POGLAVLJE 4

# Šta je opseg vidljivosti promenljive?

Jedna od najosnovnijih paradigmi gotovo svih programskih jezika jeste mogućnost smeštanja vrednosti u promenljive i zatim učitavanja ili menjanja tih vrednosti. U stvari, mogućnost čuvanja vrednosti u promenljivama i njihovog učitavanja iz promenljivih jeste ono što određuje tekuće *stanje* programa.

Bez jednog takvog koncepta, program bi mogao da obavlja određene poslove, ali bi bio veoma ograničen i ne baš mnogo zanimljiv.

Uvođenje promenljivih u program nameće najzanimljivija pitanja koja ćemo sada razmotriti: gde te promenljive *žive*? Drugim rečima, gde se one čuvaju? I, što je najvažnije, kako ih naš program pronalazi kad mu zatrebaju?

Ta pitanja dovode do potrebe za dobro definisanim skupom pravila za čuvanje promenljivih na odgovarajućem mestu, kao i za kasnije pronalaženje tih promenljivih. Taj skup pravila nazvaćemo: *opseg vidljivosti* (engl. *scope*) promenljivih.

Ali, gde se i kako definišu pravila tog *opsega vidljivosti*?

## Teorija kompajlera

Možda je za nekoga očigledno, a za drugog iznenađujuće – zavisno od vašeg nivoa poznavanja raznih programskih jezika – ali uprkos činjenici da JavaScript pripada opštoj kategoriji „dinamičkih“ ili „interpretiranih“ jezika, to je zapravo kompajliran jezik. *Nije* unapred kompajliran, kao mnogi drugi tradicionalni jezici, niti su rezultati kompajliranja prenosivi na druge distribuirane sisteme.

Ipak, JS mašina obavlja veliki broj istih koraka, mada na složenije načine nego što smo toga uglavnom svesni, kao i svaki drugi tradicionalni kompajler jezika.

U postupku tradicionalnog kompajliranja jezika, uobičajeno je da blok izvornog koda, tj. vaš program, prolazi kroz tri koraka *pre* nego što se izvrši, a sve zajedno je poznato pod opštim imenom „kompajliranje“:

*Razlaganje na tokene / leksička analiza*

Deljenje grupe znakova na celine koje imaju značenje (u datom jeziku) i koje se zovu tokeni. Na primer, razmotrite program `var a = 2;`. Ovaj program bi verovatno bio podeljen na sledeće tokene: `var`, `a`, `=`, `2` i `;`. Beline između znakova možda će se smatrati tokenima, ali to nije obavezno i zavisi od toga da li one imaju u jeziku neko značenje ili ne.

Razlika između deljenja na tokene i leksičke analize suptilna je i akademskog nivoa, ali se zasniva na tome da li se ti tokeni obrađuju kao da opisuju određeno *stanje* ili ne. Jednostavno rečeno, ako pri tumačenju da li `a` treba smatrati zasebnim tokenom ili samo delom drugog tokena, sistem za razlaganje na tokene primenjuje pravila koja podrazumevaju i prepoznavanje stanja, onda bi *to* bila *leksička analiza*.

*Raščlanjivanje*

Raščlanjivanje (engl. *parsing*) jeste raspoređivanje niza tokena na stablo sa ugnežđenim elementima, koji uzeti kao celina predstavljaju gramatičku strukturu programa. To stablo se zove „AST“ (od engl. *Abstract Syntax Tree* – apstraktno sintaksno stablo).

Stablo za iskaz `var a = 2;` moglo bi da počne od čvora najvišeg nivoa, koji se zove `VariableDeclaration` i koji ima čvor potomak nazvan `Identifier` (čija vrednost je `a`), kao i još jednog potomka, `AssignmentExpression`, koji pak ima svog potomka po imenu `NumericLiteral` (a vrednost mu je `2`).

*Generisanje koda*

Postupak pretvaranja AST-a u izvršiv kôd. Ovaj deo se prilično razlikuje u zavisnosti od jezika, ciljne platforme itd.

Zato ćemo, umesto da se izgubimo među detaljima, samo reći da postoji način da se naš prethodno opisani AST za iskaz `var a = 2;` pretvori u skup mašinskih naredaba čiji je posao da *naprave* promenljivu koja se zove `a` (što podrazumeva rezervisanje memorije za nju itd.), i da zatim smeste određenu vrednost u `a`.

Pošto se detalji načina na koji mašina jezika upravlja resursima sistema nalaze dublje nego što ćemo mi kopati, uzećemo zdravo za gotovo da mašina jezika ume da pravi i skladišti promenljive prema potrebama.

Mašina JavaScripta obavlja poslove koji su znatno složeniji od opisana tri koraka – što važi i za većinu drugih kompajlera programskih jezika. Na primer, tokom postupaka raščlanjivanja i generisanja koda, svakako ima koraka za optimizovanje performansi pri izvršavanju koda, kao što je uklanjanje dupliranih elemenata itd.

Dakle, ovde slikam samo širokim potezima četkice. Ali mislim da ćete uskoro shvatiti zbog čega su bitni ti detalji koje *ipak* razmatramo, čak i ako je to samo na visokom nivou.

Prvo, budući da JavaScript mašine nemaju taj luksuz (kao kompajleri drugih jezika) da im je na raspolaganju dovoljno vremena da optimizuju kôd, JavaScript kôd se ne kompajlira unapred, u okviru postupka sklapanja izvršne verzije koda, kao u slučaju drugih jezika.

U mnogim slučajevima, JavaScript kôd se kompajlira jedva nekoliko mikrosekundi (ili još kraće!) pre izvršavanja koda. Da bi obezbedile najbolje performanse, JS mašine primenjuju sve moguće vrste trikova (kao što je JIT, odloženo kompajliranje ili čak kompajliranje „u letu“ itd.) koji su daleko izvan „opsega vidljivosti“ našeg razmatranja.

Jednostavnosti radi, reći ćemo samo da svaki delić JavaScript koda mora da bude kompajliran pre (uglavnom *neposredno* pre!) nego što se izvrši. Tako će program `var a = 2;` JS kompajler *prvo* kompajlirati, a zatim pripremiti za izvršavanje, obično odmah nakon toga.

# Opis opsega vidljivosti

Da bismo objasnili opseg vidljivosti, razmišljaćemo o njemu kao o razgovoru između više sagovornika. Ali, *ko* su učesnici tog razgovora?

## Podela uloga

Da bismo razumeli razgovor kojem ćemo uskoro prisustvovati, moramo znati uloge likova koji učestvuju u razgovoru povodom kompajliranja programa `var a = 2;`:

*Mašina jezika*
: Odgovorna je za kompajliranje od početka do kraja našeg JavaScript programa, kao i za njegovo izvršavanje.

*Kompajler*
: Jedan od Mašininih prijatelja; obavlja prljavi posao raščlanjivanja izvornog koda i generisanja mašinskog koda (videti prethodni odeljak).

*Opseg vidljivosti*
: Drugi Mašinin prijatelj; formira i održava pretraživu listu svih deklarisanih identifikatora (promenljivih) i stara se o poštovanju skupa striktnih pravila koja određuju kako su ti identifikatori dostupni tekućem kodu koji se izvršava.

Da biste *potpuno razumeli* kako radi JavaScript, morate početi da *razmišljate* na isti način kao što razmišlja Mašina (i njeni prijatelji), postavljati pitanja koja oni postavljaju i odgovarati na ta pitanja na isti način kao oni.

## Ja tebi, ti meni

Kada vidite program kao što je `var a = 2;`, najverovatnije pomislite da je to samo jedan iskaz. Međutim, to nije oblik u kojem ga vidi naša nova prijateljica Mašina. Ona zapravo vidi dva jasno razdvojena iskaza – jedan koji će Kompajler obraditi tokom postupka kompajliranja i drugi koji će Mašina obraditi tokom izvršavanja koda.

Sada ćemo opisati način na koji Mašina i njeni prijatelji pristupaju programu `var a = 2;`.

Kompajler će prvo obaviti leksičku analizu ovog programa da bi ga razložio na pojedinačne tokene, koje će zatim raščlaniti i raspodeliti na stablo. Ali, kad Kompajler stigne u fazu generisanja koda, on će naš program obrađivati donekle drugačije nego što možda pretpostavljate.

Razumna pretpostavka bi bila da će Kompajler proizvesti kôd čiji se rad može opisati sledećim pseudokodom: „Rezervisati memoriju za promenljivu, obeležiti je sa `a`, i zatim u tu promenljivu postaviti vrednost `2`“. Nažalost, to nije sasvim tačan opis.

Kompajler će umesto toga uraditi sledeće:

1. Kada naiđe na deklaraciju `var a`, Kompajler pita Opseg vidljivosti da li promenljiva `a` već postoji u njegovoj kolekciji promenljivih. Ako postoji, Kompajler zanemaruje deklaraciju i nastavlja dalje. U suprotnom, Kompajler zahteva da Opseg vidljivosti deklariše u svojoj kolekciji novu promenljivu čije je ime `a`.
2. Kompajler onda generiše kôd koji će Mašina kasnije izvršiti i koji obrađuje iskaz `a = 2`. Kôd koji Mašina izvršava prvo će pitati Mašinu da li je u kolekciji promenljivih tekućeg opsega vidljivosti dostupna neka promenljiva čije je ime `a`. Ako jeste, Mašina koristi tu promenljivu. Ukoliko nije, Mašina traži promenljivu *na drugom mestu* (videti odeljak „Ugnežđeni opsezi vidljivosti“ na strani 69).

Ako Mašina na kraju pronađe promenljivu, dodeljuje joj vrednost `2`. Ukoliko je ne pronađe, Mašina će prijaviti da je došlo do greške.

Ukratko: pri dodeljivanju vrednosti promenljivoj, odvijaju se dve akcije: prvo, Kompajler deklariše promenljivu (ako nije prethodno deklarisana) u tekućem Opsegu vidljivosti; drugo, pri izvršavanju koda, Mašina traži promenljivu unutar Opsega vidljivosti i dodeljuje joj vrednost, ako je tamo nađe.

## Terminologija kompajlera

Da bismo nastavili proučavanje, treba nam još malo kompajlerske terminologije.

Kada Mašina izvršava kôd koji je Kompajler pripremio za korak 2, mora da potraži promenljivu `a` kako bi videla da li je promenljiva deklarisana, a ta pretraga se odvija unutar Opsega vidljivosti. Ali, rezultat pretrage zavisi od načina na koji Mašina obavi pretragu.

U našem slučaju, kaže se da bi Mašina obavila LHS pretragu za promenljivu `a`. Druga vrsta pretrage zove se RHS.

Možda ste pogodili šta znače „L“ i „R“. Tim slovima počinju izrazi *lefthand side* (leva strana) i *righthand side* (desna strana).

Strana...čega? *Operacije dodeljivanja vrednosti promenljivoj.*

Drugim rečima, LHS pretraživanje se obavlja kada se promenljiva pojavljuje na levoj strani operacije dodeljivanja vrednosti, a RHS – kada se promenljiva pojavljuje na desnoj strani te operacije.

U stvari, budimo malo precizniji. RHS pretraga se ne razlikuje, u našem primeru, od običnog traženja vrednosti određene promenljive, dok LHS pretraga pokušava da pronađe samu promenljivu, tj. kontejner za vrednost, da bi joj se dodelila vrednost. Stoga, RHS ne znači samo po sebi *zaista* „desna strana operacije dodeljivanja vrednosti“, nego, tačnije rečeno, „suprotno od leve strane“.

Pošto je vaše poznavanje materije zasad još nedovoljno duboko (ali razumete engleski), mogli biste pomisliti da RHS u stvari znači *„retrieve his/her source (value)“* /učitati njegovu/njenu izvornu (vrednost)/ što bi uputilo na to da RHS znači „učitati vrednost promenljive ...“

Razmotrimo to detaljnije.

Kada kažem:

```
console.log( a );
```

referenca `a` je RHS referenca, zato što se ovde promenljivoj `a` ništa ne dodeljuje. Umesto toga, pretražujemo kolekciju promenljivih da bismo učitali vrednost promenljive `a`, koju treba proslediti funkciji `console.log(..)`.

Nasuprot tome:

```
a = 2;
```

U ovom slučaju, referenca `a` je LHS referenca, zato što nam zapravo nije važno koja je tekuća vrednost, nego samo želimo da pronađemo promenljivu kao cilj operacije dodeljivanja vrednosti `= 2`.

Značenje akronima LHS i RHS „leva/desna strana operacije dodeljivanja vrednosti“ ne mora da znači bukvalno „leva/desna strana operatora `=` za dodeljivanje vrednosti“. Pošto se vrednost može dodeliti na još nekoliko načina, bolje je da ga konceptualno zamislite kao: „Ko je odredište u operaciji (LHS)?“ i „Ko je izvor u operaciji (RHS)?“

Razmotrite sledeći program, koji sadrži i LHS i RHS reference:

```
function foo(a) {
    console.log( a ); // 2
}

foo( 2 );
```

Poslednji red – koji poziva funkciju `foo(..)` – predstavlja RHS referencu na `foo`, koja znači „Potraži vrednost `foo` i daj mi je“. Osim toga, pošto `(..)` znači da tu vrednost `foo` treba izvršiti, bilo bi bolje da je to zaista funkcija!

U ovom slučaju imamo neočiglednu, ali važnu operaciju dodeljivanja vrednosti.

Možda vam je u ovom primeru koda promakla implicitna operacija `a = 2`. Ona se izvršava kada se vrednost `2` prosledi kao argument funkciji `foo(..)`, pri čemu se vrednost `2` *dodeljuje* parametru `a`. Da bi se (implicitno) dodelila vrednost parametru `a`, obavlja se LHS pretraga.

Postoji i RHS referenca na vrednost `a`, a ta rezultujuća vrednost prosleđuje se funkciji `console.log(..)`. Funkciji `console.log(..)` potrebna je referenca za izvršavanje. Tu se odvija RHS pretraga radi pronalaženja objekta `console`, a zatim se razrešava njegovo svojstvo radi utvrđivanja da li taj objekat ima metodu koja se zove `log`.

I najzad, ovde možemo konceptualno zamisliti da se odvija LHS/RHS razmena u kojoj se vrednost `2` prosleđuje (putem RHS pretrage promenljive `a`) funkciji `log(..)`. Unutar interne implementacije funkcije `log(..)`, možemo pretpostaviti da ona ima parametre, među kojima se za prvi (možda nazvan `arg1`) obavlja LHS pretraga reference, pre nego što mu se dodeli vrednost `2`.

Možda biste došli u iskušenje da deklaraciju funkcije `function foo(a) {…` konceptualno zamislite kao običnu deklaraciju promenljive s dodeljivanjem vrednosti, kao što je `var foo` i `foo = function(a){….` Kada tako uradite, izgledalo bi privlačno da tu deklaraciju funkcije zamislite kao da podrazumeva LHS pretragu.

Međutim, suptilna ali bitna razllika je to što Kompajler obrađuje i deklarisanje promenljive i definisanje njene vrednosti tokom postupka generisanja koda, tako da kada Mašina izvršava kôd, nije potrebna nikakva obrada da bi se funkciji `foo` „dodelila“ vrednost. Iz tog razloga, deklaracija funkcije se ne može posmatrati kao dodeljivanje vrednosti sa LHS pretragom u obliku u kojem ih ovde razmatramo.

## Razgovor između Mašine i Opsega vidljivosti

```
function foo(a) {
    console.log( a ); // 2
}

foo( 2 );
```

Zamislimo prethodnu razmenu (u kojoj se obrađuje navedeni blok koda) kao nekakv razgovor. Taj razgovor bi se odvijao otprilike ovako:

Mašina: Zdravo Opseže, imam jednu RHS referencu `foo`. Da li si nekad čuo za nju?

Opseg vidljivosti: Jesam. Kompajler ju je deklarisao pre samo jedne sekunde. To je funkcija. Evo, izvoli.

Mašina: Odlično, hvala! OK, izvršavam `foo`.

Mašina: Zdravo, Opseže, imam jednu LHS referencu `a`. Da li si nekad čuo za nju?

Opseg vidljivosti: Jesam. Kompajler ju je nedavno deklarisao kao formalni parametar finkcije `foo`. Evo, izvoli.

Mašina: Kao i uvek, ti si mi velika pomoć, Opseže. Hvala još jednom. A sada, vreme je da dodelim `2` promenljivoj `a`.

Mašina: Zdravo, Opseže, izvinjavam se što te opet gnjavim. Treba mi RHS pretraga za `console`. Da li si nekad čuo za to?

Opseg vidljivosti: Nema problema, Mašino, to je ono što radim po ceo dan. Da, imam `console`. To je ugrađena funkcija. Evo, izvoli.

Mašina: Savršeno. Tražim `log(..)`. OK, super, to je funkcija.

Mašina: Zdravo, Opseže. Možeš li mi pomoći s RHS referencom `a`? Mislim da je se sećam, ali ipak želim da proverim još jednom.

Opseg: U pravu si, Mašino. Ista promenljiva, ništa se nije promenilo. Evo, izvoli.

Mašina: Super. Prosleđujem vrednost promenljive `a`, što je `2`, funkciji `log(..)`.

...

## Kviz

Proverite znanje koje ste dosad stekli. Igrajte ulogu Mašine i obavite „razgovor“ s Opsegom vidljivosti:

```
function foo(a) {
    var b = a;
    return a + b;
}

var c = foo( 2 );
```

1. Otkrijte sve LHS pretrage (ima ih 3!).
2. Otkrijte sve RHS pretrage (ima ih 4!).

Odgovore na pitanja iz kviza potražite u sažetku poglavlja!

## Ugnežđeni opsezi vidljivosti

Rekli smo da je opseg vidljivosti skup pravila za traženje promenljivih po imenima njihovih identifikatora. Međutim, obično ima više opsega vidljivosti koje treba pregledati.

Isto kao što je blok jedne funkcije ugnežđen unutar bloka druge funkcije, tako su i jedni opsezi vidljivosti ugnežđeni unutar drugih. Zbog toga, ako određenu promenljivu ne uspe da pronađe unutar tekućeg opsega vidljivosti, Mašina pretražuje prvi sledeći spoljašnji i tako nastavlja sve dok ne pronađe promenljivu ili dok ne dođe do spoljašnjeg opsega najvišeg nivoa (poznat i kao globalni opseg vidljivosti).

Razmotrite sledeće:

```
function foo(a) {
    console.log( a + b );
}

var b = 2;

foo( 2 ); // 4
```

RHS referenca `b` ne može se razrešiti unutar funkcije `foo`, ali može u opsegu koji okružuje tu funkciju (što je, u ovom slučaju, globalni opseg vidljivosti).

Zato kada bismo se vratili na razgovor između Mašine i Opsega vidljivosti, čuli bismo sledeće:

Mašina: „Zdravo, Opseže vidljivosti funkcije `foo`, da li si čuo za `b`? Imam jednu RHS referencu takve promenljive."

Opseg vidljivosti: „Ne, nikad čuo za to. Traži dalje."

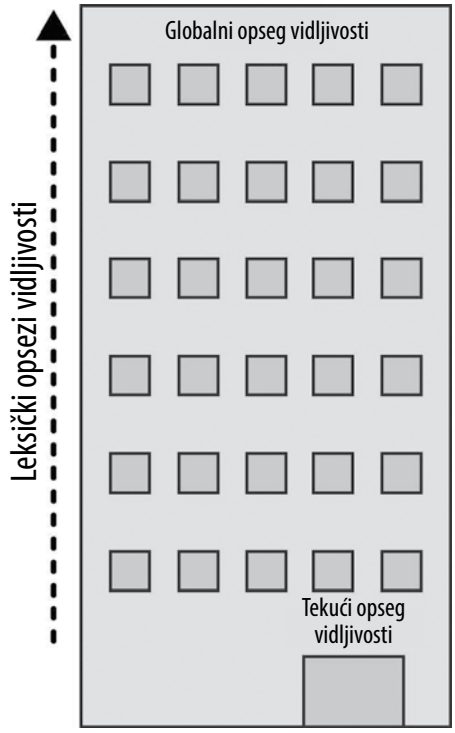
Mašina: „Zdravo, Opseže vidljivosti izvan funkcije `foo`; oh, to je globalni opseg vidljivosti, super. Da li si čuo za `b`? Imam RHS referencu takve promenljive."

Opseg vidljivosti: „Jesam, svakako. Evo, izvoli."

Jednostavna pravila za pretraživanje ugnežđenog opsega vidljivosti: Mašina prvo traži promenljivu unutar opsega vidljivosti tekuće funkcije koja se izvršava; ako je tamo ne nađe, prelazi na prvi sledeći okružujući nivo itd. Kada dođe do spoljašnjeg, globalnog opsega vidljivosti, pretraživanje se završava, bez obzira na to da li je promenljiva nađena ili nije.

## Gradnja na metaforama

Da biste sebi vizuelno predstavili pretraživanja ugnežđenih opsega vidljivosti, zamislite sledeću visoku zgradu:




Zgrada predstavlja skup pravila koja važe za pretraživanje ugnežđenih opsega vidljivosti u našem programu. Tekući sprat zgrade, koji god to bio, predstavlja opseg vidljivosti funkcije koja se trenutno izvršava. Najviši sprat zgrade je globalni opseg vidljivosti.

LHS i RHS reference razrešavate tako što referencu prvo tražite na tekućem spratu, pa ako je tamo ne nađete, uzimate lift do sledećeg višeg sprata i tražite tamo, zatim na sledećem višem spratu itd. Kada stignete na najviši sprat (globalni opseg vidljivosti), tamo ili nalazite ili ne nalazite ono što ste tražili. U svakom slučaju, morate prekinuti pretragu.

# Greške

Zbog čega je važno da li se obavlja LHS ili RHS pretraga?

Zato što se te dve vrste pretrage ponašaju različito u slučaju kada promenljiva nije deklarisana (nije pronađena ni u jednom pregledanom opsegu vidljivosti).

Razmotrite sledeće:

```
function foo(a) {
    console.log( a + b );
    b = a;
}

foo( 2 );
```

Kada se prvi put izvrši RHS pretraga za `b`, ta promenljiva neće biti pronađena. Tada se ta promenljiva smatra „nedeklarisanom“ jer nije pronađena u opsegu vidljivosti.

Ako RHS pretraga nigde ne pronađe traženu promenljivu, ni u jednom od ugnežđenih opsega vidljivosti, mašina jezika generiše grešku `ReferenceError`. Važno je imati u vidu da je tip te greške `ReferenceError`.

Nasuprot tome, ukoliko mašina obavlja LHS pretragu i dođe do najvišeg sprata (globalnog opsega vidljivosti) a da nije pronašla promenljivu, ako program ne radi u „striktnom režimu“[1], globalni opseg vidljivosti napraviće novu promenljivu s traženim imenom *u globalnom opsegu vidljivosti* i proslediti je Mašini.

*„Ne, pre nije bilo takve promenljive, ali hteo sam da pomognem, pa sam ti napravio jednu“*.

„Striktni režim“ koji je uveden u verziji ES5, ima više ponašanja različitih od onih u normalnom/opuštenom/lenjom režimu. Jedno od takvih ponašanja je da ne dozvoljava automatsko/implicitno generisanje globalnih promenljivih. U tom slučaju, ne bi bilo nijedne promenljive s globalnim opsegom vidljivosti koji bi LHS pretraga mogla da pronađe, a Mašina bi generisala grešku `ReferenceError`, slično kao pri RHS pretrazi.

Ako je promenljiva pronađena tokom RHS pretrage, ali vi pokušate da s njenom vrednošću uradite nešto što nije moguće – recimo, da izvršite kao funkciju promenljivu koja to nije, ili da referencirate svojstvo vrednosti koja je `null` ili `undefined` – onda Mašina generiše grešku drugačijeg tipa, koji se zove `TypeError`.

`ReferenceError` znači neuspešno pretraživanje opsega vidljivosti, dok `TypeError` znači da je pretraživanje opsega vidljivosti bilo uspešno, ali da ste nad rezultatom pretrage pokušali neku nedozvoljenu/nemoguću akciju.

# Sažetak poglavlja

Opseg vidljivosti je skup pravila koja određuju gde i kako se može pronaći data promenljiva (identifikator). Razlog te pretrage može biti dodeljivanje vrednosti promenljivoj, što

1. Videti u MDN-u objašnjenje striktnog režima.

se zove LHS referenca (na levu stranu), ili učitavanje njene vrednosti, što se zove RHS referenca (na desnu stranu).

LHS reference su rezultat operacija dodeljivanja vrednosti. Dodeljivanje vrednosti promenljivoj unutar opsega može se dogoditi zahvaljujući upotrebi operatora `=` ili prosleđivanju (dodeljivanju) vrednosti parametrima funkcija pomoću argumenata.

Mašina JavaScripta prvo kompajlira kôd pre nego što ga izvrši, pri čemu deli iskaze programa, kao što je `var a = 2;`, u dva koraka:

1. Prvo izvršava deo `var a`, da bi deklarisala tu promenljivu u opsegu vidljivosti programa. To se odvija na početku, pre izvršavanja koda.
2. Zatim, izvršava deo `a = 2`, da bi pronašla promenljivu (LHS referencu) i dodelila joj vrednost, ako je nađe.

Obe vrste traženja reference, i LHS i RHS, započinju postupak traženja u opsegu vidljivosti tekuće funkcije u toku izvršavanja, a ukoliko je potrebno (tj. ako tamo ne nađu ono što traže), traže identifikator redom na svim sledećim nivoima ugnežđenih opsega vidljivosti, opseg (sprat) po opseg, sve dok ne dođu do globalnog nivoa (najvišeg sprata) i prekinu pretragu, a tamo ili nađu traženi identifikator ili ga ne nađu.

Ako tražena RHS referenca ne postoji, rezultat je greška tipa `ReferenceError`. Nepostojeća LHS referenca prouzrokuje automatsko generisanje implicitno deklarisane globalne promenljive s traženim imenom (ukoliko nije uključen striktni režim), ili grešku tipa `ReferenceError` (ako je uključen striktni režim).

## Odgovori na pitanja iz kviza

```
function foo(a) {
    var b = a;
    return a + b;
}

var c = foo( 2 );
```

1. Otkrijte sve LHS pretrage (ima ih 3!).

   *`c = ..;, a = 2 (implicitno dodeljivanje vrednosti parametrima) i b = ..`*

2. Otkrijte sve RHS pretrage (ima ih 4!).

   *`foo(2.., = a;, a .. i .. b`*