

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 233

**Parsiranje domenskih jezika
koristeći Prattov parser**

Marin Jovanović

Zagreb, lipanj 2021.

Sadržaj

Uvod	1
1. Prattova tehnika parsiranja	2
1.1. Vrste parsiranja.....	2
1.2. Od klasičnog pristupa do Prattove metode	2
1.3. Obilježja Prattove metode	5
1.4. Upravljanje greskama	11
1.5. Poziv funkcija.....	12
1.6. Ostali operatori	15
2. Prattova metoda naspram drugih	17
3. Primjena.....	19
4. Prednosti i nedostaci.....	20
Zaključak	22
5. Daljnji rad	23
Literatura	24
Sažetak.....	26
Summary.....	27
Skraćenice.....	28
Programska podrška	29

Uvod

Parsiranje je postupak prepoznavanja uzoraka iz nekog objekta i gradnje strukture koja ga opisuje na temelju zadane gramatike. Struktura koja se gradi je proizvoljnog oblika te je gramatika također proizvoljna kao i njen tip [1]. Ovoliko slobodna definicija upućuje na veliku domenu primjene što se i u praksi pokazalo da i jest tako. Općenito ako je neki algoritam primjenjiv na široki spektar problema možda bi bilo mudro implementirati opće programsko rješenje koje će riješiti svaki od tih problema ili napraviti generator koji će generirati rješenje za svaki takav problem ili pak osmisliti postupak koji će biti dovoljno jednostavan da se može lako implementirati za svaki predstavljeni problem.

Mnogi su pokušali napraviti opća programska rješenja za parsiranje ili generatore parsera te su ta rješenja prihvaćena. Trećim putem krenuo je Vaughan Pratt predloživši metodu parsiranja od vrha prema dnu uz prednost operatora¹.

Kao argument za ovaj pristup naveo je da postoje dvije struje svijesti o oslanjanju na sintaksu pri oblikovanju i ostvarenju programskih jezika. Njegov prijedlog rješenja je da se pristupa hibridno kako bi se postigao kompromis između tih podvojenih mišljenja. Glavne značajke bi bile da je njegovo rješenje trivijalno za implementirati te da nije ograničeno BNFom [2].

Njegova metoda na elegantan način rješava inače dosta zamršene probleme s kojima je često suočen razvijatelj parsera kao što su na primjer lijeve rekurzije i upravljanje odnosno oporavak od pogrešaka.

U sklopu ovog rada fokus će biti na njegovoj metodi, usporedbi sa sličnim metodama, položaju njegove metode u svijetu parsera te praktični primjeri. Parsirati ćemo programske jezike iako je ideje moguće poopćiti i primijeniti na druge strukture.

¹ <https://dl.acm.org/doi/10.1145/512927.512931>

1. Prattova tehnika parsiranja

1.1. Vrste parsiranja

Sve metode parsiranja se mogu svrstati u dvije kategorije: parsiranje od vrha prema dnu i parsiranje od dna prema vrh. Prva metoda pokušava za zadani ulaz generirati strukturu (najčešće stablo, specifičnije apstraktno sintaksno stablo) tako što pokušava spojiti prvi token ulaza s nekom produkcijom gramatike. Na temelju prvotne produkcije pokušava se slijediti nadolazeće produkcije i spojiti ih s odgovarajućim tokenima i tako u konačnici izgraditi željenu strukturu. Drugi pristup je da se obrne prethodno opisani proces u nadi da će taj način biti povoljniji [1].

Vaughan Pratt je predstavio metodu koja spada u prvu skupinu te koristi prednost operatora te je stoga naziva parsiranje od vrha prema dnu uz prednost operatora.

1.2. Od klasičnog pristupa do Prattove metode

Metode se kompliciraju kada je za određeni token moguće primijeniti više produkcija odnosno kada je moguće krenuti parsirati u različitim smjerovima u nadi da će jedan od tih smjerova biti valjan za zadani ulaz [1].

Mogli bi formalno reci da je to problem pretraživanja stanja. Tada bi algoritme mogli ocjenjivati po tome koliko brzo pretražuju ta stanja te kakva im je kvaliteta intuicije odnosno heuristike. Heuristika se može zadati već u samoj specifikaciji gramatike, ali je isto tako moguće natuknuti programu kojim putem ići pri samoj implementaciji. Za neke jednostavnije implementacije (jezike) nije teško odrediti kako bi se parser trebao ponašati, i moguće je parsiranje odraditi na ad-hoc način, ali kako jezik kojim se bavimo postaje složeniji tako i sam parser postaje kompliciraniji. Dobra praksa bi nalažala da kod bude dobro strukturiran i da slijedi određena pravila odnosno da nema odstupanja od nekakve globalne logike. Prattova tehnika upravo slijedi takav princip, on nudi generičko rješenje u koje mi usađujemo heuristiku, a generirano strukturu oblikujemo po želji.

Usmjerimo li pažnju na parsiranje od vrha prema dnu vrlo je vjerojatno da ćemo algoritam temeljiti na rekurzivnom spustu. Problem s kojim ćemo se možda susresti je lijeva rekurzija [3].

Prikažimo sada naivan pristup pri rješavanju takvog problema.

Recimo da postoji produkcija

$$S \rightarrow Sa \mid a$$

Očito je da će program uz iduću funkciju zaglaviti u rekurziji za neke ulaze (Kôd 1.1):

```
def S():
    # TODO try
    S()
    a()

    # TODO try if first production failed
    a()

    # TODO raise exception if both failed
```

Kôd 1.1 – neispravna funkcija S

Rješenje kojim bi mogli ovaj problem riješiti je iduće (Kôd 1.2):

```
def S():
    a()
    while True:
        # TODO break when following production can not be done
        a()
```

Kôd 1.2 – Ispravna funkcija S

Prethodno rješenje je valjano. Pratt je to rješenje poopćio i time dobio funkciju `expression` [3]. Kostur algoritma bazira se na temelju ovog članka², u kojem je prikazana inkrementalna izrada parsera, poslužiti ćemo se istom metodom i u ovom radu.

Tim poopćavanjem je zadovoljio dobru praksu, štoviše, prisilio je da se cijeli parser fokusira na ovu (Kôd 1.3) funkciju. Uz pomoćne klase koje ćemo uvesti prikazati ćemo veliku modularnost algoritma.

```
def expression(rbp=0):
    global token
    t = token
    token = lexer.__next__()
    while rbp < token.lbp:
        t = token
        token = lexer.__next__()
        left = t.nud(left)
    return left
```

Kôd 1.3 – Funkcija `expression`

Razlika ovog pristupa od onog opisanog u Kôd 1.2 je što se poopćava mehanizam vezan za lijevu rekurziju te više nije potrebno pisati isti dio koda svaki put kada se susretnemo s njom.

Ideja je da ovo glavna funkcija `expression` pokuša spojiti trenutni (početni) token s tokenima s desne strane uz pomoć funkcije `nud` ako to ima smisla te nakon toga pokušava spojiti tokene koje je parser već prošao s trenutnim tokenom. Ta funkcionalnost se ostvaruje uz pomoć funkcije `led` te se kontrolira koliko će se ona puta zvati unutar petlje pomoću prednosti operatora. Primjeri će biti dani u idućem poglavlju.

Dio programa koji gradi tokene nećemo previše razmatrati u ovom radu. Možemo pretpostaviti da će funkcija `lexer.__next__()` vratiti objekt tokena koji bi trebao doći nakon trenutnog tokena, a u slučaju da ne postoji više tokena trebao bi se vratiti objekt `TokenEOF` koji simbolizira kraj izvornog koda kojeg želimo parsirati.

² <https://eli.thegreenplace.net/2010/01/02/top-down-operator-precedence-parsing>

Sada je potrebno dodati module koji će koristiti ovu funkciju, ona će njima upravljati i parsirati ih s obzirom na njihovu važnost (pomoću LBP i RBP, ali o tome kasnije).

1.3. Obilježja Prattove metode

Ono što je još potrebno definirati su klase tokena. Razmatrat ćemo jednostavni jezik koji će se sastojati samo od matematičkih operacija, varijabli i konstanti. To je primjer koji se većinom koristi kako bi se prikazala sva moć Prattove metode pa tako i u članku³ iz kojeg smo preuzeli kostur koda i nadogradili ga. Nadogradnjom smo željeli demonstrirati još neka zanimljiva svojstva i mogućnosti algoritma.

Razlog parsiranja samo matematičkog dijela jezika je intuitivna prednost operatora. Također je za primijetiti da veliki broj jezika koristi sličnu gramatiku za matematičke izraze i to je komponenta parsiranja koju je moguće odvojiti u posebnu rutinu parsera i onda parsirati posebno cijeli jezik, a kada se dođe do matematičkih izraza parsirati ih kao posebnu komponentu (kao zaseban dio jezika).

Pod prednosti operatora u nekom izrazu (kodu) smatrat ćemo razrješava (spaja) li se neki token s tokenom ili tokenima desno od njega ili s tokenom ili tokenima lijevo od njega. Trivijalan i očigledan primjer je da ćemo izraz $1 - 2 * 3$ razriješiti kao $1 - (2 * 3)$, a ne slijedno od lijeva na desno. Manje intuitivan primjer bi bio $int\ x = 1;$, ovdje bi puno teže bilo odrediti što ćemo smatrati lijevom, a što desnim spajanjem. O ovome će biti govora u poglavlju [Prednosti i nedostaci](#).

U svrhu ove demonstracije gradit ćemo apstraktno sintakso stablo umjesto direktnog uvrštavanja i računanja izraza kao što je rađeno u navedenom radu jer time možemo bolje razumjeti što se događa.

Kako bi mogli razlikovati tokene i ukalupiti logiku koja bi rekla na koji se način spaja određeni token poslužiti ćemo se klasama. Prvi korak je napraviti klasu Token (Kôd 1.4) koju će svaka nova klasa tokena nasljeđivati.

```
class Token(object):
    def __init__(self, value):
        id = self.__class__.__name__
```

³ <https://eli.thegreenplace.net/2010/01/02/top-down-operator-precedence-parsing>

```

self.identifier = id
self.value = value
self.lbp = LBP.get(id)
self.rbp = RBP.get(id)

def nud(self):
    pass

def led(self, left):
    pass

```

Kôd 1.4 – klasa Token

Svaki token će imati svoju identifikacijsku oznaku identifier koja će biti istovjetna imenu klase, vrijednost value, težinsku vrijednost lijevog pridruživanja lbp (eng. left binding power) i težinsku vrijednost desnog pridruživanja rbp (eng. right binding power).

LBP i RBP predstavljaju dva rječnika koji će kao ključeve imati id tokena, a kao vrijednost samu vrijednost pridruživanja. Radi preglednosti dati ćemo ta dva rječnika kao tablicu (Tablica 1.1). Tokeni koji ne koriste svoju lbp i rbp vrijednost nisu prikazani u tablici, za njih možemo pretpostaviti da imaju vrijednost manju ili jednaku nula.

Tablica 1.1 LBP i RBP rječnici

LBP		RBP	
Token	Vrijednost	Token	Vrijednost
		Subtraction	13
Power	12		
Division	10		
Multiplication	10		
		Power	9
Subtraction	8		
Addition	8		
		LeftBracket	7
		Assignment	5
Ternary	4		
Assignment	2		

		If	1
--	--	----	---

Također će svaki token morati implementirati svoju funkciju `nud` i `led` koja će nadjačati postojeće. Implementirati će te funkcije jedino ako ih koristi, za neke tokene jednostavno nema smisla nadjačavati jednu od njih, primjere ćemo dati kasnije.

Za navedeni primjer $1 - 2 * 3$ potrebno je implementirati tri klase tokena; literale (Kôd 1.5), tokene za oduzimanje (Kôd 1.6) i množenje (Kôd 1.7). Valja primijetiti da token `-` ima i `nud` i `led` metodu jer se minus može naći prije i nakon drugih tokena i da tvori sintaksni i semantički smisao.

```
class TokenLiteral(Token):
```

```
    def nud(self):
```

```
        return self.value
```

Kôd 1.5 – klasa `TokenLiteral`

```
class TokenSubtraction(Token):
```

```
    def nud(self):
```

```
        return [„-“, expression(self.rbp)]
```

```
    def led(self, left):
```

```
        return [„-“, left, expression(self.lbp)]
```

Kôd 1.6 – klasa `TokenSubtraction`

```
class TokenMultiplication(Token):
```

```
    def led(self, left):
```

```
        return [„*“, left, expression(self.lbp)]
```

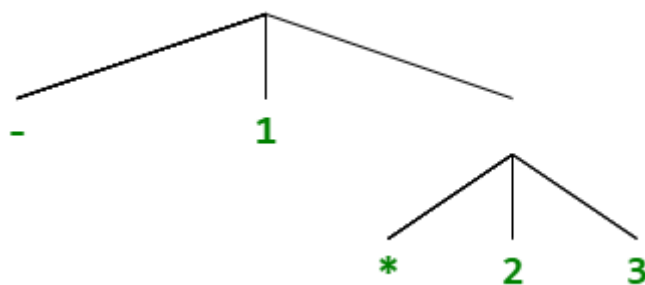
Kôd 1.7 – klasa `TokenMultiplication`

Potrebno je pozvati funkciju `expression` koja će imati zadanu vrijednost `rbp` jednaku 0, prvo što će se dogoditi je da će se za token 1 nad klasom `TokenLiteral` zvati metoda `nud` koja će vratiti vrijednost 1. Nakon toga funkcija `expression` ulazi u `while` petlju u kojoj se zove za token – nad klasom `TokenSubtraction` metoda `led` koja vraća `[-, 1, expression(8)]`, za `left` se uzima vrijednost svega predano lijevo a to je samo broj 1, a za desni argument (zadnji element liste) vraća se `expression(8)` od kojeg očekujemo da vrati parsirani oblik `(2 * 3)`.

Program ulazi u funkciju `expression(8)` i `t.nud()` vraća 2 te nakon toga ulazi u `while`, unutar `while`-a zove se `t.led(left)` koja vraća `["*", 2, expression(10)]`.

Funkcija `expression(10)` za `t.nud()` vraća 3 te ne prolazi kroz `while` jer je trenutni token `TokenEndOfFile` koji nam govori da smo sve tokene iskoristili. Funkcija `expression(10)` tada vraća samo vrijednost 3.

Sada funkcija `expression(8)` može vratiti `["*", 2, 3]` te se to ubacuje u početni `expression()` koji vraća `[-, 1, ["*", 2, 3]]`. Takva struktura predstavlja AST koji možemo grafički prikazati (**Error! Reference source not found.**).



Slika 1.1 AST za izraz `1 - 2 * 3`

Idući korak bilo bi implementacija upravitelja pogreškama. Sam kod nam djelom i nameće da razmišljamo o rubnim slučajevima odnosno o ne-standardnom načinu parsiranja (odstupanju od zadanog). Primjer toga bi mogle biti zagrade, ali za to bi morali prvo uvesti dvije nove klase (Kôd 1.8 i Kôd 1.9).

```
class TokenLeftBracket(Token):
```

```

def nud(self):

    global token

    expr = expression(self.rbp)

    if not isinstance(token, TokenRightBracket):

        print(„error right bracket“)

        import sys

        sys.exit()

    r_b = token.value

    token = lexer.__next__()

    return [self.value, expr, r_b]

```

Kôd 1.8 – klasa TokenLeftBracket

```

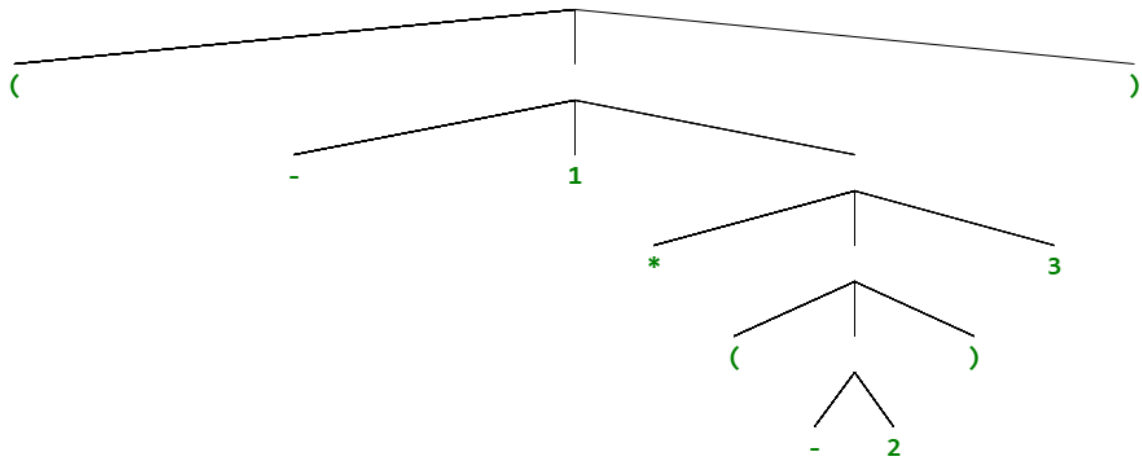
class TokenRightBracket(Token):

    pass

```

Kôd 1.9 – klasa TokenRightBracket

Zamislamo sada da parsiramo izraz $(1 - (-2) * 3)$. Kako bi to omogućili trebali smo implementirati klasu lijeve i desne zagrade. Algoritam sada prvo uzima lijevu zagradu te nad njom zove metodu nud koja prvo zove funkciju expression(self.rbp). Ideja je da se sada isparsira sve unutar te zagrade i vrati se dio AST oblika $[(„(“, expression(self.rbp), „)“)]$. Valja primijetiti ugniježdene zagrade koje će naš parser s lakoćom pravilno isparsirati. Ideja je da je vrijednost self.rbp veća od vrijednosti TokenRightBracket.lbp. Time osiguravamo da će se parsirati ulaz sve do prve desne zagrade. Mogli bi si sada postaviti pitanje što je s ugniježdenom zagradom i odgovor je da je stvar ista. Ponovno će se krenuti parsirati izraz unutar zagrada sve dok ne naleti na prvu desnu zagradu, u ovom slučaju vratit će se $[('(', ['-', '2'], ')]$. Taj će se izraz vratiti expression funkciji koju je pozvala prva lijeva zagrada te će ona vratiti $['-', '1', ['*', ['(', ['-', '2'], ')], '3']$. Nakon toga sve skupa vraća AST oblika $[('(', ['-', '1', ['*', ['(', ['-', '2'], ')], '3'], ')]$ (Slika 1.2).



Slika 1.2 AST za $(1 - (- 2) * 3)$

Zanimljivost ovog načina rješavanja zagrada je što možemo lako različite vrste zagrada tretirati isto, potrebno je samo pri stvaranju objekta definirati vrijednost `self.value`.

U ovom primjeru se za token – pozivala metoda `nud`, a ne `led`. Razlog ovoga je što je to bio prefiks broju 2, nije se oduzimao 2 od nekog drugog broja. Možemo vidjeti kako se u ovom slučaju parsirao broj desno od tog tokena, a to je upravo 2. To je tako jer se zvala funkcija `expression` s vrijednosti 13 koji je maksimalan u našoj tablici, a to onda osigurava da se neće izvršiti `while` petlja u toj funkciji i vratiti će samo `nud` koji odgovara broju 2. Ovaj primjer je moguće poopćiti; kada je potrebno da neki token vrati token koji je njemu s desna to je ostvarivo tako da se pozove funkcija `expression` s argumentom većim od svih vrijednosti `lbp`. Naravno, ako taj token u svojoj `nud` metodi zove ponovo funkciju `expression` onda će se i to morati razrješavati, ali uvijek će se vratiti samo vrijednost tog nuda, neće se izvršavati `while` dio tog `expressiona`.

Sada bi se moglo postaviti pitanje zašto se nije to onda odradilo za desnu zagradu i odgovor se nalazi u upravljanju greškama, ali i optimizaciji. Puno je lakše provjeriti odmah je li idući token desna zagrada i onda upravljati daljnjim tijekom ovisno o odgovoru nego da se poziva funkcija `expression` koja će vratiti samo jedan token i onda bi opet morali provjeriti je li vraćena desna zagrada.

1.4. Upravljanje greškama

U ovom primjeru gdje očekujemo desnu zagradu mogli bismo ispisati informaciju o grešci i probati nastaviti s parsiranjem. Ako pretpostavimo da pokušavamo parsirati neki izvorni kod i ustanovimo da bi negdje trebala postojati zagrada, a nje nema mogli bismo ispisati poruku i svejedno nastaviti program kao da je idući token bila odgovarajuća zagrada. Tada bi nastavili s parsiranjem i mogli ustanoviti postoji li još koja pogreška u semantici koda. Za to bi trebali proširiti kod klase tokena za lijevu zagradu (Kôd 1.10).

```
class TokenLeftBracket(Token):

    def nud(self):

        global token

        expr = expression(self.rbp)

        if not isinstance(token, TokenRightBracket):

            print(„error right bracket“)

            #TODO determinate which type of bracket is needed

            r_b = „err )“

        else:

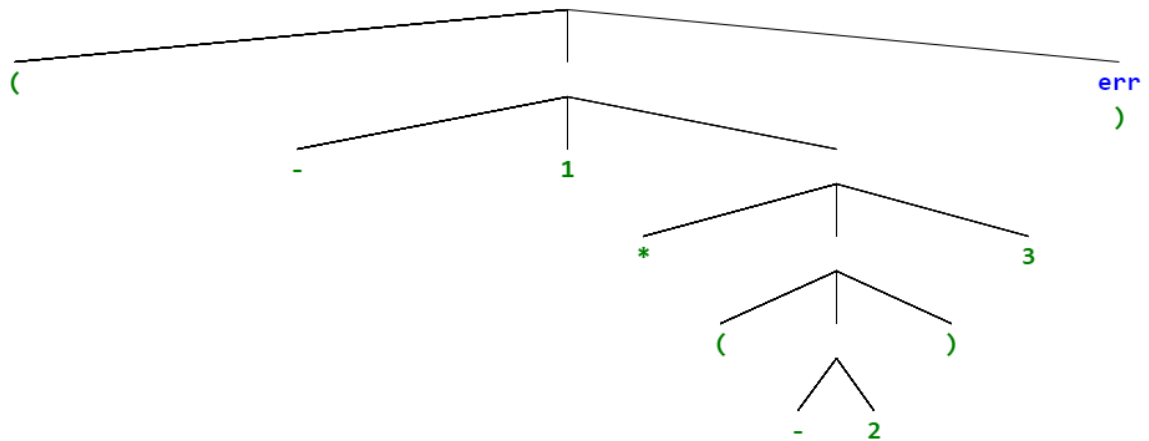
            r_b = token.value

            token = lexer.__next__()

        return [self.value, expr, r_b]
```

Kôd 1.10 – klasa TokenLeftBracket

Iz metode nud smo makli dio koji prekida program i dodali TODO u kojem je potrebno utvrditi na temelju lijeve zgrade (self.value) koja bi zagrada trebala doći s desne strane, to je lako napraviti ako je poznato koje sve vrste zagrada mogu doći s lijeve strane. U ovom primjeru smo dali fiksnu vrijednost „err)“ kako bi se bolje vidjelo to u ispisu stabla. Nismo pridijelili tokenu novu vrijednost putem funkcije lexer.__next__() jer je već vrijednost tokena iduća vrijednost. Ispis stabla je prikazan u nastavku (Slika 1.3) za ulaz (1 – (-2 * 3).



Slika 1.3 AST za (1 – (-2 * 3)

Moglo bi se sada zapitati kako za izraz $1 + (2 - (3 + 4))$ ustanoviti je li $1 + (2 - (3) + 4)$ ili $1 + (2 - (3 + 4))$ i odgovor na to ovom metodom ne možemo dati, ali je ovim pristupom osigurano parsiranje ovog dijela izvornog koda bez ispadanja programa i to nam je dovoljno. Programer će dobiti obavijest o grešci i sam će utvrditi na kojoj poziciji nedostaje desna zagrada.

1.5. Poziv funkcija

U sklopu jednog kalkulatorskog jezika imalo bi smisla dodati pozive trigonometrijskih funkcija. Pozivi funkcija u drugim (stvarnim) jezicima su slični pa bi se iz ovog primjera (Kôd 1.11) mogli svi drugi pozivi funkcija iskonstruirati stoga ovo smatramo vrijednim primjerom.

```

class TokenTrigonometry(Token):
    def nud(self):
        globa token
        if not isinstance(token, TokenLeftBracket):
            print(„error left bracket“)
            #TODO determinate which type of bracket is needed
            l_b = „err (“
        else:

```



```

        l_b = token.value

        token = lexer.__next__()

    expr = expression()

    if not isinstance(token, TokenRightBracket):

        print(„error right bracket“)

        #TODO determinate which type of bracket is needed

        r_b = „err )“

    else:

        r_b = token.value

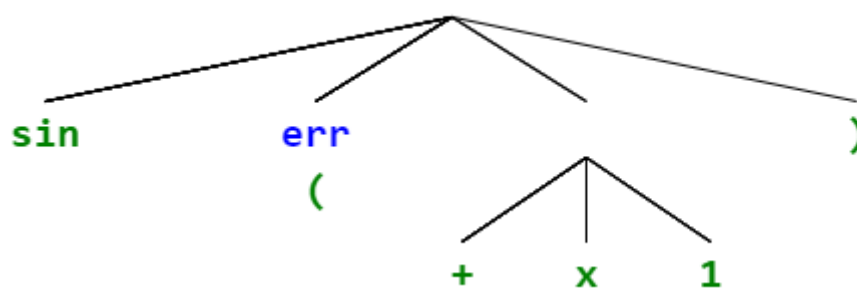
        token = lexer.__next__()

    return [self.value, expr, r_b]

```

Kôd 1.11 – klasa TokenTrigonometry

Za izraz $\sin x + 1$) dobili bi ['sin', 'err (', ['+', 'x', '1'], ')'] što je u skladu s očekivanjima, AST bi grafički izgledao ovako (Slika 1.4):



Slika 1.4 AST za $\sin x + 1$)

Još bi bilo zanimljivo pogledati kako bi se parsirao izraz `if {expr} then {expr} else {expr}`. Od ovoga ne bi imali preveliku korist u kalkulatoru i nije nešto što se pojavljuje u konvencionalnim kalkulatorima makar bi mogli imati neku korist od recimo izraza `1 + if 2`

– 6 == 5 + 1 then 4 else 3. Svrha ovoga je demonstracija važnosti vrijednosti lbp i rbp. Za ovo bi nam potrebne bile opet nove klase (Kôd 1.12, Kôd 1.13 i Kôd 1.14).

```
class TokenIf(Token):  
    def nud(self):  
        globa token  
        if_e = expression(self.rbp)  
        if not isinstance(token, TokenThen):  
            print(„error then“)  
            then_t = „then“  
        else:  
            then_t = token.value  
            token = lexer.__next__()  
        then_e = expression()  
        if isinstance(token, TokenElse):  
            else_t = token.value  
            token = lexer.__next__()  
            else_e = expression()  
            return [self.value, if_e, then_t, then_e, else_t, else_e]  
        else:  
            return [self.value, if_e, then_t, then_e]
```

Kôd 1.12 – klasa TokenIf

```
class TokenThen(Token):  
    pass
```

Kôd 1.13 – klasa TokenThen

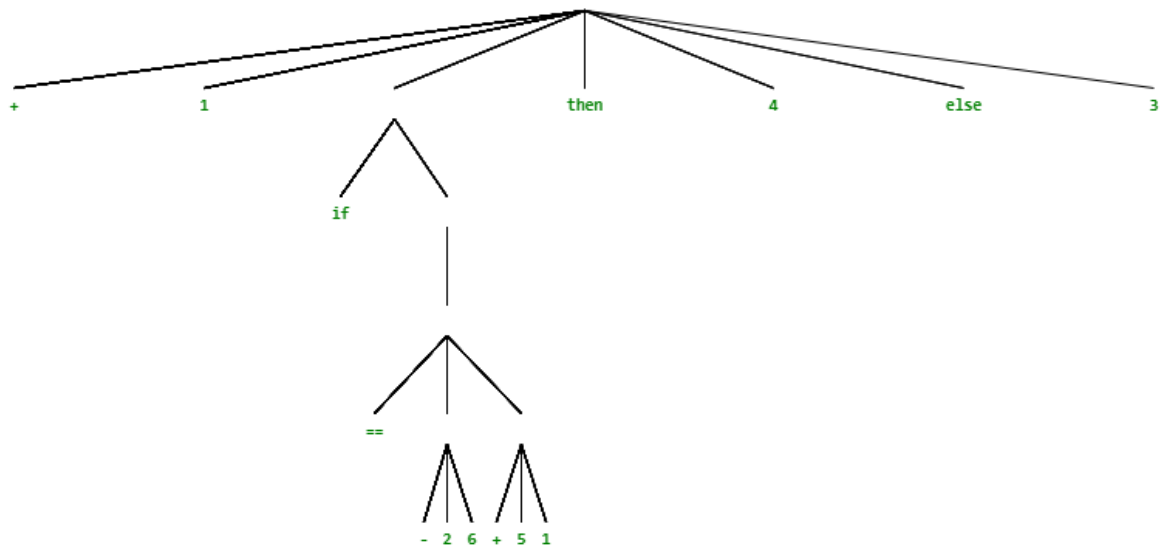
```
class TokenElse(Token):
```

pass

Kôd 1.14 – klasa TokenElse

Primijetimo kako za token TokenElse ne radimo provjeru nego na osnovu njega upravljamo hoćemo li razrješavati then dio ili ćemo razaznati da on ne postoji.

Za naš primjer AST bi izgledao (Slika 1.5):



Slika 1.5 AST za izraz $1 + \text{if } 2 - 6 == 5 + 1 \text{ then } 4 \text{ else } 3$

Ideja je da se krene parsirati expression nakon if koji vraća uvjet koji se ispituje i staje na then nakon kojeg ide ponovno expression sve do else i nakon toga se obrađuje else na jednak način.

U izvornom kodu rada je još dodan i ternarni operator nešto jednostavnije sintakse, ali jednake značenjske vrijednosti koji nema smisla ponavljati jer je slična stvar.

1.6. Ostali operatori

U kodu su dodani i drugi klasični matematički operatori, zainteresiranog čitatelja se poziva da ih prouči. Trenutni podržani tokeni su:

- ☐ literal (brojevi)
- ☐ zbrajanje

- ☐ oduzimanje
- ☐ množenje
- ☐ dijeljenje
- ☐ potenciranje
- ☐ lijeve zagrade
- ☐ desne zagrade (vise vrsta)
- ☐ neki znakovi jednakosti
- ☐ trigonometrijske funkcije (sin, cos, tan)
- ☐ boolean vrijednosti
- ☐ ternarni operator
- ☐ if-then-else struktura
- ☐ varijable

2. Prattova metoda naspram drugih

Ustanovivši da postoji široka potreba za parsiranjem napravljeni su mnogi alati i metode za generiranje parsera. Primjeri takvih alata su Yacc⁴ i Bison⁵. Za generičke metode kojima se ostvaruju parseri usko je vezana teorija automata. Česta ostvarenja su LR parseri.

Ovakva rješenja su adekvatna kada nije bitna optimalnost parsera ili kada je vrijeme bitan faktor. Također je moguće ostvariti puno bolje upravljanje greškama odnosno ispis grešaka [2]. Za slučajeve kada to nije tako razvijatelj koda ima mogućnost primijeniti Prattovu metodu parsiranja i time ostvariti puno modularniji (personaliziraniji) kod.

Postoje sličnosti između Prattovog algoritma i Shunting Yard algoritma uz razliku što drugi koristi stog umjesto rekurzije. Također je jasna sličnost između Prattove metode i algoritma „precedence climbing“ gdje je logika u principu ista, ali uz drugaciju formulaciju [6]. Također postoje i neke jasne razlike između navedenih algoritma stoga ih ne možemo smatrati identičnima [8]. Smatra se da je Dijskrta generalizirao shunting-yard metodu parsiranja koristeći samo jedan stog ili uz pomoć rekurzivnog spusta. Nakon toga Pratt ulazi u priču sa svojim algoritmom koji je po mnogočemu bolji od prethodne Dijskrtine metode [9].

Zanimljivo je primijetiti i sličnost s obradom prekida u operacijskim sustavima. Postoje programske implementacije koje čekaju prekidi i obrađuju ga samo u slučaju ako je veći od prekida koji se trenutno obrađuje. Algoritam je rekurzivan i implementacijom podsjeća na Prattovu tehniku. Pseudo kod i njegovo objašnjenje dano je na uputama za laboratorijsku vježbu predmeta Operacijski sustava na FERu⁶.

Zanimljiva je i simbioza ovog algoritma s ostalim rješenjima.

⁴ <http://dinosaur.compilertools.net/yacc/>

⁵ <https://www.gnu.org/software/bison/>

⁶ <http://www.zemris.fer.hr/predmeti/os/pripreme/z1b.html>

Pokazalo se iznimno jednostavno napraviti algoritam za parsiranje matematičkih izraza koji je opisan u ovom radu, ali za ostatak jezika već je argumentirano da i nije uvijek tako jednostavno.

Preporuka je koristiti kombinaciju parsera u smislu da se koristi glavni parser za parsiranje cijelog jezika izuzev dijela koji služi za pridruživanje vrijednosti. Za tu svrhu ima smisla koristiti Prattov pristup jer se tada izvlači ono najbolje iz njega. Naravno, implementacija će uvijek ostati na razvijatelju i njegovim afinitetima.

Teorija automata je često u službi parsera. Različite gramatike služe za opisivanje jezika dok razni generatori generiraju lexera i parsere. Većinom su to LR parseri, njegove izvedenice i algoritmi bazirani na rekurzivnom spustu.

Programer nema potrebu znati išta iz teorije automata ili gramatike jezika kako bi shvatio kako ova metoda funkcionira i to nosi nekakav značaj.

3. Primjena

Pošto smo pokazali da je najbolja primjena Prattova metoda na matematičke izraze ona je većinom našla primjenu u tom polju. Očekivana primjena su aplikacije za rješavanje matematičkih zadataka. Aplikacija koja koristi takav pristup je Desmos⁷. U svom članku⁸ objasnili su zašto su prešli na Prattovu metodu i kakva im je to poboljšanja donijelo. Glavni razlog im je bio upravljanje pogreškama koje generatori parsera nisu nudili. S novom implementacijom uspjeli su ostvariti manje memorijsko opterećenje i veću brzinu parsiranja. S druge strane navode i neke negativne strane poput velikog broja rekurzivnih poziva i laku mogućnost uvođenja neefikasnosti [5].

Drugi primjer primjene je ukalupljivanje u programski jezik. To je napravljeno u jeziku Natalie⁹. Pažnja je ponovno stavljena na matematičke izraze. Autor djela je također objavio video materijale koji prate razvoj jezika i ideje iza odabira Prattove metode¹⁰.

Vjerojatno najpoznatija implementacija je ona Douglasa Crockforda u JSLintu¹¹. Svoj kod učinio je javnim te je moguće pogledati njegovu implementaciju¹² koja je podosta različita od prethodno navedenih premda postoje i drugi izvori koji koriste metodu sličnu njegovoj¹³. Također je napisao i članak o svojoj implementaciji i argumentirao zašto je Prattova metoda dobra za parsiranje JavaScripta.

Pratt je u svom izvornom radu rekao da je njegova metoda već implementirana u SCRATCH-PAD-u i MACSYMA-u

⁷ <https://www.desmos.com/>

⁸ <https://engineering.desmos.com/articles/pratt-parser/>

⁹ <https://github.com/seven1m/natalie>

¹⁰ <https://natalie-lang.org/>

¹¹ <https://jshint.com/>

¹² <https://github.com/douglascrockford/JSLint>

¹³ <https://theantlrguy.atlassian.net/wiki/spaces/ANTLR3/pages/2687077/Operator+precedence+parser>

4. Prednosti i nedostaci

Pokazalo se trivijalno implementirati parser za matematičke izraze i za njihovo parsiranje se ova metoda smatra vrlo pogodnom. Za cijeli jezik to bi bilo dosta kompleksnije. Manje je intuitivno u nekom jeziku odrediti prednost operatora. Dobar primjer gdje se krenu gubiti svojstva prednosti operatora i metoda kreće više ličiti klasičnom rekurzivnom spustu je kada ispred ili iza određenog tokena mogu doći mnogo različitih kombinacija tokena i postaje teško odrediti prednost operatora. Tada je normalno krenuti ispitivati koji se token nalazio prije trenutnog tokena te se prestaje oslanjati na prednost operatora. Apache Thrift¹⁴ je jedna primjer takvog jezika u kojem token Identifier može imati preko 10 različitih prefiksni vrijednosti.

Drugi veliki nedostatak je što je metoda nepopularna i nema sličnosti s drugim metodama pa drugi sustavi se ne temelje niti uzimaju u obzir ovakvo ostvarenje parsera. Izrazit problem može biti zadavanje gramatike u obliku koji ne vodi računa o prednosti operatora. Kod matematičkih izraza vrlo je jasno koji operator ima prednost, dok u programskim jezicima to nažalost nije slučaj. Ako je neka gramatika zadana u nepovoljnom obliku može biti lakše parsirati taj jezik koristeći metodu koja je kompatibilna s takvim načinom zadavanja gramatike. Zaključak je da se o parsiranju treba razmišljati za vrijeme definiranja jezika.

Iz prethodnog ima smisla parser temeljiti na ovoj metodi u slučaju gradnje jezika na brzinu gdje dokumentacija (gramatika) nije previše bitna.

Ono gdje bi mogla biti veća primjena je u razvojnoj okolini gdje bi trebala se pamtit i struktura AST radi statičke analize. Takav aparat bi mogao puno bolje obavijesti o pogreškama davati od klasičnih parsera.

Ono što ovaj parser kao i svaki drugi treba je testiranje koje iziskuje puno resursa.

Kada je jezik zadan u BNF-u ili nekoj sličnoj strukturi vjerojatno ga nema smisla parsirati Prattovom tehnikom jer bi takva pravila trebalo prepisati u sami kod Prattovog algoritma.

¹⁴ <https://github.com/apache/thrift/blob/master/doc/specs/idl.md>

Tada bi se izgubilo vremena na toj migraciji. Možemo čak biti ekstremni i reći da je Prattov algoritam dovoljan za shvatiti sintaksu jezika.

Ono što ova metoda nudi, kao i svaki ručno pisani parser, je veća kontrola oko implementacijskih detalja s tim da ova metoda ima izrazito lak način upravljanja greškama.

Nažalost nije moguće ili je vrlo teško automatizirati izgradnju ovakve vrste parsera, ali uvijek postoji mogućnost korištenja boilerplate koda.

Zaključak

Prattova metoda izrazito je jednostavna za implementaciju unatoč pomalo konfuznoj ideji. Rekurzije su nekada vrlo zamršene i nerazumljive te su često zbog toga preskakane, no Prattova metoda nudi baš obrnuto. Nakon prihvatanja Prattove terminologije ispada trivijalna i uklanjanje zamršenosti u parsiranju.

Uz moć dobre obrade grešaka i brzu ručnu implementaciju nameće se kao odlična metoda za ručno pisane parsere [4]. Nedostatak je što ju je vrlo teško automatizirati za cijeli jezik.

Prattovu metodu parsiranja možemo shvatiti kao deklarativni način pisanja parsera jer cijela kontrola upravljanja parsiranja je zadana posredno. Prilikom pisanja koda dobije se dojam kao da se piše gramatika, a ne kod. To je i bila Prattova ideja kada je predstavio metodu, time je želio spojiti one koji smatraju da gramatika nije bitna pri ostvarivanju parsera i onih koji smatraju da je neophodna.

On je zamislio metodu više kao pisanje gramatike, a manje kao pisanje programa odnosno uvukao je sintaksu u kod [2].

Prattova metoda danas je dosta neprihvaćena i nepoznata, tome svjedoče malobrojni članci na internetu. Nada je da se ovim djelom pridonese popularizaciji teme.

5. Daljnji rad

Bilo bi zanimljivo promotriti strukture koje se generiraju u statičkim analizatorima koda u razvojnim okolinama. Kako se piše izvorni kod tako se gradi sintaksno stablo i to nije posebno inovativno ili zanimljivo, ali je zanimljivo kada se krenu raditi izmjene u kodu jer tada se mora zamijeniti samo dio ASTa, a ne cijelo. Argument za to da se ne gradi ponovno cijelo stablo je da se to treba obaviti brzo. Pitanje je kako bi se Prattova metoda mogla uklopiti u tu ideju i je li to uopće moguće. Za razmotriti je također frekvenciju mijenjanja određenih dijelova koda programera, nema smisla optimirati nešto što se rijetko koristi.

Postoje neke sumnje u učinkovitost korištenja Prattove metode korištenjem objektno orijentiranog pristupa te bi imalo smisla razmotriti i druge pristupe poput npr. onog koje je Crockford koristio pri pisanju jsLint-a [10].

Literatura

[1]

Grune, D i Jacobs, C. J.H. Parsing Techniques: Introduction. New York: Springer Science+Business Media, LLC, 2008.

[2]

Pratt, Vaughan R. 1973. "Top down operator precedences". Proceedings of the 1st annual ACM

SIGACTSIGPLAN symposium on Principles of programming languages, 41-51

[3]

<https://matklad.github.io/2020/04/13/simple-but-powerful-pratt-parsing.html>

[4]

CROCKFORD, Douglas. 2007. Top Down Operator Precedence. Available at:

<https://crockford.com/javascript/tdop/tdop.html> (Accessed: 13 January 2021).

[5]

<https://engineering.desmos.com/articles/pratt-parser/>

[6]

<https://eli.thegreenplace.net/2012/08/02/parsing-expressions-by-precedence-climbing>

[7]

<https://wwwantlr.org/papers/Clarke-expr-parsing-1986.pdf>

K clarke, the top down parsing of expressions

[8]

<http://www.oilshell.org/blog/2016/11/01.html>

[9]

<http://journal.stuffwithstuff.com/2011/03/19/pratt-parsers-expression-parsing-made-easy/>

[10]

<https://www.robertjacobson.dev/designing-a-pratt-parser-generator#fnref:5>

Sažetak

Parsiranje od vrha prema dnu ili kraće Prattova metoda je rijetko poznata metoda parsiranja koja ugrađuje sintaksu jezika direktno u sam programski kod parsera. Uz to svojstvo još nudi i veliku modularnost i lako upravljanje greškama. U ovom radu prezentiran je rad Prattove metode, argumentirano kada ga ima smisla koristiti te su glavni koncepti potkrijepljeni primjerima na generičkim matematičkim izrazima.

Summary

Top down operator precedence parsing or also known as Pratt method is rarely known method of parsing in which syntax of a language is directly embedded into source code. This technic also offers great modularity and easy way of error handling. This work presents Pratt method along with arguments when to use it (what are use cases). Main concepts are presented on examples of parsing on generic mathematic expressions

Rjesi pasive voice u zadnjoj recenici

Skraćenice

LBP	<i>Left Binding Power</i>	lijeva vrijednost vezanja
RBP	<i>Right Binding Power</i>	desna vrijednost vezanja
nud	<i>null denotation</i>	null denotacija (prefix)
led	<i>left denotation</i>	lijeva denotacija (infix)
AST	<i>Abstract Syntax Tree</i>	apstraktno sintaksno stablo
BNF	<i>Backus-Naur Form</i>	Backus-Naurov oblik

Programska podrška

Za pokretanje primjera programa potreban je python 3 interpreter. Nije potrebna niti jedna nestandardna biblioteka.

Cijeli programski kod se nalazi na linku¹⁵.

Za potrebe ostvarenja lexera pri testiranju programa kao ulazi su predani stringovi tokena koji su bili odvojeni razmakom. Kao alat može poslužiti i generator lexera koji se također nalazi u git repozitoriju¹⁶.

Za program su napisani unit testovi. Ako je želja proširiti postojeći kod novim tokenima preporuča se dopisivanje već postojećih testova.

¹⁵ https://github.com/marin-jovanovic/top-down-operator-precedence/blob/master/pratt_toy/main.py

¹⁶ https://github.com/marin-jovanovic/top-down-operator-precedence/tree/master/lexer_generator