16,18,20,1,3,5,7,9,11#][0] , 2),
              …
```

```
                wt::testEq( [#2,4,6,8,10,12,14,16,18,20,1,3,5,7,9,11#][15] , 11),
                wt::testEq( [#2,4,6,8,10,12,14,16,18,20,1,3,5,7,9,11#][3+16] , 8),
                …
                wt::testEq( [#2,4,6,8,10,12,14,16,18,20,1,3,5,7,9,11#][3-3*16] , 8)
        ]
},{
        sectionName: "tuple",
        tests: [
                wt::testEq(
{#2,3.0,'a',[1,2,3],3,4.0,'b',[4,5],6,5.5,3,8,'bca',7.7,0,'afd'#}.1 , 2 ),
                …
                wt::testEq(
{#2,3.0,'a',[1,2,3],3,4.0,'b',[4,5],6,5.5,3,8,'bca',7.7,0,'afd'#}.16 , 'afd' )
        ]
},{
        sectionName: "string",
        tests: [
                wt::testEq( 'nikola'[0] , 'n' ),
                …
                wt::testEq( 'nikola'[5] , 'a' ),
                wt::testEq( 'nikola'[-6] , 'n' ),
                …
                wt::testEq( 'nikola'[-1] , 'a' ),
                wt::testEq( 'nikola'[3+6] , 'o' ),
                …
                wt::testEq( 'nikola'[3-3*6] , 'o' )
        ]
},{
        sectionName: "list",
        tests: [
                wt::testEq( [1,2,3,4][0] , 1 ),
                …
                wt::testEq( [1,2,3,4][3] , 4 ),
                wt::testEq( [1,2,3,4][-1] , 4 ),
                …
                wt::testEq( [1,2,3,4][-4] , 1 ),
                …
                wt::testEq( [1,2,3,4][1+4] , 2 ),
                …
                wt::testEq( [1,2,3,4][1-3*4] , 2 )
        ]
},{
        sectionName: "record",
        tests: [
                wt::testEq( {a:1,b:2}$a , 1 ),
                wt::testEq( {a:1,b:2}$b , 2 )
        ]
},{
        sectionName: "map",
        tests: [
                wt::testEq( createMap([# 'name1', 'name2' #],[# 'value1', 'value2'
#])['name1'] , 'value1' )
        ]
}]
…
```

За приступање појединачним елементима листе, низа и ниске карактера користе се угласте заграде. Индексирање може бити непосредно (од 0 до *n*-1, при чему је *n* број елемената колекције, а 0 је индекс првог елемента) и обрнуто (од -*n* до -1, при чему је *n* број елемената колекције, а -1 је индекс последњег елемента). Оба начина

подржавају приступ по модулу – за произвољан број *m* и број елемената колекције *n*, индекс елемента је m по модулу *n*. Све комбинације су проверене за ова три типа. Преостала три типа не подржавају овакве специфичности, па су њихови тест-случајеви мање бројни.

У извештају након тестирања пише да све пролази без проблема:

```
File testCollection\syntax_constructs\get_element_test.w OK!    Pass  6 / 6
       Section array ..................................... OK!    Pass  16 / 16
       Section tuple ..................................... OK!    Pass  16 / 16
       Section string .................................... OK!    Pass  12 / 12
       Section list ...................................... OK!    Pass  4 / 4
       Section record .................................... OK!    Pass  2 / 2
       Section map ....................................... OK!    Pass  1 / 1
```

... али је приликом тестирања било непријатних изненађења. Прва провера приступа торци са 20 елемената довела је до грешке при извршавању програма. Испоставило се да верзија интерпретатора која је тада коришћена није подржавала рад са торкама које имају више од 15 елемената. Исти проблем јавио се и код приступа појединачним елементима слога, при чему је утврђено да је ограничење максималног броја чланова било 16.

Захваљујући резултатима овог тестирања, имплементација интерпретатора је унапређена, тако да, начелно, подржава неограничен број елемената у оквиру разматраних типова. У пракси је, међутим, примећено значајно успоравање аутоматске провере типова за торке и слогове са великим бројем елемената.

Тестирање приступа елементима колекције по индексу довршено је тестирањем оператора за одсецање, који раде са секвенцама, тј. са низовима, листама и нискама. При њиховом коришћењу, уместо [*indeks*], наводи се [*donja_granica:gornja_granica*], што враћа „сегмент“ или само једна граница – [*donja_granica*:], што даје „суфикс“ или [:*gornja_granica*], што даје „префикс“. Ови оператори као излаз враћају одговарајући део секвенце, обухваћен улазним аргументом, односно аргументима. Ти тест-случајеви су у фајловима *slice_segment_test.wafl*, *slice_prefix_test.wafl* и *slice_suffix_test.wafl*, који покривају позиве са обе наведене границе или само једним. Њихово тестирање пролази без проблема.

2.4. Намерно неуспешни тестови

Колико год појава неуспеха при тестирању била непријатна, откривање недостатака и прављење извештаја о њима је важна функционалност алата за тестирање. Неоспорно је, такође, да је јако корисна провера да ли нешто не ради, односно да ли је нешто и даље недозвољено понашање након одређених измена, када се разматра софтвер који трпи сталне промене. Тест-случајеви који не треба да прођу проверавају исправност понашања алата за тестирање и интерпретатора у случају грешке. Такође, уколико су планиране измене које их обухватају, пружају потпору даљем развоју. Тест-случај са торком од двадесет елемената је неко време

био издвојен у посебан фајл, а када је интерпретатор измењен, написан је нови тест-случај са сврхом постизања неуспеха при тестирању.

Садржај фајла *intentional_error_test.wafl* је:

```
[{
	sectionName: "array",
	tests: [
		wt::testEq( [#2,4,6,8,10#][0] , 13),
		wt::testEq( [#2,4,6,8,10#][0] , 2)
	]
},{
	sectionName: "tuple",
	tests: [
		wt::testEq(
{#2,3.0,'a',[1,2,3],3,4.0,'b',[4,5],6,5.5,3,8,'bca',7.7,0,'afd',7#}.2 , 3 ),
		wt::testEq(
{#2,3.0,'a',[1,2,3],3,4.0,'b',[4,5],6,5.5,3,8,'bca',7.7,0,'afd',7,3.8,'fail',0#}.1 , 2 )
	]
}]
->wt::reportResults()

where {
	wt = library file 'Test.wlib';
}
```

А извршавање нам даје следећи испис у извештају:

```
File testCollection\syntax_constructs\intentional_error_ FAIL!  Program error!
```

Ово је очекиван резултат, јер при интерпретирању првог тест-случаја из друге секције долази до поређења елемената два различита типа (реалног и целобројног), што није дозвољено. Присутан је и тест-случај који је написан тако да не пролази, односно да врати нетачан резултат (први тест из прве секције), али грешка при извршавању програма то прикрива. Ако се тест-случајеви који проузрокују овај проблем не изврше, нпр. ако су под коментаром, као у *intentional_failure_test.wafl*:

```
[{
	sectionName: "array",
	tests: [
		wt::testEq( [#2,4,6,8,10#][0] , 13),
		wt::testEq( [#2,4,6,8,10#][0] , 2)
	]
},{
	sectionName: "tuple",
	tests: [
		//wt::testEq(
{#2,3.1,'a',[1,2,3],3,4.0,'b',[4,5],6,5.5,3,8,'bca',7.7,0,'afd',7#}.2 , 1.3 ),
		//wt::testEq(
{#2,3.0,'a',[1,2,3],3,4.0,'b',[4,5],6,5.5,3,8,'bca',7.7,0,'afd',7,3.8,'fail',0#}.1 , 2 )
	]
}]
->wt::reportResults()

where {
	wt = library file 'Test.wlib';
}
```

онда се резултат неуспешног тестирања види у извештају у следећем облику:

```
File ...\intentional_failure_test.wafl ................. FAIL!  Pass  1 / 2
   Section array ....................................... FAIL!  Pass  1 / 2
    * FAIL:  wt::testEq( [#2,4,6,8,10#][0] , 13)
               ==>  2 = 13
      OK:    wt::testEq( [#2,4,6,8,10#][0] , 2)
   Section tuple ....................................... OK!    Pass  0 / 0
```

Овим примерима је описана форма фајлова са тестовима и изглед извештаја о тестирању, у различитим околностима, па ће о томе бити мање речи у даљем раду.

2.5. Листе и низови

Испитано је 28 функција за рад са листама, кроз 128 тест-случајева. Већина тестираних функција за рад са низовима подржана је тек у каснијим фазама тестирања. Оне су имплементиране по узору на постојеће функције везане за листе. Услед тога је рад са низовима испитан по узору на тестирање везано за листе, уз употребу низова у одговарајућим тест-случајевима. У наставку ће бити разматрани првенствено тест-случајеви у којима се јављају листе јер су управо они довели до откривања бројних пропуста.

Функција `count` прихвата листу и логички предикат, а резултат је број елемената листе за које је задати предикат тачан. Примећена је недоследност понашања функције када је улазна листа празна:

- позив `count([], \x:x>0)` враћао је вредност 0, што је очекивано, јер празна листа нема елемената, па ни оних елемената који су већи од нула;
- позив `count([], \x:x=x)` није враћао резултат, већ је доводио до грешке у извршавању

Функција `append` користи се за надовезивање две листе. Приликом њеног тестирања, примећено је да се не понаша исправно за улазне вредности од којих је једна празна листа, која се у програмском језику *Wafl* означава са `[]` или `nil`. Позиви `append(nil, [1,2])`, `append([1,2], nil)` и `append([],[])` унутар тестова доводили су до грешке.

Функција `intRange` служи за прављење листе целих бројева, такве да садржи све чланове полу-затвореног интервала задатог улазним вредностима. На пример, резултат позива `intRange(0,5)` је `[0,1,2,3,4]`. Тест-случајеви са непразним листама нису занимљиви (јер ти тестови пролазе), али поређење резултата позива `intRange(5,5)` са празном листом враћало је вредност `false`. Овај проблем је у међувремену отклоњен и овај тест сада пролази.

Функција `empty`, која проверава да ли је листа празна, није се исправно понашала када је њен аргумент празна листа. Такође, функција `length`, која за дату листу враћа њену дужину, односно број елемената листе, враћала је грешку приликом примене на празну листу. Није било могуће ни проверити да ли празна листа има више од

нула елемената, коришћењем функције `longerThan`, нити издвојити подлисту празне листе, коришћењем функције `subList`. Покушај сортирања празне листе, функцијом `sort`, доводио је до грешке. Неки од поменутих тест-случајева су:

```
wt::test( empty([]) ),
wt::testEq( [].length() , 0),
wt::test( [].longerThan(0) ),
wt::testEq( sort([]) , [] )
```

Важно је напоменути да су све наведене функције добро и исправно радиле у програмима, а у тестовима је долазило до грешке због проблема са провером типова. На пример, позив `empty([])` је доводио до грешке, што је откривено тестовима. У програму, таква употреба функције нема смисла, већ се јавља позив попут `empty(tl([1]))`, у којем при интерпретирању не долази до проблема са препознавањем типова. За све наведене проблеме са празним листама је заједничко да су то тест-случајеви код којих при интерпретирању не може да се установи тип елемената листе.

Функција `newArrayFn`, која прави низ и чији су аргументи величина низа који ће бити направљен и функција чијом ће применом на индекс елемената бити добијени сами елементи низа, тестирана је следећим тест-случајевима:

```
wt::testEq( newArrayFn(0,\x:x+2) , [##] ),
wt::testEq( newArrayFn(3,\x:x+2) , [#2,3,4#] ),
wt::testEq( newArrayFn(18,\x:x*0) , [#0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0#] ),
wt::testEq( newArrayFn(5,\x:x*3) , [#0,3,6,9,12#] )
```

Она се понашала правилно, али употреба функције `newArrayFn_par`, која је имплементирана са циљем паралелизације израчунавања и бржег добијања истог резултата као и `newArrayFn`, доводила је до програмске грешке.

Проблем препознавања типова решен је увођењем правила да се подразумевано користе листе целих бројева, када није могуће закључити тип елемената листе. Имплементација функције `newArrayFn_par` је унапређена. Тиме су отклоњени сви описани проблеми и у последњој верзији *Wafl* интерпретатора наведени тест-случајеви пролазе.

2.6. Функције вишег реда

У оквиру тестирања функција за рад са листама и низовима тестиране су функције вишег реда. Њих карактерише то што је један од аргумената функција или логички предикат. Програмски језик *Wafl* подржава примену ових функција на секвенце, с тим да у оквиру овог рада није проверено понашање у раду са нискама. Примери у наставку односе се на рад са листама.

Функција `map` примењује дату унарну функцију на све елементе секвенце, правећи нову секвенцу. Постоји и функција `map_par`, која ради искључиво са листама и чије је израчунавање паралелизовано, али даје исти резултат. Коришћене су исте улазне

вредности у тестирању ове две функције како би се проверила ова еквиваленција. Неки од тест-случајева су:

```
wt::testEq( (1..5).map(\x: x*2) , [2,4,6,8,10] ),
wt::testEq( [[1,2,3],[1,2],[1]].map(tl) , [[2,3],[2],[]] ),
wt::testEq( [[],[],[]].map(tl) , [[],[],[]] )
```

Функције `foldl` и `foldr`, као и њима еквивалентне `leftAggregate` и `rightAggregate`, примењују дату бинарну функцију на све елементе секвенце. При томе, `foldl` подразумева леву асоцијативност, а `foldr` десну асоцијативност при рачунању. Следећи тест-случајеви илуструју разлике:

```
wt::testEq( leftAggregate(["ovo","je","test"], \x,y:x+y, ""), "ovojetest"),
wt::testEq( rightAggregate(["ovo","je","test"], \x,y:x+y, ""), "ovojetest"),
wt::testEq( (1..5).leftAggregate(\r,x: x:r, [0]), [5,4,3,2,1,0] ),
wt::testEq( (1..5).rightAggregate(\x,r: x:r, [0]), [1,2,3,4,5,0] )
```

Функција `filter` прави нову секвенцу, применом датог логичког предиката (услова), на дату секвенцу. Елементи који не задовољавају услов се не налазе у новој секвенци. Неки од тест-случајева за ову функцију су:

```
wt::testEq( filter([1,2,3],\x:x%3=0) , [3] ),
wt::testEq( filter([1,2,3],\x:x=x) , [1,2,3] ),
wt::testEq( filter([],\x:x>0), [] )
```

Функције `exists` и `forall` проверавају да ли у датој секвенци постоји елемент који задовољава дати услов, односно да ли сви елементи дате секвенце задовољавају дати услов. Неки од тест-случајева којима се проверава понашање ових функција су:

```
wt::testFalse( [1,2,3].forall(\x:x>2) ),
wt::test( [].forall(\x:x>2) ),
wt::testEq( filter([1,2,3],\x:x%3=0) , [3] ),
wt::testEq( filter([],\x:x=x), [] )
```

Функција `filterMap` комбинује понашање функција `filter` и `map`. Прво се на дату секвенцу примењује `filter` са датим логичким предикатом, а потом се на добијену секвенцу примењује `map` са датом унарном функцијом. Следећи тест-случај настао је на основу два наведена тест-случаја за функције `map` и `filter`:

```
wt::testEq( [#[1,2,3],[1,2],[1]#].filterMap(\x: length(x)<3, \x: tl(x)) , [#[2], []#] )
```

Функција `zipWith` прави нову секвенцу применом дате бинарне функције на елементе две дате секвенце. Тест-случајеви илуструју понашање:

```
wt::testEq( zipWith([1,2,3],[4,5,6],\x,y:x*y) , [4,10,18] ),
wt::testEq( zipWith([1,2,3],[4,5],\x,y:x*y) , [4,10] ),
wt::testEq( zipWith([],[4,5,6],\x,y:x*y) , [] ),
wt::testEq( zipWith([],[],\x,y:x*y) , [] )
```

Последњи тест-случај за функцију `zipWith` враћао је грешку због раније поменутог проблема препознавања типова. Све остале функције понашале су се у складу са очекивањима током тестирања.

2.7. Ниске

У оквиру ове целине, кроз 274 тест-случаја, тестирано је понашање 27 функција за рад са нискама, као и провера приступа појединачним карактерима или поднискама преко њихових позиција. Уочено је проблематично понашање код већег броја функција и то када је један од аргумената празна ниска.

Функције као што су `strPos`, `strLastPos` и `strNextLastPos` (и њима еквивалентне функције са "I" на крају назива, које не праве разлику између малих и великих слова) служе да се у датој ниски пронађе позиција датог карактера или друге ниске. Током тестирања, ове функције понашале су се као да се празна ниска налази на крају било које непразне ниске. На пример, резултат позива `strLastPos("ovo je test", "")` је 11.

Функција `strRTrim`, која из дате ниске избацује све размаке са десне стране враћала је грешку за улазну ниску која је празна или су сви њени карактери размаци.

Функције `strReplaceAll` и `strReplaceAllI`, које у датој ниски свако појављивање неке дате ниске карактера замењују датом другом ниском. Примећено је веома чудно понашање ових функција када је тражена ниска за замену празна. Позиви попут `strReplaceAll("ovo je test", "","e")` покретали су процес које траје неочекивано дуго и узима пуно процесорских ресурса. Исход и укупно време израчунавања остали су непознати, јер је најдужи покушај прекинут после једног минута. Позив функције `strSplit`, која дели дату ниску користећи дати сепаратор, попут `strSplit("test","")` узроковао је исти проблем.

Неки од тест-случајева који су указали на проблеме су:

```
wt::testEq( strLastPos("ovo je test", "") , 11 ),
wt::testEq( strRTrim("   ") , "" ),
wt::testEq( strReplaceAll("ovo je test", "", "e") , "eoeveoe ejeee eteeesete" ),
wt::testEq( strSplit("test", "") , ['t','e','s','t'] )
```

Након ових резултата, уследиле су поправке имплементације библиотеке. Имплементација функција које траже позицију датог карактера је унапређена, проблем са функцијом `strRTrim` је уклоњен, а `strReplaceAll`, `strReplaceAllI` и `strSplit` сада враћају тачне резултате у задовољавајућим временским оквирима.

2.8. Регуларни изрази

Дванаест функција за рад са регуларним изразима тестирано је кроз 147 тест-случајева.

Програмски језик *Wafl* користи синтаксу записивања ниски језика *C++*. То значи да се при навођењу појединих карактера у оквиру ниске мора употребити тзв. *escape* карактер, како би се спречило посебно тумачење тих карактера. У програмском језику *C++*, *escape* карактер је "\". Дакле, да би се записала ниска која се тумачи као `"\"` од стране интерпретатора, потребно је унети `"\\"`. Такође, при претраживању ниске по шаблону `\w+`, позив одговарајуће функције не може да буде `regexMatch("test","\w+")`, већ мора бити `regexMatch("test","\\w+")`.

Функције `regexPosAll` и `regexPosAllI` враћају листу свих позиција од којих почиње ниска препозната датим шаблоном. Употреба празне ниске као аргумента ових функција изазивала је слично понашање као и код неких функција за рад са нискама, где после дуго времена и пуно заузетих ресурса нема никаквих резултата. На пример, позив `regexPosAll('abc','')` доводио је до тог проблема.

Много занимљивије од тога, добијен је следећи резултат за један позив функције `regexPosAllI`:

```
* FAIL:  wt::testEq( regexPosAllI('aBc09 AbC08','[A-Z]') , [0,1,2,6,7,8] )
             ==>  [0, 1, 2, 7, 8] = [0, 1, 2, 6, 7, 8]
```

Ова функција би требало да занемари разлику између малог и великог слова. Шаблон `[A-Z]` у том случају треба да проузрокује препознавање свих слова у оквиру дате ниске `“aBc09 AbC08”`, која се налазе на позицијама 0, 1, 2, 6, 7 и 8, међутим функција не препознаје слово „A“ и враћа нетачан резултат.

Даља анализа довела је до закључка да проблем потиче из *C++* библиотеке *<regex>*, у оквиру преводиоца *Microsoft Visual Studio 2017*. Имплементација функција за рад са регуларним изразима језика *Wafl* се ослања на ову библиотеку.

Даље проучавање овог проблема довело је до закључка да је проблем у самој библиотеци *MSVC*[17], чија се функција непосредно користи у имплементацији `regexPosAllI`, као и осталих функција за рад са регуларним изразима.

Указала се потреба за додатним функцијама, `regexSearchAll` и `regexSearchAllI`, које су додате у основну библиотеку језика *Wafl* и такође су обухваћене тестовима.

Сви описани проблеми су уклоњени, а они директно узроковани библиотеком *Microsoft Visual Studio 2017* су нестали увођењем библиотеке *Microsoft Visual Studio 2019* у употребу. Примећено је да је услед ове промене резултујући код интерпретатора *clwafl* и до 40% спорији на оперативном систему *Windows*.

[17] Имплементација стандардне библиотеке програмског језика *C++*, доступна на https://github.com/microsoft/STL, коју одржава компанија *Microsoft*.

2.9. Рад са фајл-системом

Поступак тестирања рада са фајл-системом је специфичан, јер се неке од функција тичу постојећих фајлова и директоријума, тј. проверавају да ли они постоје, мењају их или бришу. То у пракси значи да се неке од функција које се тестирају морају и употребити за припрему структуре која ће бити коришћена за тестирање. У супротном, успешно тестирање би подразумевало измену стања фајл-система од стране корисника пре тестирања, тј. да корисник обезбеди да постоји фајл или директоријум којем треба приступити или га обрисати. То није прихватљиво решење за тестирање које се јако често понавља.

Лакоћа коришћења је, дакле, обезбеђена науштрб једног од пет принципа писања тестова јединица кода о којима је било речи на почетку поглавља. Секције у овом фајлу са тестовима представљају кораке прављења и уклањања привремене структуре фајл-система. Неуспех у једној од секција ће вероватно проузроковати неуспехе и у наредним. Читав процес изгледа овако:

- проверава се да не постоји фајл са наведеним именом
- направи се фајл са тим именом
- проверава се да сада постоји тако назван фајл
- чита се из њега и пише се у њега
- брише се
- проверава се да не постоје директоријуми који треба да буду направљени
- прави се директоријум са наведеним именом и у њему два директоријума
- проверава се да сви ти директоријуми сада постоје
- проверава се да је структура фајл-система онаква каква се очекује
- бришу се направљени директоријуми

Главни изазов је био генерисање случајних назива за привремене фајлове и директоријуме, како не би дошло до проблема услед могућих заосталих предмета претходних тестирања. Тиме се такође ствара могућност паралелизације тестирања, јер у тренутку извршавања више тестова не долази до колизије изазване употребом идентичних назива фајлова и директоријума. За ову сврху, у првобитној имплементацији, употребљена је функција `fileNewTempName`. Она прави нови, насумично именовани, привремени фајл у директоријуму *Temp*[18] и враћа његову апсолутну путању као резултат. Прављење имена привремених чинилаца фајл-система било је урађено на следећи начин:

```
t = fileNewTempName();
tempFile = t.subStr(t.strLastPos('\\')+1, 5) + '.txt';
tempDir = 'testDir' + t.subStr(t.strLastPos('\\')+1, 5);
```

[18] На оперативном систему Windows, овде се смештају сви привремени фајлови. Апсолутна путања до овог директоријума на мом рачунару, примера ради је `C:\Users\Nikola Ninkov\AppData\Local\Temp`. На оперативном систему *Linux*, функција `fileNewTempName` се понаша идентично, с тим да се привремени фајл смешта у директоријум намењен за то, што је обично */tmp*.

Из апсолутне путање се издваја име фајла, а затим се уклања постојећа екстензија *(.wafl*) и додаје екстензија за текстуални фајл (*.txt*).

Није било неуспешних тестова при тестирању са оваквом имплементацијом, али у њој има озбиљних пропуста. Наиме, у програмском језику *Wafl* не постоје глобалне променљиве. У наведеном примеру кода, `tempFile` и `tempDir` имају семантику функција, самим тим се израчунавају сваки пут када се појаве у тест програму. То значи да се и `t` такође израчунава поново, дајући ново име при сваком позиву. Такође, овако добијена имена `tempFile` и `tempDir` нису обавезно непостојећа.

Измењено решење користи експлицитно задата имена привремених фајлова и директоријума. Неинтуитивна синтакса је промењена тако да `tempFile` и `tempDir` буду представљени као функције без аргумената, како се и понашају:

```
tempFile() = 'U11erly_d3spic4ble_file_name2021.txt';
tempDir() = 'Utt3rly_desp1cabl3_d1rectory_name2021';
subDir1() = 'Utt3rly_desp1cabl3_d1rectory_name2021\\subDir1';
subDir2() = 'Utt3rly_desp1cabl3_d1rectory_name2021\\subDir2';
file1() = 'Utt3rly_desp1cabl3_d1rectory_name2021\\subDir1\\file1.txt';
file2() = 'Utt3rly_desp1cabl3_d1rectory_name2021\\subDir1\\file2.txt';
```

Такође, функција `fileNewTempName` сада се позива само једном за време целог теста, како би се проверило да ли заиста прави привремени фајл:

```
{
	sectionName: “fileNewTempName”,
	tests: [
		wt::test( regexPos(fileNewTempName(), ‘wafltmpfile’) >= 0 )
	]
}
```

Овако измењени тестови прошли су без инцидената на оперативном систему *Windows*, али на оперативном систему *Linux*, који језик *Wafl* такође подржава, тестови нису пролазили. Прецизније, резултати позива функција нису били идентични очекиваним резултатима у тест-случајевима, а све операције над системом фајлова су радиле исправно. Узрок неочекиваних резултата су били сепаратори у именима привремених фајлова и директоријума, који су зависни од платформе. Како би се избегла зависност од платформе и исти тестови могли неометано да се извршавају на оба оперативна система, неопходно је било наћи начин за очитавање стања окружења у којем се програм извршава. Ово је омогућено увођењем нове функције `sysEnvDetails` у основну библиотеку језика *Wafl*, која као резултат враћа карактеристике извршног окружења. Кôд у којем се задају имена привремених фајлова и директоријума у коначној имплементацији изгледа овако:

```
tempFile() = 'U11erly_d3spic4ble_file_name2021.txt';
tempDir() = 'Utt3rly_desp1cabl3_d1rectory_name2021';
subDir1() = 'Utt3rly_desp1cabl3_d1rectory_name2021' + sep() + 'subDir1';
subDir2() = 'Utt3rly_desp1cabl3_d1rectory_name2021' + sep() + 'subDir2';
file1() = 'Utt3rly_desp1cabl3_d1rectory_name2021' + sep() + 'subDir1' + sep() +
'file1.txt';
file2() = 'Utt3rly_desp1cabl3_d1rectory_name2021' + sep() + 'subDir1' + sep() +
'file2.txt';
```

```
sep() = sysEnvDetails()$dirSep;
```

Овако параметризована имена су независна од платформе. Примећено је и да називи фајлова без сепаратора нису били проблематични. Тиме су сви проблеми у тестирању ове групе функција решени.

2.10. Конверзија типова

Уочено је неколико проблема при раду са функцијама за конверзију типова.

Прављење празног низа од празне листе, функцијом `asArray`, као прављење празне листе од празног низа, функцијом `asList` није било могуће, односно следећи тест-случајеви доводили су до програмске грешке:

```
wt::testEq( [].asArray() , [##] ),
wt::testEq( [##].asList() , [] )
```

Приликом пребацивања листе ниски у ниску, јавила се грешка при извршавању тест-случаја

```
wt::testEq( ['a','b','c'].asString() , '[\'a\', \'b\', \'c\']' )
```

резултат је био `'[a, b, c]'`, а не `'['a', 'b', 'c']'`. Делује безазлено, али при овом губитку знака навода заправо долази до губитка информација – не зна се да је новонастала ниска запис листе ниски, пошто `a`, `b` и `c` могу бити било шта.

Приликом прављења ниске од слога, уочено је нестајање знака навода и промена сепаратора слога. Тест-случај

```
wt::testEq( {# 64, 6.40000, 'qqq' #}.asString(), '{# 64, 6.4, \'qqq\' #}' )
```

пријављивао је грешку, јер је резултат био `'{ a:64; b:6.4; c:qqq }'`, уместо очекиваног `'{ a:64; b:6.4; c:'qqq' }'`.

Даљом анализом утврђено је да је дозвољено комбиновање сепаратора (`,` или `;`) и знака доделе (`:` или `=`) у оквиру истог слога, па је нешто попут `{ a=6, b:7; c=9 }` успешно пролазило превођење.

При раду са нискама, функција `asString` је измењена тако да чува наводнике где је то потребно, како не би дошло до губитка информација, а при раду са слоговима, увек користи сепаратор `,` и знак доделе `=`. За записивање слогова, уведено је правило да први употребљени сепаратор и први употребљени симбол доделе дефинишу синтаксу на нивоу тог слога, како би се постигла усклађеност у запису, без умањења броја доступних стилова писања.

2.11. Функције за рад са целобројним типом

Тестиране су две функције за рад са целим бројевима и откривено је лоше понашање функције `random`, која служи за избор случајних бројева.

Функција узима једну целобројну улазну вредност A и треба да врати насумично изабран број од 0 до A-1, ако је A>2, а у супротном 0. При позивима `random(0)` и `random(-0)` долазило је до недефинисаног понашања, при чему се није добијао никакав резултат.

Понашање је исправљено у наредној верзији интерпретатора, проширењем дефиниције функције, тако да следећи тест-случајеви враћају тачне вредности:

```
wt::testEq( random(0) , 0 ),
wt::testEq( random(-0) , 0 )
```

2.12. Функције за рад са реалним типом

Тестирано је 17 функција, кроз 91 тест-случај.

Испитано је понашање функција за заокругљивање када је аргумент вредност блиска нули:

```
wt::testEq( round(0.00000), 0 ),
wt::testEq( round(-0.00000), 0 ),
wt::testEq( round(0.000001), 0 ),
wt::testEq( round(-0.000001), 0 ),
wt::testEq( ceil(0.00000), 0 ),
wt::testEq( ceil(-0.00000), 0 ),
wt::testEq( ceil(0.000001), 1 ),
wt::testEq( ceil(-0.000001), 0 ),
wt::testEq( floor(0.00000), 0 ),
wt::testEq( floor(-0.00000), 0 ),
wt::testEq( floor(0.000001), 0 ),
wt::testEq( floor(-0.000001), -1 )
```

Проверено је и понашање тригонометријских функција за неке карактеристичне вредности:

```
wt::testEqEps( sin(0.0) , 0.0 , eps ),
wt::testEqEps( cos(0.0) , 1.0 , eps ),
wt::test( abs(1.0+cos(pi)) < eps ),
wt::test( abs(tan(pi/4.0)-1.0) < eps )
```

... при чему је ниже у фајлу дефинисано:

```
eps = 0.0000000001;
pi = 3.1415926535;
```

Сви тестови пролазе и ништа везано за понашање самих функција не заврeђује посебну дискусију, међутим, додавање наизглед безазленог коментара у један од наведених тест-случајева:

```
wt::test( abs(1.0+cos(pi)) < eps ) //cos(pi)=-1
```

доводи до пуцања целог теста:

```
File testCollection\float_functions.wafl ..............  FAIL!  Program error!
```

Системом елиминације, закључено је да до грешке долази због затворене заграде у коментару. Детаљније објашњење је да се приликом претпроцесирања тест програма препозна и коментар као део позива функције, а узрок је недостатак у регуларним изразима којима се врши анализа кода. У даљем раду није било навођења коментара после позива функција за тестирање, како би се избегло испољавање овог проблема, који није решен закључно са последњом библиотеке за тестирање.

2.13. Оператори

У оквиру ове целине, кроз више од 500 тест-случајева, тестирано је 27 оператора програмског језика *Wafl*. Већина оператора може се користити и у облику функције, нпр. `operator+(arg1, arg2)` је еквивалентно `arg1 + arg2` и ова особина је проверена. Тестирање је обухватило и неке „граничне“ тест-случајеве, односно испитивање понашања при раду са вредностима које могу открити мане у имплементацији и захтевају посебну пажњу.

Оператор `<<`, који дати цео број помера улево бит-по-бит, испитан је у случају када ово померање доводи до „испадања“ битова изван граница 64-битног опсега:

```
wt::testEq( 65535<<47 , 9223231299366420480 ),
wt::testEq( -65535<<47 , -9223231299366420480),
wt::testEq( 65535<<48 , -281474976710656 ),
wt::testEq( -65535<<48 , 281474976710656 ),
wt::testEq( 65535<<49 , -562949953421312 ),
wt::testEq( -65535<<49 , 562949953421312 )
```

Оператор `++`, који надовезује две листе, тестиран је у раду са празним листама:

```
wt::testEq( [1,2]++[] , [1,2] ),
wt::testEq( []++[1,2] , [1,2] ),
wt::testEq( []++[] , [] )
```

Оператори `=` и `==`, који су еквивалентни у програмском језику Wafl, тестирани су над свим типовима језика.

Оператор унарне негације, -, тестиран је за улазну вредност 0:

```
wt::testEqEps( -(0.0) , 0.0 , 0.0000000001 )
```

Упркос уложеном труду, нису откривене неправилности у овој целини.

2.14. Мапе

У програмском језику *Wafl* не постоје синтаксни литерали типа `map`, па се свака мапа мора направити позивом функције `createMap`:

```
{
        sectionName: "JoinValueSets",
        tests: [
                wt::testEq( JoinValueSets(m1,m2) , m12 ),
                wt::testEq( JoinValueSets(m2,m1) , m21 ),
                wt::testEq( JoinValueSets(m1,e) , m1 ),
                wt::testEq( JoinValueSets(e,m1) , m1 ),
                wt::testEq( JoinValueSets(e,e) , e )
        ]
}
        …
        m1 = createMap([#'a','b'#],[#'1','2'#]);
        m2 = createMap([#'c','d'#],[#'3','4'#]);
        …
        e = createMap([##],[##]);
```

Глобалне променљиве такође не постоје, као што је раније поменуто, тако да се при сваком тесту праве оне мапе које су наведене као аргумент функције. Сваки тест-случај из приказаног кода узрокује прављење три мапе.

Током тестирања, откривено је да функције `JoinValueSets`, `RemoveFromValueSet` и `UpdateValueSet` не раде како треба, односно њихово позивање не доводи до било каквог резултата, нити пријаве грешке.

Поправкама интерпретатора, отклоњени су проблеми и свих 13 функција за рад са мапама се понаша правилно у последњој верзији.

2.15. Остале функције

На самом крају процеса тестирања, испитане су преостале групе функција.

Тестирање условних израза `if` и `switch` засновано је на примеру коришћења условних израза, преузетом са веб-странице[19] *Wafl Tutorial*. Функција `repeatUntil` такође је испитана у оквиру истог теста. Нису уочене неправилности.

Последња верзија интерпретатора омогућава коришћење појединих функција за дебаговање: `debugOn`, `debugOff`, `debugIsOn` и `debugState`, помоћу којих се може укључити или искључити дебаговање и проверити да ли је дебаговање укључено или искључено. Све функције враћају неизмењен аргумент, где он постоји. У складу са тиме, тестирано је њихово понашање и установљено да нема никаквих проблема.

---

19 http://poincare.matf.bg.ac.rs/~smalkov/wafl/tutorial/tut_02.Program%20Structure_11.Conditional%20Expressions.html

Исправно понашање функција за кодирање ниски карактера је утврђено након мале забуне узроковањем приказа једног од специјалних карактера у командној линији. Наиме, карактер чији је *ASCII* код 127, који представља *Delete* команду и нема придружен симбол се заправо исписује у командној линији:

```
>clwafl -code "asChar(127)"
[WAFL 2.0]
⌂
```

Ово понашање није погрешно, али је узроковало опширну анализу разних кодирања ниски приликом овог тестирања.

Тестирање функција за коришћење и припрему објеката за слање на вебу (*MIME*) није открило проблеме.

2.16. Преглед резултата тестирања

У табели 1 су приказани резултати тестирања. За све наведене групе функција, тестиране су све постојеће функције из те групе, осим за ниске. Функције које раде са колекцијама нису тестиране над нискама (тестиране су над листама и низовима). Функције вишег реда тестиране су у оквиру тестирања функција за рад са листама и низовима. Ти тест-случајеви су издвојени у табели у посебан ред, али не учествују у укупним вредностима као посебна група (јер су већ урачунати).

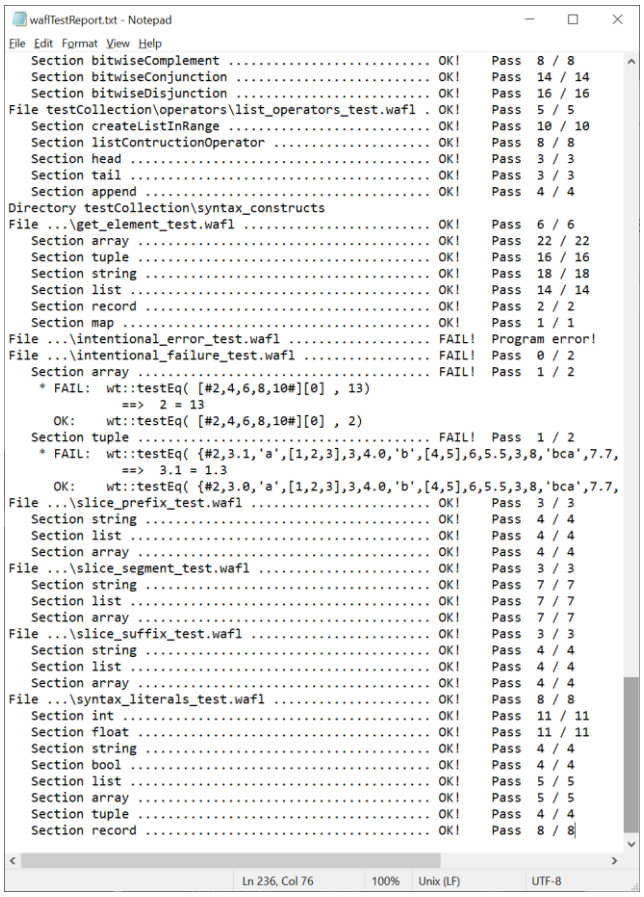
	број функција (оператора)	број тестова	просечан број тестова по функцији
синтаксни литерали	8	52	6.5
приступ члановима колекције	9	45	5
листе	28	128	4.6
низови	21	89	4.2
функције вишег реда	17	77	4.5
ниске	27	274	10.1
регуларни изрази	12	147	12.2
рад са фајл-системом	12	88	7.3
конверзија типова	9	96	10.6
рад са целобројним типом	2	10	5
рад са реалним типом	17	92	5.4
оператори	27	530	19.6
мапе	13	45	3.4
остало	17	57	3.3
укупно	**202**	**1653**	**8.2**

Табела 1: Преглед резултата тестирања по групама функција

3. Представљање извештаја о обављеном тестирању у формату *HTML*

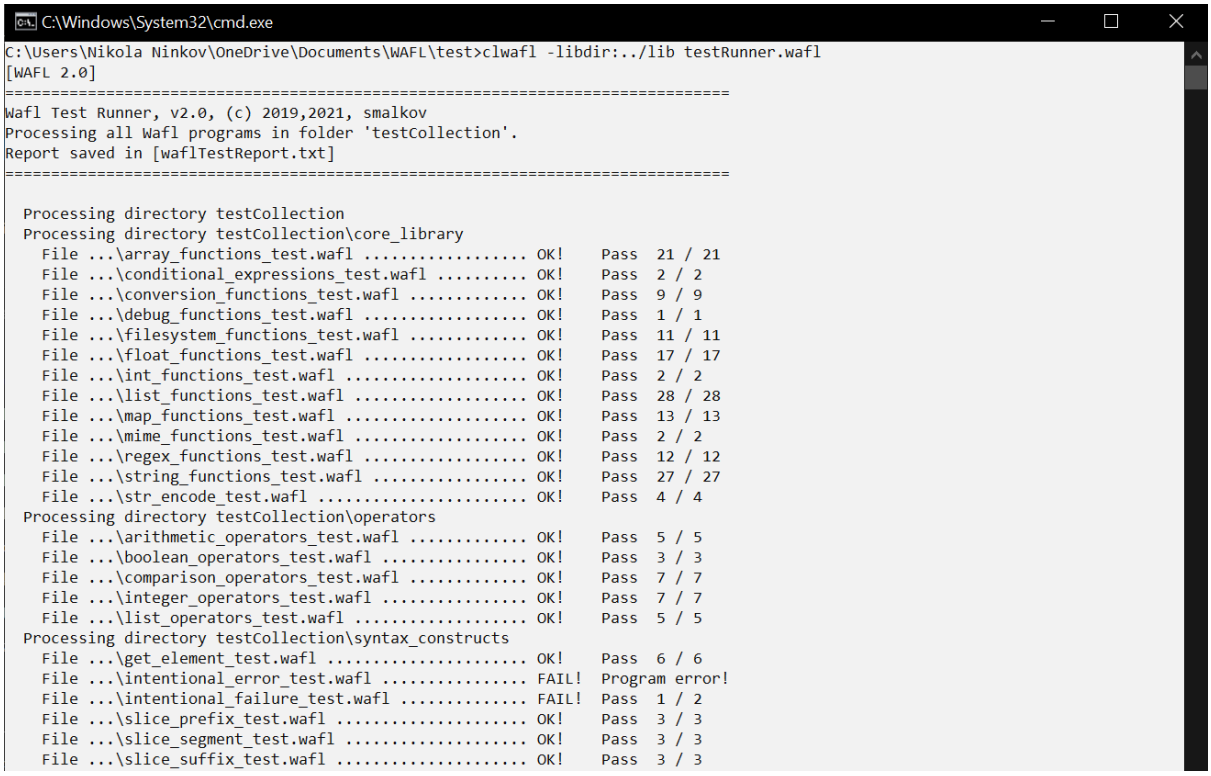
3.1. Текстуални извештај о комплетном тестирању

Извођење тестирања, односно извршавање свих тест-случајева описаних у претходном поглављу подразумева и прављење извештаја о тестирању. То је раније поменути текстуални фајл *waflTestReport.txt*, који је аутоматски направљен од стране интерпретатора и чији се изглед види на слици 1. У коначној верзији колекције тестова, овај извештај има укупно 236 линија текста и садржи информације о структури фајл-система који је обухваћен тестирањем (називе директоријума и фајлова и њихове релативне путање), називе појединачних секција унутар тест-фајлова са резултатом тестирања и посебно издвојене тест-случајеве који нису успешно прошли тестирање. На самом почетку процеса писања тестова, када је њихов број био мали и када су били садржани у свега неколико фајлова, брзо проналажење битних информација у текстуалном извештају било је могуће. Повећањем обима тестирања, преглед резултата овим путем постао је неефикасан.



Слика 1: Извештај *waflTestReport.txt* приказан у прозору текстуалног едитора

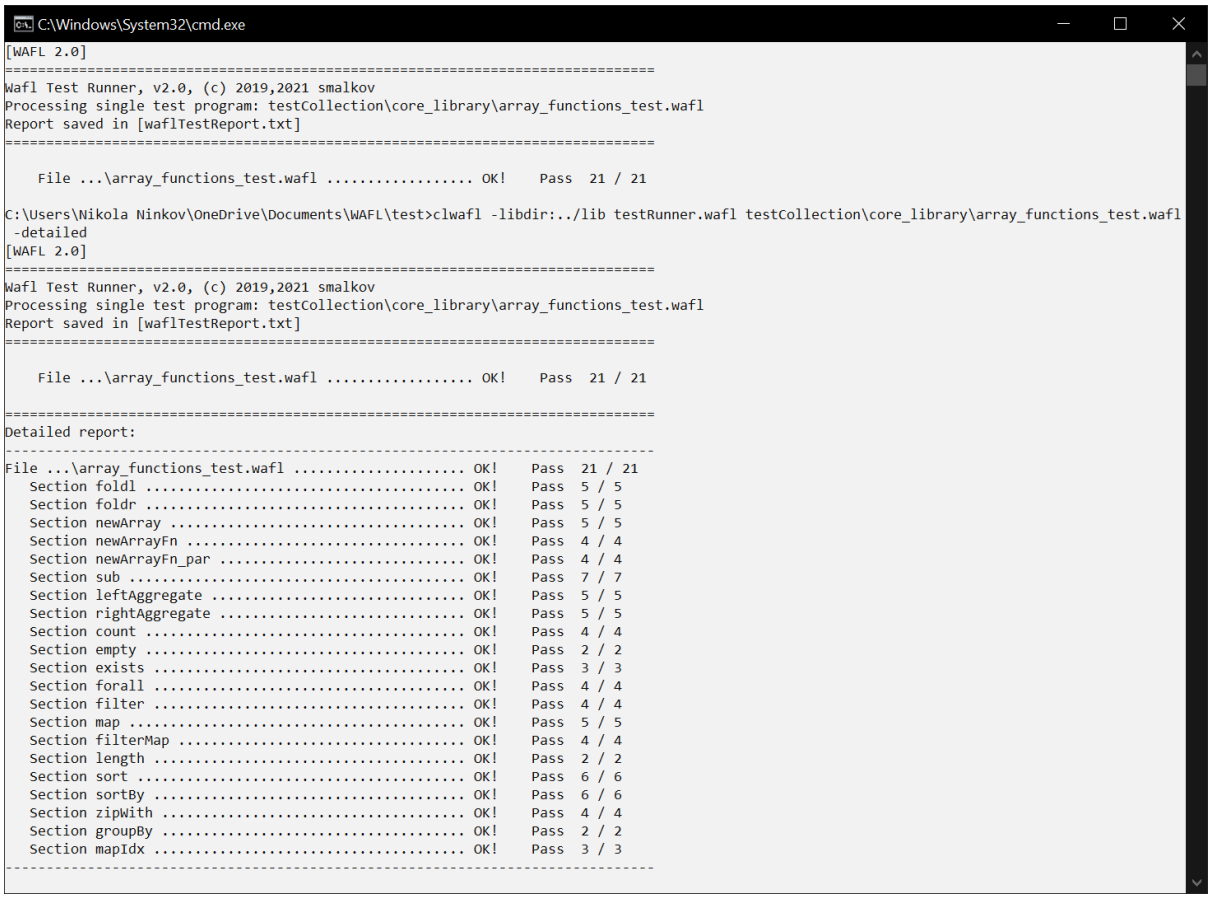
Са друге стране, у командној линији из које је упућен позив интерпретатору да изврши тестирање исписује се сажет извештај о тестирању, приказан на слици 2. Он показује пролазност тестова по секцијама, за сваки тест фајл са тестовима и опис структуре колекције тестова, али нема информација о појединачним тест-случајевима који су били неуспешни.



Слика 2: Извештај о комплетном тестирању приказан у командној линији

3.2. Извештај о појединачном тестирању

Могуће је покренути тестирање појединачног тест фајла, којим се добијају слични извештаји као при свеобухватном тестирању. Приликом појединачног тестирања, има смисла користити опцију `detailed` како би се добиле детаљне информације у прозору командне линије. На слици 3 приказане су обе верзије извештаја о појединачном тестирању у командној линији.



Слика 3: Извештај о појединачном тестирању приказан у командној линији

3.3. Извештај *test.html*

Ниједан од постојећих текстуалних извештаја не представља добро решење за проблем представљања резултата, услед чега се јавља потреба за другачијим приказом свих неопходних информација. Решење је извештај у формату *HTML*, који је потпун, пријатног изгледа[20] и помоћу којег се брзо стиже до кључних информација.

Извештај *test.html* настаје обрадом текстуалног фајла *waflTestReport.txt*. Програм *parser.wafl*, чија имплементација ће бити разматрана у наставку, парсира линије текста и на основу њих прави *HTML* фајл. Приликом тестирања основне библиотеке, прво се изврше сви тест-случајеви, а онда позивом

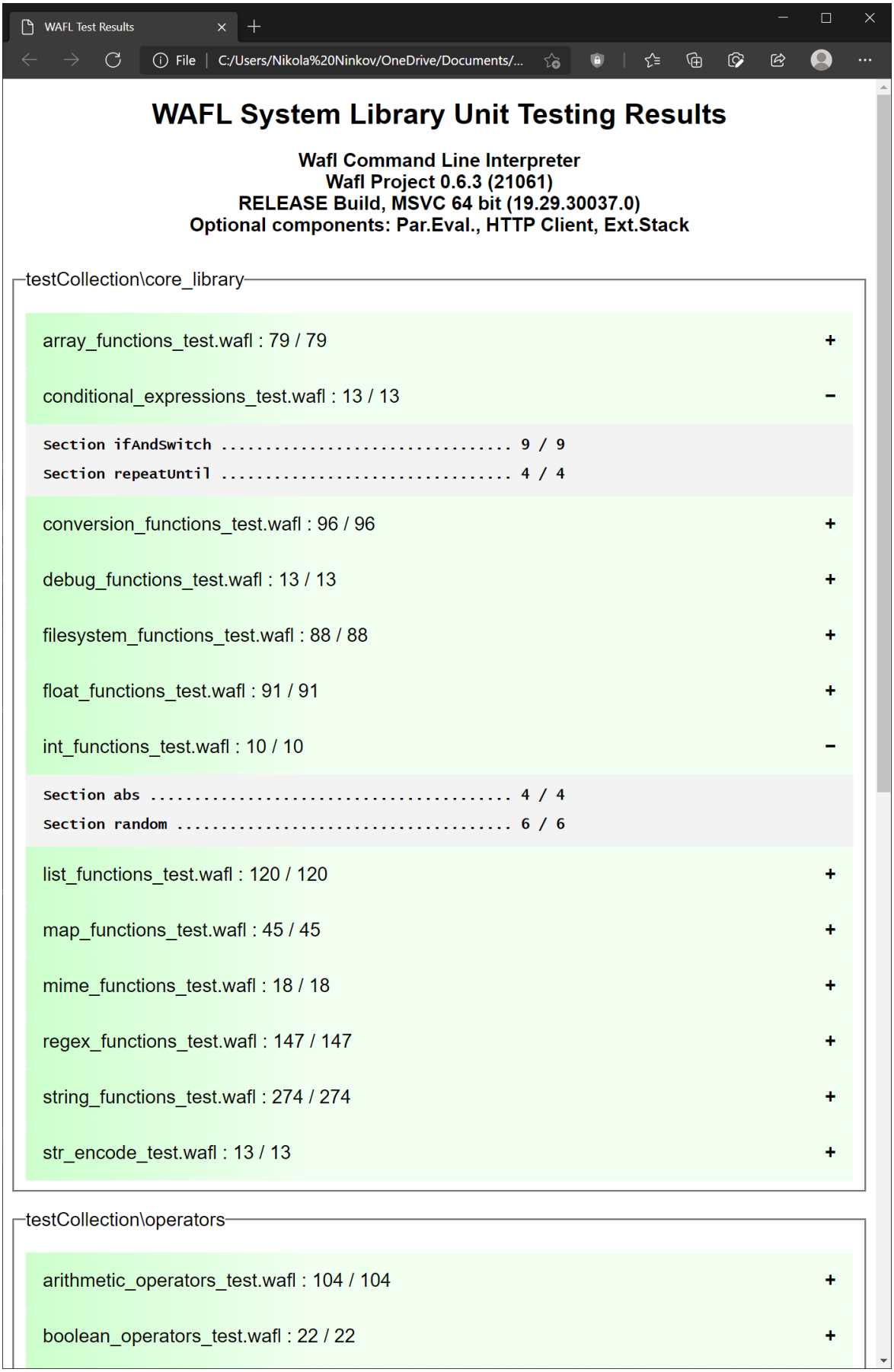
```
>clwafl parser.wafl
```

из командне линије настаје *test.html*. У командној линији се исписује обавештење да је фајл успешно направљен.

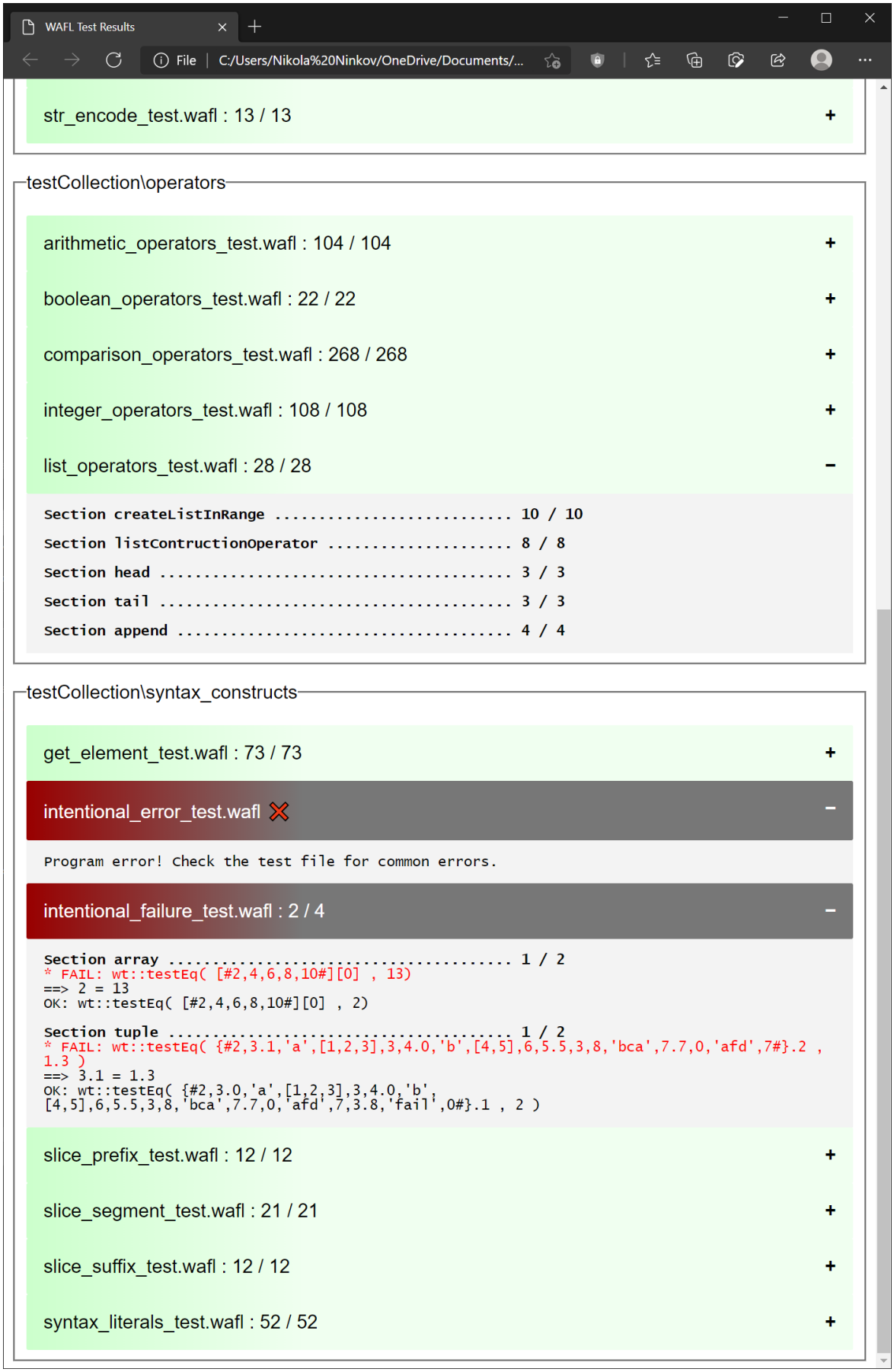
На почетку направљеног извештаја наводе се подаци о верзији интерпретатора којом је извршено тестирање, а следи приказ резултата, по директоријумима. За сваки од њих постоји један формулар (енг. *fieldset*), у чијем заглављу се налази релативна путања директоријума. Формулар садржи по један склопиви (енг. *collapsible*) одељак секција за сваки фајл са тестовима унутар директоријума. Заглавље одељка је обојено у зелено ако су сви тест-случајеви прошли или у црвено, у супротном. На заглављу је приказан број успешних и укупан број тестова одговарајућег фајла са тестовима. Кликом на заглавље одељка, он се „расклапа“, чиме се откривају резултати тестирања по секцијама за одговарајући тест-фајл. У овом приказу, истакнути су сви тест-случајеви који нису прошли и за сваку секцију је наведен број успешних и укупан број тест-случајева. Одељак се може „склопити“, кликом на заглавље, чиме се враћа у првобитни приказ.

На сликама 4 и 5 приказан је изглед извештаја *test.html* у прозору интернет претраживача.

[20] Примедба критички настројених читалаца да је ово субјективна категорија се уважава.



Слика 4: Почетак извештаја са информацијама о верзији интерпретатора



Слика 5: Приказ неуспешних тест-случајева у оквиру склопивих елемената

3.4. Имплементација *HTML*-извештаја

3.4.1. Изглед извештаја

Изрази `htmlHead` и `script` дефинисани су као *HTML* шаблони. У оквиру шаблона, *HTML* ознаке се не наводе као ниске карактера. Због тога није неопходно коришћење великог броја escape карактера, што знатно олакшава рад. Општи облик овако дефинисане функције је:

```
nazivFunkcije = html template
// telo funkcije
<#>;
```

У телу функције, све *HTML* ознаке се наводе у неизмењеном облику, а програмски код се може угнездити коришћењем специјалне ознаке `<# #>`.

Функција `htmlHead` садржи ознаку `<head>`, а унутар ње је ознака `<style>`, у оквиру које је описан изглед свих графичких елемената који се могу видети у резултујућем извештају. Приказ верзије интерпретатора употребљеног за тестирање је такође део функције `htmlHead`:

```
<h2><#
        version_info[strPos(version_info, "]")+3:]
        .strReplaceAll('\n','\n<br/>')
#></h2>
```

При томе, `version_info` се добија као извештај о верзији програма *clwafl*, а израчунава се као резултат функције `cmdExecute`, која извршава дату команду:

```
version_info = cmdExecute("clwafl -version -verbose");
```

Функција `script` садржи ознаку `<script>`, унутар које је наведен *JavaScript* код који омогућава функционисање склопивих графичких елемената који се користе за приказ резултата тестирања по фајловима.

Поменутим двема функцијама обухваћена је форма, а за прикупљање садржаја одговорна је функција `generateHtmlBody`, чији је улазни аргумент садржај текстуалног извештаја који ће бити обрађен, претходно прочитан функцијом `fileRead`.

3.4.2. Садржај извештаја

Функција `fileWrite` користи се за упис у фајл:

```
fileWrite('test.html', createHtml('waflTestReport.txt')).sink("Saved to 'test.html'")
```

Функција `createHtml` ослања се на претходно описане изразе `htmlHead` и `script`, као и на функцију `generateHtmlBody` која врши обраду прочитаног фајла:

```
generateHtmlBody(fileContents) =
	fileContents
		->splitByWord('Directory')
		->map(parseDirectory)
		->strJoin('');
```

Велика ниска прочитана из текстуалног фајла дели се у листу ниски, тако што се реч *Directory* користи као сепаратор при раздвајању ниски карактера:

```
splitByWord(text, word) =
	text
		->strSplit(word)
		->tl()
		->map(\x # word: word+x);
```

Тако раздвојени сегменти текста обрађују се појединачно:

```
parseDirectory(fileContents) =
	if regexPos(fileContents, '^File .+') >= 0
	then
		fileContents
			->splitByWord('File ')
			->map(parseSegment)
			->mergeParsedFileSegments(getDirectoryName(fileContents))
	else
		"";
```

Њихова обрада подразумева да се поделе на мање целине, према фајловима на које се односе. Сваки се обрађује појединачно:

```
parseSegment(segment) =
	if isSuccessfulTest(segment)
	then parseSuccessful(segment)
	else if isErrorTest(segment)
		then parseError(segment)
		else parseFailed(segment);
```

У тексту се проверава да ли је у датом фајлу са тестовима пронађена грешка приликом тестирања, употребом регуларних израза:

```
isSuccessfulTest(segment) =
	!(regexPos(segment, '^File.*?OK') == -1);

isErrorTest(segment) =
	!(regexPos(segment, '^File.*error') == -1);
```

У зависности од исхода тестирања за разматрани тест-фајл, у HTML код се додаје одељак секција, одговарајућег изгледа описаног у `<style>` ознаци и понашања дефинисаног у `<script>` ознаци:

- Ако сви тестови пролазе:

```
parseSuccessful(segment) =
	buttonHtmlSuccessful(segment.strSplit('\n')) +
	divHtml(segment.strSplit('\n').tl().strJoin('\n'));
buttonHtmlSuccessful(lines) = html template
	<button class="collapsibleButton2"><# extractFileNameAndTestScore(lines)
	#></button>
<#>;
```

- Ако има тестова који не пролазе, али је добијен резултат:

```
parseFailed(segment) =
	buttonHtmlFailed(segment.strSplit('\n')) +
	divHtml(segment.strSplit('\n').tl().strJoin('\n'));


buttonHtmlFailed(lines) = html template
	<button class="collapsibleButton"><# extractFileNameAndTestScore(lines)
	#></button>
<#>;
```

- Ако није добијен резултат услед програмске грешке:

```
parseError(segment) = html template
	<button class="collapsibleButton"><# getFileName(segment) #>
	&#10060</button>
	<div class="collapsibleButtonText">
		<p>Program error! Check the test file for common errors.</p>
	</div>
<#>;
```

За све тест-фајлове чије је тестирање извршено приказује се број успешних и укупан број тестова, а то се из линија текста издваја на следећи начин:

```
extractFileNameAndTestScore(lines) =
	getFileName(lines.hd()) + ' : ' + getTestScore(lines.tl());

getFileName(line) =
	line
		->regexSearch("(File )(.*?(?=( \.*(FAIL|OK!))))")
		->hd()
		->trimFileName();

trimFileName(line) =
	line[strLastPos(line, '\\') + 1:];

getTestScore(lines) =
	lines
		->filter(lineHasScore)
		->map(extractTestScore)
		->leftAggregate(addTestScores, '0 / 0');

lineHasScore(line) =
	regexPos(line, "[0-9]+ \/ [0-9]+$") >= 0;

extractTestScore(line) =
	line
		->regexSearch("[0-9]+ \/ [0-9]+$")
		->hd();
```

```
addTestScores(s1, s2) =
	(s1.strSplit('/')[0].asInt() + s2.strSplit('/')[0].asInt()).asString()
	+ ' / '
	+ (s1.strSplit('/')[1].asInt() + s2.strSplit('/')[1].asInt()).asString();
```

У наставку се врши даља подела текста који се обрађује (на сличан начин као раније), овог пута по подацима о тестирању појединачних секција у оквиру фајла са тестовима:

```
divHtml(text) =
	text
		->splitByWord('Section')
		->map(addHtmlToSection)
		->map(trimTrailingBRTag)
		->strJoin('\n')
		->nestIntoHtml()
		->trimLeadingBRTag();
```

За сваку секцију се из текста издваја први ред информација, који садржи њен назив и однос успешних и извршених тестова, а затим и остатак, у којем се налазе подаци о тестовима који нису прошли:

```
addHtmlToSection(section) =
	addHtmlToFirstLine(section.strSplit('\n').hd()) +
addHtmlToRemainingLines(section.strSplit('\n').tl());

addHtmlToFirstLine(line) =
	html template
		<b><br><# sectionName(line) + sectionSuccessRatio(line) #></b><br>
	<#>;

sectionName(line) =
	regexSearch(line, "(^\s*Section )(.*?(?=(FAIL|OK!)))").hd();

sectionSuccessRatio(line) =
	regexSearch(line, "\\d+\\s/\\s\\d+").hd();

addHtmlToRemainingLines(lines) =
	lines
		->map(addHtmlToTestResultLine)
		->strJoin('\n');

addHtmlToTestResultLine(line) =
	if regexPos(line, "FAIL:") >= 0
	then
		html template
			<font color="red"><# line #></font><br>
		<#>
	else
		html template
			<# line #><br>
		<#>;
```

Након тога се уклањају сувишне ознаке за прелазак у нови ред:

```
trimTrailingBRTag(section) =
	section[:strLastPos(section, "<br>")];
```

Добијен HTML код додаје се као садржај раније направљених склопивих форми, уз уклањање сувишних ознака за прелазак у нови ред на почетку:

```
	nestIntoHtml(sections) = html template
		<div class="collapsibleButtonText"><p><# sections #></p></div>
	<#>;
	trimLeadingBRTag(text) =
		strReplace(text, "<br>", "", 1);
```

На крају, добијене форме се спајају и групишу према директоријумима:

```
	mergeParsedFileSegments(listOfParsedFileSegments, directoryName) = html template
		<fieldset><legend><# directoryName #></legend><#
mergeParsedSegments(listOfParsedFileSegments) #></fieldset>
	<#>;
```

4. Закључак

Може се закључити да је програмски језик *Wafl* значајно унапређен као резултат спроведеног тестирања. Имплементација многих постојећих функција основне библиотеке је поправљена и уведене су нове функције за којима се указала потреба. Подсистем за тестирање је унапређен могућношћу да за корисника направи извештај о тестирању који је у складу са потребама развојног процеса.

Побољшања су сигурно могућа у виду додавања још тестова. Поједине групе функција имају огроман потенцијал за експериментисање са разним комбинацијама улазних аргумената, нпр. оне које се тичу рада са регуларним изразима. Неке функције из стандардне библиотеке нису се нашле у процесу тестирања. Када њихова имплементација буде комплетна, допуна колекције тестова новим тест-случајевима би сигурно допринела унапређењу квалитета програмског језика *Wafl*. Постоје и неки проблеми откривени у току тестирања који нису решени.

Прављење извештаја у формату *HTML* би могло да буде унапређено аутоматизацијом, тј. да се при сваком тестирању позива *parser.wafl* и прави извештај, одмах доступан кориснику. Друга опција за постизање сличног ефекта је прављење скрипта, који би покретао тестирање и потом покретао обраду извештаја, како би кориснику уштедео драгоцено време.

Литература

1. Paul Hudak, **Conception, Evolution and Application of Functional Programming Languages**, *ACM Computing Surveys*, Vol. 21, No. 3, 359-411, 1989.
2. Gerard O'Regan, **Concise Guide to Software Testing**, *Springer*, 2019, 978-3-030-28494-7
3. Paul C. Jorgensen, **Software Testing – A Craftsman's Approach** (4), *CRC Press, Taylor & Francis Group*, 2014, 978-1-4665-6069-7
4. International Software Testing Qualifications Board (ISTQB), **ISTQB Glossary**, https://glossary.istqb.org/app/en/search/ (@23.05.2021)
5. Robert C. Martin, **Clean Code**, *Prentice Hall*, 2008, 978-0-13-235088-4
6. Саша Малков, **Wafl Tutorial**, http://poincare.matf.bg.ac.rs/~smalkov/wafl/tutorial/index.html (@23.5.2021)
7. TIOBE, **TIOBE Index**, https://www.tiobe.com/tiobe-index/ (@9.6.2021)
8. W3Schools, **How To Create a Collapsible**, https://www.w3schools.com/howto/howto_js_collapsible.asp (@16.7.2021)
9. Erlang, **Erlang Programming Language**, https://www.erlang.org/ (@12.9.2021)
10. Joe Armstrong, **Making reliable distributed systems in the presence of software errors**, http://erlang.org/download/armstrong_thesis_2003.pdf (@12.9.2021)
11. Scala, **The Scala Programming Language**, https://scala-lang.org/ (@12.9.2021)
12. FSharp, **F# Software Foundation**, https://fsharp.org/ (@12.9.2021)
13. Clojure, **Clojure**, https://clojure.org/index (@12.9.2021)
14. Kotlin, **Kotlin Programming Language**, https://kotlinlang.org/ (@12.9.2021)
15. Elixir, **The Elixir programming language**, https://elixir-lang.org/ (@12.9.2021)
16. Swift, **Welcome to Swift.org**, https://swift.org/ (@12.9.2021)
17. PurelyFunctional.tv, **Top 13 Functional Programming Languages**, https://purelyfunctional.tv/functional-programming-languages/ (@12.9.2021)
18. Саша Малков, WAFL Funkcionalni programski jezik za razvoj Veb aplikacija, http://poincare.matf.bg.ac.rs/~smalkov/files/wafl_old/wafl/Doc/Wafl.Implementacija.v0.9.3.1.pdf (@12.9.2021)