UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Ivan Ristović

KREIRANJE ZAJEDNIČKE AST APSTRAKCIJE ZA RAZLIČITE PROGRAMSKE JEZIKE

master rad

Beograd, 2020.

Mentor:

doc. dr Milena Vujošević-Janičić
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Ana Anić
University of Disneyland, Nedođija

dr Laza Lazić
Univerzitet u Beogradu, Matematički fakultet

Datum odbrane: _____

TODO zahvalnica

Naslov master rada: Kreiranje zajedničke AST apstrakcije za različite programske jezike

Rezime: Fijuče vetar u šiblju, ledi pasaže i kuće iza njih i gunđa u odžacima. Nidžo, čežnjivo gledaš fotelju, a Đura i Mika hoće poziciju sebi. Ljudi, jazavac Džef trči po šumi glođući neko suho žbunje. Ljubavi, Olga, hajde pođi u Fudži i čut ćeš nježnu muziku srca. Boja vaše haljine, gospođice Džafić, traži da za nju kulučim. Hadži Đera je zaćutao i bacio čežnjiv pogled na šolju s kafom. Džabe se zec po Homolju šunja, čuvar Jožef lako će i tu da ga nađe. Odžačar Filip šalje osmehe tuđoj ženi, a njegova kuća bez dece. Butić Đuro iz Foče ima pun džak ideja o slaganju vaših željica. Džajić odskoči u aut i izbeže đon halfa Pecelja i njegov šamar. Plamte odžaci fabrika a čađave guje se iz njih dižu i šalju noć. Ajšo, lepoto i čežnjo, za ljubav srca moga, dođi u Hadžiće na kafu. Huči šuma, a iza žutog džbuna i panja đak u cveću delje seji frulu. Goci i Jaćimu iz Banje Koviljače, flaša džina i žeđ padahu u istu uru. Džaba što Feđa čupa za kosu Milju, ona juri Živu, ali njega hoće i Daca. Dok je Fehim u džipu žurno ljubio Zagu Čađević, Cile se ušunjao u auto. Fijuče košava nad odžacima a Ilja u gunju žureći uđe u suhu i toplu izbu. Bože, džentlmeni osećaju fizičko gađenje od prljavih šoljica! Dočepaće njega jaka šefica, vođena ljutom srdžbom zlih žena. Pazi, gedžo, brže odnesi šefu taj đavolji ček: njim plaća ceh. Fine džukce ozleđuje bič: odgoj ih pažnjom, strpljivošću. Zamišljao bi kafedžiju vlažnih prstića, crnjeg od čađi. Đače, uštedu plaćaj žaljenjem zbog džinovskih cifara. Džikljaće žalfija između tog busenja i peščanih dvoraca. Zašto gđa Hadžić leči živce: njena ljubav pred fijaskom? Jež hoće peckanjem da vređa ljubičastog džina iz flaše. Džej, ljubičast zec, laže: gađaće odmah pokvašen fenjer. Plašljiv zec hoće jeftinu dinju: grožđe iskamči džabe. Džak je pun žica: ćućeš tad svađu zbog lomljenja harfe. Čuj, džukac Flop bez daha s gađenjem žvaće stršljena. Oh, zadnji šraf na džipu slab: muž gđe Cvijić ljut koči. Šef džabe zvižduće: mlađi hrt jače kljuca njenog psa. Odbaciće kavgadžija plaštom čađ u željezni fenjer. Deblji krojač: zgužvah smeđ filc u tanjušni džepić. Džo, zgužvaćeš tiho smeđ filc najdeblje krpenjače. Štef, bacih slomljen dečji zvrk u džep gđe Žunjić. Debljoj zgužvah smeđ filc — njen škrt džepčić.

Ključne reči: TODO

Sadržaj

Glava 1

Uvod

Fijuče vetar u šiblju, ledi pasaže i kuće iza njih i gunđa u odžacima. Nidžo, čežnjivo gledaš fotelju, a Đura i Mika hoće poziciju sebi. Ljudi, jazavac Džef trči po šumi glođući neko suho žbunje. Ljubavi, Olga, hajde pođi u Fudži i čut ćeš nježnu muziku srca. Boja vaše haljine, gospođice Džafić, traži da za nju kulučim. Hadži Đera je zaćutao i bacio čežnjiv pogled na šolju s kafom. Džabe se zec po Homolju šunja, čuvar Jožef lako će i tu da ga nađe. Odžačar Filip šalje osmehe tuđoj ženi, a njegova kuća bez dece. Butić Đuro iz Foče ima pun džak ideja o slaganju vaših željica. Džajić odskoči u aut i izbeže đon halfa Pecelja i njegov šamar. Plamte odžaci fabrika a čađave guje se iz njih dižu i šalju noć. Ajšo, lepoto i čežnjo, za ljubav srca moga, dođi u Hadžiće na kafu. Huči šuma, a iza žutog džbuna i panja đak u cveću delje seji frulu. Goci i Jaćimu iz Banje Koviljače, flaša džina i žeđ padahu u istu uru. Džaba što Feđa čupa za kosu Milju, ona juri Živu, ali njega hoće i Daca. Dok je Fehim u džipu žurno ljubio Zagu Čađević, Cile se ušunjao u auto. Fijuče košava nad odžacima a Ilja u gunju žureći uđe u suhu i toplu izbu. Bože, džentlmeni osećaju fizičko gađenje od prljavih šoljica! Dočepaće njega jaka šefica, vođena ljutom srdžbom zlih žena. Pazi, gedžo, brže odnesi šefu taj đavolji ček: njim plaća ceh. Fine džukce ozleđuje bič: odgoj ih pažnjom, strpljivošću. Zamišljao bi kafedžiju vlažnih prstića, crnjeg od čađi. Đače, uštedu plaćaj žaljenjem zbog džinovskih cifara. Džikljaće žalfija između tog busenja i peščanih dvoraca. Zašto gđa Hadžić leči živce: njena ljubav pred fijaskom? Jež hoće peckanjem da vređa ljubičastog džina iz flaše. Džej, ljubičast zec, laže: gađaće odmah pokvašen fenjer. Plašljiv zec hoće jeftinu dinju: grožđe iskamči džabe. Džak je pun žica: ćućeš tad svađu zbog lomljenja harfe. Čuj, džukac Flop bez daha s gađenjem žvaće stršljena. Oh, zadnji šraf na džipu slab:

muž gđe Cvijić ljut koči. Šef džabe zvižduće: mlađi hrt jače kljuca njenog psa. Odbaciće kavgadžija plaštom čađ u željezni fenjer. Deblji krojač: zgužvah smeđ filc u tanjušni džepić. Džo, zgužvaćeš tiho smeđ filc najdeblje krpenjače. Štef, bacih slomljen dečji zvrk u džep gđe Žunjić. Debljoj zgužvah smeđ filc — njen škrt džepčić.

Glava 2

Pregled relevantnih pojmova

U ovom poglavlju će biti opisani koncepti i alati čije je razumevanje potrebno kako bi se razumeo opis dalje apstrakcije i implementacije samog programa. Umesto analize samog sadržaja izvornog koda analizira se *apstraktno sintaksno stablo* (eng. *Abstract Syntax Tree*, u daljem tekstu *AST*), opisano u odeljku 2.1. Alat koji je korišćen za generisanje parsera za proizvoljnu gramatiku jezika se zove *Another Tool For Language Recognition* [1], u daljem tekstu *ANTLR*, opisan u odeljku 2.3. Parser generiše AST specifičan za datu gramatiku i nema sličnosti u dobijenim apstrakcijama za različite jezike. Kako bismo poredili stabla različitih jezika, kreiramo reprezentaciju na višem nivou i specifični AST podižemo na taj nivo. Ta reprezentacija će biti opisana u narednim poglavljima, kao i načini kako se ona može analizirati. Takođe, pojmovi specifični za implementaciju će takođe biti opisani u ovom poglavlju.

2.1 Apstraktna sintaksna stabla - AST

Kako bi se kod pisan u nekom programskom jeziku (*izvorni fajl*) preveo u kod koji će se izvršavati na nekoj mašini (*izvršivi fajl*), prevodilac prolazi kroz određene korake. Da bi se izvorni kod preveo, prevodilac mora da zna njegovu formu, ili *sintaksu*, i njegovo značenje, ili *semantiku*. Deo prevodioca koji određuje da li je izvorni kod ispravno formiran u terminima sintakse i semantike se naziva *prednji deo* (engl. *front end*). Ukoliko je izvorni kod ispravan, prednji deo kreira *međureprezentaciju* koda (engl. *intermediate representation*, u daljem tekstu *IR*). Ukoliko to nije slučaj, prevođenje ne uspeva i programeru se daje poruka o detaljima zašto prevođenje nije uspelo. [5]

Postupak rada prednjeg dela će biti opisan kroz konkretan primer. Pretpostavimo da želimo da prevedemo kod pisan u programskom jeziku C prikazan na slici 2.1. Primetimo da postoji greška u datom kodu - simbol `c` koji se koristi u dodeli u liniji 8 će biti prepoznat kao identifikator koji ne odgovara nijednoj deklarisanoj promenljivoj - stoga ne možemo prevesti ovaj kod. Ovo, doduše, nije sintaksna greška - izraz `a+c` je sasvim validan u programskom jeziku C bez analize konteksta u kom se javlja. Problem će postati očigledan tek nakon parsiranja izvornog koda i provere ispunjenosti sintaksih pravila, tačnije u fazi semantičke provere. Stoga se ovakve greške nazivaju *semantičke greške*, dok se greške u sintaksi nazivaju *sintaksne greške*.

```
1   #include<stdio.h>
2
3   #define T int
4
5   int main()
6   {
7       T a, b;
8       a = a + c;          // c nije deklarisano
9       printf("%d", a);
10      return 0;
11  }
```

Slika 2.1: Primer izvornog koda pisanog u programskom jeziku C.

Pre nego što prednji kraj prevodioca uopšte dobije kod koji treba prevesti, vrši se *pretprocesiranje* od strane programa koji se naziva *pretprocesor*. U fazi pretprocesiranja se izvode samo tekstualne operacije kao što su brisanje komentara ili zamena makroa u jezicima kao što je C. Rezultat rada pretprocesora za kod sa slike 2.1 bi izgledao kao na slici 2.2 [1].

Da bi proverio sintaksu izvornog koda, prevodilac mora da uporedi strukturu istog sa unapred definisanom strukturom za određeni programski jezik. Ovo zahteva formalnu definiciju sintakse jezika. Programski jezik možemo posmatrati kao skup *pravila* koji se naziva *gramatika* [4], prikazana na slici 2.3. U prednjem

[1] U nekim implementacijama C standardne biblioteke, moguće je da se poziv funckije `printf` zameni pozivom funkcije `fprintf` sa ispisom na `stdout`. U standardu se propisuje da funkcije kao što je `printf` mogu biti implementirane kao makroi. Izlaz na slici 2.2 je generisan od strane `GCC 7.4.0` po C11 standardu i ovo nije slučaj u datom okruženju.

```
1       # 1 "<stdin>"
2       # 1 "<built-in>"
3       # 1 "<command-line>"
4       # 31 "<command-line>"
5       # 1 "/usr/include/stdc-predef.h" 1 3 4
6
7       ...
8
9       extern char *ctermid (char *__s) __attribute__
           ((__nothrow__ , __leaf__));
10      # 840 "/usr/include/stdio.h" 3 4
11      extern void flockfile (FILE *__stream) __attribute__
           ((__nothrow__ , __leaf__));
12      extern int ftrylockfile (FILE *__stream) __attribute__
           ((__nothrow__ , __leaf__)) ;
13      extern void funlockfile (FILE *__stream) __attribute__
           ((__nothrow__ , __leaf__));
14      # 868 "/usr/include/stdio.h" 3 4
15      # 2 "<stdin>" 2
16      # 2 "<stdin>"
17
18      int main()
19      {
20          int a, b;
21          a = a + c;
22          printf("%d", a);
23          return 0;
24      }
```

Slika 2.2: Prikaz rezultata rada pretprocesora za izvorni kod sa slike 2.1. Pritom, prikazano je samo par linija sa početka i kraja izlaza pretprocesora - kod iznad `main` funkcije je uključen iz `stdio.h` zaglavlja.

delu se izvode dva procesa koji određuju da li ulaz zaista zadovoljava gramatiku određenog programskog jezika. Ova dva procesa se nazivaju *skeniranje* i *parsiranje*, a komponente prednjeg dela koje vrše te procese se nazivaju *skener* (takođe se naziva i *lekser*) i *parser*, redom.

Prilikom faze prevođenja, kako prevodilac ne bi radio nad sirovim karakterima izvornog koda, potrebno je izvršiti pripremu istog - skeniranje. Prevodilac ima u vidu moguće elemente programskog jezika, tzv. *tokene*, koje treba prepoznati u datom fajlu - ključne reči, operatore, promenljive itd. Proces prepoznavanja

```
1     functionDefinition
2         :   declarationSpecifiers? declarator
3             declarationList? compoundStatement
4         ;
5
6     declarationList
7         :   declaration
8         |   declarationList declaration
9         ;
10
11    declaration
12        :   declarationSpecifiers initDeclaratorList ';'
13        |   declarationSpecifiers ';'
14        |   staticAssertDeclaration
15        ;
```

Slika 2.3: Isečak gramatike programskog jezika C po standardu C11.

tokena u izvornom fajlu se naziva *tokenizacija*. Pojednostavljen primer tokena koje lekser pokušava da prepozna se može videti na slici 2.4. Primer izlaza leksera za izlaz pretprocesora sa slike 2.2 se može videti na slici 2.5. [2]

```
1     Identifier : IdentifierNondigit
2                  (IdentifierNondigit | Digit)*
3                ;
4
5     IdentifierNondigit : Nondigit
6                        | UniversalCharacterName
7                        ;
8
9     Nondigit : [a-zA-Z_]
10              ;
11
12    Digit : [0-9]
13          ;
```

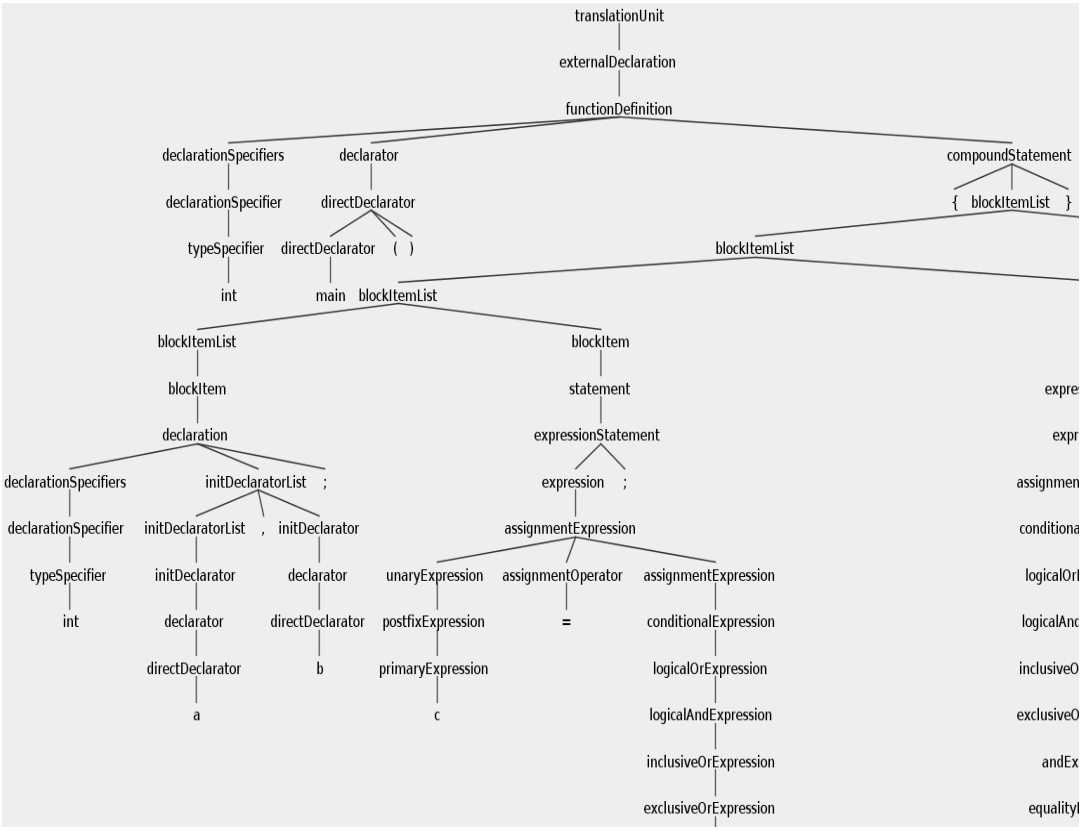
Slika 2.4: Primer delimične definicije tokena za ime promenljive po C11 standardu.

[2]Moderni kompajleri često nemaju odvojene faze u kojima se pozivaju skeniranja i parsiranja, već se skeniranje odvija paralelno sa fazom parsiranja. Međutim, to nas ne sprečava da ispišemo tokene onda kada se oni prepoznaju, i to je demonstrirano na slici 2.5.

```
1     identifier 'main'    [LeadingSpace] Loc=<sample.c:3:5>
2     l_paren '('     Loc=<sample.c:3:9>
3     r_paren ')'     Loc=<sample.c:3:10>
4     l_brace '{'  [StartOfLine]  Loc=<sample.c:4:1>
5     int 'int'    [StartOfLine] [LeadingSpace]
        Loc=<sample.c:5:5>
6     identifier 'a'   [LeadingSpace] Loc=<sample.c:5:9>
7     comma ','       Loc=<sample.c:5:10>
8     identifier 'b'   [LeadingSpace] Loc=<sample.c:5:12>
9     semi ';'        Loc=<sample.c:5:13>
10    identifier 'a'   [StartOfLine] [LeadingSpace]
        Loc=<sample.c:6:5>
11    equal '='    [LeadingSpace] Loc=<sample.c:6:7>
12    identifier 'a'   [LeadingSpace] Loc=<sample.c:6:9>
13    plus '+'     [LeadingSpace] Loc=<sample.c:6:11>
14    identifier 'c'   [LeadingSpace] Loc=<sample.c:6:13>
15    semi ';'        Loc=<sample.c:6:14>
16    identifier 'printf'  [StartOfLine] [LeadingSpace]
        Loc=<sample.c:7:5>
17    l_paren '('     Loc=<sample.c:7:11>
18    string_literal '"%d"'       Loc=<sample.c:7:12>
19    comma ','       Loc=<sample.c:7:16>
20    identifier 'a'   [LeadingSpace] Loc=<sample.c:7:18>
21    r_paren ')'     Loc=<sample.c:7:19>
22    semi ';'        Loc=<sample.c:7:20>
23    return 'return'  [StartOfLine] [LeadingSpace]
        Loc=<sample.c:8:5>
24    numeric_constant '0'     [LeadingSpace]
        Loc=<sample.c:8:12>
25    semi ';'        Loc=<sample.c:8:13>
26    r_brace '}'  [StartOfLine]  Loc=<sample.c:9:1>
27    eof ''      Loc=<sample.c:9:2>
```

Slika 2.5: Proces tokenizacije koda sa slike 2.2. Generisano uz pomoć `clang` [3] kompajlera.

Nakon faze skeniranja potrebno je parsirati dobijene tokene. Parser, imajući u vidu gramatiku jezika, pokušava da kreira *stablo parsiranja* (eng. *parse tree* ili *derivation tree*). Takvo stablo i dalje sadrži sve relevantne informacije o izvornom kodu. Vizuelni prikaz rada parsera za gramatiku sa slike C11 i izvonog koda sa slike 2.2 je dat na slici 2.6. Stablo parsiranja se koristi u narednim fazama prevođenja.

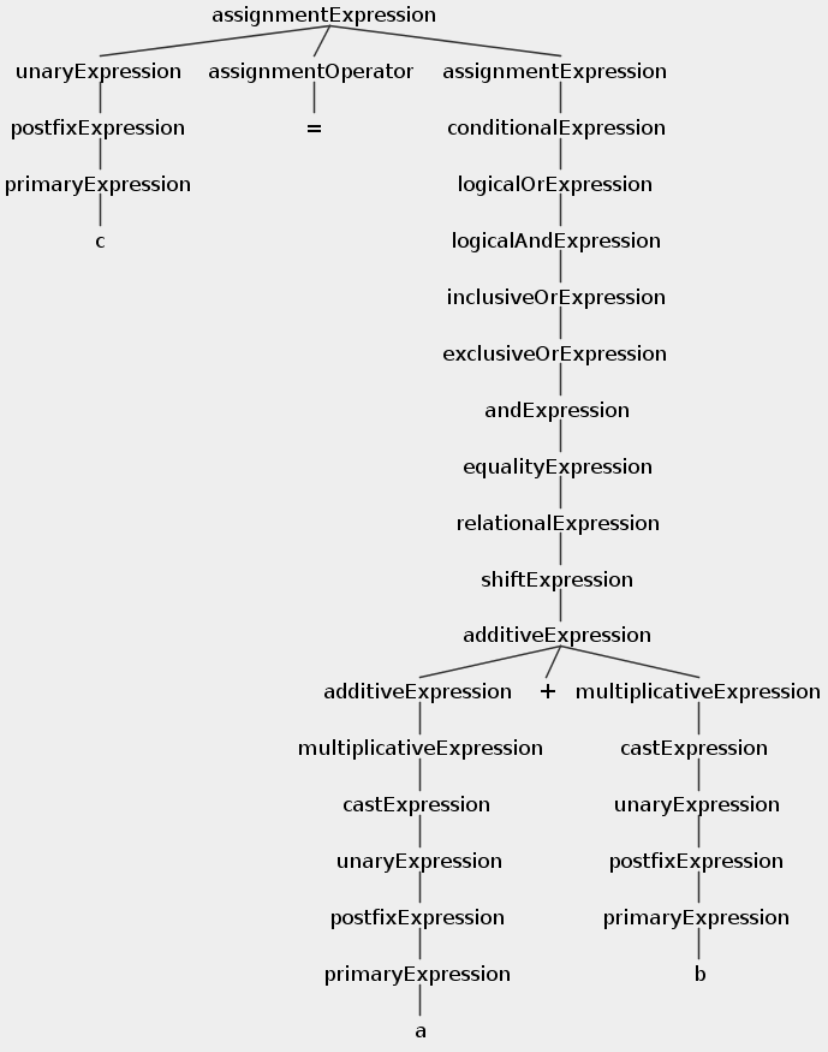


Slika 2.6: Prikaz dela stabla parsiranja koje generiše parser kreiran od strane alata ANTLR4 za kod sa slike 2.2.

Za potrebe ovog rada, što se procesa prevođenja tiče, dovoljno je pozvnavanje prednjeg dela, stoga neće biti reči o daljim koracima u fazi prevođenja (semantička provera, optimizacije, generisanje IR). Zainteresovani čitalac može više detalja pronaći u [5] i [18].

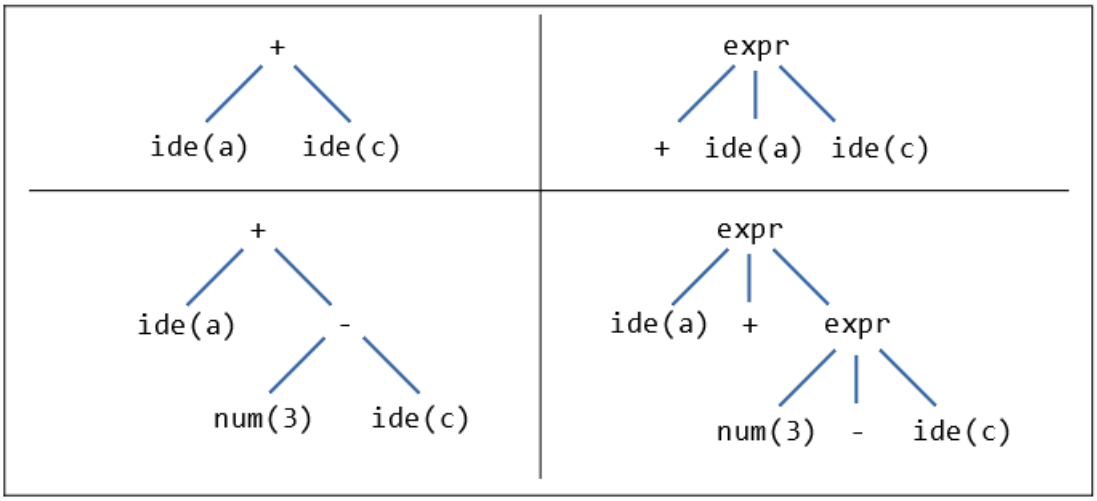
Stablo parsiranja sadrži sve informacije potrebne u fazi parsiranja uključujući detalje korisne samo za parser prilikom provere ispunjenosti gramatičkih pravila. Sa druge strane, *apstraktno sintaksno stablo* sadrži samo sintaksnu strukturu u jednostavnijoj formi. Na slici 2.7 se može videti koliko je stablo parsiranja komplikovano čak i za naizgled jednostavne aritmetičke izraze. Razlog ovolike komplikovanosti dolazi iz rekurzivnih pravila definisanih u C11 gramatici. Parseru su sve ove informacije neophodne ali na apstratknijem nivou nisu potrebne. Jedina važna semantička odlika izraza `a+c` je ta da je to zbir vrednosti nekih promenljivih - sve ostale informacije su nepotrebne. Na slici 2.8 se mogu videti različita apstraktna sintaksna stabla za pomenuti izraz, ali takođe i za malo komplikovanije izraze.

Podrazumeva se, naravno, da je ulaz već tokenizovan.



Slika 2.7: Prikaz kompleksnosti stabla parsiranja za izraz `a+c` u C11 gramatici.

Uloga apstraktnih sintaksnih stabala je da pokažu semantiku strukture koda preko stabala. Kao što se vidi na slici 2.8, postoji određeni nivo slobode u dizajniranju ovih stabala. Generalno, *terminalni simboli*, simboli koji predstavljaju listove stabla parsera, koji odgovaraju operatorima i naredbama se podižu naviše i postaju koreni podstabala, dok se njihovi operandi ostavljaju kao njihovi potomci u stablu. Desna stabla sa slike ne prate u potpunosti ovaj princip, ali se takođe koriste zbog regularnosti izraza - recimo ukoliko binarni izraz posmatramo kao koncept, mnogo je lakše raditi sa ovakvom strukturom. Ovakva struktura će biti korišćena kasnije u implementaciji programa. Primetimo takođe da se u stablima



Slika 2.8: Varijante apstraktnih sintaksnih stabala bez regularnosti(levo) i sa regularnošću (desno) za izraze `a+c` (gore) i `a + (3 - c)` (dole).

za izraz `a + (3 - c)` (dole) implicitno sačuvala informacija o prioritetu operacije oduzimanja u izrazu. Jasno je, dakle, da se računanje vrednosti aritmetičkih izraza onda vrši kretanjem od listova stabla ka korenu. Takođe, pošto su apstraktna sintaksna stabla apstrakcija stabla parsiranja, više istih izraza jezika može imati isto apstraktno sintaksno stablo ali različito stablo parsiranja; na primer, ako razmatramo izraz `(a + 5) - x / 2` i izraz `a + 5 - (x / 2)`.

Apstrakna sintaksna stabla će u daljem tekstu biti referisana skraćenicom *AST*, koja dolazi od engleskog naziva *Abstract Syntax Trees*. Takođe, reči samom dizajnu i tipovima čvorova AST-a korišćenih u implementaciji će biti u poglavlju 3. O procesu generisanja leksera i parsera za datu gramatiku programskog jezika će više biti reči u poglavlju [**?**], gde je opisan alat koji je korišćen za kreiranje parsera za C11 gramatiku (ali i za druge, proizvoljne gramatike)i koji daje mogućnost jednostavnog obilaska stabla parsiranja i pruža intuitivan način za izvršavanje logike nad tim stablom, što uključuje i izradu AST-a od stabla parsiranja.

2.2 Obrasci za projektovanje

Obrasci za projektovanje (engl. *design patterns* [7], drugačije nazvani i projektni šabloni, uzorci) predstavljaju opšte i ponovno upotrebljivo rešenje čestog problema, često implementirani koristeći koncepte objektno-orijentisanog programiranja. Svaki obrazac za projektovanje ima četiri osnovna elementa:

- ime - ukratko opisuje problem, rešenje i posledice
- problem - opisuje slučaj u kome se obrazac koristi
- rešenje - opisuje elemente dizajna kao i odnos tih elemenata
- posledice - obuhvataju rezultate i ocene primena obrasca

Obrasce za projektovanje je moguće grupisati po situaciji u kojoj se mogu iskoristiti ili načinu na koji rešavaju zadati problem. Stoga je opšte prihvaćena podela na tri grupe:

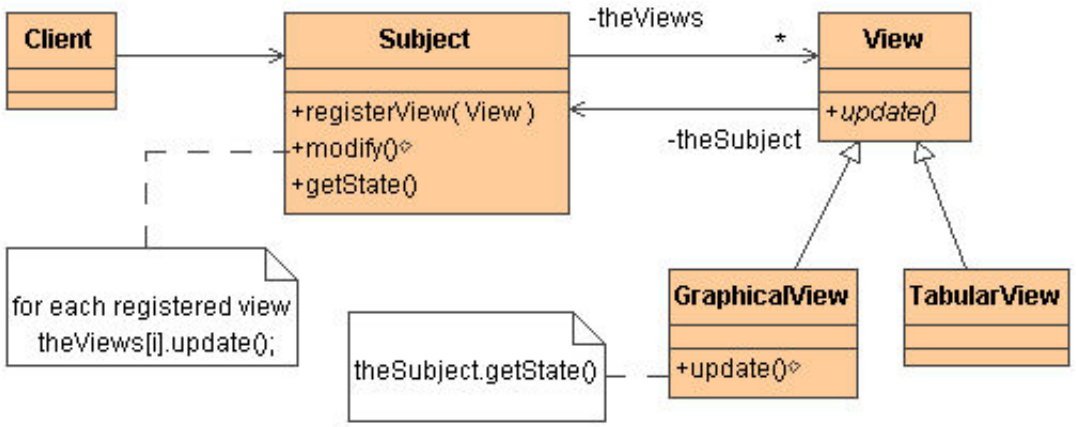
- *gradivni obrasci* (engl. *creational patterns*)
- *strukturni obrasci* (engl. *structural patterns*)
- *obrasci ponašanja* (engl. *behavioral patterns*)

Gradivni obrasci apstrahuju proces pravljenja objekata i važni su kada sistemi više zavise od sastavljanja objekata nego od nasleđivanja. Neki od najvažnijih gradivnih obrazaca su *apstraktna fabrika* (engl. *abstract factory*), *graditelj* (engl. *builder*), *proizvodni metod* (engl. *factory method*), *prototip* (engl. *prototype*) i *unikat* (engl. *singleton*). Strukturni obrasci se bave načinom na koji se klase i objekti sastavljaju u veće strukture. Neki od najvažnijih strukturnih obrazaca su *adapter* (engl. *adapter*), *most* (engl. *bridge*), *sastav* (engl. *composite*), *dekorater* (engl. *decorator*), *fasada* (engl. *facade*), *muva* (engl. *flyweight*) i *proksi* (engl. *proxy*). Obrasci ponašanja se bave načinom na koji se klase i objekti sastavljaju u veće strukture. Neki od najvažnijih strukturnih obrazaca su *lanac odgovornosti* (engl. *chain of responsibility*), *komanda* (engl. *command*), *interpretator* (engl. *interpreter*), *iterator* (engl. *iterator*), *posmatrač* (engl. *observer*), *strategija* (engl. *strategy*) i *posetilac* (engl. *visitor*).

Za potrebe ovog rada, obrasci za projektovanje će se koristiti kao opšte prihvaćeno i programerski intuitivno rešenje određenih problema. Takođe, u kontekstu stabala parsiranja i apstraktnih sintaksnih stabala, opisanih u poglavlju 2.1, obrasci *posmatrač* i *posetilac* su od velikog značaja jer pružaju interfejs za obilazak takvih stabala. Stoga se, osim u implementaciji opisanoj u narednim poglavljima, koriste i od strane drugih alata korišćenih u radu kao što je ANTLR, opisan u poglavlju 2.3. Stoga će u nastavku biti opisani samo obrasci posmatrač i posetilac, dok zainteresovani čitalac može pročitati više o ostalim obrascima u [9].

Obrazac „Posmatrač”

Obrazac za projektovanje *Posmatrač* (u daljem tekstu samo *posmatrač*) je strukturni obrazac za projektovanje koji se koristi kada je potrebno definisati jedan-ka-više vezu između objekata tako da ukoliko jedan objekat promeni stanje (subjekat) svi zavisni objekti su obavešteni o izmeni i shodno ažurirani. Posmatrač predstavlja *pogled* (engl. *View*) u MVC (engl. *Model-View-Controller*) arhitekturi. Na slici 2.9 se može videti UML dijagram [17] za obrazac posmatrač.



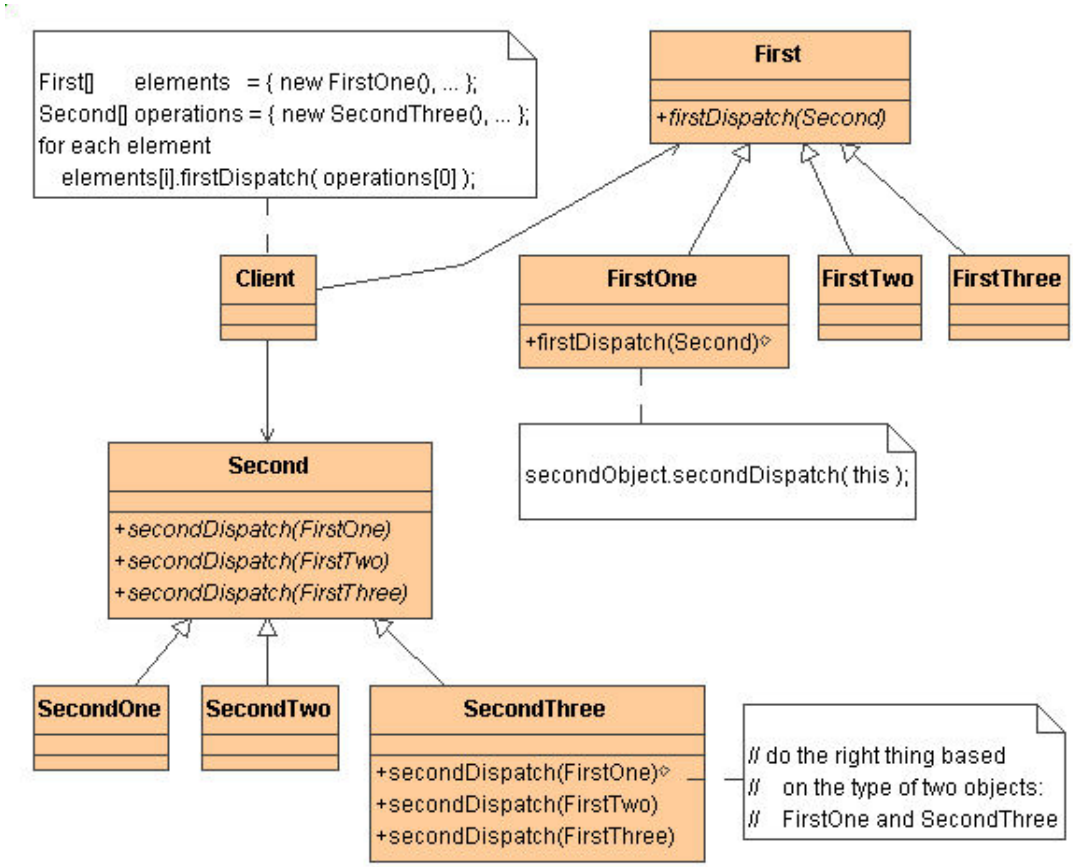
Slika 2.9: UML dijagram obrasca za projektovanje „Posmatrač”.

Primer upotrebe ovog obrasca može biti aukcija gde je aukcionar subjekat i započinje aukciju, dok učesnici aukcije (objekti) posmatraju aukcionera i reaguju na podizanje cene. Prihvatanje promene cene menja trenutnu cenu i aukcioner oglašava promenu iste, a svi učesnici aukcije dobijaju informaciju da se izmena izvršila. Za potrebe ovog rada, primer upotrebe može biti obilazak stablolike kolekcije (recimo stabla parsiranja) i obaveštavanje o nailasku na čvorove određenih tipova. Te informacije se dalje mogu iskoristiti za izračunavanja nad pomenutom strukturom ili generisanje novih struktura (recimo AST).

Obrazac „Posetilac”

Obrazac za projektovanje *Posetilac* (u daljem tekstu samo *posetilac*) je strukturni obrazac za projektovanje koji predstavlja operaciju koju je potrebno izvesti nad elementima objektne strukture. Posetilac omogućava definisanje nove operacije bez izmena klasa elemenata nad kojima operiše. Operacija koja će se izvesti

zavisi od imena zahteva, tipa posetioca i tipa elementa kog posećuje. Na slici 2.10 se može videti UML dijagram [17] za obrazac posetilac.



Slika 2.10: UML dijagram obrasca za projektovanje „Posetilac".

Primer upotrebe ovog obrasca može biti operisanje taksi kompanija. Kada osoba pozove taksi kompaniju (prihvatanje posetioca), kompanija šalje vozilo osobi koja je pozvala kompaniju. Nakon ulaska u vozilo (posetilac), mušterija ne kontroliše svoj transport već je to u rukama taksiste (posetioca). Za potrebe ovog rada, primer upotrebe može biti prikupljanje informacija o kolekciji stablolike strukture (recimo stablo parsiranja) i korišćenje istih za neko izračunavanje ili generisanje novih struktura (recimo AST, za potrebe ovog rada).

2.3 ANTLR

Pretpostavljajući da imamo gramatiku proizvoljnog programskog jezika, postavlja se pitanje: *Da li je moguće definisati postupak i zatim napraviti program koji će generisati kodove leksera i parsera napisane u određenom programskom jeziku za proizvoljnu gramatiku datu na ulazu?*. Odgovor je potvrdan i postoji veliki broj alata koji se mogu koristiti u ove svrhe, od kojih je navedeno par njih:

- *GNU Bison* [11]
  GNU Bison je generator parsera i deo GNU projekta [12], često referisan samo kao *Bison*. Bison generiše parser na osnovu korisnički definisanih kontekstno slobodnih gramatika [4], upozoravajući pritom na dvosmislenosti prilikom parsiranja ili nemogućnost primena gramatičkih pravila. Generisani parser je najčešće C a ređe C++ kod, mada se u vreme pisanja ovog rada eksperimentiše sa Java podrškom. Generisani kodovi su u potpunosti prenosivi i ne zahtevaju specifične kompajlere. Bison može da, osim podrazumevanih *LALR(1)* [13] parsera, generiše i kanoničke *LR* [15], *IELR(1)* [6] i *GLR* [10] parsere.

- *Flex* [8]
  Kreiran kao alternativa *lex*-u [14], Flex generiše samo leksere pa se stoga najčešće koristi u kombinaciji sa drugim alatima koji mogu da generišu parsere, kao što je *BYACC*, opisan u nastavku.

- *BYACC* [2]
  Berkeley YACC, skraćeno *BYACC*, pisan po ANSI C standardu i otvorenog koda, se smatra od strane mnogih kao *najbolja varijanta YACC-a* [14]. BYACC dozvoljava tzv. *reentrant* kod - memorija je deljenja između poziva pa je bezbedno konkurentno izvršavanje koda - na način kompatibilan sa Bison-om i to je delom razlog njegove popularnosti.

- *ANTLR* [1]
  Another Tool for Language Recognition, ili kraće *ANTLR*, je generator *LL(*)* [**?**] leksera i parsera pisan u programskom jeziku Java sa intuitivnim interfejsom za obilazak stabla parsiranja. Verzija 3 podržava generisanje parsera u jezicima Ada95, ActionScript, C, C#, Java, JavaScript, Objective-C, Perl, Python, Ruby, i Standard ML, dok verzija 4 u vreme pisanja ovog rada samo

generiše parsere u narednim jezicima: Java, C#, C++, JavaScript, Python, Swift i Go.

ANTLR, verzije 4, je izabran u ovom radu zbog svoje jednostavnosti, intuitivnosti i podrške za mnoge moderne programske jezike. Verzija 4 je izabrana umesto verzije 3 po preporuci autora, na osnovu eksperimentalne analize brzine i pouzdanosti verzije 4 u odnosu na verziju 3. Lekseri i parseri za ulazne gramatike će u implementaciji biti generisani u programskom jeziku C#.

Parseri generisani koristeći ANTLR koriste novu tehnologiju koja se naziva *Prilagodljiv LL(*)* (engl. *Adaptive LL(*)*) ili *ALL(*)* [16], dizajniranu od strane Terensa Para, autora ANTLR-a, i Sema Harvela. *ALL(*)* vrši *dinamičku analizu* gramatike u fazi izvršavanja, dok su starije verzije radile analizu pre pokretanja parsera, tzv. *statičku analizu*. Ovaj pristup je takođe efikasniji zbog značajno manjeg prostora ulaznih sekvenci.

Najbolji aspekt ANTLR-a je lakoća definisanja gramatičkih pravila koji opisuju sintaksne konstrukte nalik na aritmetičkim izrazima u programskim jezicima. Primer jednostavnog pravila za definisanje aritmetičkog izraza je dat na slici 2.11. Pravilo `exp` je levo rekurzivno jer barem jedna od njegovih alternativnih definicija referiše na baš pravilo `exp`. ANTLR4 automatski zamenjuje levo rekurzivna pravila u nerekurzivne ekvivalente. Jedini zahtev koji mora biti ispunjen je da levo rekurzivna pravila moraju biti *direktna* - da pravila odmah referišu sama sebe. Pravila ne smeju referisati drugo pravilo sa leve strane definicije takvo da se eventualno kroz rekurziju stigne nazad do pravila od kog se krenulo bez poklapanja sa nekim tokenom.

```
1      exp : (exp)
2          | exp '*' exp
3          | exp '+' exp
4          | INT
5          ;
```

Slika 2.11: Definicija uprošćenog aritmetičkog izraza u ANTLR4 gramatici.

Preduslovi za pokretanje ANTLR4

Kako bi ANTLR generisao parser u proizvoljnom programskom jeziku, potrebno je instalirati ANTLR i imati *Java Runtime Environment* (skr. *JRE*) instaliran na sistemu i dostupan globalno pokretanjem putem komande `java`. Instalacija se sastoji od preuzimanja najnovijeg *.jar* fajla [3], sa zvanične stranice [1] ili recimo korišćenjem *curl* alata:

```
1 $ curl -O http://www.antlr.org/download/antlr-4-complete.jar
```

Na UNIX sistemima moguće je kreirati alias `antlr4` ili *shell* skript unutar direktorijuma `/usr/local/bin` sa imenom `antlr4` koji će pokrenuti *.jar* fajl na sledeći način (pretpostavljajući da se *.jar* fajl nalazi u direktorijumu `/usr/local/lib`):

```
1 #!/bin/sh
2 java -cp "/usr/local/lib/antlr4-complete.jar:$CLASSPATH"
     org.antlr.v4.Tool $*
```

Na Windows sistemima moguće je kreirati *batch* skript sa imenom `antlr4.bat` koji će pokrenuti ANTLR4, na sledeći način (pretpostavljajući da se *.jar* fajl nalazi u direktorijumu `C:\lib`):

```
1 java -cp C:\lib\antlr-4-complete.jar;%CLASSPATH%
     org.antlr.v4.Tool %*
```

Ukoliko su aliasi ili skript fajlovi imenovani kao iznad, moguće je iz komandne linije pojednostavljeno pokretati ANTLR4:

```
1 $ antlr4
2 ANTLR Parser Generator Version 4.0
3 -o ___    specify output directory where all output is
     generated
4 -lib ___  specify location of .tokens files
5 ...
```

Dodatno, za Unix sisteme [4], moguće je kreirati dodatni alias `grun` (ili alternativno, kreirati `shell script`) za biblioteku `TestRig`. Biblioteka TestRig se može

[3] Takođe je moguće kompajlirati izvorni kod dostupan na servisu GitHub `https://github.com/antlr/antlr4`

[4] Za Windows operativni sistem je moguće kreirati *batch* skript po opisu na `https://github.com/antlr/antlr4/blob/master/doc/getting-started.md`.

koristiti za brzo testiranje parsera - moguće je pokrenuti parser od bilo kog pravila i dobiti izlaz parsera u raznim formatima. TestRig dolazi uz ANTLR `.jar` fajl i moguće je napraviti prečicu za brzo pokretanje (nalik na ANTLR alias):

```
1 $ alias grun='java -cp
      "/usr/local/lib/antlr-4-complete.jar:$CLASSPATH"
      org.antlr.v4.gui.TestRig'
```

Generisanje parsera koristeći ANTLR4

Prvi korak u izradi aplikacije koja u sebi koristi parsiranje nekog jezika je definisanje gramatike jezika i kreiranje leksera i parsera za isti. U nastavku će biti opisan proces kreiranja interfejsa za parsiranje programa pisanih u pseudo-programskom jeziku (u nastavku *pseudo-jezik*), nalik na pseudokod. Ovako dobijeni interfejs će moći da se koristi u opšte svrhe, za potrebe ovog rada će se koristiti za generisanje AST stabla za program pisan u pseudo-jeziku.

Definišimo gramatiku pseudo-jezika prateći ANTLR pravila za definisanje gramatika. Kao i za svaki drugi programski jezik, treba obezbediti da postoje određeni koncepti koji se pojavljuju u programskim jezicima: *identifikatori*, *izrazi*, *naredbe*, *funkcije* i slično. Za sada se fokusirajmo na naredbe, kao samostalne izvršive jedinice koda. Stoga program možemo smatrati kao niz naredbi. U nekim slučajevima će biti potrebno definisanje kompleksnih naredbi koje se sastoje od više drugih naredbi, i ovakve složene naredbe ćemo zvati *blok* ili *blok naredbi*. Stoga, radi konzistentnosti, program će biti blok naredbi. Kako bismo označili da su naredbe deo bloka naredbi, koristićemo reči `begin` i `end`, osim ukoliko je reč o samo jednoj naredbi. Ovakve situacije rešavamo definisanjem *alternativa* u definiciji pravila - više definicija razdvojenih simbolom `|`. Specijalne reči kao što su `begin` i `end` će biti rezervisane reči našeg pseudo-jezika, tzv. *ključne reči*. Na slici 2.12 se može videti definicija programa [5] i bloka naredbi pseudo-jezika, pri čemu se ključne reči u pravilima navode između apostrofa. ANTLR dozvoljava jednostavne definicije pravila u kojima figuriše promenljiv broj drugih pravila, pri čemu se koriste simboli kao u regularnim izrazima [6], što je iskorišćeno za definiciju pravila bloka naredbi.

[5] Drugim rečima, jedan program u pseudo-jeziku je jedinica prevođenja, pa je zato pravilo nazvano *unit*.

[6] U regularnim izrazima, simbol `a?` označava opciono pojavljivanje simbola `a`, simbol `a+` označava jedno ili više pojavljivanja simbola `a`, a simbol `a*` označava proizvoljan broj pojavljivanja simbola `a` - kombinacija simbola `?` i `+`.

`NAME` predstavlja ime programa, što je zapravo identifikator. Identifikatore ćemo definisati kasnije, za sada možemo posmatrati identifikator kao nisku karaktera s tim što će postojati restrikcije vezane za to koji karakteri se mogu naći unutar identifikatora ali o tome će biti reči kasnije.

```
1      unit
2          : 'algorithm' NAME block EOF
3          ;
4
5      block
6          : 'begin' statement+ 'end'
7          | statement
8          ;
```

Slika 2.12: Definicija jedinice prevođenja i bloka naredbi za pseudo-jezik.

Sledeći korak je definisanje naredbe pseudo-jezika. Slično kao i u drugim programskim jezicima, potrebno je podržati koncept deklaracije promenljive, dodele vrednosti izraza promenljivoj, naredbe kontrole toka - grananje i petlje. Na slici 2.13 je definisano šta se sve smatra jednom naredbom. Naredbe mogu biti i prazne, što je označeno ključnom rečju `pass`. Iz definicije naredbe sa slike se jasno vidi šta sve može biti naredba (prateći redosled alternativa pravila):

- deklaracija
- dodela
- poziv funkcije (označen kao `cexp`, skraćeno od *function call expression*) [7]
- vraćanje vrednosti izraza (ključna vrednost `return`) iz funkcije
- prekidanje izvršavanja davanjem poruke o grešci
- naredba grananja
- *while* petlja

[7]Funkcije mogu vratiti vrednosti pa se stoga njihovi pozivi mogu naći u izrazima - dakle poziv funkcije je validan izraz (stoga `expression` u imenu `function call expression`). Naravno, ta vrednost se može ignorisati ili pak sama funkcija može biti takva da nema povratnu vrednost već je samo neophodno izvršiti je zbog sporednih efekata.

- *repeat-until* petlja
- inkrementiranje/dekrementiranje vrednosti promenljive

```
1     statement
2         : 'pass'
3         | declaration
4         | assignment
5         | cexp
6         | 'return' exp
7         | 'error' STRING
8         | 'if' exp 'then' block ('else' block)?
9         | 'while' exp 'do' block
10        | 'repeat' block 'until' exp
11        | ('increment' | 'decrement') var
12        ;
```

Slika 2.13: Definicija naredbe za pseudo-jezik.

Deklaracija, prikazana na slici 2.14, uvodi pojavljivanje simbola datog preko identifikatora `NAME` kao oznaku za promenljivu, funkciju ili proceduru - funkciju bez povratne vrednosti. Svaka promenljiva mora biti određenog tipa, što se postiže pravilom `type`. Promenljivoj se, opciono, može pridružiti početna vrednost, drugim rečima promenljiva se može *inicijalizovati* tako da joj se pridruži vrednost nekog izraza. Procedure i funkcije imaju opcione parametre, vrednosti izraza koje im se prosleđuju kasnije u pozivu kao argumenti. Lista parametara, takođe prikazana na slici 2.14, se navodi kao lista proizvoljno mnogo parova `NAME : type`, što se vidi iz definicije pravila `parlist`.

Identifikatori su niske karaktera koje predstavljaju ime koje odgovara određenoj memorijskoj adresi. Identifikatori se koriste umesto sirovih vrednosti adresa kako bi kod bio čitljiviji i lakši za pisanje - na nivou asemblera se većinom koriste adrese ili automatski generisane oznake. Na slici 2.15 se može videti definicija identifikatora. Identifikator se sastoji od slova, cifara i simbola `_`, s tim što ne sme početi cifrom. Ovo je konvencija koju prati dosta jezika, uključujući programski jezik C. Primetimo da je identifikator nešto što bi lekser trebalo da prepozna tokom tokenizacije. Međutim, kada definišemo gramatiku od koje će ANTLR praviti lekser i parser, možemo i tokene definisati preko gramatičih pravila dajući regularni izraz za njihovo poklapanje. Listovi stabla parsiranja su uvek tokeni, drugim

```
1     declaration
2         : 'declare' type NAME ('=' exp)?
3         | 'procedure' NAME '(' parlist? ')' block
4         | 'function' NAME '(' parlist? ')' 'returning' type
              block
5         ;
6
7     parlist
8         : NAME ':' type (',' NAME ':' type)*
9         ;
```

Slika 2.14: Definicija deklaracije za pseudo-jezik.

rečima se nazivaju i *terminalni simboli*. Tokeni se, naravno, mogu naći bilo gde u stablu parsiranja.

```
1     NAME
2         : [a-zA-Z_][a-zA-Z_0-9]*
3         ;
```

Slika 2.15: Definicija identifikatora za pseudo-jezik.

Pošto želimo da pseudo-jezik bude strogo tipiziran, potreban je koncept tipa (što smo videli u deklaracijama), čija je definicija data na slici 2.16. Tip može biti *primitivan* (drugim rečima *prost*) ili *složen*. Primitivni tipovi su podržani u samoj sintaksi jezika - u našem slučaju brojevi i niske. Brojevi mogu biti celi ili realni. U složene tipove spadaju korisnički definisani tipovi (sa imenom `NAME`, u četvrtoj alternativi pravila `typename` sa slike 2.16) i kolekcije. Od kolekcija su podržani nizovi, liste i skupovi. Prilikom definicije kolekcije mora se navesti tip elemenata kolekcije i taj tip mora biti uniforman - isti za sve elemente kolekcije.

Izrazi, iako se definišu rekurzivno, se mogu posmatrati kao kombinacija promenljivih, operatora i poziva funkcija sa odlikom da se mogu *evaluirati*, tj. moguće je izračunati njegovu vrednost. Iz definicije pravila `exp` na slici 2.17, mogu se uočiti tipovi izraza, pri čemu nije vođeno računa o matematičkom prioritetu operatora, radi jednostavnosti. Izraz može biti *literal*, koji predstavlja konstantu, bilo brojevnu, logičku ili nisku karaktera. Promenljive, definisane pravilom `var` su takođe izrazi, jer se trenutna vrednost promenljive posmatra kao vrednosti izraza. Pri-

```
1       type
2           : typename 'array'?
3           | typename 'list'?
4           | typename 'set'?
5           ;
6
7       typename
8           : 'integer'
9           | 'real'
10          | 'string'
11          | NAME
12          ;
```

Slika 2.16: Definicija tipa podataka za pseudo-jezik.

metimo da promenljiva može biti kolekcijskog tipa, u kom slučaju se navodi redni broj elementa nakon identifikatora promenljive - taj redni broj može biti rezultat evaluacije drugog izraza, ali ne bilo kakvog, stoga se u pravilu `iexp` definiše šta sve može biti korišćeno da se indeksira element kolekcije. Izrazima se može dati prioritet pomoću zagrada, što se vidi u trećoj alternativi pravila `exp`. U naredne tri alternative su opisani tipovi izraza: aritmetički, relacioni i logički. Aritmetički izrazi su vezani aritmetičkim operatorima definisanim preko pravila `aop`, slično važi i za ostala dva tipa. Svi tipovi izraza navedeni iznad su binarni, što znači da operatori zahtevaju dva argumenta. Postoje i unarni izrazi, od kojih su podržane promena znaka i logička negacija, što se vidi iz pravila `uop`.

Definicija literala je prikazana na slici 2.18. Literali mogu bili istinitosne konstante `True` i `False`, brojevne konstante ili niske karaktera. Brojevne konstante mogu bili celobrojni ili realni dekadni brojevi. Realne konstatne je moguće definisati u fiksnom ili pokretnom zarezu. Niske se mogu definisati između navodnika ili apostrofa. Pritom, kao i u modernim programskim jezicima, moguće je navesti sekvence koje predstavljaju specijalne karaktere kao što su novi red, tabulator itd. Oznaka `fragment` označava optimizaciju, naime nije potrebno da postoji na primer pravilo `Digit`, već samo dajemo simbol za regularni izraz koji će se koristiti u više drugih pravila i poklapati jednu dekadnu cifru.

Poslednje što treba definisati je sve ono što lekser treba da preskoči tokom prolaska kroz izvorni kod programa. To su beline (nevidljivi karakteri kao što su razmaci, tabulatori i novi redovi) i komentari. Definicije ovih pravila se mogu

```
1       exp
2           : literal
3           | var
4           | '(' exp ')'
5           | exp aop exp
6           | exp rop exp
7           | exp lop exp
8           | uop exp
9           | cexp
10          ;
11
12      var
13          : NAME ('[' iexp ']')?
14          ;
15
16      iexp
17          : literal
18          | var
19          | aexp
20          ;
21
22      cexp
23          : 'call' NAME '(' explist? ')'
24          ;
25
26      explist
27          : exp (',' exp)*
28          ;
29
30      aop : '+' | '-' | '*' | '/' | 'div' | 'mod' ;
31      rop : '>' | '>=' | '<' | '<=' | '==' | '=/=' ;
32      lop : 'and' | 'or' ;
33      uop : '-' | 'not' ;
```

Slika 2.17: Definicija izraza za pseudo-jezik.

videti na slici 2.19. Vidimo da se u njima koristi posebna oznaka `-> skip`, koja predstavlja instrukcije lekseru da preskoči sve ono što ovo pravilo poklopi. Komentari su u stilu kao u programskom jeziku C (ali naravno, isti stil se koristi i u mnogim jezicima) i mogu biti jednolinijski ili višelinijski. Beline koje treba preskočiti su definisane u pravilu `WS`, skraćeno od *whitespace*, što u prevodu sa engleskog znači *beli prostor, belina.*

```
literal : 'True' | 'False' | INT | FLOAT | STRING ;

STRING
    : '"' ( EscapeSequence | ~('\\'|'"') )* '"'
    ;

fragment
EscapeSequence
    : '\\' [abfnrtvz"'\\]
    | '\\' '\r'? '\n'
    ;

INT
    : Digit+
    ;

FLOAT
    : Digit+ '.' Digit* ExponentPart?
    | '.' Digit+ ExponentPart?
    | Digit+ ExponentPart
    ;

fragment
ExponentPart
    : [eE] [+-]? Digit+
    ;

fragment
Digit
    : [0-9]
    ;
```

Slika 2.18: Definicija konstanti za pseudo-jezik.

Ovako definisanu gramatiku možemo sačuvati u fajl sa imenom `Pseudo.g4`, potrebno je samo navesti ime gramatike na početku fajla, kao na slici 2.20. Naredni korak je kreiranje leksera i parsera koristeći ANTLR4, predpostavljajući da je instaliran na način opisan u 2.3. Pokretanjem ANTLR-a generišemo lekser i parser za gramatiku pseudo-jezika:

```
$ antlr4 Pseudo.g4
```

```
1       BlockComment
2           :   '/*' .*? '*/'  -> skip
3           ;
4       LineComment
5           :   '//' ~[\r\n]*  -> skip
6           ;
7       WS
8           : [ \t\u000C\r\n]+ -> skip
9           ;
```

Slika 2.19: Definicija komentara i belina za pseudo-jezik.

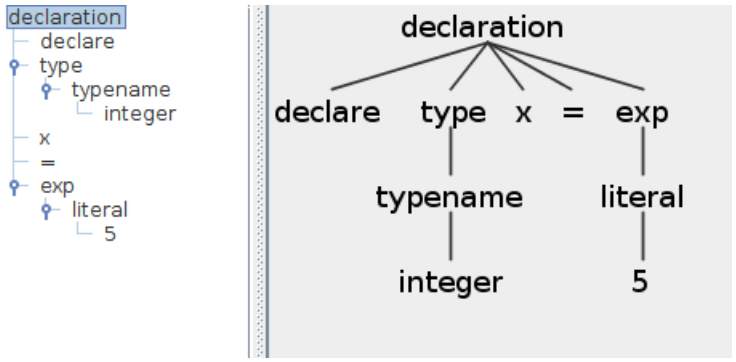
```
1       grammar Pseudo;
```

Slika 2.20: Definicija imena gramatike za pseudo-jezik.

ANTLR4 će generisati lekser i parser podrazumevano napisane u programskom jeziku Java. Ukoliko želimo to da promenimo, možemo koristiti opciju `-Dlanguage=...`. Kako bismo testirali generisani lekser i parser, možemo koristiti ANTLR TestRig da vizualno prikažemo stablo parsiranja, s tim što moramo prvo kompajlirati generisane Java kodove. TestRig pozivamo navođenjem ime gramatike (koje se poklapa sa imenom leksera i parsera) i imenom pravila od koga će parser krenuti. Opcija `-gui` pokreće vizualni prikaz stabla parsiranja pokazan na slici 2.21 (vizualni prikaz je moguće preskočiti i samo ispisati stablo u LISP formi koristeći opciju `-tree`), mada je moguće i ispisati samo tokene koristeći opciju `-tokens`. Ulaz se prosleđuje programu dok se ne naiđe na simbol `EOF`, ili alternativno se može preneti ulaz putem UNIX pipeline-a (na slici 2.21 se može videti izlaz koji se dobija korišćenjem opcije `-gui`):

```
1 $ javac *.java
2 $ echo "declare integer x = 5" | grun Pseudo declaration
     -tokens
3 [@0,0:6='declare',<'declare'>,1:0]
4 [@1,8:14='integer',<'integer'>,1:8]
5 [@2,16:16='x',<NAME>,1:16]
6 [@3,18:18='=',<'='>,1:18]
```

```
[@4,20:20='5',<INT>,1:20]
[@5,22:21='<EOF>',<EOF>,2:0]
$ echo "declare integer x = 5" | grun Pseudo declaration
    -tree
(declaration declare (type (typename integer)) x = (exp
    (literal 5)))
$ echo "declare integer x = 5" | grun Pseudo declaration -gui
```



Slika 2.21: Grafički prikaz stabla parsiranja koje generiše parser kreiran od strane alata ANTLR4 TestRig za kod pisan u pseudo-jeziku.

Obilazak stabla parsiranja

ANTLR, osim leksera i parsera za datu gramatiku, može da kreira interfejse i bazne klase koji prate obrasce za projektovanje *posetilac* (engl. *visitor*) i osluškivač (engl. *listener*, varijanta obrasca *posmatrač* engl. *observer*) opisane u 2.2. Tako kreirani interfejsi i klase imaju metodi za obilazak stabla parsiranja. ANTLR podrazumevano generiše interfejs osluškivača (slika 2.22) kao i baznu klasu koja implementira generisani interfejs tako što su sve implementirane metodi prazne. Stoga, ukoliko korisnik želi da definiše operaciju samo u slučaju da se prilikom obilaska stabla parsiranja naiđe na samo jedan tip čvora, nije potrebno implementirati ceo interfejs osluškivača, već je moguće naslediti baznu klasu i predefinisati samo jedan metod. ANTLR može da generiše i posetilac (slika 2.23), ukoliko se navede odgovarajuća opcija `-visitor` prilikom pokretanja. Slično, ukoliko nije potrebno generisati osluškivač, može se koristiti opcija `-no-listener` kako se ne bi generisao osluškivač.

```
1  public interface IPseudoListener : IParseTreeListener
2  {
3      void EnterUnit([NotNull] PseudoParser.UnitContext
           context);
4      void ExitUnit([NotNull] PseudoParser.UnitContext
           context);
5      void EnterBlock([NotNull] PseudoParser.BlockContext
           context);
6      void ExitBlock([NotNull] PseudoParser.BlockContext
           context);
7      void EnterStatement([NotNull]
           PseudoParser.StatementContext context);
8      void ExitStatement([NotNull]
           PseudoParser.StatementContext context);
9
10     ...
11 }
```

Slika 2.22: Delimični prikaz interfejsa osluškivača generisanog od strane ANTLR4 za pseudo-jezik definisan u prethodnom odeljku (C#).

Sa slike 2.22 se vidi da je moguće definisati metodi koje će se pozivati prilikom ulaska ali i prilikom izlaska iz čvora određenog tipa prilikom obilaska stabla parsiranja. Pritom, važno je kako se stablo obilazi. U slučaju ANTLR, to je pretraga u dubinu (engl. *depth-first search, DFS*) [8], stoga će se metod `Exit` za proizvoljni čvor pozvati tek kad se obiđu sva deca tog čvora - dakle nakon poziva njihovih `Enter` i `Exit` metoda. Pošto se DFS obično implementira putem LIFO [9] strukture, može se reći da se `Enter` metod poziva onog trenutka kad se čvor ubaci u strukturu, a `Exit` metod onda kada se čvor ukloni iz strukture.

Za razliku od osluškivača, posetilac je prirodnije koristiti ukoliko je potrebno izvršiti neko izračunavanje nad strukturom koja se obilazi. Interfejs posetioca (slika 2.23) je šablonski, i metodi imaju povratnu vrednost šablonskog tipa za razliku od metoda osluškivača i, u odnosu na osluškivač, nema para metoda za

[8] DFS je obilazak stabla takav da se obilazak duž grane stabla nastavlja sve dok je moguće ići dublje, a ako to nije moguće vratiti se unazad i obići druge grane.

[9] *Last In, First Out* struktura podataka je apstraktna struktura podataka sa operacijama ubacivanja i izbacivanja elemenata, pri čemu je element koji se izbacuje onaj koji je poslednji ubačen. Primer LIFO strukture je držač za CD-ove - ne mogu se ukloniti CD-ovi ispod CD-a na vrhu (poslednji ubačen) a da se ne ukloni isti. U slučaju opisanom iznad, implementacija LIFO strukture se naziva stek *stack*.

```
1 public interface IPseudoVisitor<Result> :
     IParseTreeVisitor<Result>
2 {
3      Result VisitUnit([NotNull] PseudoParser.UnitContext
          context);
4      Result VisitBlock([NotNull] PseudoParser.BlockContext
          context);
5      Result VisitStatement([NotNull]
          PseudoParser.StatementContext context);
6      Result VisitDeclaration([NotNull]
          PseudoParser.DeclarationContext context);
7
8      ...
9 }
```

Slika 2.23: Delimični prikaz interfejsa posetioca generisanog od strane ANTLR4 za pseudo-jezik definisan u prethodnom odeljku (C#).

svaki čvor već samo jedan metod. Dodatna razlika, ali i najveća, je ta što se metodi posetioca ne pozivaju automatski. Stoga je na programeru da nastavi obilazak i da odluči u koje čvorove želi da se spusti. Jasno je da i osluškivač i posetilac imaju svoje primene - ukoliko je potrebno obići stablo parsiranja i dovući neke informacije može se iskoristiti osluškivač jer onda ne moramo brinuti o obilasku. S druge strane, ukoliko je potrebno izračunati neku vrednost prirodno je iskoristiti rekurziju i iskoristiti posetilac - rekurzivni pozivi prilikom obilaska nam idu u prilog jer koristimo povratne vrednosti tih metoda da gradimo rezultat od listova ka korenu stabla parsiranja. U nastavku će se koristiti posetilac zbog kontrole obilaska ali i činjenice da se stablo parsiranja obilazi sa ciljem da se izgradi AST, koji je takođe rekurzivna struktura i gradi se inkrementalno kroz rekurziju.

Bilo da se koristi osluškivač ili posetilac, potrebno je nekako proslediti informacije o samom čvoru na koji se naišlo tokom obilaska stabla parsiranja. Te informacije se metodima osluškivača i posetioca prosleđuju putem potklasa apstraktne klase konteksta pravila `ParserRuleContext` - u primeru iznad `UnitContext`, `BlockContext` itd. Svaki kontekst pravila po imenu odgovara pravilima definisanim u gramatici i sadrži informacije bitne za trenutni čvor u stablu parsiranja koji odgovara tipu konteksta. Takođe, u svakom kontekstu su prisutne i metodi čija imena odgovaraju pravilima koja se javljaju u definiciji samog pravila koje odgo-

vara kontekstu. Tako da, za `BlockContext`, imajući u vidu definiciju sa slike 2.12, pošto se u definiciji osim tokena koristi i pravilo `statement`, u okviru `BlockContext` klase biće implementiran i metod `statement()` koji vraća kontekst pravila u ovom slučaju tipa `StatementContext[]` jer u prvoj alternativi stoji `statement+` - dakle možemo imati više `statement` poklapanja. Sa ovim u vidu, moguće je odrediti kako će se obilazak nastaviti (u slučaju posetioca) ili dovući informacije o delovima definicije pravila. Ukoliko pravilo ima više alternativa, metodi koje vraćaju kontekst pravila koje figuriše u alternativi koja nije korišćena za poklapanje pravila će vratiti `null`. Pošto se `statement()` pravilo javlja u obe alternative pravila `block` (i nije opciono), možemo biti sigurni da povratna vrednost `statement()` metoda nikada neće biti `null`.

U poglavlju 3 će se koristiti posetilac za obilazak stabla parsiranja i kreiranje AST apstrakcije od istog. Pritom, koristiće se implementacija posetioca u programskom jeziku C#.

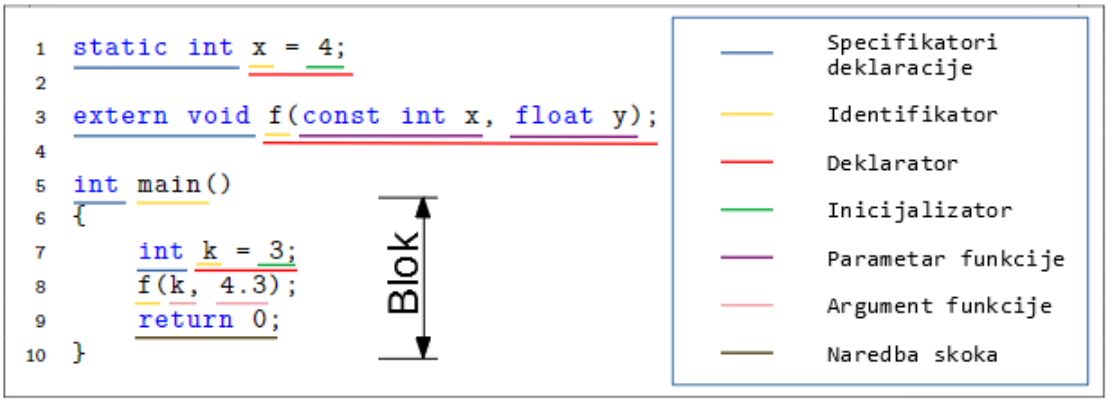
Glava 3

Opis zajedničke AST apstrakcije

Kako bi se kreirala smislena apstrakcija stabla parsiranja, potrebno je identifikovati bitnih informacije u stablu parsiranja ali i koncepte same gramatike koji su od značaja. Najjednostavnije rešenje je mimikovati čvorove stabla parsiranja, ukoliko su gramatička pravila kreirana tako da oslikaju koncepte jezika koji gramatika definiše. Na primer, ukoliko u gramatici imamo pravilo `deklaracija` sa alternativama `deklaracijaPromenljive` i `deklaracijaFunkcije`, možemo kreirati apstraktnu klasu `Deklaracija` sa konkretizacijama `DeklaracijaPromenljive` i `DeklaracijaFunkcije`. Kako se definišu deklaracije promenljivih i funkcija zavisi dalje od definicija pravila `deklaracijaPromenljive` i `deklaracijaFunkcije`. Naravno, nije uvek moguće primeniti ovakav postupak. Takođe, nekada u gramatici definišemo pomoćna pravila kako bismo se izborili sa rekurzijom ili izbegli neke tipove rekurzije - ta pravila ne bi trebalo da imaju odgovarajuće tipove u apstrakciji. Dakle, identifikovanje toga šta čini gramatiku nije jednostavno u opštem slučaju. Međutim, pošto su u pitanju gramatike programskih jezika, onda je jasno da dosta različitih gramatika dele slične koncepte i da je moguće definisati tipove čvorova koji odgovaraju tim konceptima. Neki od njih mogu biti: naredba, izraz, deklaracija, poziv funkcije, dodela... Jasno, postoji i hijerarhija u navedenom - recimo poziv funkcije se može smatrati kao samostalna naredba ali može biti i deo izraza. Dakle, treba biti jako pažljiv u definisanju hijerarhije tako da ne dozvoli nešto što u opštem slučaju ne bi trebalo da bude dozvoljeno (npr. ako je dozvoljeno višestruko nasleđivanje i poziv funkcije je i naredba ali i izraz, onda se izrazi u kojima figurišu pozivi funkcija sastoje od više naredbi, što nema smisla.).

Bez obzira na to koje koncepte izdvojimo iz stabla parsiranja kao relevantne, potrebno ih je sve unifikovati. Slično kao što svi čvorovi stabla parsiranja moraju

biti derivati određene klase (pošto svi elementi stabla moraju biti istog tipa), isto treba da važi i za apstraktno sintaksno stablo. Stoga, postojaće apstraktna bazna klasa sa informacijama koje svaki čvor AST-a poseduje. Ova bazna klasa će u nastavku rada biti referisana kao `ASTNode`. Pošto je AST stablo u kome svaki čvor može imati proizvoljno mnogo potomaka, očekivano je da svaki čvor ima pokazivače na potomke gde svaki pokazivač pokazuje na objekat tipa `ASTNode` (jasno, ne možemo napraviti objekat klase `ASTNode` jer je klasa apstraktna ali konkretizacije klase `ASTNode` su takođe tipa `ASTNode`). Radi jednostavnosti implementacije nekih operacija biće dodatno dostupna informacija o roditelju svakog čvora putem pokazivača na roditelja, tipa `ASTNode` - koreni čvor neće imati postavljenu vrednost ovog polja. Dodatne informacije recimo mogu biti linija u izvornom fajlu gde se nalazi kod koji odgovara čvoru. Dodatno, možemo definisati i metode koje treba svaki čvor da poseduje - recimo metode za serijalizaciju [1].

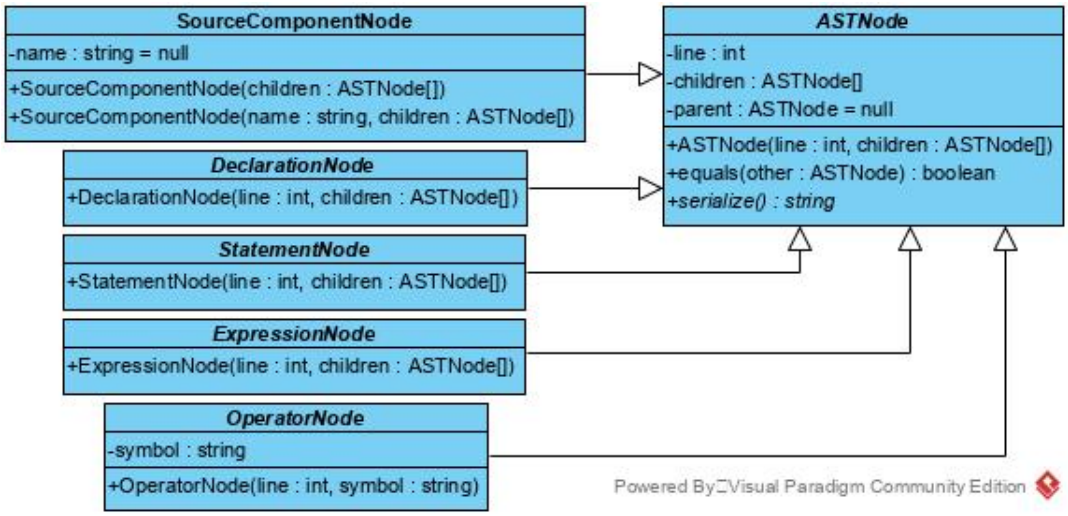


Slika 3.1: Vizualni prikaz delova C koda na osnovu pravila iz C11 gramatike.

Krenućemo od gramatika strogo tipiziranih proceduralnih programskih jezika, a zatim ćemo koncepte proširiti i na skript jezie. U strogo tipiziranim proceduralnim jezicima se mogu identifikovati koncepti koji se pojavljuju u većini od njih. Na primeru C koda sa slike 3.1 moguće je videti neke od njih. Koncepti naredbe i izraza su očigledni i nisu prikazani. Nije očigledno, doduše, kako bi se deklaracija promenljive dekomponovala na delove. Po C11 gramatici, svaka deklaracija se sastoji od *specifikatora deklaracije* i od jednog ili više *deklaratora*. Specifikatori deklaracije su ključne reči koje bliže određuju deklaraciju - modifikatori vidljivosti ili specifikatori pristupa. Deklaratori uvode simbol (*identifikator*) u kontekst kao

[1] Serijalizacija predstavlja proces konvertovanja stanja objekta u niz bajtova da bi se isti sačuvao u bazu podataka, fajl, ili pak prosledio nekom drugom programu.

promenljivu, niz ili funkciju. Deklaratori, u zavisnosti od toga šta se deklariše, mogu imati razne delove. Ukoliko je u pitanju deklarator promenljive, sastoje se i od opcionog izraza za inicijalizaciju, u daljem tekstu *inicijalizator*. Ukoliko je u pitanju deklarator funkcije, nakon identifikatora se nalazi lista parametara funkcije - svaki od njih se sastoji od specifikatora deklaracije i deklaratora.



Slika 3.2: UML klasni dijagram direktnih derivata klase `ASTNode`.

Ovakvi koncepti će biti inspiracija za definisanje AST čvorova, pri čemu će detaljno biti opisana motivacija i razlozi za odabir nekih specifičnosti. Na slici 3.2 se mogu videti osnovni apstraktni derivate klase `ASTNode`. Jedini direktna konkretizacija klase `ASTNode` koja nije apstraktna je klasa `SourceComponentNode`. Svrha ove klase je da označi granice izvornih fajlova u slučaju da se pravi AST od više različitih izvornih fajlova - stoga će koren svakog AST-a kreiranog od izvornog fajla biti tipa `SourceComponentNode` [2]. U nastavku će biti detaljno opisane apstraktne konkretizacije klase `ASTNode` sa slike 3.2 zajedno sa svojim derivatima.

[2]Navedeno važi pod pretpostavkom da se stablo parsiranja čitavog izvornog fajla prevodi u AST. Biće dozvoljeno i prevođenje delova izvornog fajla, u kom slučaju koren AST-a neće biti tipa `SourceComponentNode`.

Glava 4

Zaključak

Fijuče vetar u šiblju, ledi pasaže i kuće iza njih i gunđa u odžacima. Nidžo, čežnjivo gledaš fotelju, a Đura i Mika hoće poziciju sebi. Ljudi, jazavac Džef trči po šumi glođući neko suho žbunje. Ljubavi, Olga, hajde pođi u Fudži i čut ćeš nježnu muziku srca. Boja vaše haljine, gospođice Džafić, traži da za nju kulučim. Hadži Đera je zaćutao i bacio čežnjiv pogled na šolju s kafom. Džabe se zec po Homolju šunja, čuvar Jožef lako će i tu da ga nađe. Odžačar Filip šalje osmehe tuđoj ženi, a njegova kuća bez dece. Butić Đuro iz Foče ima pun džak ideja o slaganju vaših željica. Džajić odskoči u aut i izbeže đon halfa Pecelja i njegov šamar. Plamte odžaci fabrika a čađave guje se iz njih dižu i šalju noć. Ajšo, lepoto i čežnjo, za ljubav srca moga, dođi u Hadžiće na kafu. Huči šuma, a iza žutog džbuna i panja đak u cveću delje seji frulu. Goci i Jaćimu iz Banje Koviljače, flaša džina i žeđ padahu u istu uru. Džaba što Feđa čupa za kosu Milju, ona juri Živu, ali njega hoće i Daca. Dok je Fehim u džipu žurno ljubio Zagu Čađević, Cile se ušunjao u auto. Fijuče košava nad odžacima a Ilja u gunju žureći uđe u suhu i toplu izbu. Bože, džentlmeni osećaju fizičko gađenje od prljavih šoljica! Dočepaće njega jaka šefica, vođena ljutom srdžbom zlih žena. Pazi, gedžo, brže odnesi šefu taj đavolji ček: njim plaća ceh. Fine džukce ozleđuje bič: odgoj ih pažnjom, strpljivošću. Zamišljao bi kafedžiju vlažnih prstića, crnjeg od čađi. Đače, uštedu plaćaj žaljenjem zbog džinovskih cifara. Džikljaće žalfija između tog busenja i peščanih dvoraca. Zašto gđa Hadžić leči živce: njena ljubav pred fijaskom? Jež hoće peckanjem da vređa ljubičastog džina iz flaše. Džej, ljubičast zec, laže: gađaće odmah pokvašen fenjer. Plašljiv zec hoće jeftinu dinju: grožđe iskamči džabe. Džak je pun žica: ćućeš tad svađu zbog lomljenja harfe. Čuj, džukac Flop bez daha s gađenjem žvaće stršljena. Oh, zadnji šraf na džipu slab:

muž gđe Cvijić ljut koči. Šef džabe zvižduće: mlađi hrt jače kljuca njenog psa. Odbaciće kavgadžija plaštom čađ u željezni fenjer. Deblji krojač: zgužvah smeđ filc u tanjušni džepić. Džo, zgužvaćeš tiho smeđ filc najdeblje krpenjače. Štef, bacih slomljen dečji zvrk u džep gđe Žunjić. Debljoj zgužvah smeđ filc — njen škrt džepčić.

Fijuče vetar u šiblju, ledi pasaže i kuće iza njih i gunđa u odžacima. Nidžo, čežnjivo gledaš fotelju, a Đura i Mika hoće poziciju sebi. Ljudi, jazavac Džef trči po šumi glođući neko suho žbunje. Ljubavi, Olga, hajde pođi u Fudži i čut ćeš nježnu muziku srca. Boja vaše haljine, gospođice Džafić, traži da za nju kulučim. Hadži Đera je zaćutao i bacio čežnjiv pogled na šolju s kafom. Džabe se zec po Homolju šunja, čuvar Jožef lako će i tu da ga nađe. Odžačar Filip šalje osmehe tuđoj ženi, a njegova kuća bez dece. Butić Đuro iz Foče ima pun džak ideja o slaganju vaših željica. Džajić odskoči u aut i izbeže đon halfa Pecelja i njegov šamar. Plamte odžaci fabrika a čađave guje se iz njih dižu i šalju noć. Ajšo, lepoto i čežnjo, za ljubav srca moga, dođi u Hadžiće na kafu. Huči šuma, a iza žutog džbuna i panja đak u cveću delje seji frulu. Goci i Jaćimu iz Banje Koviljače, flaša džina i žeđ padahu u istu uru. Džaba što Feđa čupa za kosu Milju, ona juri Živu, ali njega hoće i Daca. Dok je Fehim u džipu žurno ljubio Zagu Čađević, Cile se ušunjao u auto. Fijuče košava nad odžacima a Ilja u gunju žureći uđe u suhu i toplu izbu. Bože, džentlmeni osećaju fizičko gađenje od prljavih šoljica! Dočepaće njega jaka šefica, vođena ljutom srdžbom zlih žena. Pazi, gedžo, brže odnesi šefu taj đavolji ček: njim plaća ceh. Fine džukce ozleđuje bič: odgoj ih pažnjom, strpljivošću. Zamišljao bi kafedžiju vlažnih prstića, crnjeg od čađi. Đače, uštedu plaćaj žaljenjem zbog džinovskih cifara. Džikljaće žalfija između tog busenja i peščanih dvoraca. Zašto gđa Hadžić leči živce: njena ljubav pred fijaskom? Jež hoće peckanjem da vređa ljubičastog džina iz flaše. Džej, ljubičast zec, laže: gađaće odmah pokvašen fenjer. Plašljiv zec hoće jeftinu dinju: grožđe iskamči džabe. Džak je pun žica: ćućeš tad svađu zbog lomljenja harfe. Čuj, džukac Flop bez daha s gađenjem žvaće stršljena. Oh, zadnji šraf na džipu slab: muž gđe Cvijić ljut koči. Šef džabe zvižduće: mlađi hrt jače kljuca njenog psa. Odbaciće kavgadžija plaštom čađ u željezni fenjer. Deblji krojač: zgužvah smeđ filc u tanjušni džepić. Džo, zgužvaćeš tiho smeđ filc najdeblje krpenjače. Štef, bacih slomljen dečji zvrk u džep gđe Žunjić. Debljoj zgužvah smeđ filc — njen škrt džepčić.

Literatura

[1] ANTLR4. https://www.antlr.org/.

[2] BYACC. http://byaccj.sourceforge.net/.

[3] Clang. https://clang.llvm.org/.

[4] Context-Free Grammars. https://www.cs.rochester.edu/~nelson/courses/csc_173/grammars/cfg.html.

[5] Keith Cooper and Linda Torczon. *Engineering a Compiler, 2nd Edition*. 2013.

[6] Joel Denny and Brian Malloy. IELR(1): practical LR(1) parser tables for non-LR(1) grammars with conflict resolution. pages 240–245. Association for Computing Machinery, New York, NY, United States.

[7] Design Patterns. http://www.vincehuston.org/dp/.

[8] Flex. https://www.gnu.org/software/flex/.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1995.

[10] GLR. https://www.gnu.org/software/bison/manual/html_node/GLR-Parsers.html.

[11] GNU Bison. https://www.gnu.org/software/bison/.

[12] GNU Project. https://www.gnu.org/gnu/thegnuproject.en.html.

[13] LALR1. https://web.cs.dal.ca/~sjackson/lalr1.html.

[14] LexYacc. http://dinosaur.compilertools.net/.

[15] LR. http://pages.cpsc.ucalgary.ca/~robin/class/411/LR.1.html.

[16] Terrence Parr. *The Definitive ANTLR 4 Reference*. 2012.

[17] Design Patterns. `https://www.uml.org/`.

[18] William M. Waite and Gerhard Goos. *Compiler Construction*. 1995.

Biografija autora

Ivan Ž. Ristović rođen je 17.01.1995. godine u Užicu. Osnovnu školu, kao i prirodno-matematički smer Užičke gimnazije, završio je kao nosilac Vukove diplome. Tokom navedenog perioda školovanja isticao se u oblastima matematike, informatike, fizike, hemije i engleskog jezika, što potvrđuje veći broj nagrada na Državnim takmičenjima.

Smer Informatika na Matematičkom fakultetu Univerziteta u Beogradu upisuje 2014. godine. Na navedenom smeru je diplomirao 2018. godine, posle tri godine studija sa prosečnom ocenom 9,17. Master studije upisuje na istom fakultetu odmah nakon diplomiranja.

U avgustu 2018. biva izabran u zvanje „Saradnik u nastavi“ na Matematičkom fakultetu paralelno sa master studijama. Drži vežbe iz kurseva „Računarske mreže”, „Funkcionalno programiranje”, „Programske paradigme” i „Objektno orijentisano programiranje” na kasnijim godinama osnovnih studija.

Oblasti interesovanja uključuju pre svega razvoj i verifikaciju softvera, mikroservise i računarske mreže.