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 i gradnje strukture koja ju 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 sto se i u praksi pokazalo da i jest tako. Općenito ako je neki algoritam primjenjiv na široki spektar stvari možda bi bilo mudro implementirati programsko rješenje koje ce riješiti svaki od tih problema iz spektra ili napraviti generator koji će generirati rješenje za svaki takav problem ili pak osmisliti postupak koji ce biti dovoljno jednostavan da se implementira za svaku primjenu.

Mnogi su pokusali napraviti opća programska rješenja za parsiranje ili generatore parsera te su ta rješenja prihvaćena. Trećom opcijom (putom?) 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 sto su na primjer lijeve rekurzije i upravljanje greškama (oporavak od pogresaka).

U sklopu ovog rada fokus će biti na njegovoj metodi, usporedbi sa sličnim metodama, položaju njegove metode u svijetu parsera te praktični primjeri.

[1] https://dl.acm.org/doi/10.1145/512927.512931

1. Prattova tehnika parsiranja

1.1. Vrste parsiranja

Opcenito o parsiranju

Sve metode parsiranja se mogu svrstati u dvije kategorije: parsiranje od vrha prema dnu i parsiranje od dna prema vrh. Prva metoda pokusava za zadani ulaz generirati strukturu (najcesce stablo, specificnije apstraktno sintaksno stablo) tako sto pokusava spojiti prvi token ulaza s nekom produkcijom gramatike. Na temelju prvotne produkcije pokusava se slijediti nadolazece produkcije i spojiti ih s odgovarajucim tokenima i tako u konacnici izgraditi zeljenu strukturu. Drugi pristup je da se obrne prethodno opisani proces u nadi da ce taj nacin 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 klasicnog pristupa do Prattove metode

Metode se kompliciraju kada je za odredeni token moguce primijeniti vise produkcija odnosno kada je moguce krenuti parsirati u razlicitim smjerovima u nadi da ce jedan od tih smjerova biti valjan za zadani ulaz [1].

Mogli bi formalno reci da je to problem pretrazivanja stanja. Tada bi algoritme mogli ocjenjivati po tome koliko brzo pretrazuju ta stanja te kakva im je kvaliteta intuicije odnosno heuristike. Heuristika se moze zadati vec u samoj specifikaciji gramatike, ali je isto tako moguce natuknuti programu kojim putem ici pri samoj implementaciji. Za neke jednostavnije implementacije (jezike) nije tesko odrediti kako bi se parser trebao ponasati, i moguce je parsiranje odraditi na ad-hoc nacin, ali kako jezik kojim se bavimo postaje slozeniji tako i sam parser postaje kompliciraniji. Dobra praksa bi nalagala da kod bude dobro strukturiran i da slijedi odredena pravila odnosno da nema odstupanja od nekakve globalne logike.

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

To je trenutak kada dolazi do ad-hoc (ovo izmjeni) rješenja. Prattova metoda na vrlo suptilan nacin rjesava ovaj problem. Jednostavno je poopciti iduci primjer.

Recimo da postoji produkcija

S -> Sa | a

Očito je da će uz idući algoritam program zaglaviti u rekurziji:

```
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

Rjesenje kojim bi mogli ovaj problem rjesiti je iduce:

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

Kôd 1.2 – Ispravna funkcija S

Prethodno rjesenje je valjano. Pratt je to rjesenje poopćio i time dobio funkciju expression [3]. Kostur algoritma bazira se na temelju ovog članka[2], u tom članku je prikazano inkrementalna izrada parsera te je to dobar primjer grade osnova parsera.

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

Time je zadovoljio dobru praksu, štoviše, prisilio je da se cijeli parser fokusira na ovu funkciju. Uz pomocne klase koje cemo uvesti prikazati cemo 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

Razilika ovog pristupa od prethodnog je …. (TODO dodaj link na clanak)

Ideja je da ovo glavna funkcija koja pokusava spojiti trenutni (pocetni) token s tokenima s desne strane uz pomoc funkcije nud ako to ima smisla te nakon toga pokusava spojiti tokene koje je parser vec prosao s trenutnim tokenom. Ta funkcionalnost se ostvaruje uz pomoc funkcije led te se kontrolira koliko ce se ona puta zvati unutar petlje pomocu prednosti operatora. Primjeri ce biti dani u iducem pogalavlju

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

1.3. Obiljezja Prattove metode

Ono sto je jos potrebno definirati su klase tokena. Razmatrat cemo jednostavni jezik koji ce se sastojati samo od matematickih operacija, varijabli i konstanti. To je primjer kkoji se vecinom koristi kako bi se prikazala upotreba (korisnost, jacina) metodeu Prattove metode pa tako i u clanku(link) iz kojeg smo preuzeli kostur koda i nadogradili ga kako bi pokazali pravu moc 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 i onda kad se dode do matematike parsirati samo tu matematiku

Pod prednosti operatora u nekom izrazu (kodu) smatrat cemo razrjesava (spaja) li se neki token s tokenom(tokenima) desno od njega ili s tokenom(tokenima) lijevo od njega. Trijvijalan i ocigledan primjer (ponovo samo na matematici) je da cemo izaz 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 (djelu).

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[5:])
    self.rbp = RBP.get(id[5:])

    def nud(self):
        pass

    def led(self, left):
        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 dictionary-a koji ce kao kljuceve imati id tokena, a kao vrijednost samu vrijednost pridruzivanja. Radi preglednosti dati cemu ta dva dictionarya

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 (provjeri ovo).

Tablica 1.1 LBP i RBP dictionary

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.

```
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)]
```

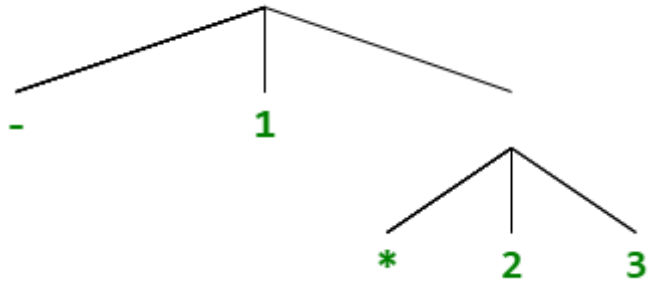
Kôd 1.7 – klasa TokenMultiplication

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]
```

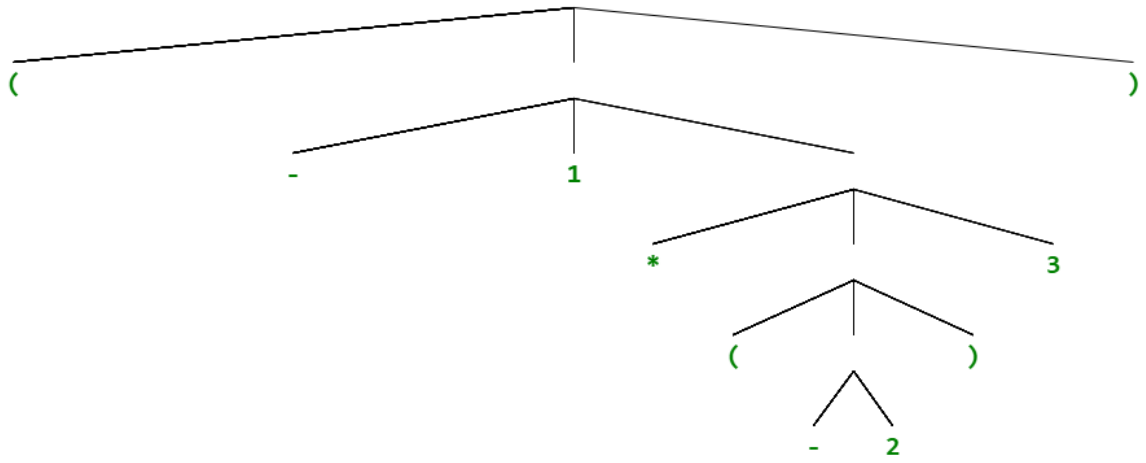
Kôd 1.8 – klasa TokenLeftBracket

```
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 ovaj izraz, todo poziv

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 odma 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.

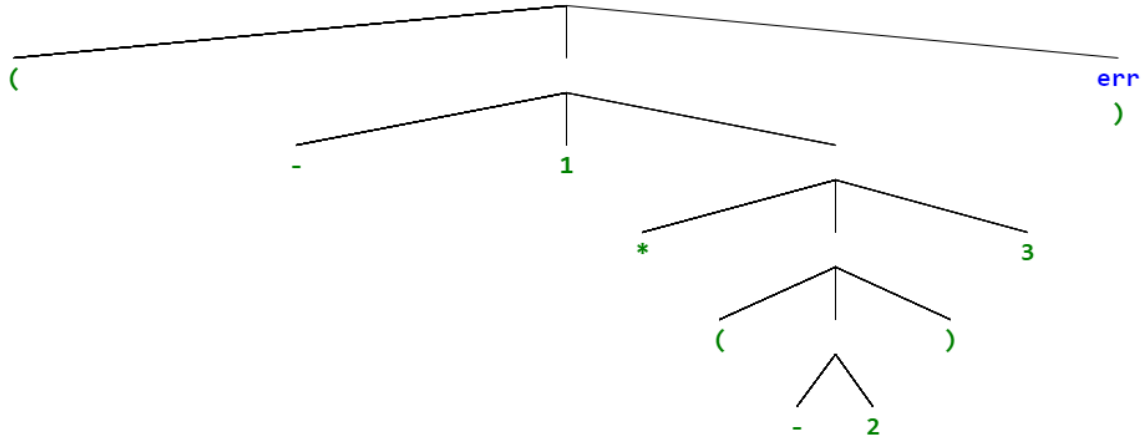
1.4. Upravljanje greskama

U ovom primjeru gdje ocekujemo desnu zagradu mogli bismo ispisati pogresku o gresci i probati nastaviti s parsiranjem. Ako pretpostavimo da je pokusavamo parsirati neki izvorni kod i ustanovimo da bi negdje trebala postojati zagrada, a nje nema (kao u ovom primjeru) 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 s error, todo referenciranje

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 nemam. TODO

1.5. Poziv funkcija

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

TODO vidi jel objasnjeno sta je lexer__next__

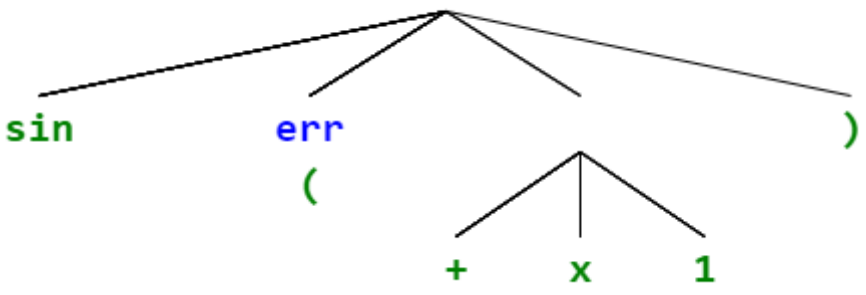
```
class TokenTrigonometry(Token):
    def nud(self):
        globa token
        if not isinstance(token, TokenLeftBracket):
            print(„error left bracket“)
```

```
                #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
```

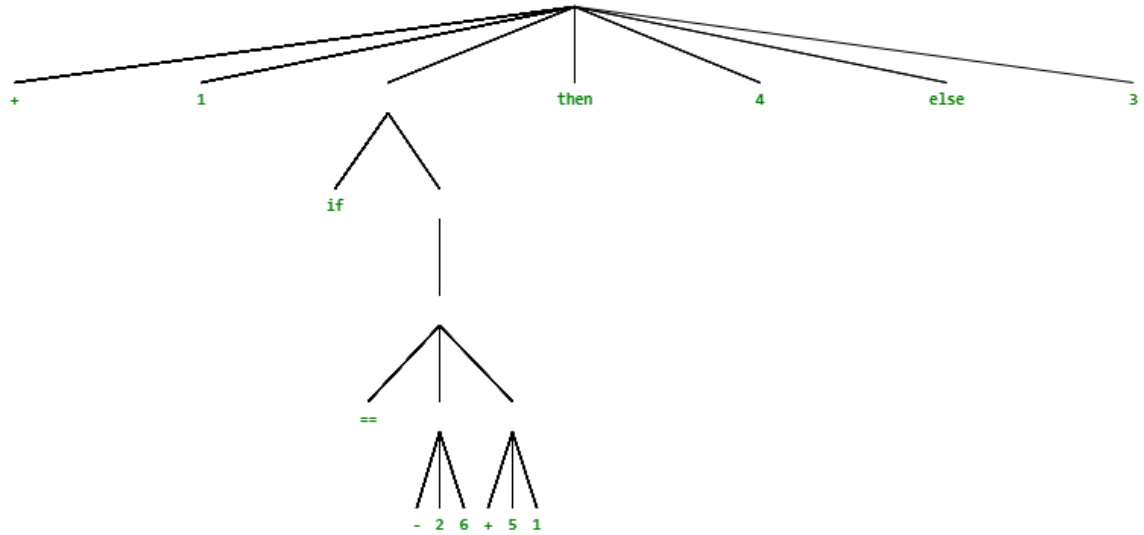
Kôd 1.13 – klasa TokenThen

```
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 ovo

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

Zasto pratt metodu korsitit

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.

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

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

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.

Neke spoznaje za vrijeme implementacije

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].

Svaki token prolazi kroz funkciju expression te to olaksava debagiranje

2. 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[5]. U svom clanku[6] 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[7]. Paznja je ponovno stavljena na matematicke izraze. Autor djela je takoder objavio videomaterijale koji prate razvoj jezika i ideje iza odabira Prattove metode[8].

Vjerojatno najpoznatija implementacija je ona Douglasa Crockforda u JSLintu[9]. Svoj kod ucinio je javnim te je moguce pogledati njegovu impelementaciju[10] koja je podosta razlicita od prethodno navedenih premda postoje i drugi izvori koji koriste metodu slicnu njegovoj[11]. 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

---

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

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

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

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

[9] https://jslint.com/

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

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

3. Prednosti i nedostatci

3.1. Prednosti

Pokazalo se trivijalno implementirati parser za matematicke izraze dok bi za cijeli jezik to 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[12] 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.

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

3.2. Nedostatci

4. Alternative

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].

Takoder bi se mogla povuci paralela izmedu

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.

Literatura

Popis literature dolazi na kraju rada, iza zaključka, a prije ostalih priloga.

Na naslov **Literatura** primijenite stil Heading 1, a zatim ručno maknite brojčanu oznaku (to je važno kako bi i naslov „Literatura“ ušao u sadržaj na početku rada, prije uvoda).

Pri kreiranju navoda u popisu literature koristite stil *literatura*.

Primjeri u nastavku ilustriraju navođenje raznih izvora u popisu literature: (1) knjige, (2) članka u časopisu, (3) članka u zborniku konferencije, (4) doktorskog, magistarskog ili diplomskog rada, (5) web-stranice.

[1] Tanenbaum, A. S., Wetherall, D. J. *Computer Networks*. 5. izdanje. London: Pearson, 2013.

[2] Brady, P.T. *A Statistical Analysis of On-off Patterns in 16 Conversations*, Bell System Technical Journal, 47,1 (1998), str. 55-62.

[3] Brady, N. *A Statistical Analysis of Use Case*. Proceedings of the 7th International Conference on Telecommunications ConTEL, Zagreb, (2003), str. 45-52.

[4] Ivić, M. *Analiza ponašanja korisnika u digitalnim igrama namijenjenima učenju*. Diplomski rad. Sveučilište u Zagrebu Fakultet elektrotehnike i računarstva, 2016.

[5] Epstein M., The *best VR headset in 2019*, PC Gamer, (2019, listopad). Poveznica: https://www.pcgamer.com/best-vr-headset/; pristupljeno 4. listopada 2019.

Uz svaki preuzeti sadržaj u svom radu – bilo da je riječ o tekstu (izravno citiranome ili „prepričanome“), slici ili grafičkom prikazu – treba navesti oznaku izvora (članak, knjiga, web-stranica ...) u popisu literature te se na nju „pozvati“, na primjer:

> Međusobno povezivanje mreža zasniva se na primjeni komunikacijskih protokola (Tanenbaum i Wetheral, 2014).
>
> Podaci o karakteristikama uređaja za virtualnu stvarnost preuzeti su s portala PC Gamer [5].
>
> Početna verzija programa preuzeta je iz diplomskog rada [5].

U danim primjerima mogli ste uočiti dva načina referenciranja:

- (Tanenbaum i Wetheral, 2014),
- [1].

Kad izaberete jedan od njih svakako ga se držite konzistentno u cijelome radu.

Sažetak

Summary

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

Privitak

Instalacija programske podrške

Za pokretanje primjera programa potreban je python 3 interpreter. Nije potrebna niti jedna nestandarna biblioteka. Cijeli programski kod se nalazi na stranici (link).

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 (TODO link).

Upute za korištenje programske podrške

Za program su napisani testovi (TODO link).

Ako definirani ispis nije pozeljan moguce je rucno promijeniti vrijednosti koje se vracaju u funkcijama led i nud tako da ili direktno racunaju vrijednosti pa ce se dobiti kalkulator ili se moze postaviti da se vraca klasa tokena te tako dobiti vise informiranije AST nego sto je trenutno