vepar

Distribucija 2.4

Veky

ožu 12, 2022

Sadržaj:

1 Leksička analiza **1**
- 1.1 Tipovi tokena . . . 1
- 1.2 Lexer . . . 2
- 1.3 Metode lexera . . . 3
- 1.4 Primjer . . . 4

2 Sintaksna analiza **5**
- 2.1 Beskontekstne gramatike . . . 5
- 2.2 Razine prioriteta . . . 6
- 2.3 Parser . . . 6
- 2.4 Primjer . . . 7
- 2.5 Iterativno parsiranje . . . 8
- 2.6 Asociranost . . . 9

3 Semantička analiza **11**
- 3.1 Evaluacija . . . 12
- 3.2 Memorija . . . 12
- 3.3 Optimizacija . . . 13

4 vepar module **15**

POGLAVLJE 1

Leksička analiza

Program (izvorni kod) nam je najčešće zadan kao string, dakle niz znakova Unikoda. Prvo što trebamo učiniti s njime je prepoznati osnovne podnizove koji predstavljaju podlogu za dalju obradu. Ti podnizovi tvore *tokene*. Iako može biti beskonačno mnogo tokena (npr. u mnogim programskim jezicima, svako legalno ime varijable je mogući token), ključno je da se svi oni razvrstavaju u konačno mnogo *tipova*.

Tipovi tokena su ono što zapravo upravlja kasnijom fazom sintaksne analize programa. To znači da, idealno, u sintaksnoj analizi ne bi smjelo biti bitno koji je točno niz znakova na određenom mjestu u programu (to zovemo *sadržaj* tokena), već samo njegov tip. Odnosno, ako je program sintaksno ispravan, ostat će takav i ako bilo koji token u njemu zamijenimo tokenom drugog sadržaja a istog tipa (recimo, broj `3` zamijenimo brojem `58`, ili ime varijable `xy` zamijenimo imenom `t`).

Pored tipa i sadržaja, tokeni prepoznati u izvornom kodu imaju brojne metapodatke koji kazuju gdje token počinje (u kojem retku odnosno stupcu izvornog koda), gdje završava, je li uspješno obrađen od strane sintaksnog analizatora itd. No svaki token mora imati tip i sadržaj, koji su na tokenu `t` dostupni kao `t.tip` i `t.sadržaj`.

1.1 Tipovi tokena

Da bismo započeli leksičku analizu, moramo definirati enumeraciju — klasu koja nabraja sve moguće tipove tokena. U vepru postoji bazna klasa `TipoviTokena` nasljeđivanjem od koje dobijemo svu potrebnu funkcionalnost. Evo primjera:

```
from vepar import *

class T(TipoviTokena):
    PLUS = '+'
    MINUS, PUTA, KROZ = '-*/'
    STRELICA, RAZLIČITO = '->', '!='
    class BROJ(Token):
        def vrijednost(t): return int(t.sadržaj)
    class I(Token):
        literal = 'i'
```

(continues on next page)

(nastavak sa prethodne stranice)

```
        def vrijednost(t): return 1j
    class IME(Token): pass
```

Upravo navedeni primjer pokazuje nekoliko mogućnosti za definiranje tokena, od jednostavnijih prema složenijima.

- obične inertne tokene koji uvijek (ili gotovo uvijek) imaju isti sadržaj (*literale*) definiramo navođenjem tipa lijevo od = i sadržaja desno. Dakle, token tipa `T.PLUS` ima podrazumijevani sadržaj `'+'`.
- više literala možemo definirati koristeći Pythonovo ugrađeno raspakiravanje stringova (za jednoznakovne tokene) i slogova (za višeznakovne). Dakle, token tipa `T.PUTA` ima podrazumijevani sadržaj `'*'`, a token tipa `T.RAZLIČITO` ima podrazumijevani sadržaj `'!='`.
- tokene koji nisu inertni (imaju metode koje sudjeluju u semantičkoj analizi) definiramo kao unutarnje klase koje nasljeđuju klasu `Token`. Tako `T.BROJ` nema podrazumijevani sadržaj, ali jednom kad ga prepoznamo i podvrgnemo semantičkoj analizi, moći ćemo iz njegovog sadržaja dobiti njegovu *vrijednost* kao Pythonov `int`.
- ako želimo neinertni literal (ima metode za semantičku analizu, a također ima konstantni sadržaj), možemo takvoj unutarnjoj klasi dati atribut `literal`. Tako se za potrebe prepoznavanja `T.I` ponaša kao da je definiran s `I = 'i'`, a u semantičkoj analizi za bilo koji token `t` tipa `T.I` možemo pozvati `t.vrijednost()` i dobiti Pythonov kompleksni broj `1j` (imaginarnu jedinicu).
- u suprotnom smjeru, ako želimo inertni token bez podrazumijevanog sadržaja, možemo jednostavno definirati unutarnju klasu s `pass` (ili `...` ako želimo signalizirati da je situacija privremena i da ćemo kasnije možda dodati neku semantiku za taj tip tokena).

1.2 Lexer

Lexer je funkcija (*generator* u Pythonu) koja prolazi kroz ulazni string i redom daje (`yield`) tokene koje u njemu prepoznaje. Vepar zahtijeva da bude dekorirana s `@lexer`. Jedini argument joj se obično zove `lex`, i predstavlja objekt klase `Tokenizer` čije metode pomažu pri leksičkoj analizi. Ovdje navodimo samo najčešće načine korištenja `lex`a — za potpunije informacije pogledajte dokumentaciju (recimo pomoću `help(Tokenizer)`).

Lexer najčešće počinje linijom poput `for znak in lex:`, dakle sastoji se od petlje koja u svakom prolazu varijabli `znak` pridružuje sljedeći znak ulaza. Nakon toga obično slijedi grananje s obzirom na to koji odnosno kakav znak smo našli. To utvrđujemo najčešće običnom usporedbom poput `znak == '$'` ili pozivom metode klase `str` poput `znak.islower()` (što se može zapisati i kao `str.islower(znak)`). Često korištene metode ovdje su:

`str.isspace` Je li znak bjelina, poput razmaka, tabulatora, prelaska u novi red i sličnog.

`str.isalpha` Je li znak slovo, poput `A`, `č`, , ili . Ako želite samo ASCII-slova, provjerite `znak in string.ascii_letters`, ali najčešće za tim nema potrebe. Trudite se biti inkluzivni!

`str.isdecimal` Je li znak znamenka u dekadskom sustavu. Opet, za ASCII-znamenke možete pitati `znak in string.digits`.

Mnoge `lex`ove metode primaju argument imena `uvjet`. On može biti pojedini znak koji se traži, metoda oblika `str.is...` poput ovih upravo navedenih, bilo koja funkcija (možete je i sami napisati, bilo preko `def` ili `lambda`) koja prima znak i vraća `bool`; ili pak skup takvih, interpretiran kao disjunkcija. Recimo, `{str.islower, str.isdecimal, '_', '$'}` znači „ili malo slovo, ili dekadska znamenka, ili donja crta, ili znak dolara”.

1.3 Metode lexera

Raznim metodama možemo unutar jednog prolaza pročitati i više znakova. Ovdje su neke.

`lex.zvijezda(uvjet)`, skraćeno `lex * uvjet` čita nula ili više znakova (od trenutne pozicije) koji ispunjavaju uvjet

`lex.plus(uvjet)`, skraćeno `lex + uvjet` čita jedan ili više znakova (od trenutne pozicije) koji ispunjavaju uvjet; prijavljuje grešku ako takvih nema

`lex.pročitaj_do(uvjet, *, uključivo=True, više_redova=False)`, skraćeno `lex - uvjet` ili `lex <= uvjet`, ili pak `lex` čita znakove dok ne naiđe na prvi znak koji ispunjava uvjet, prijavljuje grešku ako takvog nema; `uključivo` kazuje hoće li pročitati i taj znak, a `više_redova` hoće li tražiti znak i u kasnijim recima a ne samo u trenutnom

`lex.prirodni_broj(znak, nula=True)` čita prirodni broj (niz znamenaka bez vodećih nulâ) s početkom `znak` (najčešće znamo da dolazi prirodni broj tek kad vidimo prvu njegovu znamenku, ali ne uvijek — tada možemo za prvi argument staviti `''`); `nula` kazuje dozvoljavamo li čitanje `0` kao prirodnog broja

Također, razne metode su nam na raspolaganju ako želimo pročitati samo jedan znak.

`lex.čitaj()`, skraćeno `next(lex)` čita i vraća sljedeći znak bez ikakve provjere; na kraju ulaza vraća `''`

`lex.vidi(uvjet)`, skraćeno `lex > uvjet` ispituje ispunjava li znak, koji bi sljedeći bio pročitan, uvjet

`lex.nužno(uvjet)`, skraćeno `lex >> uvjet` čita sljedeći znak ako ispunjava uvjet, inače prijavljuje grešku

`lex.slijedi(uvjet)`, skraćeno `lex >= uvjet` čita sljedeći znak ako i samo ako ispunjava uvjet; pokrata za `if lex > uvjet: lex >> uvjet`.

Kad zaključimo da smo pročitali dovoljno znakova (što smo pročitali od zadnjeg stvorenog tokena možemo vidjeti u `lex.sadržaj`), vrijeme je da od njih konstruiramo neki token. Na raspolaganju nam je nekoliko metoda.

`yield lex.token(T.TIP)` stvara i šalje dalje token tipa `T.TIP` i sadržaja `lex.sadržaj`

`yield lex.literal(T, *, case=True)` stvara i šalje dalje literal onog tipa koji ima odgovarajući (pročitani) sadržaj; recimo ako je `lex.sadržaj == '->'`, uz gore definirani `T`, to bi odaslalo `Token(T.STRELICA, '->')` (skraćeno `T.STRELICA'->'`); ako takvog nema prijavljuje grešku; `case` govori traži li sadržaj uzimajući u obzir razliku velikih i malih slova

`yield lex.literal_ili(T.DEFAULT)` kao `lex.literal`, osim što ako takvog literala nema, vraća token tipa `T.DEFAULT`

`lex.zanemari()` resetira `lex.sadržaj`; možemo zamisliti da konstruira neki token, i uništi ga umjesto da ga pošalje dalje; česta linija u lexeru je `if znak.isspace(): lex.zanemari()`, čime zanemarujemo bjeline u izvornom kodu (ali nam i dalje služe za razdvajanje tokena).

Ako želimo sami prijaviti grešku, to možemo učiniti pomoću `raise lex.greška(poruka)` (ne moramo navesti poruku ako vepar ima dovoljno podataka za konstrukciju dovoljno dobre poruke).

1.4 Primjer

Jednom kad smo napisali lexer i dekorirali ga s `@lexer`, možemo ga pozvati s nekim stringom da vidimo kako funkcionira i eventualno ispravimo greške. Evo jednog primjera s obzirom na gornji `T`:

```
@lexer
def moj(lex):
    for znak in lex:
        if znak == '-':
            if lex >= '>': yield lex.token(T.STRELICA)
            else: yield lex.token(T.MINUS)
        elif znak == '!':
            lex >> '='
            yield lex.token(T.RAZLIČITO)
        elif znak.isdecimal():
            lex.prirodni_broj(znak, nula=False)
            yield lex.token(T.BROJ)
        else: yield lex.literal(T)

>>> moj('-+->ii!=234/')
```

```
Tokenizacija: -+->ii!=234/
                Znak #1        : MINUS'-'
                Znak #2        : PLUS'+'
                Znakovi #3-#4  : STRELICA'->'
                Znak #5        : I'i'
                Znak #6        : I'i'
                Znakovi #7-#8  : RAZLIČITO'!='
                Znakovi #9-#11 : BROJ'234'
                Znak #12       : KROZ'/'
```

POGLAVLJE 2

Sintaksna analiza

Naš program (niz znakova) pretvorili smo u niz tokena — sada je vrijeme da taj niz obogatimo dodatnom strukturom. Nizovi su linearni, a nama trebaju grafovi u više dimenzija, koji predstavljaju kako se tokeni slažu u semantički smislene cjeline. U ogromnom broju slučajeva ti grafovi su zapravo stabla, i zovu se apstraktna sintaksna stabla (AST).

U nizu tokena `BROJ'5'` `PLUS'+'` `BROJ'12'` dobivenom leksičkom analizom `5+12`, tokeni nisu ravnopravni: u izvjesnom smislu ovaj `PLUS` je „iznad", predstavlja *vrstu* izraza s kojom radimo (zbroj), a `BROJ`evi su samo operandi, koji dolaze u igru tek nakon što znamo o kojoj vrsti izraza se radi. Čak njihov sintaksni identitet može ovisiti o onom što se očekuje na danom mjestu: recimo, u programskom jeziku C, nakon `if` i otvorene zagrade dolazi uvjet, a onda nakon zatvorene dolazi naredba. Svaki uvjet je ujedno i naredba, ali ne i obrnuto. Dakle, na najvišoj razini možemo reći „aha, radi se o `if`-naredbi, `IF OTV uvjet ZATV naredba`", a onda `uvjet` i `naredba` predstavljaju dijelove koji se na nižim razinama popunjavaju tokenima koji se nalaze između odgovarajućih mjesta. Recimo, `if(i==3)break;` bi se moglo prikazati kao

```
Ako:                    @[Znakovi #1-#13]
  uvjet = Jednakost:    @[Znakovi #4-#7]
    lijevo = IME'i'     @[Znak #4]
    desno = BROJ'3'     @[Znak #7]
  onda = BREAK'break'   @[Znakovi #9-#13]
```

Ta ideja razina odnosno slaganja tokena u stabla, ključna je za sintaksnu analizu. Uglavnom ćemo dalje raditi s aritmetičkim izrazima jer su nam bliski, ali sve se može primijeniti na sintaksnu analizu brojnih formalnih jezika.

2.1 Beskontekstne gramatike

Formalizam koji nam omogućuje koncizno zapisivanje sintaksne strukture našeg jezika zove se *beskontekstna gramatika* (skraćeno BKG, engleski *context-free grammar*). BKG se sastoji od jednog ili više pravila oblika `varijabla -> riječ`, gdje je `riječ` konačni niz varijabli i tipova tokena. Pravilo predstavlja jednu moguću realizaciju niza tokena koji pripada jeziku određenom tom varijablom (kažemo da varijabla *izvodi* niz tokena). Više pravila za istu varijablu često se piše u skraćenom obliku kao `varijabla -> riječ1 | riječ2`.

Recimo, ako hoćemo reprezentirati zbrojeve jednog do tri broja, mogli bismo napisati gramatiku s 3 pravila

```
# zbroj -> BROJ | BROJ PLUS BROJ | BROJ PLUS BROJ PLUS BROJ
```

— ali što ako hoćemo proizvoljno mnogo pribrojnika? U tom slučaju možemo koristiti rekurzivno pravilo

```
# zbroj -> BROJ | BROJ PLUS zbroj
```

samo, naravno, moramo paziti da uvijek imamo i pravilo koje završava rekurziju.

Što se samog vepra tiče, ne moramo pisati BKG (u Pythonu pravila pišemo u komentare), ali iskustvo pokazuje da je tako puno lakše napisati sintaksni analizator (parser).

2.2 Razine prioriteta

Pribrojnici u `zbroju` ne moraju biti `BROJ`evi. Recimo, za ulazni string `5+8*12` dobijemo niz tokena `BROJ'5' PLUS'+' BROJ'8' PUTA'*' BROJ'12'`, te znamo (na osnovi dogovora o prioritetu operacija) da predstavlja zbroj broja `5` i umnoška brojeva `8` i `12`. (Također predstavlja i broj `101`, ali o tome tek u sljedećoj fazi.) Dakle, cilj nam je proizvesti nešto poput `Zbroj(lijevo=BROJ'5', desno=Umnožak(lijevo=BROJ'8', desno=BROJ'12'))`, ili, kako to vepar zna „stabloliko" prikazati:

```
Zbroj:                @[Znakovi #1-#6]
  prvi = BROJ'5'      @[Znak #1]
  drugi = Umnožak:    @[Znakovi #3-#6]
    prvi = BROJ'8'    @[Znak #3]
    drugi = BROJ'12'  @[Znakovi #5-#6]
```

Da bismo to zapisali u našoj BKG, moramo uvesti još jednu varijablu. U tom kontekstu ona se obično zove „član" (*term*), jer nam se ne da pisati „pribrojnik" zbog duljine.

```
# izraz -> član | član PLUS izraz
# član -> BROJ | BROJ PUTA član
```

Vidimo da smo preimenovali i početnu varijablu iz `zbroj` u `izraz`. Vepru nije previše bitno: ako postoji varijabla imena `start`, od nje kreće sintaksna analiza, a ako ne, kreće od prve varijable.

Sada je jasno kako bismo dodali još razina prioriteta: možete za vježbu dodati potenciranje. No što je sa zagradama? One nam omogućavaju da u član umjesto `BROJ`a utrpamo cijeli izraz, samo ga moramo staviti u zagrade. Dakle

```
# izraz -> član | član PLUS izraz
# član -> faktor | faktor PUTA član
# faktor -> BROJ | OTV izraz ZATV
```

2.3 Parser

Ovo je već dovoljno kompliciran BKG da je zanimljivo vidjeti kako bi izgledao parser za nju. Parser pišemo kao potklasu veprove klase `Parser`, čije metode (uglavnom) odgovaraju varijablama gramatike. Svaka metoda prima instancu parsera koja se uobičajeno zove `p`, ali nitko vam ne brani da pišete tradicionalno `self` ili kakvo god ime želite.

Parser ima slično sučelje kao lexer, što se tiče „konzumiranja" tokena:

`p.vidi(T.TIP)`, skraćeno `p > T.TIP` ako bi sljedeći pročitani token bio tipa `T.TIP`, vraća ga (ali ne čita), inače vraća `nenavedeno`

`p.nužno(T.TIP)`, skraćeno `p >> T.TIP` čita sljedeći token koji mora biti tipa `T.TIP`, i vraća ga; inače prijavljuje grešku

`p.slijedi(T.TIP)`, skraćeno `p >= T.TIP` čita sljedeći token (i vraća ga) ako i samo ako je tipa `T.TIP`; inače vraća `nenavedeno`.

Umjesto `T.TIP` može stajati i skup tipova tokena, koji se shvaća kao disjunkcija.

Za razliku od lexera, gdje možemo čitati bilo kakve znakove (npr. s `next`) pa ih poslije zgurati nekamo kao dio `lex.sadržaja`, parser mora svaki token *razriješiti* jednom od gornjih metoda; inače ćemo dobiti sintaksnu grešku. Nju možemo i sami prijaviti pomoću `raise p.greška(poruka)`; kao i za leksičku grešku, poruku ne moramo navesti ako smo zadovoljni veprovom konstrukcijom poruke. Također, grešku ćemo dobiti ako stanemo prije kraja (jer, opet, nismo razriješili sve tokene); možemo zamisliti da nakon svih lexerovih tokena postoji još jedan token tipa `KRAJ`, koji ne smijemo pročitati (pročitat će ga vepar na kraju parsiranja) ali ga možemo koristiti (operatorom >) za upravljanje sintaksnom analizom: recimo, `while not p > KRAJ:`.

Objekt `nenavedeno` je specijalni singleton koji predstavlja odsustvo tokena u ASTu (iz tehničkih razloga ne koristi se Pythonov `None`). Važno je da je lažan (dok su svi tokeni istiniti) pa se može koristiti za grananje potpuno jednako kao `None`.

2.4 Primjer

Na primjer, zadnji red u našoj BKG može se kodirati kao

```
def faktor(p):
    if p >= T.OTV:
        u_zagradi = p.izraz()
        p >> T.ZATV
        return u_zagradi
    else: return p >> T.BROJ
```

Pomoću „morž-operatora” `:=` možemo određene rezultate gornjih metoda iskoristiti za grananje, a ujedno ih i zapamtiti za kasnije vraćanje. Recimo, ako iz nekog razloga želimo zamijeniti grane u upravo napisanoj metodi, imat ćemo

```
def faktor(p):
    if broj := p >= T.BROJ: return broj
    p >> T.OTV
    u_zagradi = p.izraz()
    p >> T.ZATV
    return u_zagradi
```

Sve je to lijepo ako sve mogućnosti za neku varijablu počinju različitim tokenima: tako smo odmah znali koje pravilo za faktor trebamo slijediti ovisno o tome jesmo li započeli faktor tokenom tipa `T.OTV` ili tokenom tipa `T.BROJ`. (Ponekad je potrebno gledati više od jednog tokena unaprijed, ali vepar to ne podržava jednostavno; prvenstveno jer vodi do jezika koji su i ljudima teški za razumijevanje. Stručnim rječnikom, veprovo parsiranje je **LL(1)**.) No kako napisati metodu `izraz`? Imamo dvije mogućnosti, i obje počinju `član`om. U tom slučaju možemo *faktorizirati*, odnosno „izlučiti” `član` slijeva. Drugim riječima, možemo sigurno pročitati `član`, i onda donijeti odluku što dalje ovisno o tome slijedi li `PLUS` ili ne.

```
def izraz(p):
    prvi = p.član()
    if p >= T.PLUS:
        drugi = p.izraz()
```

(continues on next page)

(nastavak sa prethodne stranice)

```
        return Zbroj(prvi, drugi)
    else: return prvi
```

Sasvim je isto za `član`: napišite sami za vježbu. Sada od te tri metode možemo sklopiti klasu: važno je da metoda `izraz` bude prva, kako bi vepar znao da od nje počinje. Alternativno, mogli bismo je preimenovati u `start`, ali vjerojatno je svejedno dobro da bude prva.

```
class P(Parser):
    def izraz(p): ...   # prepišite odozgo
    def član(p): ...    # napišite sami po uzoru na izraz
    def faktor(p): ...  # prepišite jednu od varijanti odozgo
```

Da bismo mogli doista pokrenuti parser, trebamo još implementirati ASTove `Zbroj` i `Umnožak`. Zapravo, dovoljno je samo navesti atribute, a metode i anotacije možemo dodati kasnije.

```
class Zbroj(AST):
    lijevo: ...
    desno: ...

class Umnožak(Zbroj): pass
```

Trebali bismo svaku klasu koja predstavlja AST naslijediti od klase `AST`, ali zapravo, s obzirom na to da imaju iste atribute, možemo jednu naslijediti od druge. Sada napokon možemo vidjeti 1D i 2D prikaz našeg stabla:

```
>>> P('5+8*12')
Zbroj(prvi=BROJ'5', drugi=Umnožak(prvi=BROJ'8', drugi=BROJ'12'))

>>> prikaz(_)
Zbroj:  @[Znakovi #1-#6]
  prvi = BROJ'5'  @[Znak #1]
  drugi = Umnožak:  @[Znakovi #3-#6]
    prvi = BROJ'8'  @[Znak #3]
    drugi = BROJ'12'  @[Znakovi #5-#6]
```

2.5 Iterativno parsiranje

Dosad korišteni (rekurzivni) način pisanja parsera je izuzetno jednostavan (samo prepisujemo pravila gramatike u Python), ali ne radi uvijek, a i kada radi, ponekad ne daje intuitivne rezultate.

Lako je vidjeti (isprobajte) da će upravo napisani parser `2+3+4` parsirati na isti način kao `2+(3+4)` (kažemo da je `PLUS` *desno asociran*), a ne na isti način kao `(2+3)+4`. U ovom slučaju nije toliko bitno jer je zbrajanje asocijativna operacija pa će rezultat biti isti, ali to ne odgovara baš onome kako smo učili u školi (da je zbrajanje *lijevo asocirano*); i naravno, imali bismo problema pri uvođenju oduzimanja, jer `6-3-2` nije isto što i `6-(3-2)` čak ni po vrijednosti. Dakle, moramo naučiti parsirati i lijevo asocirane operatore.

Na prvi pogled, vrlo je jednostavno: samo umjesto `član PLUS izraz` u drugo pravilo za `izraz` napišemo `izraz PLUS član`. Iako matematički sasvim točno, to nam ne pomaže za pisanje parsera, jer bi rekurzivna paradigma koju smo dosad koristili zahtijevala da unutar metode `izraz` prvo pozovemo metodu `izraz`, što bi naravno dovelo do rušenja zbog beskonačne rekurzije. Taj fenomen se u teoriji parsiranja zove *lijeva rekurzija* (*left recursion*).

Možemo li bez rekurzije parsirati to? Sjetimo se gramatičkih sustava. Po lemi o fiksnoj točki (lijevolinearno), rješenje od `izraz = izraz PLUS član | član` je `izraz = član (PLUS član)*`. Drugim riječima, samo trebamo pročitati `član`, i onda nakon toga nula ili više puta (u neograničenoj petlji) čitati `PLUS` i `član` sve dok možemo.

```
def izraz(p):
    stablo = p.član()
    while p > T.PLUS:
        p >> T.PLUS
        novi = p.član()
        stablo = Zbroj(stablo, novi)
    return stablo
```

Naravno, koristeći >=, cijela petlja se u jednoj liniji može zapisati kao:

```
while p >= T.PLUS: stablo = Zbroj(stablo, p.član())
```

2.6 Asociranost

Potrebno je neko vrijeme da se priviknete na takav način pisanja, ali jednom kad uspijete, vidjet ćete da istu tehniku možete primijeniti na mnogo mjesta. Na primjer, iterativno možete parsirati i *višemjesne* (asocijativne) operatore — gdje ne želite stablo povećavati u dubinu nego u širinu, držeći sve operande kao neposrednu djecu osnovnog operatora (npr. u listi).

```
def izraz(p):
    pribrojnici = [p.član()]
    while p >= T.PLUS: pribrojnici.append(p.član())
    return Zbroj(pribrojnici)
```

Zadnji napisani kod ima jedan problem: uvijek vraća `Zbroj`, čak i kad nikakvog zbrajanja nema (onda će lista biti duljine 1). Da to izbjegnete, možete koristiti alternativni konstruktor ASTova, `return Zbroj.ili_samo(pribrojnici)`. Način na koji to radi je ono što biste očekivali: ako `A` ima točno jedan atribut, i njegova vrijednost je lista s jednim elementom `[w]`, tada `A.ili_samo([w])` umjesto `A([w])` vraća `w`.

Jednostavnom zamjenom `while` s `if`, možete implementirati i *neasocirane* operatore, koji uopće nemaju rekurzivna pravila: recimo, da smo imali pravilo `izraz -> član | član PLUS član`, bili bi dozvoljeni samo zbrojevi jednog ili dva pribrojnika, i za bilo koji veći broj pribrojnika morali bismo eksplicitno koristiti zagrade.

Idealno, svaka metoda parsera bi trebala koristiti samo rekurzivni ili samo iterativni pristup. Doista, asociranost je svojstvo *razine prioriteta*, ne pojedinog operatora. Recimo, ako imamo + i - na istoj razini prioriteta kao što je uobičajeno, ne može + biti desno a - lijevo asociran: tada bi `6-2+3` bio dvoznačan (mogao bi imati vrijednost 1 ili 7), dok `6+2-3` ne bi uopće imao značenje.

Drugim riječima, da bismo parsirali `a$b¤c`, moramo odlučiti „kome ide b”. Ako su operatori `$` i `¤` različitog prioriteta, dohvatit će ga onaj koji je većeg prioriteta. Ali ako su na istoj razini, tada asociranost te razine određuje rezultat: ako je asocirana lijevo, to je `(a$b)¤c`, a ako je asocirana desno, to je `a$(b¤c)`.

U istu paradigmu možemo uklopiti i unarne operatore, zamišljajući da su prefiksni operatori „desno asocirani” (rekurzivno parsirani), dok su postfiksni „lijevo asocirani” (iterativno parsirani). I opet, u skladu s upravo navedenim argumentom, ne možemo imati prefiksne i postfiksne operatore na istoj razini prioriteta, jer ne bismo znali kako parsirati `$b¤` („kome ide b”).

POGLAVLJE 3

Semantička analiza

Već smo vidjeli kako deklarirati ASTove u poglavlju o sintaksnoj analizi. Ponovimo: pišemo ih kao potklase veprove klase `AST`, navodeći odmah ispod toga deklaracije njihovih atributa. Primjer:

```
class Petlja(AST):
    uvjet: 'logički_izraz'
    tijelo: 'naredba|Blok'
```

Anotacije (ovo iza dvotočke u deklaracijama) mogu biti proizvoljne, vepar ih ne čita. Pristojno je napisati ili varijablu gramatike (metodu parsera) koja vraća nešto što može doći na to mjesto, ili tip (klasu) ASTa koji se direktno može tamo staviti. Anotacije treba pisati kao stringove, osim ako ćete jako paziti na poredak tako da ne koristite ništa što već nije definirano. (Možete i `from __future__ import annotations` na početku ako ne želite paziti na to.) Kao što rekoh, vepra nije briga što ćete tamo napisati: `atribut: ...` je sasvim u redu ako vam se ne da razmišljati, ali slično kao za BKG, pažljiv dizajn ASTova i njegova dokumentacija kroz tipove atributa vrlo je korisna stepenica u uspješnoj implementaciji semantičkog analizatora.

Krenuvši od korijena (to je AST kojeg stvara početna varijabla gramatike) i slijedeći atribute, prije ili kasnije doći ćemo do *listova*, što su najčešće tokeni. Dualno, možemo zamisliti ASTove kao nekakve obrasce u koje parser slaže tokene koje lexer generira, kako bi prikazao sintaksnu strukturu programa. Obilascima ASTova (koristeći najčešće rekurzivne metode) sada možemo učiniti sljedeći korak i našem programu „udahnuti dušu" odnosno dati mu nekakvu semantiku.

S ASTovima možemo raditi praktički što god želimo: pretvarati ih u jednostavnije ili po nekom drugom kriteriju bolje ASTove (optimizacija), preslikavati ih u nekakvu međureprezentaciju, bajtkod ili čak strojni kod (kompilacija), ili ih direktno preslikavati u stvarni svijet efekata i izlaznih vrijednosti (izvršavanje/evaluacija). Ovo je mjesto gdje vepar prestaje biti rigidni *framework* i postaje biblioteka raznih alata za koje se pokazuje da su često korisni u semantičkim analizama mnogih jezika. Događa se inverzija kontrole: dok je pri leksičkoj i sintaksnoj analizi vepar sam po potrebi pozivao lexer i metode parsera, i od njih slagao ASTove, sada ASTovi sami brinu za svoju kontrolu toka, eventualno koristeći veprove strukture i pozivajući njegove pomoćne funkcije tamo gdje je potrebno.

Tako ovdje samo navodimo dobre prakse za neke najčešće obrasce semantičke analize; prostor mogućnosti ovdje je daleko veći nego što se to u ovakvom tutorialu može opisati.

3.1 Evaluacija

Metode pišemo na pojedinim ASTovima, pozivajući (iste ili različite) metode na drugim ASTovima koji su najčešće neposredni potomci početnog, sve dok ne dođemo do poziva metode na tokenu u listu ASTa, koja predstavlja bazu rekurzije. Recimo, u klasi `Zbroj` mogli bismo napisati metodu

```
def vrijednost(self):
    return self.lijevo.vrijednost() + self.desno.vrijednost()
```

i sasvim analogno (napišite sami!) za `Umnožak`, što bi zajedno s metodom `vrijednost` na `BROJevima` (koju smo napisali u poglavlju o leksičkoj analizi) dalo da napokon možemo dobiti

```
>>> P('5+8*12').vrijednost()
101
```

Prilično analogno, samo bez povratne vrijednosti, možemo *izvršavati* naredbe u programima. Recimo, `Petlja` definirana na početku ovog poglavlja mogla bi imati metodu

```
def izvrši(self):
    while self.uvjet.vrijednost(): self.tijelo.izvrši()
```

(pod pretpostavkom da sve što metoda parsera `logički_izraz` može vratiti ima metodu `vrijednost` koja vraća `bool`, te da svaka `naredba`, kao i `Blok`, imaju metodu `izvrši`). Implementacija bloka naredbi bila bi u tom slučaju

```
class Blok(AST):
    naredbe: 'naredba*'
    def izvrši(self):
        for naredba in self.naredbe(): naredba.izvrši()
```

3.2 Memorija

Što ako u izrazu imamo varijable? Treba nam (a) neki način da im pridijelimo vrijednost (naredbom pridruživanja, ili prijenosom izvana), (b) način da te vrijednosti učinimo dostupnom svakoj metodi `vrijednost` na svim ASTovima i Tokenima, te (c) način da vepru kažemo da prijavi suvislu grešku u slučaju nepostojanja (ili, u nekim slučajevima, redefinicije) tražene varijable.

Za (a) možemo koristiti Pythonov rječnik, ali u tom slučaju je komplicirano riješiti (c) (opet, zbog inverzije kontrole; vepar ne zna ništa o tome što mi radimo s tim rječnikom). Također, nezgodno je uvijek misliti jesu li ključevi u rječniku tokeni ili njihovi sadržaji (ponekad je bolje jedno, ponekad drugo). Sve te (i druge) probleme rješava `Memorija`. To je klasa čije instance su specijalno prilagođene za praćenje metapodataka (vrijednosti, tipova, ...) raznih tokena, bilo direktno bilo po sadržaju. Pri konstrukciji možemo (a i ne moramo) navesti inicijalne podatke u obliku rječnika, druge memorije, ili čak objekta poput `zip(imena, vrijednosti)`. Također možemo navesti opciju `redefinicija=False` (podrazumijevano je `True`), čime zabranjujemo *rebinding* (ponovo pridruživanje) tokenima koji već postoje u memoriji. Naravno, ako su metapodaci promjenjivi Pythonovi objekti, možemo ih mijenjati bez obzira što smo tu naveli.

Kako riješiti (b)? *Low-tech* rješenje je jednostavno memoriju, jednom kad je konstruiramo, poslati kao argument u svaki poziv. To je sasvim izvedivo, ali zahtijeva puno mijenjanja (posebno ako se kasnije sjetimo da još nešto trebamo prenijeti na isti način), a i djeluje pomalo šašavo pisati da `T.BROJ.vrijednost` prima argument koji uopće ne koristi. Za to možemo koristiti *rt* (od *runtime*), veprov globalni objekt koji u svojim atributima drži sve potrebno za izvršavanje, pa tako i memoriju ako nam je potrebna. Dakle, samo treba negdje pri početku izvršavanja inicijalizirati `rt.mem = Memorija()`, realizirati nekakvu naredbu pridruživanja poput

```
class Pridruživanje(AST):
    čemu: 'IME'
    pridruženo: 'izraz'
    def izvrši(self):
        rt.mem[self.čemu] = self.pridruženo.vrijednost()
```

i klasi `T.IME` dodati metodu

```
def vrijednost(t): return rt.mem[t]
```

3.3 Optimizacija

Jedna konceptualno jednostavna operacija na ASTovima je optimizacija: od jednog ASTa napravimo drugi, najčešće tako da uklonimo neke nepotrebne dijelove. Recimo, uz globalnu definiciju `nula = Token(T.BROJ, '0')`, mogli bismo u klasi `Zbroj` napisati

```
def optim(self):
    if self.lijevo == nula: return self.desno.optim()
    elif self.desno == nula: return self.lijevo.optim()
    else: return self
```

ali to **nije dobra** metoda za optimizaciju, jer gleda samo „plitko" izraze do dubine 1: recimo, neće optimizirati `(0+0)+5`. Bolje je rekurzivno optimizirati prvo potomke, što može otkriti neke obrasce koje možemo iskoristiti na osnovnoj razini:

```
def optim(self):
    ol, od = self.lijevo.optim(), self.desno.optim()
    if ol == nula: return od
    elif od == nula: return ol
    else: return Zbroj(ol, od)
```

Alternativno, možemo i *mijenjati* ASTove direktno, što ima svoje prednosti poput čuvanja podataka o rasponu pojedinih ASTova, ali ima i mane jer je teže pisati rekurzivne funkcije koje mijenjaju svoje argumente. To se obično koristi u složenijim jezicima, gdje inkrementalna kompleksnost takvog postupka nije prevelika, a ogromnu većinu „gornjih slojeva" nećemo mijenjati optimizacijom, pa nema smisla da ih prepisujemo svaki put (tzv. lokalni ili *peephole* optimizatori).

POGLAVLJE 4

vepar module

Vepar vam omogućuje da koristeći Python pišete vlastite programske jezike, prolazeći kroz uobičajene faze kao što su leksiranje, parsiranje, proizvodnja apstraktnih sintaksnih stabala (AST), optimizacija, generiranje bajtkoda za virtualni stroj, te izvršavanje (interpretaciju u užem smislu).

Ne trebate znati puno Pythona za to: vepar je framework (poput npr. Djanga) koji ima svoje konvencije i prilično rigidan uobičajeni stil pisanja, iz kojih ćete rijetko trebati izaći. Ipak, poznavanje Pythona sigurno će pomoći, jer ćete manje toga morati naučiti i mnogi detalji u dizajnu imat će vam više smisla.

Iako se vepar može koristiti u profesionalnom okruženju, prvenstveno je zamišljen kao akademski alat: cilj je naučiti što više o uobičajenim tehnikama interpretacije programa, a ne napisati najbrži parser, ni semantički analizator s najboljim mogućim oporavkom od grešaka, niti generator strašno optimiziranog koda.

To je ujedno i glavna prednost vepra pred uobičajenim alatima kao što su flex, yacc i LLVM: ne morajući cijelo vrijeme razmišljati o performansama, vepar može puno bolje izložiti osnovne koncepte koji stoje u pozadini raznih faza interpretacije programa, razdvajajući ih i tretirajući svaki zasebno. Spoj toga i uobičajene Pythonove filozofije „sve je interaktivno, dinamično i istraživo" predstavlja dobru podlogu za učenje.