Apstraktno sintaksno stablo

Dujmović, David

Undergraduate thesis / Završni rad

2023

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, School of Applied Mathematics and Informatics / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet primijenjene matematike i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:126:234640

Sveučilište J. J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni preddiplomski studij matematike i računarstva

David Dujmović

Apstraktno sintaksno stablo

Završni rad

Osijek, 2023.

Sveučilište J. J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni preddiplomski studij matematike i računarstva

David Dujmović

Apstraktno sintaksno stablo

Završni rad

Mentor: doc. dr. sc. Domagoj Ševerdija

Osijek, 2023.

Abstract syntax tree

Sažetak

Abstract Syntax Tree (AST), ili apstraktno sintaksno stablo, je struktura podataka koja predstavlja sintaktičku strukturu programskog ili izvornog koda. Rezultat je parsiranja izvornog koda i pruža apstraktnu, hijerarhijsku reprezentaciju tog koda. U tradicionalnom parsiranju, izvorni kod se prolazi kroz leksičku analizu (tokenizacija) i sintaksnu analizu kako bi se odredila sintaktička struktura. Umjesto da zadržava sve detalje i specifičnosti jezika, AST se fokusira na bitne dijelove koda, ignorirajući nepotrebne detalje poput razmaka i specifičnosti pojedinih izraza.Sastoji se od čvorova koji predstavljaju različite kons- trukte jezika, poput izjava, operatora, identifikatora, literala i ostalih elemenata. Svaki čvor stabla ima svoje podčvorove koji predstavljaju njegove sastavne dijelove. Na taj način, AST reprezentira strukturu izvornog koda kao hijerarhijsko stablo, pri čemu korijen predstavlja glavni program, a djeca čvorova predstavljaju njegove podizraze ili izjave.

Ključne riječi

AST, stablo, sintaksa

Abstract

Abstract Syntax Tree (AST), or abstract syntax tree, is a data structure that represents the syntactic structure of the program or source code. It is the result of parsing source code and provides an abstract, hierarchical representation of that code. In the traditional parsing, the source code is passed through lexical analysis (tokenization) and syntax analysis the syntactic structure would be determined. Instead of keeping all the details and specifics of the language, AST focuses on the essential parts of the code, ignoring unnecessary details like whitespace and specificities of individual expressions. It consists of nodes that represent different cons- language tricks, such as statements, operators, identifiers, literals and other elements. Each node the tree has its own subnodes that represent its constituent parts. In this way, AST represents the source code structure as a hierarchical tree, where the root represents main program, and child nodes represent its subexpressions or statements.

Key words

AST, tree, syntax

Sadržaj

Uvod

Stabla apstraktne sintakse, poznata kao AST (kratica za Abstract Syntax Tree), predstavljaju računalni program na apstraktan način. AST je oblik sintaktičkog stabla koje nam pomaže razumjeti strukturu jezika. Sintaksa je jedinstvena za svaki jezik jer definira pravila koja iskazi moraju slijediti kako bi imali smisla. To se ne odnosi samo na programski jezik; pisani i govorni jezik također imaju svoju sintaksu.

Kako bismo stvorili stablo od napisanog koda, prvo moramo osigurati da je taj kod sintaktički ispravan, što znači da slijedi gramatička pravila jezika. AST je oblik strukture podataka s granama poput stabla te se često koristi u procesima kao što su kompilacija, interpretacija i statička analiza programa. Primjene AST-a uključuju optimizaciju koda, provjeru sintakse, izvođenje programa, generiranje međukoda i često služe kao temelj za daljnje analize programskog jezika.

AST-ovi su ključni u svijetu programiranja i često se koriste u procesu razvoja. Bez obzira radite li na izradi prevoditelja, razvoju općih alata ili drugim projektima, ono je moćno sredstvo koje olakšava analizu i manipulaciju programskim kodom. U ovom radu detaljnije ćemo istražiti što su AST-ovi, gdje se koriste i kako ih iskoristiti u svakodnevnom razvoju softvera.

1 O apstraktnom sintaksnom stablu

Kod pisanja koda, velike su šanse da su AST-ovi uključeni u tijek razvoja softvera. AST je kratica za Abstract Syntax Tree i oni pokreću mnoge dijelove toka razvoja. Najčešće se spominju u kontekstu prevoditelja, ali se koriste i u drugim alatima. U nastavku ćemo navesti gdje se sve to koriste i kako ih se može iskoristiti. Stabla apstraktne sintakse ili AST predstavljaju stablo koda. Oni su temeljni način na koji kompilator radi. Kada prevoditelj transformira neki kod, u osnovi postoje sljedeći koraci:

```
isPanda('🐼');
```

Slika 1 : Linija koda

Snimka zaslona

1.0.1 Leksička analiza aka tokenizacija

Tijekom ovog koraka, napisani kod bit će pretvoren u skup tokena koji opisuju različite dijelove vašeg koda. Ovo je u osnovi ista metoda koja koristi osnovno isticanje sintakse. Tokeni ne razumiju kako se stvari uklapaju i isključivo se fokusiraju na komponente datoteke. Nešto kao popis ili niz različitih vrsta tokena, kao da je uzet tekst i rastavljen na riječi. Moći će se razlikovati interpunkcijski znakovi, glagoli, imenice, brojevi itd., ali u ovoj fazi neće biti dubljeg razumijevanja onoga što je dio rečenice ili kako se rečenice uklapaju.

```
[
  { type: 'Identifier', value: 'isPanda' },
  { type: 'Punctuator', value: '(' },
  { type: 'String', value: "'🐼'" },
  { type: 'Punctuator', value: ')' },
];
```

Slika 2 : Skup tokena

Snimka zaslona

1.0.2 Analiza sintakse aka raščlanjivanje

Ovo je korak u kojem se popis tokena pretvara u stablo apstraktne sintakse. Tokeni se pretvaraju u stablo koje predstavlja stvarnu strukturu koda. Dok je prije u tokenima bio samo par, sada postoji ideja je li to poziv funkcije, definicija funkcije, grupiranje ili nešto drugo. Ekvivalent bi bilo pretvaranje popisa riječi u podatkovnu strukturu koja predstavlja stvari kao što su rečenice, kakvu ulogu određena imenica igra u rečenici ili nalazimo li se na popisu. Potrebno je napomenuti da ne postoji specifičan AST format. Mogu se razlikovati ovisno o jeziku koji se pretvara u AST, kao i o alatu koji se koristi za raščlanjivanje. U JavaScriptu je uobičajeni standard ESTree , ali također različiti alati mogu dodati dodatna svojstva. Općenito, AST je struktura stabla gdje svaki čvor ima barem tip koji specificira što predstavlja. Na primjer, tip može biti a Literal koji predstavlja stvarnu vrijednost ili CallExpression koji predstavlja poziv funkcije. Čvor Literal može sadržavati samo vrijednost dok CallExpression čvor sadrži puno dodatnih informacija koje bi mogle biti relevantne, poput "ono što se zove" (callee) ili što argumente prosljeđuje.

Slika 3 : Stablo apstraktne sintakse

Snimka zaslona

1.0.3 Generiranje koda

Ovaj korak može biti višestruki. Kad se dođe do stabla apstraktne sintakse, njime se može manipulirati, kao i "ispisati" u drugu vrstu koda. Korištenje AST-a za manipuliranje kodom sigurnije je od obavljanja tih operacija izravno na kodu kao tekstu ili na popisu tokena. Manipuliranje tekstom uvijek je opasno; pokazuje najmanje konteksta. Kod pokušaja manipuliranja tekstom koristeći zamjene nizova ili regularne izraze jednostavno je pogriješiti. Čak ni manipuliranje tokenima nije trivijalno. Iako se možda zna što je varijabla, ako se želi preimenovati, ne bi se imalo uvid u stvari poput opsega varijable ili bilo koje varijable s kojima bi se mogla sukobiti. AST pruža dovoljno informacija o strukturi koda da ga možemo modificirati s više povjerenja. Može se, primjerice, odrediti gdje je varijabla deklarirana i točno znati na koji dio programa to utječe zbog strukture stabla. Nakon što se izmanipuliralo stablom, može se ispisati stablo za ispis bilo kojeg koda koji se očekuje. Na primjer, kad bi se izgradio prevoditelj poput TypeScript prevoditelja, prvi bi ispisao JavaScript, dok bi drugi prevoditelj mogao ispisati strojni kod. Opet, to se lakše postiže s AST-om jer različiti izlazi mogu imati različite formate za istu strukturu. Bilo bi teže generirati izlaz poput teksta ili popisa tokena.

```
{
  type: 'Program',
  body: [
    {
      type: 'ExpressionStatement',
      expression: { type: 'Literal', value: true, raw: 'true' },
    },
  ],
  sourceType: 'script',
}
```

Slika 4 : Generirani kod

Snimka zaslona

1.1 Primjena u prevoditeljima

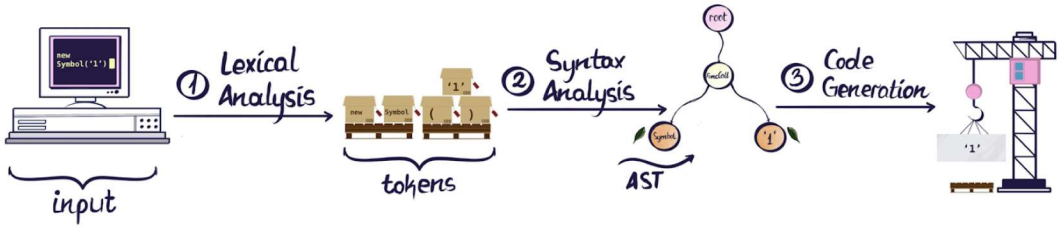
Stabla apstraktne sintakse su podatkovne strukture koje se naširoko koriste u prevoditeljima za predstavljanje strukture programskog koda. AST je obično rezultat faze analize sintakse prevoditelja. Često služi kao posredni prikaz programa kroz nekoliko faza koje prevoditelj zahtijeva i ima snažan utjecaj na konačni izlaz.

1.1.1 Motivacija

AST ima nekoliko svojstava koja pomažu u daljnjim koracima procesa kompilacije. AST se može uređivati i poboljšavati informacijama kao što su svojstva i bilješke za svaki element koji sadrži. Kao što je napomenuto takvo uređivanje i bilješke nemoguće je s izvornim kodom programa, budući da bi impliciralo njegovu promjenu. U usporedbi s izvornim kodom, AST ne uključuje nebitne interpunkcijske znakove i razdjelnike (vitičaste zagrade, točke i zareze, zagrade itd.). AST obično sadrži dodatne informacije o programu, zbog uzastopnih faza analize od strane kompilatora. Na primjer, može pohraniti položaj svakog elementa u izvornom kodu, dopuštajući kompilatoru da ispiše korisne poruke o pogrešci. AST su potrebni zbog inherentne prirode programskih jezika i njihove dokumentacije. Jezici su po prirodi često višeznačni. Kako bi se izbjegla ova dvosmislenost, programski jezici se često navode kao kontekstno-slobodna gramatika (CFG). Međutim, često postoje aspekti programskih jezika koje CFG ne može izraziti, ali su dio jezika i dokumentirani su u njegovoj specifikaciji. To su detalji koji zahtijevaju kontekst da bi se odredila njihova valjanost i ponašanje. Na primjer, ako jezik dopušta deklariranje novih tipova, CFG ne može predvidjeti imena takvih tipova niti način na koji bi se trebali koristiti. Čak i ako jezik ima unaprijed definiran skup tipova, provođenje pravilne upotrebe obično zahtijeva neki kontekst.

1.1.2 Dizajn

Dizajn AST-a često je usko povezan s dizajnom kompilatora i njegovim očekivanim značajkama. Osnovni zahtjevi uključuju da se vrste varijabli moraju sačuvati, kao i mjesto svake deklaracije u izvornom kodu. Redoslijed izvršnih iskaza mora biti eksplicitno predstavljen i dobro definiran. Lijeva i desna komponenta binarnih operacija moraju biti pohranjene i ispravno identificirane. Identifikatori i njihove dodijeljene vrijednosti moraju biti pohranjeni za izjave o dodjeli.
Ovi se zahtjevi mogu koristiti za dizajn strukture podataka za AST. Neke će operacije uvijek zahtijevati dva elementa, kao što su dva člana za zbrajanje. Međutim, neke jezične konstrukcije zahtijevaju proizvoljno velik broj djece, kao što su popisi argumenata koji se prosljeđuju programima iz naredbene ljuske. Kao rezultat toga, AST koji se koristi za predstavljanje koda napisanog u takvom jeziku također mora biti dovoljno fleksibilan da omogući brzo dodavanje nepoznate količine djece. Kako bi se podržala provjera prevoditelja, trebalo bi biti moguće rastaviti AST u oblik izvornog koda. Proizvedeni izvorni kod trebao bi biti dovoljno sličan originalnom izgledu i identičan u izvršenju, nakon ponovnog kompajliranja. AST se intenzivno koristi tijekom semantičke analize, gdje prevoditelj provjerava ispravnu upotrebu elemenata programa i jezika. Prevoditelj također generira tablice simbola na temelju AST-a tijekom semantičke analize. Kompletan obilazak stabla omogućuje provjeru ispravnosti programa.



Slika 5 : Postupak transformacije koda

Dostupno na :(https://assets.cdn.prod.twilio.com/images/-vnpXSyzxyPpPK3TqYdV1pedY3MnsEnyyGUH1WlTd0O29Cl.original.png)

1.2 Što učiniti s AST-ovima?

Slučajevi upotrebe za AST su široki i općenito se mogu podijeliti u tri sveobuhvatne radnje: čitanje, modificiranje i ispis. Oni su vrsta dodataka, što znači da ako ispisujete AST, velike su šanse da ste prethodno također pročitali AST i modificirali ga. U sljedećim poglavlja pokrit će se načini na koje se može izvršiti odgovarajuća radnja.

1.2.1 Čitanje AST-ova

Tehnički prvi korak rada s AST-ovima je raščlanjivanje teksta za stvaranje AST-a, ali u većini slučajeva biblioteke koje nude korak raščlanjivanja također nude način za prolazak kroz AST. Kretanje AST-om znači posjećivanje različitih čvorova stabla radi dobivanja uvida ili izvođenja radnji. Jedan od najčešćih slučajeva upotrebe za ovo je linting. ESLint, na primjer, koristi espree za generiranje AST-a i ako se želi napisati prilagođena pravila, onda se zapisuju ona temeljena na različitim AST čvorovima. ESLint dokumenti sadrže opsežnu dokumentaciju o tome kako možete izgraditi prilagođena pravila, dodatke i formatere. Ako pretražujete npm, također ćete pronaći kolekciju drugih alata za raščlanjivanje i prelaženje AST-ova. Obično se razlikuju po dizajnu API-ja i povremeno po mogućnostima raščlanjivanja JavaScripta.

1.2.2 Modificiranje AST-ova

Kao što je prethodno navedeno, postojanje AST-a čini modificiranje navedenog stabla lakšim i sigurnijim od modificiranja koda kao tokena ili neobrađenog niza. Postoji širok izbor razloga zašto bi se moglo modificirati neki kod pomoću AST-ova. Babel, na primjer, modificira AST-ove za transpiliranje novijih značajki ili za pretvaranje JSX-a u pozive funkcija. To će se dogoditi, na primjer, pri kompajliranju React ili Preact kod. Drugi slučaj upotrebe bio bi paketni kod. U svijetu modula, grupiranje koda često je mnogo složenije od pukog dodavanja datoteka. Bolje razumijevanje strukture odgovarajućih datoteka olakšava spajanje tih datoteka i prilagođavanje uvoza i pozivanja funkcija gdje je to potrebno. Baze kodova alata kao što su webpack, parcel ili rollup koriste AST-ove kao dio svog radnog tijeka povezivanja. Slučaj upotrebe koji bi se mogao činiti manje očitim je pokrivenost testom. Problem je u tome što ubacuju dodatni kod koji povećava različite brojače za svaki redak,

funkciju i izjavu. Da nakon izvođenja svih vaših testova mogu pregledati navedene brojače i dati vam detaljan uvid u to što je izvršeno, a što nije. Učiniti to bez AST-a nevjerojatno je teško i manje predvidljivo. Sve su to alati o kojima vjerojatno nećete puno čuti. Ali postoji jedan slučaj upotrebe koji može biti koristan za vaš redoviti tijek razvoja. A to je izrada koda za optimizaciju, makronaredbe ili ažuriranje većih dijelova vaše baze koda odjednom. React tim, na primjer, održava kolekciju skripti pod nazivom react-codemod koje mogu izvoditi uobičajene operacije povezane s ažuriranjem vaše verzije Reacta. Alat koji koriste ispod haube se zove jscodeshifti mi ga možemo koristiti za pisanje vlastitih skripti transformacije. Recimo da, na primjer, preferiramo otklanjanje pogrešaka koristeći, alert() ali želimo izbjeći slanje toga našim klijentima. Također je moguće upotrijebiti isti skup vještina za izradu, primjerice, vlastitih Babel ili ESLint dodataka koji će obavljati popravke koda.

1.2.3 Ispis AST-ova

U većini slučajeva ispis i modificiranje AST-ova idu ruku pod ruku budući da se mora ispisati AST koji se upravo izmijenio. No dok se neke biblioteke izričito fokusiraju na ispis AST-a u istom stilu koda kao i izvornik, postoji i niz slučajeva upotrebe u kojima želite izričito drugačije ispisati svoj AST. Prettier, na primjer, koristi AST-ove za preoblikovanje vašeg koda prema vašoj konfiguraciji bez mijenjanja sadržaja vašeg koda. Način na koji to rade je pretvaranje vašeg koda u AST koji je potpuno neshvatljiv za formatiranje i zatim ga ponovno pišu na temelju vaših pravila. Drugi uobičajeni slučajevi upotrebe bili bi ispis koda na drugom ciljnom jeziku ili izrada vlastitog alata za smanjivanje. Postoji nekoliko različitih alata koji se mogu koristiti za ispis AST-ova kao što su escodegen ili astring. Također moguće je uložiti sve i izgraditi vlastiti formater ovisno o vašem slučaju upotrebe ili izgraditi dodatak za Prettier .

1.3 Zašto je AST važan u razvoju softvera?

AST-ovi su važni jer omogućuju programerima da analiziraju i manipuliraju kodom. Programerima omogućuju izradu alata koji mogu automatski refaktorirati, optimizirati ili analizirati kod. Na primjer, AST se može koristiti za otkrivanje neiskorištenih varijabli, identificiranje uskih grla u izvedbi, pa čak i generiranje dokumentacije. AST-ovi se također mogu koristiti za izradu uređivača koda i drugih alata koji pružaju isticanje sintakse, dovršavanje koda i druge značajke. Mogu se koristiti na mnoge načine u razvoju softvera. Neki uobičajeni slučajevi upotrebe uključuju:

1.3.1 1. Optimizacija koda

AST-ovi se mogu koristiti za prepoznavanje neučinkovitog koda i predlaganje optimizacija. Na primjer, razmotrimo sljedeći isječak koda:

```
x = 1
y = 2
z = x + y
```

Kod 1

AST za Kod 1 bi izgledao kao sljedeće stablo:

```
Module(body=[Assign(targets=[Name(id='x', ctx=Store())],
                    value=Constant(value=1, kind=None)),
             Assign(targets=[Name(id='y', ctx=Store())],
                    value=Constant(value=2, kind=None)),
             Assign(targets=[Name(id='z', ctx=Store())],
                    value=BinOp(left=Name(id='x', ctx=Load()),
                                op=Add(),
                                right=Name(id='y', ctx=Load())))])
```

Kod 2

Iz Koda 2 možemo vidjeti da se varijable x i y koriste samo jednom, tako da možemo optimizirati kod tako da ih uklonimo i zamijenimo izraz konstantnom vrijednošću 3.

1.3.2 2. Analiza koda

AST-ovi se mogu koristiti za analizu koda i prepoznavanje potencijalnih grešaka ili sigurnosnih problema. Na primjer, razmotrimo sljedeći isječak koda u Pythonu:

```
password = input("Enter your password: ")
if password == "secret":
    print("Access granted")
```

Kod 3

AST za Kod 3 bi izgledao kao sljedeće stablo:

```
Module(body=[Assign(targets=[Name(id='password', ctx=Store())],
                    value=Call(func=Name(id='input', ctx=Load()),
                               args=[Constant(value='Enter your
                               password: ', kind=None)],
                               keywords=[])),
               If(test=Compare(left=Name(id='password', ctx=Load()),
                               ops=[Eq()],
                               comparators=[Constant(value='secret',
                               kind=None)]),
                  body=[Expr(value=Call(func=Name(id='print',
                                        ctx=Load()),
                                        args=[Constant(value='Access
                                        granted', kind=None)],
                                        keywords=[]))],
                  orelse=[])])
```

Kod 4

Iz ovog AST-a možemo vidjeti da se lozinka izravno uspoređuje s literalom niza "password", što nije siguran način za rukovanje lozinkama. Možemo upotrijebiti AST da identificiramo ovaj problem i predložimo bolje sigurnosne prakse, kao što je upotreba sigurnog algoritma raspršivanja za pohranjivanje lozinke.

1.3.3 Transformacija koda

AST se mogu koristiti za transformaciju koda iz jednog oblika u drugi. Na primjer, možemo koristiti stablo za transformaciju Python koda u drugi jezik kao što je JavaScript. Ovo je korisno za programere koji trebaju pisati kod na više jezika.

1.3.4 3.1. Transpiliranje koda s jednog jezika na drugi

AST-ovi se mogu koristiti za transformaciju koda iz jednog jezika u drugi. Na primjer, recimo da imamo Python isječak koda koji želimo transpilirati u JavaScript. Možemo koristiti AST za pretvaranje Python AST-a u JavaScript AST, a zatim generirati JavaScript kod iz dobivenog AST-a. Evo primjera:

```
import ast
import astor

code = ""
x = 1
y = 2
print(x + y)


stablo = ast.parse(code)
```

```
#Pretvaranje stabla u JavaScript AST
js_tree = astor.to_source(tree, 'js ')

# Ispis dobivenog JavaScript koda
print (js_tree)
```

Kod 5

Izlaz ovog koda bio bi (u Javascriptu):

```
var x = 1;
var y = 2 ;
konzola . log (x + y);
```

Kod 6

1.3.5 3.2. Uklanjanje nepotrebnog koda

AST-ovi se također mogu koristiti za uklanjanje nepotrebnog koda iz programa. Na primjer, recimo da imamo Python funkciju koja sadrži blok koda koji se nikad ne izvršava. Možemo koristiti stablo da uklonimo ovaj blok koda iz funkcije. Evo primjera:

```
import ast
code = """
def foo(x):
    if x > 0:
        print("Pozitivno")
    else:
        print(''Negativno")
    print("Gotovo")
"""
tree = ast.parse(code)
# Uklanjanje druge izjave za ispis
for node in ast.walk(tree):
    if isinstance (node, ast.If):
        node.body.pop()
# Ispis koda
print (astor.to_source(tree))
```

Kod 7

Izlaz ovog koda bio bi:

```
def foo ( x ):
    if x>0:
        ispis ( "Pozitivno" )
    else :
        ispis ( "Negativno" )
```

Kod 8

Kao što vidimo, druga izjava za ispis je uklonjena iz funkcije.

1.3.6 3.3. Optimizirajući kod

AST-ovi se također mogu koristiti za optimizaciju koda izvođenjem različitih optimizacija na AST-u. Na primjer, recimo da imamo Python funkciju koja sadrži petlju koja se može optimizirati. Možemo koristiti AST za izvođenje ove optimizacije. Evo primjera:

```
import ast
code = """
def foo(n):
    total = 0
    for i in range(n):
        total += i
    return total
"""
tree = ast.parse(code)
# Optimiziranje petlje
for node in ast. walk(tree):
    if isinstance (node, ast.Za):
        node.iter = ast.Call(
            ast.Name( "range" , ast.Load()),
            [node. iter .args[ 0 ]])],
            []
    )
# Ispis dobivenog koda
print (astor.to_source(tree))
```

Kod 9

Izlaz ovog koda bio bi:

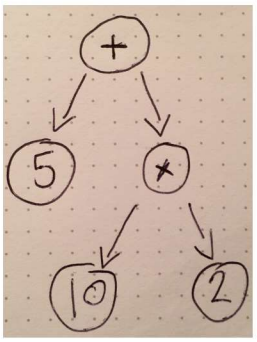
```
def foo (n):
    total = 0
     total += sum (range (n))
    return total
```

Kod 10

Petlja je optimizirana za korištenje sum() funkcije umjesto ručnog zbrajanja vrijednosti. To može dovesti do bržeg i učinkovitijeg koda.

1.3.7 Još jedan primjer primjene AST-a

Pretvorimo neki kod u stabla apstraktne sintakse. Uzmimo na primjer: 5 + (10 * 2) Ovo bi moglo izgledati ovako kao apstraktno stablo sintakse:



Slika 6 : Primjer apstraknog sintaksnog stabla

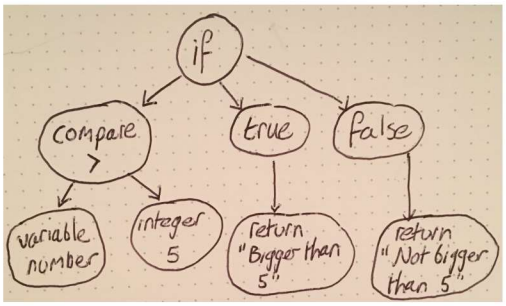
Dostupno na :(`https://tosbourn.com/img/mathmatical_AST.jpg`)

Izraz 5 + (10 * 2) je predstavljen kao AST Imajmo na umu kako nas nije briga za zagrade, jer ono što ovdje govorimo (čitajući iz korijenskog čvora) je da prvo što radimo jest da nešto dodamo. Lijeva strana našeg zbrajanja je broj 5. Desna strana našeg zbrajanja je rezultat množenja. Lijeva strana našeg množenja je 10. Desna strana množenja je 2. Recimo da imamo sljedeći dio koda:

```
if number > 5
    return 'Bigger than 5'
  else
    return 'Not bigger than 5'
```

Kod 11

Ono što ovdje govorimo je da ako je broj veći od pet, vratit će niz "Veći od 5", a ako nije, vratit će "Nije veći od 5". Nije osobito uzbudljiv dio koda, ali pogledajmo kako bi to moglo izgledati kao AST:



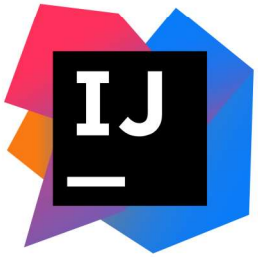
Slika 7 : apstrakno sintaksno stablo if-petlje

Dostupno na :(`https://tosbourn.com/img/if_statement_AST.jpg`)

Ovdje možemo vidjeti primjer čvora s tri djece, imamo jedno koje predstavlja ono što if izjava traži, a zatim dva koja predstavljaju je li ta izjava istinita ili lažna. Vrijedno je napomenuti da bismo mogli zamoliti 5 ljudi da smisle AST za neki kod i mogli bismo dobiti 5 različitih prikaza. Sve dok kod koji tumači AST zna što treba tražiti, u igri mogu biti osobne preferencije.

2 Platforma IntelliJ

Platforma IntelliJ nije proizvod sama po sebi, već pruža platformu za izgradnju IDE-a. Koristi se za napajanje JetBrains proizvoda poput IntelliJ IDEA. Platforma pruža svu infrastrukturu koja je ovim IDE-ovima potrebna za pružanje bogate jezične alatne podrške. To je host aplikacije temeljen na različitim platformama JVM-a s alatom za korisničko sučelje visoke razine za stvaranje prozora alata, prikaza stabla i popisa (podržava brzo pretraživanje), kao i skočnih izbornika i dijaloških okvira. Platforma IntelliJ ima uređivač punog teksta s apstraktnim implementacijama isticanja sintakse , preklapanja koda, dovršavanja koda i drugih značajki za uređivanje obogaćenog teksta . Uključen je i uređivač slika.
Također pruža infrastrukturu za bogato iskustvo otklanjanja pogrešaka, s jezično-agnostičkom naprednom podrškom za prijelomne točke, skupovima poziva, prozorima za praćenje i procjenom izraza. Međutim prava snaga platforme IntelliJ dolazi iz sučelja programske strukture (PSI). Radi se o skupu funkcionalnosti koje se koriste za analizu datoteka, izgradnju bogatih sintaktičkih i semantičkih modela koda i izgradnju indeksa iz tih podataka. PSI pokreće mnoge funkcije, od brze navigacije do datoteka, tipova i simbola, do sadržaja prozora za dovršetak koda i pronalaženja upotrebe, pregleda koda i ponovnog pisanja koda, za brze popravke ili refaktoring, kao i mnoge druge značajke. IntelliJ uključuje parsere i PSI model za mnoge jezike, a njezina proširiva priroda znači da je moguće dodati podršku za druge jezike.



Slika 8 : ikona IntelliJ IDEA

Dostupno na :(`https://upload.wikimedia.org/wikipedia/commons/thumb/9/9c/IntelliJ_IDEA_Icon.svg/768px-IntelliJ_IDEA_Icon.svg.png`)

2.1 Samostalna instalacija

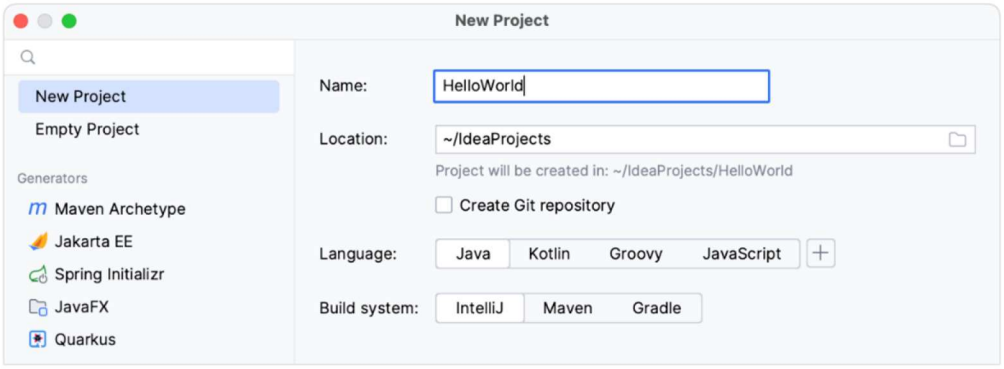
Preuzmimo na originalno stranici JetBrainsa instalacijski program Intellij.exe . Pokrenimo instalacijski program i slijedimo korake čarobnjaka. Napravimo prečac na radnoj površini za pokretanje IntelliJ IDEA. Dodajmo direktorij s IntelliJ IDEA pokretačima naredbenog retka u PATH varijablu okruženja kako bismo ih mogli pokrenuti iz bilo kojeg radnog direktorija u naredbenom retku. Dodajmo radnju "Otvori mapu" kao projekt u kontekstni izbornik sustava.

2.2 Konfiguracija

Pritisnimo "Prilagodi" i kliknimo "Sve postavke" za otvaranje dijaloškog okvira postavki. Postavke koje u ovom trenutku promijenimo postat će nova zadana konfiguracija za projekte i IDE. Pritisnimo "Dodaci" u lijevom oknu i preuzmimo i instalirajmo dodatne dodatke s JetBrains Marketplacea. Za potrebe ovog projekta potrebno je provjeriti jesu li dodaci DevKit i Gradle omogućeni te instalirati dodatke Grammar-Kit i PsiViewer. Napravili smo prazan projekt dodatka za IntelliJ platformu.

2.3 Napravimo novi Java projekt

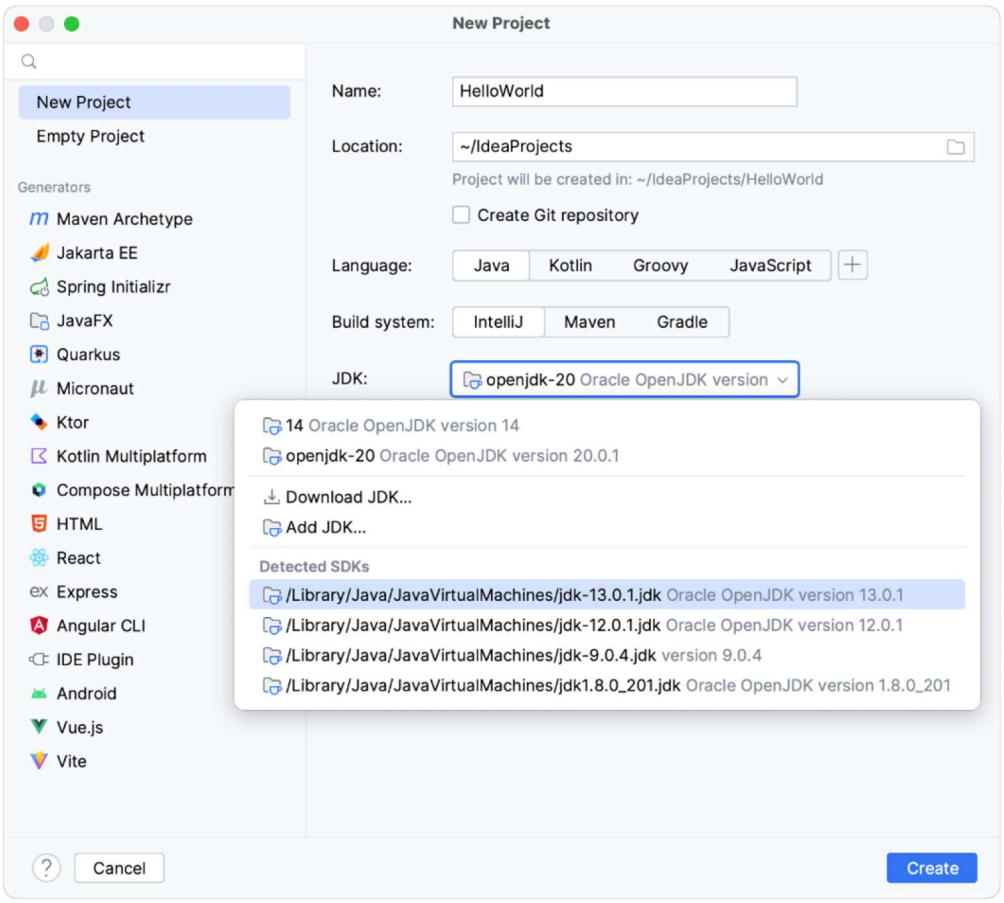
1. Pokrenimo IntelliJ IDEA.
2. Odaberimo File | Novi projekt .
3. U čarobnjaku za novi projekt odaberimo Novi projekt s popisa s lijeve strane.
4. Imenujmo projekt (na primjer HelloWorld) i promijenimo zadanu lokaciju ako je potrebno.
5. Provjerimo je li Java odabrana u Language , a IntelliJ odabrana u Build system .



Slika 9 : Napravimo novi Java projekt

Dostupno na :(`https://resources.jetbrains.com/help/img/idea/2023.2/java-tutorial-new-project.png`)

6. Za razvoj Java aplikacija u IntelliJ IDEA potreban nam je Java SDK (JDK).



Slika 10 : Napravimo novi projekt i dodajmo JDK

Dostupno na
:(`https://resources.jetbrains.com/help/img/idea/2023.2/download-jdk.png`)

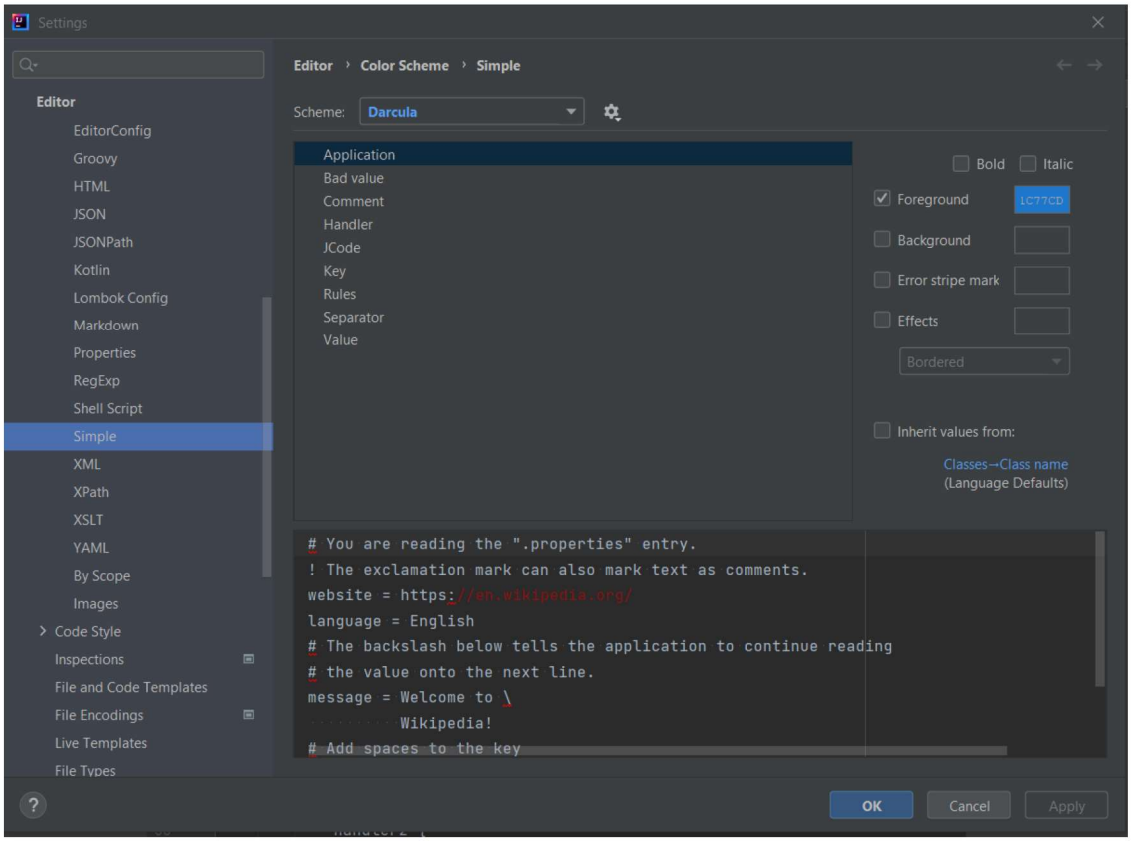
Slika 11 : Instalirajmo JDK

Dostupno na :(`https://resources.jetbrains.com/help/img/idea/2023.2/java-t-create-new-project.png`)

Nakon toga, IDE će stvoriti i učitati novi projekt.

2.4 Sintaksni označivač

Sintaksni označivač (syntax highlighter) igra ključnu ulogu u poboljšanju iskustva programiranja. Za projekt, implementirali smo sofisticirani sintaksni označivač za domensko specifičan jezik kako bismo olakšali razumijevanje i analizu izvornog koda. Sintaksni označivač koristi moćan lekser (lexer) i odgovarajuće definicije tokena kako bi prepoznao različite dijelove koda. Neke od ključnih značajki su bojenje sintakse, komentari te ispravci grešaka. Bojenjem su obuhvaćene ključne riječi, identifikatore, brojeve, nizove i mnoge druge dijelove koda, koji će svi biti obojeni u odgovarajuće boje kako bi se olakšalo razlikovanje. Komentari će biti označeni posebnom bojom kako bismo ih istaknuli i olakšali čitanje i praćenje u kodu, dok će kod grešaka nepoznati znakovi i greške bit jasno označene kako bismo odmah znali kada je nešto u redu s vašim kodom.

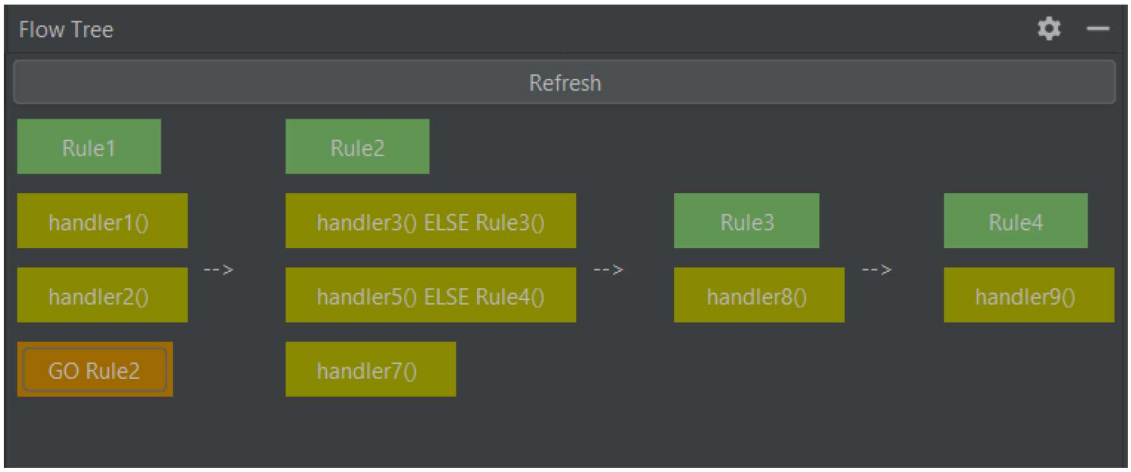


Slika 12 : Sintaksni označivač

Snimka zaslona

2.5 Stablo tijeka

Stablo tijeka (Flow Tree) je platforma koja omogućuje grafičko modeliranje i upravljanje složenim procesima, olakšavajući analizu, praćenje i optimizaciju tokova podataka.



Slika 13 : Stablo tijeka

Snimka zaslona

3 Dokumentacija skripti

3.1 Definiranje jezika

```
public class SimpleLanguage extends Language {

  public static final SimpleLanguage INSTANCE = new SimpleLanguage();

  private SimpleLanguage() {
    super("Simple");
  }

}
```

Kod 12

3.2 Definiranje ikone

Koda 13 predstavlja definiciju klase SimpleIcons, koja se koristi za dohvaćanje ikone (Icon) iz zadanog izvora ikona pomoću biblioteke IconLoader.

```
public class SimpleIcons {

  public static final Icon FILE =
  IconLoader.getIcon("/icons/jar-gray.png", SimpleIcons.class);

}
```

Kod 13

Često koristi u grafičkim aplikacijama kako bi se omogućilo jednostavno dohvaćanje i korištenje ikona za različite elemente korisničkog sučelja, kao što su gumbi, ikone datoteka, ili druge vizualne komponente.

3.3 Definiranje FileType-a

Kod 14 se obično koristi u razvojnom okruženju ili aplikaciji kako bi se omogućilo prepoznavanje i manipuliranje određenim vrstama datoteka. SimpleFileType definira kako će se tip datoteke prikazati u okruženju i što će se događati kada korisnik radi s datotekama tog tipa.

```
public class SimpleFileType extends LanguageFileType {

  public static final SimpleFileType INSTANCE = new SimpleFileType();

  private SimpleFileType() {
    super(SimpleLanguage.INSTANCE);
  }
```

```
  @NotNull
  @Override
  public String getName() {
    return "Simple File";
  }

  @NotNull
  @Override
  public String getDescription() {
    return "Simple language file";
  }

  @NotNull
  @Override
  public String getDefaultExtension() {
    return "simple";
  }

  @Nullable
  @Override
  public Icon getIcon() {
    return SimpleIcons.FILE;
  }

}
```

Kod 14

3.4 Registriranje FileType-a

XML zapis definira kako će se programsko okruženje, poput IntelliJ IDEA, nositi s datotekama ovog tipa.

```
<extensions defaultExtensionNs="com.intellij">
  <fileType
      name="Simple File"
      implementationClass="org.intellij.sdk.language.SimpleFileType"
      fieldName="INSTANCE"
      language="Simple"
      extensions="simple"/>
</extensions>
```

Kod 15

3.5 Definiranje vrste tokena

Klasa se obično koristi u procesu analize izvornog koda. Tokom parsiranja izvornog koda, različiti dijelovi koda predstavljaju različite tipove tokena, kao što su identifikatori, ključne riječi, brojevi, operatori itd. Svaki od tih tipova tokena ima svoj SimpleTokenType koji se koristi za identifikaciju i razlikovanje tokova tokom analize.

```
public class SimpleTokenType extends IElementType {

  public SimpleTokenType(@NotNull @NonNls String debugName) {
    super(debugName, SimpleLanguage.INSTANCE);
  }

  @Override
  public String toString() {
    return "SimpleTokenType." + super.toString();
  }

}
```

Kod 16

3.6 Definiranje vrste elementa

Koristi se u procesu analize izvornog koda. Tijekom analize, različite dijelove koda možemo predstaviti različitim tipovima elemenata kako bi se jasno definirala struktura koda. Primjeri tipova elemenata uključuju ključne riječi, identifikatore, brojeve, operatore i mnoge druge.

```
public class SimpleElementType extends IElementType {

  public SimpleElementType(@NotNull @NonNls String debugName) {
    super(debugName, SimpleLanguage.INSTANCE);
  }

}
```

Kod 17

3.7 Definiranje gramatike

Opisuje gramatiku za jezik u okruženju razvojnog alata IntelliJ IDEA. Ova gramatika opisuje strukturu i pravila za analizu izvornog koda u domensko specifičnom jeziku.

```
{
  parserClass="org.intellij.sdk.language.parser.SimpleParser"

  extends="com.intellij.extapi.psi.ASTWrapperPsiElement"

  psiClassPrefix="Simple"
  psiImplClassSuffix="Impl"
  psiPackage="org.intellij.sdk.language.psi"
  psiImplPackage="org.intellij.sdk.language.psi.impl"

  elementTypeHolderClass="org.intellij.sdk.language.psi.SimpleTypes"
  elementTypeClass="org.intellij.sdk.language.psi.SimpleElementType"
  tokenTypeClass="org.intellij.sdk.language.psi.SimpleTokenType"
}

simpleFile ::= item_*

private item_ ::= (property|COMMENT|CRLF)

property ::= (KEY? SEPARATOR VALUE?) | KEY
```

Kod 18

3.8 Definiranje leksera

Kod 19 je dio definicije leksera, za analizu izvornog koda napisanog u nekom jeziku. Lekser se koristi za razdvajanje niza znakova na tokene. Svaki token predstavlja jedan tip znakova, poput ključnih riječi, brojeva, identifikatora ili simbola.

```
SPACE=[ \t\n\x0B\f\r]+
COMMENT="//".*
NUMBER=[0-9]+(\.[0-9]*)?
ID=[:letter:][a-zA-Z_0-9]*
STRING=('([^'\\]|\\.)*'|\"([^\"\\]|\\.)*\")
Z=("/"[:letter:][:letter:]*)*
SYNTAX=;|\.|\+|-|\*\*|\*|==|=|,|\(|\)|\^|\!=|\!|>=|<=|>|<|\{|\}

%%
<YYINITIAL> {
   {WHITE_SPACE}        { return WHITE_SPACE; }

   "else"               { return ELSE; }
   "GO"                 { return GO; }
   "SEPARATOR"          { return SEPARATOR; }
   "KEY"                { return KEY; }
   "VALUE"              { return VALUE; }
   "float"              { return FLOAT; }
   "Application"        { return APPLICATION; }
   "Rules"              { return RULES; }
   "Handler"            { return HANDLER; }
   "JCode"              { return JCODE; }
   "JCodeSig"           { return JCODESIG; }

   {SPACE}              { return SPACE; }
   {COMMENT}            { return COMMENT; }
   {NUMBER}             { return NUMBER; }
   {ID}                 { return ID; }
   {STRING}             { return STRING; }
   {Z}                  { return Z; }
   {SYNTAX}             { return SYNTAX; }

}

[^] { return BAD_CHARACTER; }
```

Kod 19

3.9 Definiranje parsera

Lekser zajedno s odgovarajućim parserom često se koristi za pretvaranje izvornog koda u strukturu podataka koja se može dalje analizirati i obraditi.

```
public class Parser {

    public static void main(String[] args) {
        String document =
                "        Rules { // keyword\n" +
                "            Rule1 = {\n" +
                "                handler1()\n" +
                "                handler2()\n" +
                "                GO Rule2\n" +
                "            }.\n" +
                "            Rule2 = {\n" +
                "                handler3() else Rule3\n" +
                "                handler5() then Rule4\n" +
                "                handler7()\n" +
                "            }.\n" +
                "            Rule3 = {\n" +
                "                handler8()\n" +
                "            }.\n" +
                "            Rule4 = {\n" +
                "                handler9()\n" +
                "            }.\n" +
                "        } ";

        Map<String, List<RuleEntry>> rulesMap = extractRules(document);
        rulesMap.forEach((key, value) -> {
            System.out.println(key);
            value.forEach(v -> System.out.println(v.toString()));
            System.out.println("\n");
        });
    }
```

Kod 20

4 Zaključak

Apstraktno sintaksno stablo (AST) predstavlja moćnu strukturu podataka koja omogućava apstraktno predstavljanje sintaktičke strukture izvornog koda. Ova apstrakcija eliminira suvišne detalje i omogućava programerima analizu i manipulaciju programskim konstrukcijama.

AST ili jednostavno stablo sintakse, pruža sliku apstraktne sintaktičke strukture teksta napisanog u formalnom jeziku. Svaki čvor u AST-u označava konstrukciju prisutnu u izvornom kodu. Važno je napomenuti da AST zadržava samo bitne strukturalne i sadržajne detalje, ne opterećujući se svakim pojedinim aspektom stvarne sintakse.

Ova apstrakcija razdvaja AST od stabala konkretnih sintaksa koja često prate svaki detalj izvornog koda. AST se široko koristi u analizi programa i sustavima za transformaciju koda.

Iako se AST-ovi možda ne koriste svakodnevno u radu većine programera, važno je prepoznati njihovu vrijednost kao korisnog alata za dublje razumijevanje i učinkovitu manipulaciju kodom. AST dolazi najviše do izražaja u radu s kompilatorima, u smislu reprezentacija sintakse, analize, optimizacije i generiranje koda te pri sintaksnoj analizi grešaka.

Literatura

[1] Understanding Abstract Syntax Trees (AST) in Software Development, dostupno na: (https://levelup.gitconnected.com/understanding-abstract-syntax-trees\-ast-in-software-development-e8c2b1957f0a)

[2] Wikipedia - Abstract Syntax Tree, dostupno na: (https://en.wikipedia.org/wiki/Abstract_syntax_tree)

[3] ASTs - What are they and how to use them (https://www.twilio.com/blog/abstract-syntax-trees)

[4] Abstract Syntax Trees, dostupno na: (https://tosbourn.com/abstract-syntax-trees/)

[5] Abstract Syntax Tree (AST) - Explained in Plain English, dostupno na: (https://dev.to/balapriya/abstract-syntax-tree-ast-explained-in-plain\-english-1h38)

Popis slika