

SINTAKSNO TESTIRANJE

Sintaksno testiranje

- Sintaksno testiranje (**Syntax Testing**), poznato i kao **Grammar-based testing**, je metoda za testiranje ulaza koji su definisani preko bezkontekstnih gramatika (Context-Free Grammars, CFGs) ili BNF (Backus-Naur Form) forme.
- Ova metoda se fokusira na generisanje test sekvenci direktno iz gramatike kako bi se proverilo da li softver ispravno parsira validne ulaze i, što je još važnije, da li adekvatno odbija i rukuje nevalidnim ulazima.

Kako funkcioniše Sintaksno Testiranje?

Metoda se sastoji od dva aktivnosti:

- 1) Generisanje validnih ulaza** (Positive Testing): Prolazi se kroz pravila gramatike (izvedbena stabla) kako bi se kreirale niske koje sistem mora da prihvati. Cilj je pokriti sva pravila produkcije (Production Coverage).
- 2) Generisanje nevalidnih ulaza** (Negative / Mutation Testing): Namerno se uvode greške u legalne izlaze (npr. zamena znakova, ubacivanje nelegalnih znakova, izostavljanje obaveznih elemenata) kako bi se proverila robusnost parsera i rukovanje izuzecima.

Primer

- Razmotrimo modul koji proverava da li vrednost ulazne promenljive **float_in** odgovara sintaksi broja u pokretnom zarezu. Izlaz (promenljiva **check_res**) uzima vrednosti *valid* ili *invalid* u zavisnosti od validnosti sintakse

Primer

- Opis sintakse broja u pokretnom zarezu u Backus Naur-ovoj formi (BNF):
float = int "e" int.
int = ["+"|" -"] nat.
nat = {dig}.
dig = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".
- Terminali su prikazani pod navodnicima – to su elementarni delovi sintakse ulaza, tj. znaci od kojih se sastoji ulaz.
- | je sint. operator koji razdvaja alternative
- [] ograđuje neobavezni element (može se izostaviti)
- { } uokviruje element sintakse koji se u ulazu može pojaviti jednom ili više puta.

Aplikacija koju testiramo (System under test)

```
public class MojFloatParser {
```

```
    public void parsiraj(String ulaz) throws
        SintaksnaGreskaException, SemantickaGreskaException {
        if (ulaz == null || ulaz.isEmpty()) {
            throw new SintaksnaGreskaException("Ulaz ne sme biti prazan.");
        }
    }
```

```
    // 1. SINTAKSNA PROVERA: Da li postoji jedno slovo 'e'?
```

```
    int eIndex = ulaz.indexOf('e');
```

```
    if (eIndex == -1 || eIndex != ulaz.lastIndexOf('e')) {
        throw new SintaksnaGreskaException("Mora biti int 'e' int.");
    }
```

```
    // Razdvajanje na mantisu (pre 'e') i eksponent (posle 'e')
```

```
    String mantisaStr = ulaz.substring(0, eIndex);
```

```
    String eksponentStr = ulaz.substring(eIndex + 1);
```

```
    // Validacija strukture celih brojeva (intRule i natRule)
```

```
    validirajIntSintaksu(mantisaStr);
```

```
    validirajIntSintaksu(eksponentStr);
```

```
    // 2. SEMANTIČKA PROVERA: Da li brojevi mogu da stanu u Java Long?
```

```
    try {
        // Sint. ispravno (npr. "9999999999999999e1"), ali semantički puca
```

```
        long mantisaVal = Long.parseLong(mantisaStr);
```

```
        long eksponentVal = Long.parseLong(eksponentStr);
```

```
        // Dodatno semantičko pravilo: Eksponent ne sme izazvati Overflow }
```

```
        if (Math.abs(eksponentVal) > 38) {
            throw new SemantickaGreskaException(
                "Eksponent je van opsega za float tip (max 10^38).");
        }
    }
```

```
    catch (NumberFormatException e) {
        throw new SemantickaGreskaException(
            "Broj prevazilazi granice skladištenja u memoriji.");
    }
}
```

```
    private void validirajIntSintaksu(String s)
        throws SintaksnaGreskaException {
```

```
        if (s.isEmpty()) {
            throw new SintaksnaGreskaException("Nedostaje broj.");
        }
    }
```

```
    // Skidanje opcionog znaka
```

```
    if (s.startsWith("+") || s.startsWith("-")) {
        s = s.substring(1);
    }
```

```
    if (s.isEmpty()) { // Samo "+" ili "-" bez cifara
        throw new SintaksnaGreskaException("Znak mora biti
        praćen ciframa.");
    }
```

```
    // Provera da li su sve ostalo cifre (natRule)
```

```
    for (char c : s.toCharArray()) {
```

```
        if (!Character.isDigit(c)) {
            throw new SintaksnaGreskaException(
                "Dozvoljene su samo cifre.");
        }
    }
```

Sintaksne greške (strukturne greške)

- **Definicija:** Kršenje formalnih pravila gramatike (BNF-a). Odnose se isključivo na formu i strukturu teksta, bez obzira na to šta taj tekst znači.
- **U prirodnom jeziku:** "Dečak trčati brzo kuća." (Reči su tu, ali je struktura rečenice pogrešna).
- **U našem primeru:**
 - 5++e2 (Dva znaka plus – krši pravilo (PLUS | MINUS)?)
 - 5e (Nedostaje celobrojni eksponent – krši pravilo floatRule)
- **Uloga alata:** Grammarinator je dizajniran da unparsira i mutira isključivo formu. On generiše i hvata ove greške sa 100% tačnosti jer one direktno krše stablo gramatike.

Semantičke greške (Značenjske/Logičke greške)

- **Definicija:** Tekst je sintaksno savršeno ispravan, ali njegovo značenje ili kontekst nemaju smisla u stvarnom sistemu (krše se statička pravila tipova, opsega ili logike).
- **U prirodnom jeziku:** "Matematika je zelena." (Sintaksa i gramatika su 100% tačne, ali rečenica nema nikakav smisao).
- **U našem primeru:**
1e9999999999999999 (Sintaksa je ispravna: imamo int, slovo 'e', i int. Međutim, ovaj broj je nemoguće smestiti u memoriju računara niti ga interpretirati kao standardni float).
- **Ograničenja alata:** Tradicionalni unparseri (poput Grammarinatora) ne mogu sami da predvide semantičke greške samo na osnovu BNF-a, jer je 9999999999999999 za gramatiku jednako validan niz cifara kao i broj 2.

Teorijski kriterijumi pokrivenosti gramatike

- Kada testiramo softver pomoću gramatika, ne možemo generisati beskonačno mnogo stringova. Moramo definisati **kriterijume pokrivanja** (Coverage Criteria) gramatike generisanim stringovima.
- Kriterijumi se dele od najslabijeg do najjačeg:
 - 1) Pokrivenost terminala (Terminal / Token Coverage)
 - 2) Pokrivenost gramatičkih pravila (Production / Rule Coverage)
 - 3) Pokrivenost izvođenja (Derivation / Boundary Coverage)

1. Pokrivenost terminala (Terminal / Token Coverage)

- **Definicija:** Svaki terminalni simbol (token) definisan u gramatici mora se pojaviti bar jednom u generisanim test primerima.
- **U našem primeru** (float = int "e" int): Testovi moraju da sadrže znak +, znak -, slovo e i bar jednu cifru (npr. od 0 do 9).
- Primer skupa testova:
+1e-2 (Ovaj jedan test već pokriva skoro sve terminale).

2. Pokrivenost gramatičkih pravila (Production/Rule Coverage)

- **Definicija:** Svaka grana i svaka alternativa u pravilima gramatike mora biti izvršena bar jednom.
- **U našem primeru:** Pravilo int ima alternative: može imati +, može imati -, a može biti i bez znaka. Da bismo imali 100% Production Coverage, moramo generisati sva tri slučaja.
- Primer skupa testova:
 - +1e2 (pokriven + u prvom int-u, i bez znaka u drugom int-u)
 - 1e-2 (pokriven - u oba int-u)

3. Pokrivenost izvođenja (Derivation / Boundary Coverage)

- **Definicija:** Fokusira se na petlje i ponavljanja (poput {dig} ili DIGIT+). Testira se ponašanje na granicama: nula ponavljanja (ako je opciono), tačno jedno ponavljanje, i više ponavljanja.
- **U našem primeru:** Za $\text{nat} = \{\text{dig}\}$, granični slučajevi su:
 - 1 cifra (npr. 5e2)
 - Više cifara (npr. 5555e2222)

NAPOMENA: Ovo su bili samo primeri koji ilustruju definiciju, ova pravila treba sistematski primeniti na sve tokene, pravila i ponavljanja

Primena ove tehnike u praksi

Praktični pipeline:

- 1) **Definicija**: BNF gramatika → ANTLR .g4 fajl.
- 2) **Generisanje** (alat Grammarinator): Unparser generiše testove šetnjom kroz stablo (osigurava Production Coverage).
- 3) **Mutacija** (radamsa fuzzer): Namerno kršenje pravila na nivou čvorova (generisanje nelegalnih test sekvenci).
- 4) **Verifikacija**: Pokretanje testova u JUnitu → proveru da li legalni testovi prolaze bez greške ili izazivaju samo semantičke greške. Provera da li nelegalni testovi izazivaju sintaksne greške.

1. Korak: Prevođenje BNF gramatike u ANTLR .g4 format

(FloatGrammar.g4)

```
grammar FloatGrammar;
```

```
// Glavno pravilo (Start rule)
```

```
floatRule : intRule 'e' intRule ;
```

```
// Pomoćna pravila
```

```
intRule  : (PLUS | MINUS)? natRule ;
```

```
natRule  : DIGIT+ ; // {dig} u vašoj gramatici znači jedna ili više cifara
```

```
// Terminalni simboli (Leksika)
```

```
PLUS     : '+' ;
```

```
MINUS    : '-' ;
```

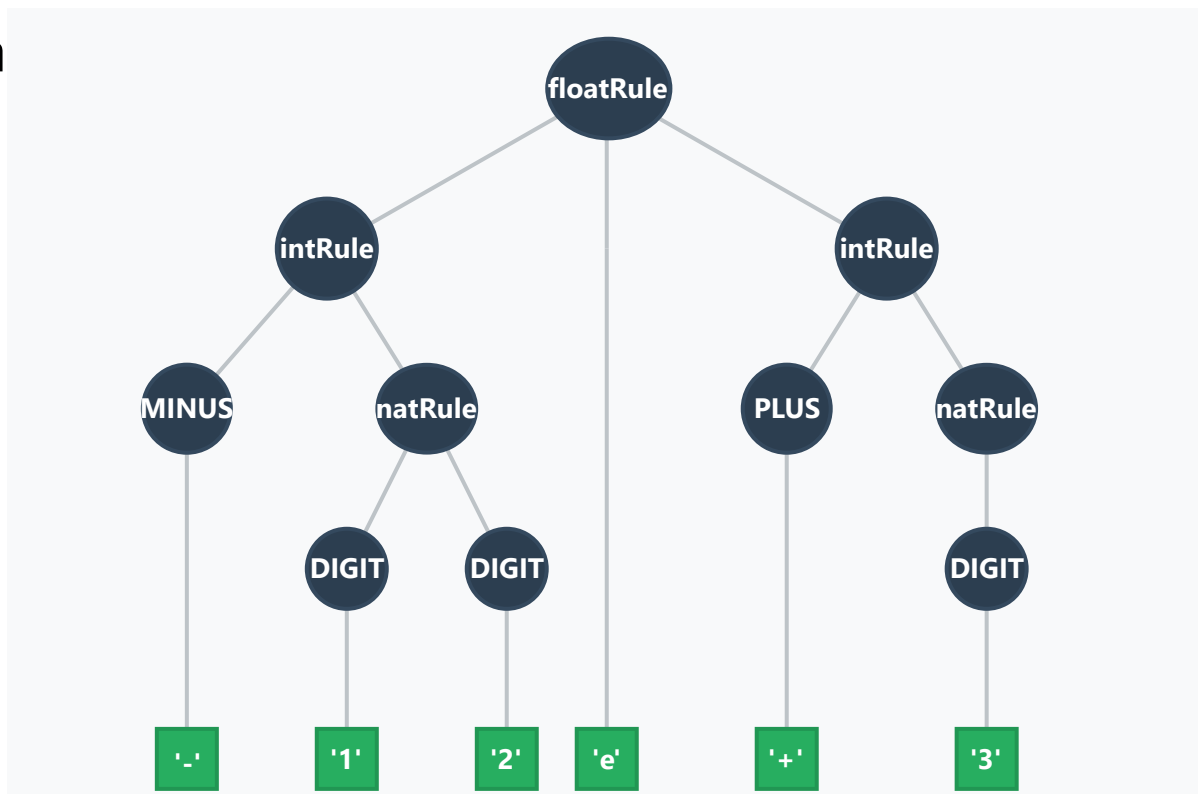
```
DIGIT     : [0-9] ;
```

2. Grammarinator radi "Unparsing" (Generisanje validnih stringova)

- Kada pokrenete Grammarinator nad ovim fajlom, on pravi stablo izvođenja (Parse Tree) i kreira unparsers koji nasumično (ali strukturno ispravno) donosi odluke na mestima gde postoje alternative (? ili * ili +).
- Kontrola generisanja: *kontrola dubine stabla izvođenja* i *broja generisanih stringova*
- *Svi ovi stringovi su 100% u skladu sa gramatikom i služe za pozitivno testiranje parsera float broja.*

2. Grammarinator radi "Unparsing" (Generisanje validnih stringova)

- Primer stabla izvođenja:



Kontrola generisanja: Parametar dubine stabla (`--max-depth`)

- Kada unparser generiše validne stringove iz vaše gramatike, on to radi rekurzivno. Ako gramatika ima rekurziju (ili ponavljanja), stablo bi teorijski moglo da ide u beskonačnost.
- Grammarinator ima opciju `--max-depth` (npr. `--max-depth 5`).
- Ograničavanjem dubine na malu vrednost, osigurava se pokrivenost minimalnih graničnih vrednosti (jednostavni stringovi). Povećanjem dubine, Grammarinator je "nateran" da prođe kroz dublje produkcije i generiše kompleksnije strukture, čime se ostvaruje napredna pokrivenost izvođenja.

Kontrola generisanja: broj generisanih testova (-n)

- Grammarinator koristi pseudo-slučajni algoritam (random walk) kroz stablo gramatike.
- Opcijom -n 1000 kažemo alatu da generiše 1000 različitih stringova.
- **Zakon velikih brojeva:** Za jednostavnu gramatiku kao u primeru, generisanje već 100-200 primeraka statistički garantuje 100% Production i Terminal Coverage. Alat interno vodi računa da prođe kroz sve grane (+, -, svaku cifru od 0-9) tokom tih iteracija.

3. Generisanje nevalidnih test primera

- Alat Radamsa je black-box generički mutator (fuzzer) koji ne poznaje strukturu gramatike, već radi direktno na nivou bajtova i karaktera
- Radamsa koristi skup internih mutacionih heuristika (operatora) i kombinatorički algoritam. Ona uzima validan ulazni uzorak, analizira ga kao niz podataka, a zatim nasumično primenjuje mutacije na kritičnim mestima (npr. na granicama reči, brojeva ili specijalnih karaktera)

Primeri mutacija koje izvodi Radamsa

1) Dupliranje (Replication): Beskonačno ponavlja delove stringa.

1) Primer: $-12e+3 \rightarrow -12e+++++++3$ ili $-12121212e+3$

2) Brisanje (Deletions): Uklanja pojedinačne karaktere ili cele delove.

1) Primer: $-12e+3 \rightarrow -12+3$ (brisanje e) ili $12e+3$ (brisanje znaka)

3) Ubacivanje ekstremnih vrednosti (Boundary/Magic Numbers): Menja postojeće cifre ekstremno velikim brojevima, nula-vrednostima ili maksimalnim vrednostima za tipove podataka (integer overflow).

1) Primer: Umesto 3 ubacuje 2147483647 ili 4294967295

4) Ubacivanje specijalnih i "neželjenih" karaktera (Junk / Bad Characters): Ubacuje kontrolne karaktere, null-byte ($\backslash x00$), Unicode karaktere, emotikone ili format stringove ($\%s\%x$).

1) Primer: $-12\backslash x00e+3$

5) Premeštanje i zamena mesta (Permutations / Swapping): Rotira ili menja mesta blokovima karaktera unutar stringa.

6) Primer: $-12e+3 \rightarrow +3e-12$

7) Bit-flipping (Izmena na nivou bitova): Nasumično izvrće bitove u bajtovima, što često pretvara regularne karaktere u potpuno nevalidne ASCII simbole

Arhitektura test sistema

