UNIVERZITET U NIŠU
ELEKTRONSKI FAKULTET

Sistem tipova podataka u Tajpskriptu

SAMOSTALNI ISTRAŽIVAČKI RAD

Studijski program: Računarstvo i informatika

Kandidat
dipl. inž. Lazar LJUBENOVIĆ

Mentor
doc. dr Ivan PETKOVIĆ

Niš, 2018.

Sadržaj

> Snaga Javaskripta je što u njemu možeš da uradiš sve. Slabost je što stvarno hoćeš.
>
> Reg Brejtvejt

Tajpskript (*TypeScript*) je jezik koji podseća na Javaskript i služi kao njegova zamena za pisanje velikih aplikacija. Dodaje opcione tipove i TC39 predloge koji još uvek nisu ušli u standard, ali su barem u fazi 3. Tajpskript se kompajlira (preciznije, „transpajlira") u čitljiv Javaskript.

1 Klasifikcija sistema tipova

Skup pravila na osnovu kojih se svojstvo poznato kao **tip podatka** dodeljuje različitim delovima kompjuterskog programa (promenljive, izrazi, funkcije, moduli, itd) naziva se **sistem tipova podataka**. Služe da bi se formalizovali koncepti koje programeri koriste za algebarske vrednosti, strukture podataka i druge komponente programa, kao što su „logička vrednost", „niz stringova" i „funkcija koja vraća broj".

Glavna namera postojanja tipova podataka jeste da minimizuje šanse za pojavu bagova u programima i pruži mogućnost za bolje isksutvo tokom razvoja softvera, ali se može koristiti i kao sredstvo za sprovođenje optimizacije. **Provera tipova** (*type checking*) je proces kojim se utvrđuje da li su ispoštovana pravila i ograničenja koja nameću tipovi podataka u nekom programskom jeziku. Postoji nekoliko osobina na osnovu kojih se može izvršiti podela načina na koje se vrši provera tipova.

1.1 Podele

Prvo pitanje koje se nameće jeste *kada* se dešava provera: da li tokom kompilacije ili tokom izvršenja programa. Za jezike koji se interpretiraju (Javaskript, Pajton, Rubi) ne postoji faza kompilacije pa je jedino mesto gde se može vršiti provera tipova – tokom izvršenja. Za ovakve jezike – u kojima se provera tipova sprovodi tokom izvršenja – se kaže da imaju **dinamičku** proveru tipova. Interpretatori za ovakve jezike obično svakom objektu dodeljuju oznaku tipa u kojoj se nalaze informacije o tipu. Ukoliko se provera vrši tokom kompjaliranja, radi se o **statičkoj** proveri. Ako program zadovolji uslove koje nameće analizator kod jezika sa statičkom proverom tipova, onda to daje garanciju da program tokom izvršenja neće da naiđe na određeni skup mogućih grešaka u vezi sa tipovima podataka.

Poznajući trenutak izvršenja provere tipova podataka, postavlja se pitanje *kako* se ona sprovodi. Kod jezika sa **jakim** tipovima podataka, nad kodom su dozvoljene samo one operacije koje imaju smisla u skladu sa semantikom jezika i one koje

neće dovesti do gubitka informacija. Za jezike koje ne nameću ova ograničenja kaže se da imaju **slabe** tipove podataka. Ne postoji jasna granica između jakih i slabih tipova, pa se ovi termini koriste uglavnom za poređenje jezika (koji ima „jače" a koji „slabije" tipove) ili za određene osobine jezika (za koju se može reći da je „jaka" a za koju „slaba").

Javaskript nema podršku za statičko definisanje tipova podataka, a uz to je i slabo tipiziran jezik. Promenljiva kojoj je dodeljen string može u sledećem redu da ima referencu na niz brojeva, a zatim da u nju bude upisana logička promenljiva. Kako je Javaskript jezik koji se interpretira, ni nema prilike da se ovo zabeleži kao greška prilikom kompajliranja; taj proces ne postoji. Međutim, čak i tokom izvršenja programa, ovakvo ponašanje koda se neće tretirati kao greška i interpretator će bez problema upisivati bilo koju vrednost u promenljivu.

Ovakvo ponašanje ne samo da može negativno da utiče na performanse (jer optimizator nije u stanju da predvidi koliko memorije je potrebno da se zauzme za promenljivu), već čini kod nečitljivim i podložnim greškama. Glavna prednost Tajpskripta je uvođenje tipova podataka u Javaskript, ali samo prilikom kompajliranja.

Tajpskript ni na koji način ne utiče na fazu interpretiranja i izvršenja koda. Tajpskript dolazi uz kompajler koji postojeći Tajpskript kod kompajlira u Javaskript. Dobijeni Javaskript kod ne sadrži nikakve informacije o tipovima podataka; ne postoje nikakvi uslovi koji će dovesti do greške ukoliko se promenljivoj dodeli drugačiji tip od navedenog. Umesto toga, Tajpskript ima zadatak da tokom kompajliranja utvrdi tipove i da se postara da developera obavesti o nastaloj nekonzistentnosti tako što emituje grešku.

Da bi se ovo postiglo, Tajpskriptov kompajler obavlja statičku analizu koda tokom koje zaključuje tipove i proverava da li su određeni operatori izvodljivi nad njima. Na primer, iako je se Javaskript interpretator ne buni za pokušaj sabiranja praznog niza i praznog objekta (`[] + {}`), Tajpskript će prijaviti grešku „Operator '+' cannot be applied to types '`undefined[]`' and '`{}`'".

1.2 Deklaracije

Kako bi bilo moguće da se koriste postojeće biblioteke pisane u Javaskriptu, u Tajpskript je uveden pojam *deklaracije*. Na primer, Tajpskript ne može automatski da bude svestan postojanja eksterne biblioteke učitane kroz `script` tag u `head` sekciji HTML dokumenta za koju se piše Tajpskript kod. Deklaracija promenljive ili nekog drugog simbola navodi se ili u posebnom fajlu sa ekstenzijom `.d.ts` ili direktno u postojeći fajl, uz prefiks `declare`.

Deklaracije za mnoge postojeće Javaskript biblioteke se mogu naći na GitHab repozitorijumu Majkrosoftovog projekta *DefinitelyTyped* koji održava zajednica. Svi paketi se mogu preuzeti sa npm repozitorijuma. Radi lakšeg pronalaska deklaracionih fajlova koristi se imenski prostor (*namespace*) `@types`, a za ime

deklaracionog paketa se koristi isto ime koje je dodeljeno samom paketu za koji se instaliraju deklaracije. Na primer, za paket `jquery` deklaracije se nalaze na `@types/jquery`.

Deklaracije se mogu dostaviti i zajedno sa samim paketom; ovaj način se preferira u situacijama kada je sam paket već napisan u Tajpskriptu. Koristeći opciju `-declaration`, Tajpskriptov kompajer može, pored `.js` fajlova, generisati i `.d.ts` fajlove, a iz `package.json` vrednosti pod ključem `types` može se pročitati putanja do ovog fajla. Ovako Tajpskriptov kompajler zna gde da pronađe potrebne deklaracije.

1.3 Definisanje tipova

Neki stariji jezici sa statičkim tipovima podataka od developera zahtevaju da se tipovi uvek definišu eksplicitno. Tajpskript je specifičan po tome što su se njegovi stvaraoci postarali da održe barijeru prelaska od Javaskripta na TajkSkript izuzetno niskom. To je postignuto skroz nekoliko osobina.

Uz odgovarajuća podešavanja kompajlera, migracija na Tajpskript se sastoji od samo jednog koraka: promeniti ekstenziju svim `.js` datotekama u `.ts`. Ovo je omogućeno činjenicom da su tipovi podataka **opcioni**. Svaki Javaskript kod može se formalno prevesti u Tajpskript kod promenom ekstenzije, a Tajpskriptov kompajer će u procesu kompajliranja od `.ts` datoteke generisati kod identičan polaznoj `.js` datoteci. Tajpskript je zbog ovoga s namerom zamišljen kao **nadskup** Javaskripta.

Za **eksplicitno** definisanje tipova koristi se postfiksna sintaksa koja je popularna među jezicima koji pružaju opcione tipove (npr. *EkšnSkript* i F♯). Ispred tipa se navode dve tačke (`:`).

```
const foo: string = "Hello"
```

Informacije o tipovima često nije neophodno ručno specificirati jer tipovi u Tajpskriptu mogu biti **implicitni**. Analizator će, gde god je to moguće, na osnovu analize koda zaključiti o kom je tipu reč. Na primer, ukoliko se konstanta inicijalizuje brojnom vrednošću, za konstantu se implicitno zaključuje da ima tip `number`. Slično, ako se iz funkcije koja prima dva stringa vrati vrednost dobijenja njihovom konkatenacijom pomoću polimorfnog operatora `+`, povratnoj vrednosti funkcije se implicitno dodeljuje tip ‘string‘ jer je upravo tog tipa rezultat primene ovog operatora nad dva stringa.

```
const foo = 2103
const concat = (a: string, b: string) => a + b
```

1.4 Strukturalni sistem tipova

Tradicionalni objektno-orijentisani jezici kao što su Java i C koriste sistem tipova u kome se podudaranje između tipova podataka vrši na osnovu njihovih eksplicitno navedenih *imena*. Ovakvi sistemi nazivaju se **nominalni**. Na primer, iako su definisane dve strukture istog oblika, njihova međusobna dodela neće biti moguća jer su imena tipova različita.

```
struct Foo { int baz; };
struct Bar { int baz; };

int main () {
  Foo foo;
  Bar bar;
  foo = bar; // error: no viable overloaded '='
}
```

S druge strane su jezici kod kojih je za određivanje privila u sistemu tipova presudna njihova *struktura*. Oni se nazivaju **strukturalni** sistemi, u koje spada Tajpskript.

```
interface Foo { baz: number }
interface Bar { baz: number }

let foo!: Foo
let bar!: Bar
foo = bar // ok
```

2 Konfiguracija

Skup fajlova koje će biti obrađene od strane kompajlera naziva se **kontekst kompilacije** (*compilation context*). Mada je sva podešavanja moguće proslediti direktno kao argumente CLI aplikacije, obično se, radi preglednosti, za ovo koristi konfiguracioni fajl. U pitanju je JSON5[1] fajl za koji se očekuje da bude imenovan `tsconfig.json` i da se nalazi u korenom direktorijumu projekta. Ukoliko to nije slučaj, putanja do konfiguracione datoteke se može proslediti kao argument opcije `--project` (`-p`).

Za uključivanje i isključivanje datoteka iz projekta koriste se tri polja u konfiguracionoj datoteci: `files`, `include` i `exclude`. Svi primaju niz stringova. Ukoliko se ne navede nijedna od ove tri opcije, biće uključene sve `.ts` datoteke koje se nalaze u korenom direktorijumu i poddirektorijumima, rekurzivno.

[1] Nadskup JSON specifikacije u kome se, između ostalog, dopuštaju viseći zarezi i komentari, a omeđivanje ključeva znacima navoda je opciono ukoliko ne sadrži karaktere koji bi doveli do dvosmislenosti; zbog ovoga dobija naziv „JSON za ljude”.

Da bi se naznačilo da datoteka ne pripada projektu, potrebno je dodati je u niz `exclude`. Pored putanja do datoteka, `exclude` opcija prepoznaje i imena direktorijuma i glob[2] šablone.

Slično, `include` opcija se koristi da bi se ograničio opseg projekta. Takođe razume putanje do datoteka, direktorijuma i blobove. Na primer, sve izvorne datoteke se često nalaze u direktorijumu `src`. Opcija `files` je specifičnija od opcije `include` jer prima samo putanje do datoteka i uglavnom se koristi za manje projekte.

Ponašanje sistema tipa podatka i način na koji će Tajpskript generisati izlaznu datoteku ili datoteke definiše se u sklopu objekta `compilerOptions`. Ovaj ključ prima objekat od preko osamdeset podešavanja. U nastavku će biti iznesene samo najznačajnije.

2.1 Izbor ES verzije

Osim što uvodi tipove podataka u Javaskript, Tajpskript služi i za izbor standarda koji aplikacija treba da podrži. Kod je moguće pisati ne samo u tekućoj verziji standarda, već i koristiti predloge koji još uvek nisu ušli u standard.

Kako ne podržavaju svi brauzeri sve TC39 predloge, autor treba da odluči po kom standardu treba da bude napisan emitovan kod. Ovo se određuje poljem `target`.

Najstariji podržani standard je `es3`, i ovo je podrazumevana vrednost za `target`. Moguće je izabrati i konkretne standarde `es5`, `es2015`, `es2016`, `es2017` i `es2018`. Ukoliko developer ne želi da se kod adaptira ni za jedan standard već da ostane u izvornom obliku (naravno, bez tipova), onda se koristi vrednost `esnext`; ovo znači da će izlazni kod sadržati Javaskript koji još uvek nije postao standard, pa postoji dobra šansa da kod bude nevalidan u nekim brauzerima.

2.2 Moduli

Da bi Tajpskript zadovoljio široke potrebe autorā (v. **??**), podržava više načina da generiše module. Opcija `module` prima jednu od sledećih vrednosti: `None`, `CommonJS`, `AMD`, `System`, `UMD`, i `ES2015`.

Ovo polje je usko povezano sa poljem `target`; `ES2015` je moguće koristiti samo ako je `target` podešeno na `es5` ili niže. S druge strane, jedino ako se izabere `AMD` ili `System` je moguće definisati opciju `outFile` kojom se podešava ime fajla u kojem će se naći izlazni kod.

[2] Glob je string kojim se na koncizan i čitljiv način definiše skup datoteka i/ili direktorijuma; koristi specijalne simbole kao `*`, `**` i `?` da bi se zadale određene „komande”. Primeri: `*.txt`, `node_modules/**/*`, `*.tsx?`.

2.3 Globalni tipovi

Osim jezičkih konstrukcija, novi ES standardi u jezik uvode i nove globalne promenljive ili nove metode i svojstva nad postojećim prototipovima. Da bi se Tajpskript kompajlirao bez greške i da bi pružio podršku u vidu tipova podataka za sve metode, autor mora da naglasi koje tipove ili koje grupe tipova želi da vidi globalno dostupne, odnosno za šta očekuje da postoji tokom izvršenja koda u okruženju za koje piše softver.

U ovu svrhu se opciji `lib` prosleđuje niz stringova. Pored celih standarda (od `ES5` do `ES2018`), mogu se uključiti i `ESNext` (kojim se pokrivaju deklaracije za predloge koji još uvek nisu postali standard), `WebWorker` (deklaracije za pokretanje koda u okviru veb-vorkera), kao i određene deklaracije iz standardā pojedinačno: na primer, `ES2015.Core` (osnovni skup funkcionalnosti), `ES2015.Promise` (deklaracije za objekte tipa `Promise` iz 2015), `ES2016.Array.Include` (dodata metoda `include` nad prototipom globalnog objekta `Array`), `ES2018.Promise` (deklaracije za novi `Promise` gde je uključena metoda `finally`), itd.

2.4 Emitovanje koda

Sem izbora načina na koji se kod parsira i ciljne verzije standarda, moguće je definisati i *šta* treba da se emituje.

Ukoliko je Tajpskript potrebno koristiti samo da bi se proverili da li su zadovoljeni tipovi podataka, postavlja se fleg `noEmit`; ovime kompilator neće generisati nijedan fajl ali će izvršiti sve provere nad kodom o tome da li bi se kompilacija završila uspešno ili ne. Slično, fleg `emitDeclarationOnly` se koristi da bi se generisale samo `.d.ts` datoteke – izlaz neće biti kod koji je moguće pokrenuti već će biti kod kojim je moguće obećati Tajpskriptu da postoji neki kod. Ovaj način kompilacije se koristi kada se `.js` datoteke generišu od strane nekog eksternog alata kao što je Babel.

Ukoliko se u kodu koriste određene konstukcije koje nisu dostupne u Ekmaskript verziji koja je navedena u `target` opciji, Tajpskript može emitovati neke pomoćne funkcije kojim premošćava njihov nedostatak, a koje imaju isti ili dovoljno sličan efekat na kod kao konstrukcija iz novog standarda koju developer koristi u kodu. Na primer, definisanje klase koja nasleđuje neku drugu klasu koristi ključne reči `class` i `extend`, ali ES5 ne podržava nijednu od njih. Iako je klasu lako imitirati funkcijom, nasleđivanje nije tako trivijalno, pa kompilator dodaje pomoćnu funkciju `__extends` koja prima dva argumenta: izvedenu i osnovnu klasu. Slično, ukoliko se koriste generatorske funkcije, umeće se pomoćna funkcija `__generator`, a ukoliko se koriste ključne reči `async` i `await` onda se pored `__generator` umeće i `__awaiter`.

Potencijalni problem leži u tome što se ove funkcije, ukoliko se koriste, definišu u svakom fajlu ponaosob. Nad velikim projektima ovo može da bude puno bespotrebno ponovljenog koda. Zato je uz verziju 1.5 dodat je fleg `noEmitHelpers`.

Kada je ova opcija uključena, Tajpskript i dalje generiše *pozive* ovih pomoćnih funkcija, ali ne i njih same.

Međutim, sada je na developeru da definiše ove pomoćne funkcije – Tajpskript podrazumeva da su globalno definisane. Kako developer ne bi morao ručno da ih definiše, a kako ne bi bile ni generisane u svakoj datoteci, u verziji 2.1 dolazi fleg `importHelpers` sa kojim će generisani kod importovati potrebne module iz biblioteke `tslib`. Ova biblioteka je dostupna na npm repozitorijumu i dovoljno je da je developer instalira u projekat kao zavinost da bi generisani kod radio.

Greške prilikom kompilacije Tajpskript koda se mogu svrstati u dve grupe. Jedna su greške zbog kojih ni Javaskript ne bi bio sintaksno ispravan, i zbog njih se kompilacija uvek obustavlja neuspešno, bez emitovanja datoteka. Drugu grupu čine semantičke greške u vidu nelegalnih operacija u vezi s tipovima podataka – iako je dodela stringa broju nemoguća, ovo je moguće kompajliarti u validan Javaskript. Postavkom flega `noEmitOnError`, ponašanje prilikom nailaska na drugi tip greške se može promeniti, tj. moguće je zabraniti da se emituju `.js` datoteke sve dok ima bilo kakvih grešaka u `.ts` kodu. Ova opcija se obično ostavlja isključena prilikom migracije sa Javaskript projekta na Tajpskript, ali i prilikom developmenta – nekada je korisno da se pokrene kod iako nije u potpunosti ispravan, kako bi se nešto brzo testiralo.

2.5 Lintanje koda

Iako za to postoji poseban alat pod nazivom `tslint`, i sam Tajpskript kompajler nudi neka osnovna podešavanja u vezi s lintanjem koda. Na primer, ukoliko se uključi fleg `noFallthroughCasesInSwitch`, kompilator će prijaviti grešku ukoliko naiđe na `case` segment koji se pretapa u sledeći (odnosno ne završava se sa `break`, `return` ili `throw`).

Flegovi `noUnusedLocals` i `noUnusedParameters` služe da bi autor koda bio obavešten o tome da postoje definicije lokalnih promenljivih ili konstanti, odnosno parametara funkcije, a da su one neiskorišćene u ostatku koda. Ovakve pojave obično znače da postoji logička greška u kodu ili da postoji zaostatak iz neke ranije verzije koda.

2.6 Strogi režim

Kao što je već rečeno, Tajpskript je nadskup Javaskripta i zamišljen je tako da migracija sa Javaskripta bude izuzetno jednostavna. Međutim, za projekte koji se otpočetka pišu u Tajpskriptu (ili za projekte koji su u potpunosti migrirani), često se javlja potreba za strožim proveravama.

Na primer, u slučaju da kompilator ne može da zaključi o kom je tipu reč, automatski mu se dodeljuje univerzalni tip `any`. Ovakvo ponašanje je korisno prilikom migracije, ali zapravo može dovesti do greške prilikom izvršenja koda

ukoliko se autor nije postarao da vodi računa o tipu takvog simbola. Kako bi se ovakve situacije izbegle, koristi se fleg `noImplicitAny`.

Postoji čitav skup opcija sličnih ovoj: `noImplicitAny`, `noImplicitThis`, `alwaysStrict`, `strictNullChecks`, `strictFunctionTypes` i `strictPropertyInitialization`. Navedena lista je vezana za verziju 3.0, ali se može promeniti u budućim verzijama. Kako autor koji želi da provera tipova bude što stroža ne bi morao da prilikom svakog ažuriranja verzije Tajpskripta vodi računa o tome da istraži novododata podešavanja i uključi nove flegove, postoji omni-opcija `strict` kojom se uključuju svi flegovi koji se tiču strogog režima.

3 Tipovi

U Javaskriptu, podatak koji nije objekat i nema nijednu metodu naziva se `primitivni tip`. Ima ih šest: `string`, `number`, `boolean`, `null`, `undefined` i `symbol`. Za svaki od ovih tipova postoji i odgovarajući statički tip u Tajpskriptu, sa istim imenom.

```
const aNumber: number = 2103
const aString: string = 'hello'
const aBoolean: boolean = true
```

Kao kontrast primitivnim tipovima, svi neprimitivni tipovi pripadaju tipu `object` (od verzije 2.2).

Osim primtivnih tipova, Tajpskript nudi i neke ambijentalne tipove, u zavinosti od izabranih opcija kroz polje `lib` u konfiguracionom fajlu. Ambijentalni tipovi vezani za postojeće globalne Javaskript objekte, što znači da postoje tipovi poput `Object` i `Function`. Na primer, neke od metoda definisane nad tipom `Object` su `toString (): string` i `hasOwnProperty (v: string): boolean`. Kada se kaže da je u JavaSkritu sve objekat, misli se na to da se u svakom prototipskom lancu na vrhu nalazi `Object`, što znači da su metode kao `toString` i `hasOwnProperty` dostupne svakoj vrednosti, bilo da se radi o primitivnim ili neprimitivnim tipovima.

Prema tome, `Object` opisuje ono što je zajedničko za svaki objekat u Javaskriptu (uključujući i primitivne vrednosti kao što su brojevi); `object` odgovara užem skupu jer isključuje pritmivne tipove.

Za preciznije definisanje objekata koristi se posebna sintaksa, koja podseća na inicijalizaciju objekata. Unutar para vitičastih zagrada navode se ključevi za koje se očekuje da postoje, a zatim se, posle dvotačke, navodi njihov tip. Ova lista se odvaja zarezima.

```
const anObject: { foo: number, bar: string } = {
  foo: 2103,
  bar: 'hello',
```

```
}
```

Počev od verzije 1.4, moguće je definisati alijase za tipove. Ovime je omogućeno da se tipovi lakše koriste kroz program i da se međusobno referenciraju.

```
type MyObject = { foo: number, bar: string }
const anObject: MyObject = { foo: 2103, bar: 'hello' }
```

Svojstvo u definiciji strukture objekta se može označiti kao opcioni pomožu simbola `?` koji se dodaje ispred dvotačke.

Od verzije 2.0 se ispred imena svojstva može dodati modifikator `readonly`. Ukoliko on postoji, dodela vrednosti tom svojstvu je moguća samo prilikom inicijalizacije.

Mada se za tipiziranje funkcija može koristiti tip `Function`, ovo je retko dovoljno jer ne pruža uvid u to koliko argumenata funkcija ima, kog su oni tipa i koja im je povratna vrednost. Za preciznije definisanje tipa funkcije koristi se posebna sintaksa koja podseća na streličastu funkciju u Javaskriptu. Argumenti funkcije su oblika `arg: Tip`, razdvojeni zarezom i omeđeni oblim zagradama. Desno od ove liste se, nakon debele strelice (`=>`), daje tip povratne vrednosti.

```
type MyFn = (foo: number, bar: string) => boolean
const myFn: MyFn = (foo, bar) => foo > 21 && bar.length < 3
```

Tipovi argumenata i povratne vrednosti se mogu definisati i prilikom same definicije funkcije.

```
const myFn = (foo: number, bar: stirng): boolean => {
  return foo > 21 && bar.length < 3
}
```

U gornjem primeru bi tip `boolean` kao tip povratne vrednosti mogao i sâm analizator da zaključi na osnovu poznavanja tipova koje vraćaju operatori `&&`, `>` i `<`, pa ga nije neophodno definisati. Prednost definisanja tipova jeste što predstavlja dokumentaciju čitljivu ljudima, pa čovek može brže ustanoviti koji je povratni tip (ne mora da analizira telo fukcije, već je dovoljno da pročita samo potpis). Eksplicitno navođenje tipa povratne vrednosti je korisno i zbog provere tipa povratne vrednosti u samom telu funkcije.

Osim streličastih funkcija, ovaj način se koristi i kod klasične definicije funkcije pomoću ključne reči `function`.

```
function myFn (foo: number, bar: string): boolean {
  return foo > 21 && bar.length < 3
}
```

Prilikom provere da li je funkcija pozvana ispravno, pored tipova argumenata posmatra se i njihov broj. Javaskript nema ograničenje po pitanju broja argumenata sa kojima se funkcija može pozvati; ukoliko postoje argumenti viška, oni se igrnoišu, a ukoliko se neki argumenti ne dodele, za njihovu vrednost se uzima `undefined`.

U Tajpskriptu se na broj argumenata može uticati na dva načina. Proizvoljan broj argumenata zdesna se može označiti kao opcioni, dodavanjem simbola `?` posle simboličnog imena argumenta. Svi opcioni argumenti moraju biti uzastopni i jedan od njih mora biti poslednji. Ovime je sprečeno da se obavezan argument nađe posle opcionog. S druge strane, broj argumenata se može povećati na beskonačnost ukoliko se ispred simboličnog imena poslednjeg argumenta doda token `...`.

Na primer, globalno dostupna funkcija `parseInt` kao prvi parametar prima string koji treba parsirati, a drugi parametar, koji je opcioni, predstavlja brojnu osnovu.

```
declare function parseInt (s: string, radix?: number): number
```

Metoda `push` definisana nad prototipom globalnog objekta `Array` prima proizvoljan broj argumenata – to su elementi koje treba dodati na kraj niza.

```
interface Array<T> {
  push (...items: T[]): number;
}
```

3.1 Unija i presek

Čest slučaj je da se nekoj promenljivoj mogu dodeliti može dodeliti vrednost koja može biti različitog tipa. Ovaj slučaj se u Tajpskriptu iskazuje pomoću **unija**. Mogući tipovi se navode razdvojeni simbolom `|`.

```
const a: number | string = 2103
const b: number | string = 'hello'
```

Kada se pristupa svojstvima sa unije, moguće je pristupiti samo onim koji postoje nad oba tipa. U opštem slučaju je ovo `Object`, ali u zavinosti od tipova koji se nalaze u uniji ih može biti više. Na primer, i nizovi i stringovi imaju svojstvo `length`.

3.2 Interfejsi i klase

Drugi način za definisanje oblika objekata su interfejsi. Definišu se ključnom rečju `interface`, nakon koje se navodi ime a potom lista parova oblika `ime: Tip`. Glavna sintaksna razlika liste jeste što se umesto zareza za razdvajanje parova mogu koristiti tačka-zarez ili novi red.

```
interface User {
  name: string
  age: number
}
```

Na sličan način se definišu i klase.

```
class User {
  name: string
  age: number
}
```

Mada se mogu koristiti na sličan način kao interfejsi, suštinska razlika između klasa i interfejsa je to što klasa postoji tokom izvršenja programa, a interfejs je samo tip podatka koji služi Tajpskriptovom kompilatoru za proveru. Interfejs ni na koji način nije odražen u kodu tokom izvršenja, pa je nemoguće koristiti operatore kao `instanceof` da bi se proverilo da li objekat implementira neki interfejs.

3.3 Osnovni tipovi

Pored šest tipova izvedenih iz primitivnih tipova koje opisuju Javaskript objekte i jednog tipa za sve neprimitivne vrednosti, Tajpskript definše još četiri osnovna tipa: `unknown` (od 3.0), `any`, `void` i `never` (od 2.0).

Jedan od glavnih dodataka koji dolaze uz verziju 3.0 jeste **tip `unknown`**. Pomoću `unknown` se opisuju simboli za koje je tip nepoznat, odnosno za simbole za koje se ne može garantovati kakva će se vrednost naći u njima. Tip `unknown` se najčešće koristi za dinamički učitane podatke, kao što su odgovori sa servera. `unknown` je **vršni tip** (*top type*). Vršni tip je tip kojem se može dodeliti bilo koji drugi tip, ali se on ne može dodeliti nijednom drugom tipu.

Tip `any` je **univerzalni tip**. Koristi se za opis tipa promenljivih za koje tip nije od važnosti. Najčešće se koristi tokom migracije ili kada bi pisanje tipa oduzelo previše vremena, a učinak bi bio previše mali. Na primer, ako za neku biblioteku ne postoje deklaracije, programer može najjednostavnije deklarisati globalnu promenljivu kao `declare const foo: any`. Sa ovime, pokušaji pristupa `foo` neće prouzrokovati Tajpskript grešku; s druge strane, kako Tajpskript ne zna o kom je tipu reč, sve operacije nad njime će bit dozvoljene tokom kompajliranja – Tajpskriptova statička analiza se u ovom slučaju ne može koristiti kao garancija da neće doći do greške tokom izvršenja programa.

Svaka funkcija u Javaskriptu, pod uslovom da se uspešno okonča njeno izvršenje (nema beskonačne petlje i ne dođe do greške pre kraja funkcije), mora da vrati neku vrednost. Povratna vrednost navodi se u istom redu posle ključne reči `return`. Ukoliko se `return` ne navede (ili se kontrolom toka zaobiđe), funkcija se završava kada se izvrši ceo blok koji predstavlja njeno telo. U tom slučaju se implicitno iz funkcije vraća vrednost `undefined`.

Međutim, Tajpskript ipak uvodi **tip `void`** koji se koristi za povratne vrednosti funkcije koje ne vraćaju ništa (izostavljanjem ključne reči `return`). Ovime se naglašava da potrošač ne treba da očekuje nikakvu povratnu vrednost od funkcije ili metode, već da se njenim pozivom dešava neki sporedni efekat. Na ovaj način je moguće razlikovati slučaj kada je povratna vrednost funkcije ekspli-

citno `undefined` (na primer, kada se u nizu ne nađe traženi rezulta pozivom `Array#find`) i kada nju treba zanemariti (na primer, `Array#forEach` samo implicitno vraća `undefined`, a sporedni efekat se definiše funkcijom koja se prosleđuje kao argument).

3.4 Čuvari tipova

Kako je Tajpskript jezik koji se oslanja na Javaskript, mnoge osobine Tajpskripta su prouzrokovane obrascima i čestim šablonima koje developeri koriste dok pišu JavaScript kod.

Na primer, često se na osnovu nekog svojstva utvrđuje o kom tipu objekta je reč.

```
const pointOnPlane = { x: 1, y: 2 }
const pointInSpace = { x: 9, y: 8, z: 7 }

function getHalfPoint (p) {
  if ('z' in p) return { x: p.x / 2, y: p.y / 2 }
  else return { x: p.x / 2, y: p.y / 2, z: p.z / 2 }
}
```

Međutim, utvrđivanje tipova može da bude jako kompleksno, pa u praksi postaje nemoguće utvrditi o kom je tipu reč – barem ne automatskom statičkom analizom koju Tajpskript sprovodi.

```
interface PlanePoint { x: number, y: number }
interface SpacePoint { x: number, y: number, z: number }

function getHalfPoint (p: PlanePoint | SpacePoint): PlanePoint
                                                  | SpacePoint {
  if ('z' in p) return { x: p.x / 2, y: p.y / 2, z: p.z / 2 }
  else return { x: p.x / 2, y: p.y / 2 }
}
```

U prethodnom primeru, verzija 2.6 Tajpskripta prijavljuje grešku na pretposlednjoj liniji prilikom pristupa `p.z`, sa greškom „Property ‘z’ does not exist on type ‘`PlanePoint`’”. Zaista, pošto `p` može da ima bilo koji od dva navedena tipa, u slučaju da se pristupa `p.z` nad tipom `PlanePoint`, dolazi do greške. Kompilator ne može da zaključi da je uslovom `'z' in p` zapravo napravljena razlika između tipova.

Da bi se ovaj čest način pisanja koda u Javaskriptu podržao, Tajpskript u verziji 1.6 dodaje mogućnost da autor definiše funkciju koja vraća `true` ili `false`, ali kao tip ima defisan **čuvar tipa** (*type guard*). Čuvari se pišu u formi `x is T`, gde je `x` deklarisan parametar u potpisu funkcije, a `T` je bilo koji tip.

```
function isSpacePoint (p: PlanePoint
                        | SpacePoint): p is SpacePoint {
```

```
  return 'z' in p
}
```

Ako se `'z' in p` iz funkcije `getHalfPoint` iz primera zameni pozivom funkcije-čuvara `isSpacePoint(p)`, Tajpskript više ne prijavljuje grešku; sada može da zaključi da je promenljiva `p` u pozitivnoj grani tipa `PlanePoint`, a u negaitvnoj grani ono što ostaje kada se iz `PlanePoint | SpacePoint` odbaci `PlanePoint`, dakle `SpacePoint`. Naravno, ovo se oslanja na to je autor dobro definisao čuvara.

U određenim delovima koda, Tajpskript može implicitno da zaključi da se radi o čuvaru i da na taj način poboljša tipove podataka na mestima u kodu gde se dešava grananje. Iz verzije u verziju je ovih tačaka u kodu sve više; najpre samo u uslovima kod `if` konstrukcije nad pozitivnom i negativnom granom i kod ternarnog operatora, zatim u `switch` naredbama, da bi se na kraju došlo do toga da se radi potpuna analiza kontrole toka gde su uključeni i rani izlasci iz funkcija i operatori kao `throw`. Sem toga, vremenom se sve više načina pisanja koda prepoznaje kao čuvar.

Dva osnovna i najstarija načina za definisanje čuvara jesu korišćenje `typeof` i `instanceof` operatora.

```
function doSomething (n: string | number): number {
  if (typeof n == 'string') { /* n is of type string here */ }
  else { /* n is of type number here */ }
}
```

Tipovi *null* i *undefined* se iz tipa mogu odstraniti klasičnim poređenjem sa vrednošću `null`.

```
function inc (n: number | null) {
  return n == null ? 0 : n + 1
}
```

Počev of verzije 2.7, operator `in` se takođe može koristiti kao čuvar, pa se sa ovim dodatkom primer koda za funkciju `getHalfPoint` uspešno kompajlira jer se na osnovu `'z' in p` može napraviti jasna razlika između dva tipa iz unije.

3.5 Tip `never`

Uz verziju 2.0 dolazi tip `never`. U pitanju je primitvni tip koji predstavlja tip vrednosti koja se nikad neće dobiti. `never` je pod-tip svakog tipa, a nijedan tip nije pod-tip tipa `never`, osim samog tipa `never`. Ovaj tip se prirodno javlja u delovima koda koji nisu dosegljivi, bilo zbog Tajpskriptove analize tipova podataka ili zbog same prirode Javaskripta.

Na primer, funkcije koje se nikad ne završe imaju kao povratni tip `never`. Ovo se može postići beskonačnom petljom ili bezuslovnim bacanjem izuzetka unutar tela funkcije. Sličan tip za povratnu vrednost funkcije je `void`, ali se on koristi za

funkcije čije se izvršenje okonča, ali se iz nje ništa ne vrati eksplicitno (odnosno vrati se implicitni `undefined`).

Drugi čest slučaj gde se ovaj tip javlja jeste prilikom ispitivanja tipa simbola pomoću čuvara (bilo implicitnih ili eksplicitnih).

```
function doSomething (x: number | string): any {
  if (typeof x == 'string') return 1
  else if (typeof x == 'number') return 2
  else { /* x is of type never */ }
}
```

3.6 Preklapanje funkcija

U jezicima sa jakim tipovima podataka se često dozvoljava da funkcije imaju isto ime, a da se na osnovu broja i tipa argumenata određuje koju od njih treba pozvati. Međutim, kako je Javaskript jezik sa izuzetno slabim tipovima podataka, ovaj način preklapanja funkcija nije moguć. Kod koji ispituje broj i topve argumenata mora da definiše autor.

Kako Tajpskript nije u stanju da uvek automatski zaključi o kom je tipu reč, i kako bi bilo potrebno previše menjati izvorni kod i izvoditi zaključke o tome koja je bila developerova namera (što se kosi sa ideologijom Tajpskripta), ovakav kod se i dalje mora pisati u telu funkcije, čak i u Tajpskriptu. Ipak, Tajpskript prepoznaje ovaj šablon i omogućuje da se dobro definišu tipovi ovakvih funkcija; ovo se zove **preklapanje funkcije** (*function overloading*, *function overloads*).

Na primer, iako je u telu funckije iz primera u prošlom odeljku jasna razlika između dva tipa, povratna vrednost je dvosmislena. Koji god tip da se prosledi u funkciju `getHalfPoint`, iako je on jasno statički definisan, rezultat će biti unija tipova.

```
function getHalfPoint (p: PlanePoint): PlanePoint
function getHalfPoint (p: SpacePoint): SpacePoint
function getHalfPoint (p) {
  if ('z' in p) return { x: p.x / 2, y: p.y / 2, z: p.z / 2 }
  else return { x: p.x / 2, y: p.y / 2 }
}
```

Ovako definisana funkcija je preklopljena sa dve deklaracije (prva dva reda). Deklaracija u definiciji funkcije se neće koristiti prilikom poziva funkcije, već samo prilikom provere tipova u samom telu funkcije. Na ovaj način je obezbeđeno da tip povratne vrednosti funkcije bude isti kao tip prosleđenog argumenta.

3.7 Generički tipovi

Generički tipovi omogućuju da se ista funkcija koristi sa više različitih tipova, pri čemu je moguće biti eksplicitan o kome se tipu radi. U prethodnom primeru su deklarisana dva potpisa-preklopa kojima se ovo omogućava, ali je istu funkciju moguće zapisati i kraće, korišćejnjem generičkih tipova prilikom definicije funkcije.

```
function getHalfPoint<T> (p: T): T { /* ... */ }
```

U gornjem isečku koda, dodata je tipska promenljiva `T` koja omogućava da se uhvati tip koji potršač funkcije prosledi (npr. `SpacePoint`) i da se ta informacija iskoristi u ostataku funkcije (bilo u telu ili u deklaraciji). Ovde se `T` koristi kao tip povratne vrednosti.

Međutim, na ovaj način definisana funkcija će zapravo dati grešku prilikom pokušaja pristupa u negativnoj grani uslova iz tela. Naime, iako se čuvarom obezbeđuje da je tip u pozitivnoj grani `SpacePoint`, u negativnoj grani se zna jedino da *nije* u pitanju `SpacePoint`. Ovo je zato što, kako je funkcija trenutno napisana, `T` može biti bilo šta.

Da bi se ograničio skup tipova koji se može koristiti kao `T`, koristi se format `<T extends U>`, pri čemu je `T` generički tip a `U` tip kojim se određuje kojem skupu tipova mora da pripada tip `T`; drugim rečima, `U` je tip takav da je `T` pod-tip `U`.

```
function getHalfPoint<T extends SpacePoint | PlanePoint>
  (p: T): T { /* ... */ }
```

Sada ne samo da je tip unutar tela funkcije ispravno definisan, već se i od potrašača zahteva da se za tip unese isključivo `SpacePoint` ili `PlanePoint` (ili tip koji je njihov podskup).

Kada se poziva funkcija sa generičkim argumentima, posle imena se navodi tip.

```
getHalfPoint<SpacePoint>({ x: 9, y: 8, z: 7 })
```

Međutim, TajpSkipt omogućava i drugačiji način da se funkcija pozove. Ukoliko je moguće zaključiti stvarni tip generičkog tipa na osnovu stvarnih argumenata prosleđenih funckiji, onda se specificiranje generičkog tipa može izostaviti – kompajler će sam popuniti prazninu.

```
getHalfPoint({ x: 9, y: 8, z: 7 })
```

Osim toga, ako se funkciji prosledi tip `SpacePoint | PlanePoint`, povratna vrednost će takođe biti `SpacePoint | PlanePoint`. Ovo je posledica toga što takva unija zadovljava `extends` uslov naveden u deklaraciji generičkog argumenta funkcije.

Pored metoda, i interfejsi i klase mogu biti generički. Na primer, `Array` je generički tip, pa se umesto kraćeg zapisa `T[]` može koristiti pun zapis `Array<T>`.

3.8 Uslovni tipovi

Veliki pomak u mogućnostima koje pruža Tajpskript načinjen je verzijom 2.8 uz koju dolaze uslovni tipovi. Uslovni tipovi omogućavaju autoru da bude izabran jedan od dva tipa na osnovu ispunjenja uslova postavljenim kao test veze sa `extends`. Imaju oblik `T extends U ? X : Y` – kada se `T` može dodeliti `U`, tada je tip `X` a inače je tip `Y`.

Kada je tip koji se proverava „go", odnosno nije upleten deo nekog tipa-omotača, tada se za uslovni tip kaže da je **distributivan**. Distribucija se vrši automatski nad tipovima koji su delovi unije. Na primer, za `T` se može proslediti `A | B`, pa se rezultat izraza `(A | B) extends U ? X : Y` dobija kao `(A extends U ? X : Y) | (B extends U ? X : Y)`.

3.9 Manipulacija tipovima

Tajpskript dolazi uz neke pomoćne tipove koji nemaju nikakav Javaskript ekvivalent, odnosno ne opisuju postojeće ambijentalne simbole. Oni služe da bi developeru omogućili da transformiše tipove, odnosno da manipuliše njima, kako bi se na osnovu postojećih dobili novi. Zaključno sa verzijom 3.1 ih ima deset.

- Za označavanje da su sva svojstva tipa `T` opcioni, koristi se tip `Partial<T>`.
- Obrnuto, `Required<T>` služi da se označi da su svi propertiji obavezni.
- `Readonly<T>` označava sve propertije kao `readonly`.

Neki generički tipovi definsisani su zahvaljujući uslovnim tipovima.

- `Exclude<T, U>` iz tipa `T` odbacuje tipove koji se mogu dodeliti tipu `U`. Na primer, `Exclude<A | B | C, C | D>` vraća tipa `A | B`. Definisan je kao `T extends U ? never : T`.
- Suprotno od `Exclude`, tip `Extract<T, U>` definisan je kao `T extends U ? T : never` i služi da bi se iz `T` izdvojili samo tipove koje je moguće dodeliti tipu `U`.

3.10 Primeri tipova iz `lib`

U ovom odeljku će biti prokomentarisano nekoliko značajnih primera iz `lib.*` deklaracionih datoteka koje Tajpskript koristi za ambijentalne deklaracije.

Metoda `filter` deklarisana je nad prototipom `Array` na sledeći način.

```
interface Array<T> {
  filter<S extends T>(callbackfn: (value: T,
                                   index: number,
```

```
                                    array: ReadonlyArray<T>,
                                  ) => value is S,
                        thisArg?: any): S[]

  filter(callbackfn: (value: T,
                      index: number,
                      array: ReadonlyArray<T>,
                     ) => any,
         thisArg?: any): T[]
}
```

Metoda je preklopljena sa dva potpisa; pošto se radi o deklaracionoj `.d.ts` datoteci, nema definicije funkcije pa su oba potpisa vidljiva potrošaču. Prilikom poziva metode, preklopljeni potpisi se obilaze redom koji su navedni, tj. traži se prvo poklapanje.

Zato, ukoliko se kao `callbackfn` prosledi čuvar koji tip `T` svodi na tip `S`, dobijeni rezultat će biti niz čiji su elementi tipa `S`. Ako se pak radi o običnoj funkciji, tip povratne vrednosti neće biti promenjen u odnosu na početni tip koju ima niz nad kojim se metoda poziva.

Osim deklaracija ambijentalnih simbola koji su dostupni tokom izvršenja programa, u `lib` datotekama se nalaze i neki pomoćni tipovi koji mogu poslužiti developeru da izvede jedan tip iz drugog.

Jedan takav tip je `NonNullable`. U pitanju je generički tip koji od prosleđenog argumenta sklanja `undefined` i `null`.

```
type NonNullable<T> = T extends null | undefined ? never : T;
```

Ukoliko je stvarni tip koji se prosledi umesto `T` neki od pod-tipova tipa `null | undefined` (tj. `null | undefined`, `null`, `undefined` ili `never`), onda se dobija tip `never`. Zaista, ako se od nečega što je obavezno *nullable* ukloni ta osobina, dobija se tip koji nikad ne može da ima vrednost. Ako stvarni tip *nije* neki od pod-tipova `null | undefined`, onda se dobija taj isti tip.

Na primer, za `NonNullable<number | string | null>` se radi o distributivnom uslovnom tipu, pa se rezultujući tip dobija na sledeći način.

```
(number | string | null) extends null | undefined ? never : T
  (number extends null | undefined ? never : T) |
  (string extends null | undefined ? never : T) |
  (null   extends null | undefined ? never : T)
    number | string | never
      number | string
```

Uopštenje ovog pomoćnog tipa je `Exclude`, koji prima dva argumenta `T` i `U` i iz `T` odbacuje `U`.

```
type Exclude<T, U> = T extends U ? never : T;
```