SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 233

Parsiranje domenskih jezika koristeći Prattov parser

Marin Jovanović

Zagreb, lipanj 2021.

Sadržaj

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 implementira 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 operatora[1].

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 nalagala 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].

Prikazimo sada naivan pristup pri rjesavanju takvog problema.

Recimo da postoji produkcija

```
S -> Sa | a
```

Očito je da će uz iduća funkcija zaglaviti u rekurziji za neke ulaze:

```
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:

```
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 članka , u tom članku je prikazano inkrementalna izrada parsera te je to dobar primjer grade osnovnog parsera.

Time je zadovoljio dobru praksu, štoviše, prisilio je da se cijeli parser fokusira na ovu 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 1.2 je sto se poopćava mehanizam vezan za lijevu rekurziju te vise nije potrebno pisati isti dio koda svaki put kada se susretnemo s njom.

Ideja je da ovo glavna funkcija koja pokušava 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.

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

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 članku[2] iz kojeg smo preuzeli kostur koda i nadogradili ga kako bi pokazali pravu moć Prattove metode.

Razlog parsiranja samo matematickog dijela jezika je sto je intuitivna prednost operatora. Takoder je za primjetiti da veliki broj jezika koristi slicnu gramatiku za matematicke izraze i to je komponenta parsiranja koju je moguce odvojiti u posebnu rutinu parsera i onda parsirati posebno cijeli jezik, a kada se dode do matematike parsirati samo tu komponentu kao zaseban dio jezika.

Pod prednosti operatora u nekom izrazu (kodu) smatrat cemo razrjesava (spaja) li se neki token s tokenom ili tokenima desno od njega ili s tokenom ili tokenima lijevo od njega. Trijvijalan i ocigledan primjer (ponovo samo na matematickom izrazu) je da cemo izraz 1 - 2 * 3 razrjesiti kao 1 - (2 * 3), a ne slijedno od lijeva na desno. Manje intuitivan primjer bi bio int x = 1;, ovdje bi puno teze bilo odrediti sto cemo smatrati lijevim, a sto desnim spajanjem. O ovome ce biti govora u poglavlju ???? prednosti i nedostatci

U svrhu ove demonstracije gradit cemo apstraktno sintaksno stablo umjesto direktnog uvrstavanja i racunanja izraza kao sto je radeno u navedenom clanku.

Kako bi mogli razlikovati tokene i ukalupiti logiku koja bi rekla na koji se nacin spaja odredeni token posluzit cemo se klasama. Prvi korak je jedna klasa Token koju ce svaka nova klasa tokena nasljedivati

```
class Token(object):
        def __init__(self, value):
        id = self.__class__.__name__
        self.identifier = id
        self.value = value
        self.lbp = LBP.get(id)
        self.rbp = RBP.get(id)

        def nud(self):
                pass

        def led(self, left):
```

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

```
pass
```

Kôd 1.4 – klasa Token

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

LBP i RBP predstavljaju dva rjecnika koji ce kao kljuceve imati id tokena, a kao vrijednost samu vrijednost pridruzivanja. Radi preglednosti dati cemu ta dva rjecnika kao tablicu (Tablica 1.1). Tokeni koji ne koriste svoju lbp i rbp vrijednost nisu prikazani u tablici, za njih mozemo pretpostaviti da imaju vrijednost manju ili jednaku nula.

Tablica 1.1 LBP i RBP rjecnici

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

Takoder ce svaki token morati implementirati svoju funkciju nud i led koja ce nadjacati postojece. Implementirati ce te funkcije jedine ako ih koristi, za neke tokene jednostavno nema smisla nadjacavati jednu od njih, primjere cemo dati kasnije.

Za navedeni primjer 1 - 2 * 3 potrebno je implementirati tri klase tokena; literale, tokene za zbranje i mnozenje. Valja primjetiti da token „–„ ima i nud i led metodu jer se minus moze naci prije i nakon drugih tokena i da tvori sintaksni i semanticki 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

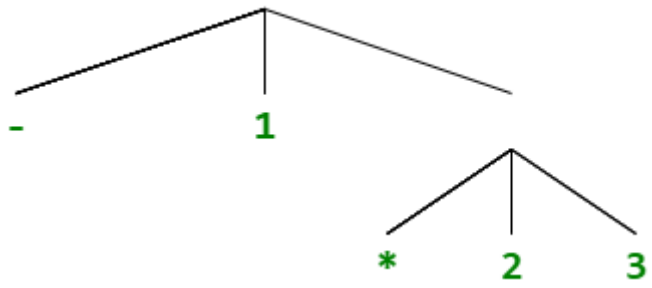
Potrebno je pozvati funkciju expression koja ce imati zadanu vrijednost rbp jednaku 0, prvo sto ce se dogoditi je da ce se za token 1 nad klasom TokenLiteral zvati funkcija nud koja ce vratiti vrijednost 1. Nakon toga funkcija expression ulazi u while petlju u kojoj se zove za token – nad klasom TokenSubtraction funkcija led koja vraca [„-„, 1, expression(8)], za left se uzima vrijednost svega predano lijevo a to je samo broj 1, a za

desni argument (zadnji element liste) vraca se expression(8) od kojeg ocekujemo da vrati parsirani oblik (2 * 3).

Program ulazi u funkciju expression(8) i t.nud() vraca 2 te nakon toga ulazi u while, unutar while-a zove se t.led(left) koji vraca [„*“, 2, expression(10)].

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

Sada funkcija expression(8) moze vratiti [„*“, 2, 3] te se to ubacuje u pocetni expression() koji vraca [„-“, 1, [„*“, 2, 3]]. Takva struktura predstavlja AST koji mozemo graficki prikazati (Sl 1.1).



Sl 1.1 AST za izraz 1 + 2 * 3

Iduci korak bilo bi implementacija upravitelja pogreskama. Sam kod nam djelom i namece da razmisljamo o rubnim slucajevima odnosno o ne-standardnom nacinu parsiranja (odstupanju od zadanog). Primjer toga bi mogle biti zagrade, ali za to bi morali prvo uvesti dvije nove klase

```
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]
```

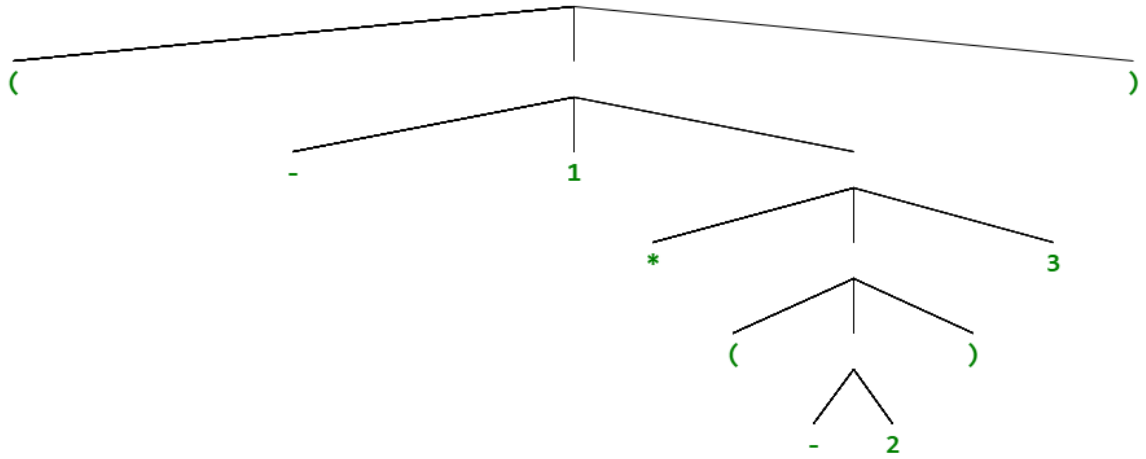
Kôd 1.8 – klasa TokenLeftBracket

```
class TokenRightBracket(Token):
    pass
```

Kôd 1.9 – klasa TokenRightBracket

Zamislimo sada da parsiramo izraz (1 - (- 2) * 3). Kako bi to omogucili trebali smo implementirati klasu lijeve i desne zagrade. Algoritam sada prvo uzima lijevu zagradu te nad njom zove funkciju nud koja prvo zove funkciju expression(self.rbp). Ideja je da se sada isprarsira sve unutar te zagrade i vrati se dio AST oblika [„(“, expressiont(self.rbp), „)“]. Valja primjetiti ugnjezdene zagrade koje ce nas parser s lakocom pravilno isparsirati. Ideja je da je vrijednost self.rbp veca od vrijednosti TokenRightBracket.lbp. Time osiguravamo da ce se parsirati ulaz sve do prve desne zagrade. Mogli bi si sada postaviti pitanje sto je s ugnjezdenom zagradom i odgovor je da je stvar ista. Ponovno ce se krenuti parsirati izraz unutar zagrada sve dok ne naleti na prvu desnu zagradu, u ovom slucaju vratit ce se ['(', ['-', '2'], ')']. Taj ce se izraz vratiti expression funkciji koju je pozvala prva lijeva zagrada te ce ona vratiti ['-', '1', ['*', ['(', ['-', '2'], ')'], '3']]. Nakon toga sve skupa vraca AST oblika ['(', ['-', '1', ['*', ['(', ['-', '2'], ')'], '3']], ')'].

Slika [[(] [[-] [1] [[*] [[(] [[-] [2]] [)]] [3]]] [)]]



Sl 1.2 AST za (1 - (- 2) * 3)

Zanimljivost ovog nacina rjesavanja zagrada je sto mozemo lako razlicite vrste zagrada tretirati isto, potrebno je samo pri stvaranju objekta definirati vrijednost self.value.

U ovom primjeru se za token – pozivala funkcija nud, a ne led. Razlog ovoga je sto je to bio prefiks broju 2, nije se oduzimao 2 od nekog drugog broja. Mozemo vidjeti kako se u ovom slucaju 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 nasoj tablici, a to onda osigurava da se nece izvrsiti while petlja u toj funkciji i vratiti ce samo nud koji odgovara broju 2. Ovaj primjer je moguce poopciti; kada je potrebno da neki token vrati token koji je njemu s desna to je ostvarivo tako da se pozove funkcija expression s argumentom vecim od svih vrijednosti lbp. Naravno, ako taj token u svojoj nud funkciji zove ponovo funkciju expression onda ce se i to morati razrjesavati, ali uvijek ce se vratiti samo vrijednost tog nuda, nece se izvrsavati while dio tog expressiona.

Sada bi se moglo postaviti pitanje zasto se nije to onda odradilo za desnu zagradu i odgovor se nalazi u upravljanju greskama, ali i optimizaciji. Puno je lakse provjeriti odma je li iduci token desna zagrada i odmah upravljati daljnjim tijekom ovisno o odgovoru nego da se poziva funkcija expression koja ce vratiti samo jedan token i onda bi opet morali provjeriti je li vracena desna zagrada.

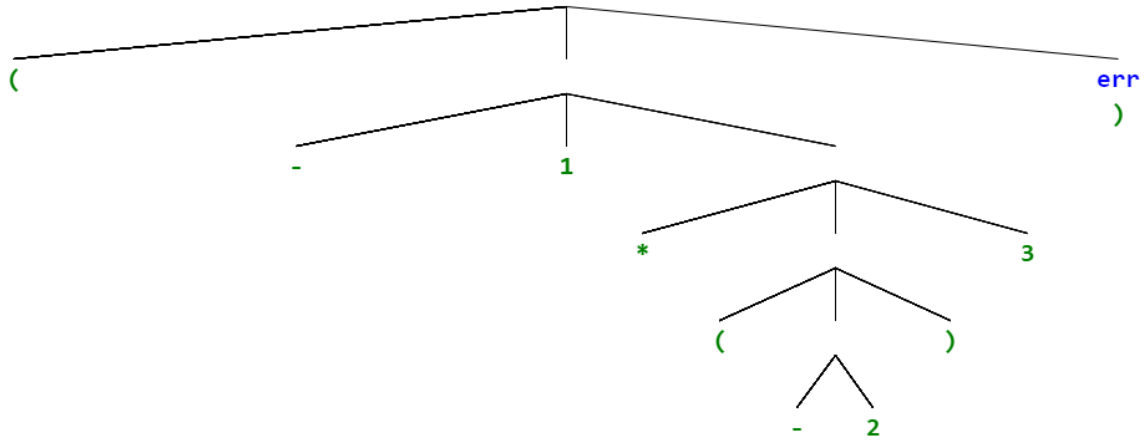
1.4. Upravljanje greskama

U ovom primjeru gdje ocekujemo desnu zagradu mogli bismo ispisati pogresku o gresci i probati nastaviti s parsiranjem. Ako pretpostavimo da pokusavamo parsirati neki izvorni kod i ustanovimo da bi negdje trebala postojati zagrada, a nje nema (kao u proslom primjeru proslog poglavlja) mogli bismo ispisati poruku i svejedno nastaviti program kao da je iduci token bila odgovarajuca zagrada. Tada bi nastavili s parsiranjem i mogli ustanoviti postoji li jos koja pogreska u semantici koda. Za to bi trebali prosirit kod klase tokena za lijevu zagradu

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



Sl 1.3 AST za (1 – (-2 * 3)

Moglo bi se sada argumentirati kako izraz 1 + (2 – (3 + 4) ustanoviti je li 1 + (2 – (3) + 4) ili 1 + (2 – (3 + 4)) i odgovor na to ovom metodom ne mozemo dati, ali je ovim pristupom osigurano parsiranje ovog dijela izvornog koda bez ispadanja programa i to nam je dovoljno. Programer ce dobiti obavijest o gresci i sam ce 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 slicni pa bi se iz ovog primjera mogli svi drugi pozivi funkcija iskonstruirati stoga ovo smatramo vrijednim primjerom.

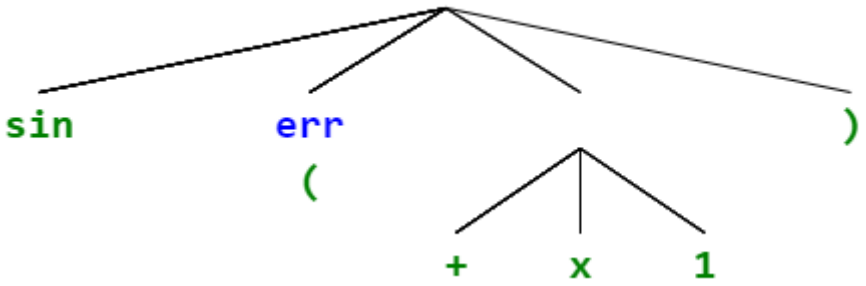
```
class TokenTrigonometry(Token):
    def nud(self):
        globa token
        if not isinstance(token, TokenLeftBracket):
            print(„error left bracket“)
            #TODO determinate which type of bracket is needed
            l_b = „err (“
        else:
            l_b = token.value
```

```
            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'], ')'] sto je u skladu s ocekivanjima, AST bi graficki izgledao ovako



Sl 1.4 ast za ovo gornje

Jos bi bilo zanimljivo pogledati kako bi se parsirao izraz if {expr} then {expr}else {expr}. Od ovoga nebi imali preveliku korist u kalkulatoru i nije nesto sto 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 vaznosti vrijednosti lbp i rbp. Za ovo bi nam potrebne bile opet nove klase.

```
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
```

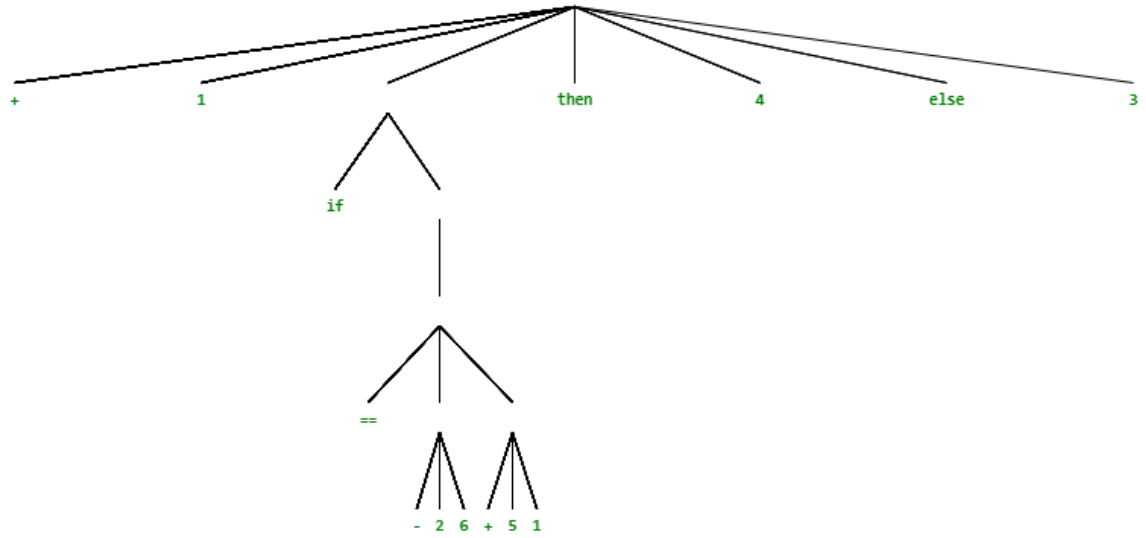
Kôd 1.13 – klasa TokenThen

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

Kôd 1.14 – klasa TokenElse

Primjetimo kako za token TokenElse ne radimo provjeru nego na osnovu njega upravljamo hocemo li razjrjesavati then dio ili cemo razaznati da on ne postoji.

Za nas primjer AST bi izgledao



Sl 1.5 AST za izraz 1 + if 2 – 6 == 5 + 1 then 4 else 3

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

U izvornom kodu rada je jos dodan i ternarni operator nesto jednostavnije sintakse, ali jednake znacenjske vrijednosti koji nema smisla ponavljati jer ovo pokriva to.

1.6. Ostali operatori

U kodu su dodani i drugi klasicni matematicki operatori, zainteresiranog citatelja se poziva da ih prouci. Trenutni podrzani tokeni su:

- literali (brojevi),
- zbrajanje,
- oduzimanje,
- mnozenje,
- djeljenj,

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

2. Prattova metoda naspram drugih

Ustanovivsi da postoji siroka potreba za parsiranjem napravljeni su mnogi alati i metode za generiranje parsera. Primjeri takvih alata su Yacc[3] i Bison[4]. Za genericne metode kojima se ostvaruju parseri usko je vezana teorija automata. Cesta ostvaranje su LR parseri.

Ovakva rjesenja su adekvantna kada nije bitna optimalnost parsera ili kada je vrijeme bitan faktor. Takoder je moguce ostvariti puno bolje upravlajnje greskama ondnoso ispis gresaka [2]. Za slucajeve kada to nije tako razvijatelj koda ima mogucnost primijeniti Prattovu metodu parsiranja i time ostvariti puno modularniji (personaliziraniji) kod.

Postoje metode slicne Prattovoj, neki argumentiraju da su cak neke metode istovjetne Prattovoj [8], dok neki usporeduju te slicnosti i govore da nisu isti. Smatra se da je Dijskrta generalizirao shunting-yard metodu parsiranja koristeci samo jedan stog ili uz pomoc rekurzivnog spusta. Nakon toga Pratt ulazi u pricu sa svojim algoritmom koji je po mnogocemu bolji od prethodne Dijskrtine metode [9].

Zanimljivo je primjetiti i slicnost s obradom prekida u operacijskim sustavima. Postoje programske implementacije koje cekaju prekidi i obraduju ga samo u slucaju ako je veci od prekida koji se trenutno obraduje. Algoritam je rekurzivan i implementacijom podsjeca na Prattovu tehniku. Pseudokod i njegovo objasnjenje dano je na uputama za laboratorijsku vjezbu predmeta Operacijski sustava na FERu[5].

Zanimljiva je i simbioza ovog algoritma s ostalim rjesenjima.

Pokazalo se iznimno jednostavno napraviti algoritam za parsiranje matematickih izraza koji je opisan u ovom radu, ali za ostatak jezika vec 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 sluzi za pridruzivanje vrijednosti. Za tu svrhu ima smisla

---

[3] http://dinosaur.compilertools.net/yacc/

[4] https://www.gnu.org/software/bison/

[5] http://www.zemris.fer.hr/predmeti/os/pripreme/z1b.html

koristiti Prattov pristup jer se tada izvlaci ono najbolje iz njega. Naravno, implementacija ce uvijek ostati na razvijatelju i njegovim afinitetima.

Teorija automata je cesto u sluzbi parsera. Razlicite gramatike sluze za opisivanje jezika dok razni generatori generiraju lexera i parsere. Vecinom su to LR parseri, njegove izvedenice i algoritmi bazirani na rekurzivnom spustu.

Postoje slicnosti izmedu Prattovog algoritma i Shunting Yard algoritma uz razliku sto drugi koristi stog umjesto rekurzije. Takoder je jasna slicnosti izmedu Prattove medote i algoritma „precedence climbing“ gdje je logika u principu ista, ali uz drugaciju formulaciju [6].

Takoder postoje i neke jasne razlike izmedu navedenih algoritma stoga ih ne mozemo smatrati identicnima [8].

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

3. Primjena

Posto smo pokazali da je najbolja primjena Prattova metoda na matematicke izraze ona je vecinom nasla primjenu u tom polju. Ocekivana primjena su aplikacije za rjesavanje matematickih zadataka. Aplikacija koja koristi takav pristup je Desmos[6]. U svom clanku[7] objasnili su zasto su presli na Prattovu metodu i kakva im je to poboljsanja donjelo. Glavni razlog im je bio upravljanje pogreskama koje generatori parsera nisu nudili. S novom implementacijom uspjeli su ostvariti manje memorijsko opterecenje i vecu brzinu parsiranja. S druge strane navode i neke negativne strane poput velikog broja rekurzivnih poziva i laku mogucnost uvodenja neefikasnosti [5].

Drugi primjer primjene je ukalupljivanje u programski jezik. To je napravljeno u jeziku Natalie[8]. Paznja je ponovno stavljena na matematicke izraze. Autor djela je takoder objavio videomaterijale koji prate razvoj jezika i ideje iza odabira Prattove metode[9].

Vjerojatno najpoznatija implementacija je ona Douglasa Crockforda u JSLintu[10]. Svoj kod ucinio je javnim te je moguce pogledati njegovu impelementaciju[11] koja je podosta razlicita od prethodno navedenih premda postoje i drugi izvori koji koriste metodu slicnu njegovoj[12]. Takoder je napisao i clanak o svojoj impelemntaciji i argumentirao zasto je Prattova metoda dobra za parsiranje JavaScripta.

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

---

[6] https://www.desmos.com/

[7] https://engineering.desmos.com/articles/pratt-parser/

[8] https://github.com/seven1m/natalie

[9] https://natalie-lang.org/

[10] https://jslint.com/

[11] https://github.com/douglascrockford/JSLint

[12] https://theantlrguy.atlassian.net/wiki/spaces/ANTLR3/pages/2687077/Operator+precedence+parser

4. Prednosti i nedostatci

Pokazalo se trivijalno implementirati parser za matematicke 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 krece vise liciti klasicnom rekurzivnom spustu je kada ispred ili iza odredenog tokena mogu doci mnogo razlictih kombinacija tokena i postaje tesko odrediti prednost operatora. Tada je normalno krenuti ispitivati koji se token nalazio prije trenutnog tokena te se prestaje oslanjati na prednost operatora. Apache Thrift[13] je jedna primjer takvog jezika u kojem token Identifier moze imati preko 10 razlicitih prefiksnih vrijednosti.

Drugi veliki nedostatak je sto je metoda nepopularna i nema slicnosti s drugim metodama pa drugi sustavi se ne temelje niti uzimaju u obzir ovakvo ostvarenje parsera. Izrazit problem moze biti zadavanje gramatike u obliku koji ne vodi racuna o prednosti operatora. Kod matematickih izraza vrlo je jasno koji operator ima prednost, dok u programskim jezicima to nazalost nije slucaj. Ako je nega gramatika zadana u nepovoljnom obliku moze biti lakse parsirati taj jezik koristeci metodu koja je kompatibilna s takvim nacinom zadavanja gramatike. Zakljucak je da se o parsiranju treba razmisljati za vrijeme definiranja jezik.

Iz prethodnog ima smisla parser temeljiti na ovoj metodi u slucaju gradnje jezika na brzinu gdje dokumentacija (gramatika) nije previse bitna. Postoje naznake o brzem izvodenju kada se parser temelji na ovoj metodi makar nije znacajna [].

Ono gdje bi mogla biti veca primjena je u razvojnoj okolini gdje bi trebala se pamtiti struktura AST radi staticke analize. Takav aparat bi mogao puno bolje obavijesti o pogreskama davati od klasicnih parsera.

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

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

[13] https://github.com/apache/thrift/blob/master/doc/specs/idl.md

Tada bi se izgubilo vremena na toj migraciji. Mozemo cak biti ekstremni i reci da je Prattov algoritam dovoljan za shvatiti sintaksu jezika.

Ono sto ova metoda nudi, kao i svaki rucno pisani parser, je veca kontrola oko implementacijskih detalja s tim da ova metoda ima izrazito lak nacin upravljenja greskama.

Zaključak

Prattova metoda izrazito je jednostvna za implementaciju unatoc pomalo konfuznoj ideji. Rekurzije su nekada vrlo zamrsene i nerazumljive te su cesto zbog toga preskakane, no Prattova metoda nudi bas obrnuto. Nakon prihvacanja Prattove terminologije ispada trivijalna i uklanja zamrsenost u parsiranju.

Uz moc dobre obrade gresaka (TODO nesto o ovome) i brzu rucnu impelementaciju namece se kao odlicna metoda za rucno pisane parsere [4]. Nedostatak je sto ju je vrlo tesko automatizirati za cijeli jezik.

Prattova metoda danas je dosta neprihvacena i nepoznata, tome svjedoce malobrojni clanci na internetu. Nada je da se ovim djelom pridonese popularizaciji teme.

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

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

5. Daljnji rad

Bilo bi zanimljivo promotriti strukture koje se generiraju u statickim analizatorima koda u razvojnim okolinama. Kako se pise izvorni kod tako se gradi sintaksno stablo i to nije posebno inovativno ili zaniljivo, 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 stable je da se to treba obaviti brzo. Pitanje je kako bi se Prattova metoda mogla uklopiti u tu ideju i je li to uopce moguce. Za razmotiriti je takoder frekvenciju mijenjanja odredenih dijelova koda programera, nema smisla optimirati nesto sto se rijetko koristi.

Postoje neke sumnje u ucinkovitost koristenja Prattove metode koristenjem objektno orijentiranog pristupa te bi imalo smisla razmotriti i druge pristupe poput npr. onog koje je Croockford 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.

poglavlja

uvod

3

[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://www.antlr.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 krace Prattova metoda je rijetko poznata metoda parsiranja koja ugraduje sintaksu jezika direktno u sam programski kod parsera. Uz to svojstvo jos nudi i veliku modularnost i lako upravljanje greskama. U ovom radu prezentiran je rad Prattove metode, argumentirano kada ga ima smisla koristiti te su glavni koncepti poktrijepljeni primjerima na generickim matematickim 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 recenicic

Skraćenice

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

Programska podrska

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

Cijeli programski kod se nalazi na linku[14].

Za potrebe ostvarenja lexera pri testiranju programa kao ulazi su predani stringovi tokena koji su bili odvojeni razmakom. Kao alat moze posluziti i generator lexera koji se takoder nalazi u git repozitoriju[15].

Za program su napisani unit testovi. Ako je zelja prosiriti postojeci kod novim tokenima preporuca se dopisivanje vec postojecih testova.

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

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