Univerza v Mariboru

**Fakulteta za elektrotehniko,
računalništvo in informatiko**

Implementacija dinamičnih konceptov čistega statičnega objektno usmerjenega jezika

MAGISTRSKO DELO

Maribor, junij 2004 Sašo Greiner

Avtor: Sašo Greiner

Naslov: Implementacija dinamičnih konceptov čistega statičnega objektno usmerjenega jezika

UDK: [004.43+004.7]:007

Ključne besede: programski jeziki, objektna usmerjenost, prevajalniki, virtualni stroj, računalniška arhitektura

ZAHVALA

Zahvaljujem se vsem in vsakemu posamezniku, ki je na kakršenkoli način nepobitno prispeval k izvedbi tega dela.

Naslov: Implementacija dinamičnih konceptov čistega statičnega objektno usmerjenega jezika
UDK: [004.43+004.7]:007

Ključne besede: programski jeziki, objektna usmerjenost, prevajalniki, virtualni stroj, računalniška arhitektura

POVZETEK

V delu predstavljamo načrtovanje in implementacijo čistega, objektno usmerjenega programskega jezika. Razvoj jezika je prikazan z vidika teoretičnih dognanj in praktično uporabnih konceptov. Osnovno vodilo jezika Z_0 je čistost objektnega modela, učinkovit sistem tipov in nekateri dinamični mehanizmi, značilni za dinamično tipizirane, interpretirane jezike. Jezik Z_0 omogoča preprost semantični model večkratnega dedovanja na podlagi metod. Velik del k čistosti objektnega modela prispeva promocija instančnih spremenljivk v metode.

Implementiran je varen mehanizem dinamičnega spreminjanja metod, ki omogoča bistveno večjo prilagodljivost in neprimerno več dinamike, kot jo imajo klasični statično tipizirani programski jeziki. Ker je jezik Z_0 preveden, je bila načrtovana in implementirana tudi virtualna računalniška arhitektura, ki omogoča izvajanje prevedenega razrednega koda.

Title: Implementation of dynamic concepts in a statically typed object-oriented programming language
UDK: [004.43+004.7]:007

Keywords: programming languages, object-orientation, compilers, virtual machine, computer architecture

ABSTRACT

We present the design and implementation of a pure object-oriented programming language. The development process is presented both from theoretical viewpoint and concepts with practical values. The main idea of language Z_0 is purity of object model, efficient type system, and some dynamic mechanisms commonly found in dynamically typed interpreted languages. The language Z_0 enables a very simple and strict semantic model for multiple inheritance that is based entirely on methods. The purity of object model is attributable mainly to the promotion of instance variables into methods.
A dynamic method update mechanism has been implemented which provides a high degree of adaptability and substantially more dynamics than the majority of statically-typed programming languages. Because Z_0 is a compiled language a virtual architecture for execution of Z_0 classes has been designed and implemented.

Kazalo

Slike

Uporabljeni simboli in kratice

ϵ	Prazna beseda.
G	Gramatika.
V_N	Množica neterminalov.
V_T	Množica terminalov.
Φ	Množica produkcij.
$\Rightarrow^+$	Tranzitivno zaprtje.
$\mathcal{L}(G)$	Jezik gramatike G.
$\circ$	Konkatenacija.
$<:$	Relacija podtipa.
$\equiv$	Ekvivalenca.
$\mathcal{C}^p$	Parametriziran razred.
Π	Okolje.

1 Predgovor

Kadar o komunikaciji in njeni osnovni ideji razmišljamo načrtno ali posredno preko nekih dejavnikov, se običajno osredotočimo na komuniciranje, katerega glavni in edini akterji so ljudje. Razmišljamo povprečno in govorimo o naravni komunikaciji, tj. o komunikaciji, ki jo s pomočjo ustreznih mehanizmov izvaja človek z namenom obvestiti drugega. Bistven aspekt naravne komunikacije je naravni jezik, ki se ga je potrebno učiti. Naravni jezik – jezik človekove komunikacije – je s svojimi lingvističnimi in semantičnimi pomembnostmi, z izjemno živopisno sintaktično in pomensko strukturo, in sploh z vso svojo obsežnostjo in raznolikostjo v izrazju, edinstveni in mnogokrat edini prikladni medij naravne komunikacije. Za naravne jezike načelno velja, da imajo izjemno veliko izrazno moč, saj zajemajo zelo širok spekter področij, na katerih funkcionirajo kot komunikacijsko sredstvo. Prav zaradi te vseobsegajoče uporabnosti v vsakdanjem življenju so večkrat pri določenih pomenih zelo neenoumni. Navkljub vsem pomenskim zaprekam jih ljudje razumemo z razmeroma malo vloženega truda, potem ko smo se jezika priučili. Vendar ne gre pozabiti, da smo ljudje inteligentna bitja.

Podobna in vendarle različna komunikacija obstaja med človekom in strojem. Bistvena in vplivna lastnost pri tej nenaravni oz. polnaravni komunikaciji je način, na katerega se sporočanje vrši. Pri "pogovoru" s strojem je običajno veliko težje besede in njihov pomen zajemati iz naravnega jezika. V tej komunikaciji raje uporabimo programski jezik, tj. jezik s katerim se s strojem na enostaven način sporazumevamo. V splošnem lahko povzamemo, da je tovrstna komunikacija osnovana na manj abstraktnih predstavah kot tista v naravnem jeziku. Lahko bi celo rekli, da gre za primitivno komunikacijo oz. komunikacijo nekega nižjega reda. Vendar se kljub tej obzervaciji stopnja naravnosti programskih jezikov dviguje, kar še posebej velja za jezike umetne inteligence. Cilj, smoter in zaenkrat še nekoliko idealizirana želja, je predstavitev problema računalniku v naravnem, človeškem jeziku [66], iz katerega bo računalnik sposoben razbrati korake rešitve. Značilnosti in prednosti programskega jezika so v nedvoumnosti pomena fraz, zgoščenem slovarju besed in sploh v preproščini izražanja. Za učinkovitost programskega jezika je potrebno zadostiti večim ciljem. Prvi je vsekakor

ta, da jezik zadošča nekim postavljenim specifikacijam. Drugi, bistveni cilj je, da je programski jezik možno zadovoljivo učinkovito implementirati na ciljnem sistemu, kjer se bo uporabljal (v večini primerov govorimo o računalniku). Programski jezik mora seveda tudi omogočati zapis poljubne rešljive naloge. Torej mora biti na nek način univerzalen. Ker je splošna univerzalnost prezahteven pogoj, se zadovoljimo s tem, da je jezik domensko-univerzalen, da torej omogoča zapis rešljive naloge zgolj z nekega ožjega področja. Programski jezik je orodje za razvoj in ima zelo velik pomen na reševanje izbranega problema. Prav s tega stališča mora biti zasnovan z visoko stopnjo odgovornosti in preudarnosti. Programski jezik je netoleranten, saj zahteva neobhodno enoumnost v sintaksi in semantiki programskih stavkov oz. fraz.

Med naravnimi in programskimi jeziki lahko izluščimo veliko podobnosti. Tako naravnega kot programskega jezika se je potrebno naučiti. Prav tako se oboji s časom razvijajo, postajajo bolj dovršeni v svoji univerzalnosti, evolvirajo v neke višje oblike. Sorodnosti najdemo tudi v nekaterih jezikovnih konceptih kot so sintaksa, pomen oz. semantika in pragmatika.

V tem delu bo tekla beseda o razvoju programskega jezika, od zgodnjih faz načrtovanja in konceptov pa vse do dejanske implementacije. Načrtovanje in implementacija programskega jezika sta zelo zahtevni opravili, tako v časovnem kot inženirskem smislu, saj obe fazi terjata strokovno in visoko specializirano znanje z različnih področij teoretičnega in pragmatičnega računalništva. Pomemben faktor pri razvoju programskega jezika predstavljajo tudi izkušnje od že implementiranih jezikov. Izkušnje so bistvena prednost, saj običajno nastanejo iz težav, ki smo jih imeli v preteklosti. Vendar pa tudi izkušnje niso vse. Programski jeziki s časom namreč evolvirajo, pojavljajo se novi koncepti, ideje in stališča, vključujejo se novi trendi in sodobna praksa. To so razlogi, da programski jeziki v svojem razvoju neprestano sledijo vsem tem smernicam in jim poskušajo zadostiti v kar največji meri.

Zgodovina programskih jezikov sega prav v začetek digitalnega računalništva kot ga poznamo danes. Jeziki so se razvijali od prvotnih strojnih oblik (strojno programiranje)

do zbirnih in kasneje do prvih visokonivojskih, kot je npr. Fortran. S časom je razvoj prinesel širše poglede v razvoju programske opreme in s tem funkcijske, logične, objektne in aspektne programske jezike, katere uporabljamo v dobršni meri še dandanes. Vpliv nekaterih spoznanj ni vselej prinesel večjih sprememb v obstoječih paradigmah razvoja. V nekaj primerih, kot je npr. Smalltalk, pa je vendarle pomenil pravo revolucijo tedanjega programiranja in posledično seveda radikalne spremembe v vzorcih načrtovanja. Če govorimo o vzorcu programskega jezika, lahko v splošnem rečemo, da ta vzorec prehaja od imperativnih k funkcijskim, logičnim in objektnim. Strokovnjak lahko spričo tega razvojnega cikla opazi, da je narava jezikov vedno bolj abstraktna. Stopnja abstraktnosti pa ni vezana zgolj na samo programiranje, kar je najbolj opazno v razvoju funkcijskih jezikovnih vrst, temveč tudi na sam pristop k razvoju programskih aplikacij, kar se nazorno izkazuje v uporabi objektno usmerjenih načrtovalnih orodij in jezikov.

Čeprav so objektni jeziki prisotni že lep čas, zadnji trendi v načrtovanju in razvoju programske opreme nazorno kažejo, da jim čas še zmeraj ni potekel. To dejstvo verjetno izhaja iz tega, da so koncepti in ideje objektne usmerjenosti, kot jo razumemo danes, dovolj dovršeni za modeliranje večine entitet realnega sveta. Tehnika modeliranja realnih entitet se tako lepo preslika v objektni model načrtovanja v okviru nekega programskega jezika, ki omogoča načrtovanje z objektnimi abstrakcijami [42]. Zdi se, da je trend objektnega programiranja zares zaživel šele s pojavitvijo jezika C++ in kasneje Jave. Navkljub temu, da Java, še manj pa seveda C++, nista nova, so se trendi objektnega programiranja, kot že rečeno, v polni meri pokazali razmeroma pozno. Razlog za tako nazorno in široko uporabo objektnega načrtovanja in programiranja je ta, da objektno usmerjeni jeziki v primerjavi s svojimi imperativnimi proceduralnimi predhodniki, skorajda nimajo pomanjkljivosti. Senčni učinek je zgolj hitrost izvajanja programskega koda, ki pa je pri prevajanju in polovičnem prevajanju ob sodobnih strojnih rešitvah vse manj opazen. Poleg povsem pragmatične uporabe objektnega modeliranja, je po drugi strani tudi sama teorija objektov zelo nazorno razdelana. Na tem področju obstaja namreč dobršna mera zanimivih teoretskih spoznanj in dognanj. Še posebej dobro so ta teoretska znanja vidna na področju abstraktnih podatkovnih

tipov in njihovega polimorfnega obnašanja. Navkljub strokovno priznani in dorečeni paradigmi objektnega načrtovanja v razvoju programske opreme, se mi s to vejo ne bomo posebej seznanjali. To je domena drugih področij. Obravnavali pa bomo enega osnovnih nivojev v tej paradigmi; to je programski jezik, ki služi kot osnovno implementacijsko orodje objektnega načrtovanja.

Upoštevajoč zgodovino razvoja jezikov, lahko rečemo, da je jezik po svojem bistvu množica izbranih in dodelanih konceptov, združena v smiselno celoto. Seveda si želimo, da bi naš jezik v vseh pogledih bil čimbolj učinkovit in opravljal nalogo interakcije med računalnikom in programerjem; ob napačnem razumevanju konceptov lahko jezik namreč kaj hitro postane disfunkcija v komunikaciji, namesto da bi jo lajšal. Želja pri načrtovanju jezika je karseda visoka stopnja univerzalnosti na področju, za katerega je jezik v prvi meri namenjen. V kolikor je jezik splošnonamenski, je domena uporabe ničmanj kot univerzum. Pomemben predikat jezika je vsekakor njegova izrazna moč, ki pove na kakšen način lahko neko idejo realiziramo z neko jezikovno abstrakcijo. Izrazna moč oz. stopnja izraznosti se med posameznimi jeziki velikokrat meri v obsežnosti programskega koda, potrebnega za realizacijo neke ideje. S tega gledišča lahko zatrdimo, da je npr. jezik Lisp izrazno močnejši od Jave. Seveda sta ta dva jezika namenjena povsem različnim ciljnim skupinam. Bolj upravičeno je primerjati sorodne jezike z istim vzorcem programiranja. Če potemtakem naredimo primerjavo med Javo [50] in C++, uvidimo da je slednji izrazno močnejši.

Postopek načrtovanja jezika lahko upravičeno imenujemo multikriterijska optimizacija, saj je potrebno dosledno upoštevati celo množico parametrov, ki so večkrat tudi medsebojno izključujoči. Najti zadovoljivo in povrhu optimalno pot med vsemi skrajnimi rešitvami, ki se ponujajo, pomeni kompleksno načrtovanje. Navkljub vsem težavam, potencialnim problemom, pastem in oviram, ki jih implementacija programskega jezika razkriva, smo se odločili prav za to. Namreč za načrtovanje in implementacijo splošnonamenskega, objektno usmerjenega programskega jezika. Proces načrtovanja programskega jezika nedvomno predstavlja jedrišče naših raziskovanj in je zato v delu prikazan na karseda pretanjen in subtilen način, ki omogoča lažje razumevanje široki

skupini posameznikov. Vodilo razprave tako ne bo namerjeno v diskusijo neke specifičnosti, temveč bo osredotočeno na širši nabor idej in konceptov. Za dovolj hitro sledenje celotnemu procesu razvoja jezika je namreč potrebno poglobljeno razumevanje nekaterih jezikovnih aspektov. Tukaj imamo v mislih nekatere najbolj bistvene koncepte, katerih implementacija v veliki meri vpliva na same korake načrtovanja. Temeljne koncepte, ki jih bomo podrobneje obravnavali, so splošne značilke abstraktnih podatkovnih tipov, razrednih hierarhij in objektov, dedovanje kot kompozicijski mehanizem, tipi povezovanj in polimorfne lastnosti tipov. Seveda se vsega omenjenega ne bi mogli uspešno lotiti brez temeljnih teoretskih dognanj na področju osnov razvoja programskih jezikov. S tega stališča se bomo najprej lotili osnov z opisi jezikovnih gramatik in nato logično nadaljevali s semantičnim modelom. Nato sledi, tako s praktičnega kot teoretičnega gledišča, kratka preambula v analizo objektno usmerjenega programskega jezika, katere osnovni namen bo razjasnitev nekaterih že znanih in tistih manj znanih pojmov. V tem delu se bomo oprli na že dorečeno in dobro definirano teorijo objektov, abstraktnih podatkovnih tipov in njihove predstavitve. Šele nato bomo začeli koncepte, ki smo jih že obdelali, dosledno povezovati v celoto programskega jezika; vključno z mehanizmi dedovanja in polimorfizma tako skozi operacijske kot strukturalne poglede. Ker gre za idejo novega jezika, bomo poseben pogled namenili jezikovnim konstruktom, njihovemu idiomatičnemu pomenu in vplivu na sam jezik. Konstrukti po svojem bistvu namreč služijo temeljni predstavitvi podatkovnih in kontrolnih entitet in so s tega stališča bistvenega pomena za načrtovanje jezika. Osredotočili se bomo predvsem na njihov opis, način predstavitve in ortogonalnost. Z gledišča implementacije se bomo na tem mestu dotaknili tudi nekaterih osnovnih principov funkcijskih jezikov in jih smiselno vpletli v razpravo, povezano z izvedbo konstruktov. Kakršenkoli je že obseg raziskave, vseh konceptov ni mogoče natančno zajeti. Lahko pa se usmerimo na določeno podskupino jezikovnih konceptov, v našem primeru objektno orientiranost, in na ta način poskušamo doseči smotrno celoto.

Delo je sestavljeno iz šestih poglavij. V drugem poglavju bomo predstavili teoretične osnove s področja programskih jezikov. Poudarili bomo sintaktično in semantično analizo ter nekatere teoretične zasnove razpoznavalnikov. V tretjem poglavju bomo

opisali znane in manj znane koncepte, ki se pojavljajo v sodobnih objektno usmerjenih jezikih. Četrto poglavje predstavlja jedro našega dela, saj v njem podamo zasnovo in implementacijo našega jezika. Peto poglavje služi za predstavitev virtualne arhitekture, katero smo zasnovali z namenom izvajati programe napisane v našem jeziku. Šesto poglavje sklene delo z zgoščenim povzetkom konceptov, ki so v jeziku implementirani. Podane so tudi stvari, ki morebiti v prihodnosti še bodo implementirane in tiste, za katere je zmanjkalo časa.

2 Splošen uvod v osnove načrtovanja jezikov

Temeljni namen pričujočega poglavja leži v razjasnitvi tistih najpomembnejših pojmov in konceptov, brez katerih si načrtovanja in izvedbe visoko nivojskega programskega jezika ni mogoče logično zamisliti. Z namenom kar najbolje predstaviti prve korake v načrtovalni proces jezika, se bomo najprej usmerili v teoretične osnove in nekatere bistvene definicije programskih jezikov. V okviru teoretskih osnov bomo na preprost način predočili predvsem tiste, ki so ključnega pomena za nadaljnji razvoj jezika. Formalne metode opisa jezika je najprej razdelal Noam Chomsky v 50. letih, ki je podal matematični model gramatik v povezavi z naravnimi jeziki [56]. Koncept gramatičnega pristopa je dobil pomemben praktični zagon z razvojem Algol-a, katerega sintaktična struktura je bila opisana prav z gramatiko. Koncept gramatike bomo podali kot preprost matematični sistem za definicijo jezika. V osnovi nam gramatična struktura jezika pove, če nek stavek pripada temu jeziku ali ne. Ker obstaja več vrst gramatik, glede na omejitve produkcijkih pravil, bomo gramatike klasificirali v razrede. Po definiciji osnovnih gramatičnih pojmov se bomo seznanili z opisom produkcijskih pravil v metajeziku BNF (Backus Naur Form) in sintaktičnimi drevesi izpeljav. Seveda obstajajo alternativni pristopi k opisu jezikov [46], vendar ti za nas ne bodo pomembni.

V nadaljevanju bomo načrtovanje zgodnjih faz jezika razdelili v leksikalno analizo, katere naloga je poiskati osnovne simbole jezika in sintaktično analizo, ki se ukvarja s preverjanjem pravilne strukture zaporedja vhodnih simbolov.

2.1 Jezik in gramatika

Sleherni jezik, naravni ali programski, lahko opišemo s sintakso, ustreznim semantičnim modelom in pragmatiko [104]. Natančna definicija programskega jezika vključuje množico simbolov ali abecedo, iz katere lahko gradimo pravilne programe, množico sintaktičnih pravil, s katerimi sestavljamo jezikovne konstrukte in predpisan pomen vseh sintaktično pravilnih programov. Abeceda je končna, neprazna množica simbolov:

$$V = \{a, b, c, \ldots, z, 0, \ldots, 9\}$$

Zaporedje simbolov tvorimo z združevanjem elementov iz abecedne množice. Operacijo združevanja oz. konkatenacije zapišemo $w_1 \circ w_2$, kjer sta w_1 in w_2 neka osnovna ali sestavljena elementa abecede V ali pa predstavljata že sestavljeno zaporedje elementov. Posamezen simbol ali konkatenacijo večih simbolov iz abecede V imenujemo *beseda nad abecedo* V. Besedo, ki je dolga 0 simbolov, imenujemo prazna beseda in jo označimo z ϵ. Če besedo dolžine n zapišemo kot $V \circ V \circ V \ldots \circ V = V^n$, lahko zaprtje V^+ nad V definiramo na naslednji način:

$$V^+ = V \cup V^2 \cup V^3 \cup \ldots$$

Upoštevajoč prazno besedo, lahko zdaj zapišemo še zaprtje $V^* = \{\epsilon\} \cup V \cup V^2 \cup V^3 \cup \ldots = \{\epsilon\} \cup V^+$. Prazna beseda ima v operaciji $\circ$ lastnost identitete, saj zanjo velja $\epsilon \circ x = x \circ \epsilon = x$, za vsako besedo x iz V^*. Druga lastnost je asociativnost, saj velja $(x \circ y) \circ z = x \circ (y \circ z)$, za vse x, y in z iz V^*.

Jezik lahko imenujemo podmnožico zaprtja V^* abecede V. Ker je tako določen jezik v svoji sestavi zelo obširen, se omejimo na ožjo definicijo, ki pravi, da je jezik L množica besed nad neko končno abecedo V_f, kar zapišemo $L \subseteq {V_f}^*$. Jezik lahko predstavimo s končno ali neskončno množico besed iz naše abecede. Pri končnem zaporedju lahko preprosto preštejemo vse elemente, pri neskončnem je to nemogoče. Za namen končne predstavitve opisa jezika, vpeljemo pojem *gramatike*.

Gramatika je formirana iz končne, neprazne množice pravil ali produkcij, ki določajo sintaktične značilnosti oz. strukturo jezika. Gramatiko jezika definiramo kot četvorček $G = (V_N, V_T, S, \Phi)$, kjer množici V_N in V_T predstavjata sintaktični razred neterminalnih in terminalnih simbolov. S je začetni simbol, Φ pa končna, neprazna podmnožica relacije $(V_T \cup V_N)^* V_N (V_T \cup V_N)^* \Rightarrow (V_T \cup V_N)^*$.

Za primer si poglejmo zelo preprosto gramatiko za razpoznavanje aritmetičnih izrazov:
$V_N = \{E, T, F\}, \quad V_T = \{+, -, *, /, (,)\}, \quad S = E$

Množica produkcij Φ je definirana na naslednji način:

$E = E + T, \quad E = E - T, \quad E = T$
$T = T * F, \quad T = T/F, \quad T = F$
$F = stevilo, \quad F = (E)$

Z izpeljavo lahko preverimo, da npr. izraz $5 * 8 - (6/2) + 7$, pripada jeziku, podanemu z gornjo gramatiko.

Imejmo sedaj gramatiko $G = (V_N, V_T, S, \Phi)$ in $\sigma, \psi \in V^*$. Pravimo, da je σ neposredna izpeljava ψ, kar zapišemo $\psi \Rightarrow \sigma$, če obstajata takšni besedi γ_1, γ_2, da velja $\psi = \gamma_1\alpha\gamma_2$ in $\sigma = \gamma_1\beta\gamma_2$, pri čemer je $\alpha \rightarrow \beta$ produkcija iz G.
Če velja $\psi \Rightarrow \sigma$ pravimo tudi, da ψ neposredno producira σ oz. da se σ neposredno reducira v ψ.
Zapišimo, da ψ producira σ z $\psi \Rightarrow^+ \sigma$, če obstajajo takšne besede $\gamma_1, \gamma_2, \ldots, \gamma_n$, da je $\psi = \gamma_1 \Rightarrow \gamma_2, \gamma_2 \Rightarrow \gamma_3, \ldots, \gamma_{n-1} \Rightarrow \gamma_n = \sigma$. Relacija $\Rightarrow^+$ je tranzitivno zaprtje relacije $\Rightarrow$. Če za vrednosti n vključimo tudi vrednost 0, lahko refleksivno tranzitivno zaprtje k $\Rightarrow$ definiramo kot:

$$\psi \Rightarrow^* \sigma \quad \Leftrightarrow \quad \psi \Rightarrow^+ \sigma \ \vee \ \psi = \sigma$$

Stavčna oblika je poljubna izpeljava neterminalnega simbola S. Jezik $\mathcal{L}$, dobljen iz gramatike G je potemtakem množica vseh stavčnih oblik, katerih simboli so terminali:

$$\mathcal{L}(G) = \{\sigma | S \Rightarrow^* \sigma \wedge \sigma \in {V_T}^*\}$$

2.2 Klasifikacija gramatik

Predstavimo sedaj gramatike na način, kot jih je klasificiral Chomsky v svojem matematičnem modelu. Gramatike so v splošnem razdeljene v štiri razrede glede na omejitve v množici produkcij. V prvi razred sodijo gramatike, ki nimajo omejenih pravil in jih zato imenujemo neomejene gramatike. V drugi razred se uvrščajo gramatike z omejitvijo na produkcijskih pravilih:

$$\alpha \rightarrow \beta, \quad |\alpha| \leq |\beta|,$$

kjer je $|\alpha|$ dolžina besede α. Ta omejitev pomeni, da simbol β ne sme biti prazna beseda. Gramatika s takšno omejitvijo se imenuje kontekstno odvisna gramatika.

Nasprotno pa kontekstno proste gramatike vsebujejo samo produkcije oblike:

$$\alpha \rightarrow \beta, \quad |\alpha| \leq |\beta| \ \wedge \ \alpha \in V_N$$

Kontekstno proste gramatike vsebujejo produkcije, katerih leva stran sestoji samo iz enega razreda simbolov. Te gramatike so bistveno preprostješe in nimajo moči, da bi definirale naravni jezik, kjer je kontekstna odvisnost temeljnega pomena za pravilno interpretacijo. Po drugi strani pa te gramatike zadovoljijo večini programskih jezikov, saj so ti v primerjavi z naravnimi precej preprostejši po pomenu in sintaktični strukturi.

Še zadnja omejitev nas pripelje do definicije regularnih gramatik:

$$\alpha \rightarrow \beta, \quad |\alpha| \leq |\beta| \ \wedge \ \alpha \in V_N,$$

kjer ima β obliko aB ali a, kjer je $a \in V_T$ in $B \in V_N$. Jeziki, definirani s takšno gramatiko, se imenujejo regularni jeziki. Če neomejene, kontekstno odvisne in proste ter regularne gramatike klasificiramo v razrede C_0, C_1, C_2 in C_3 in z $\mathcal{L}(C_i)$ predstavimo razred jezikov, ki jih generira množica gramatik C_i, velja hierarhija:

$$\mathcal{L}(C_3) \subset \mathcal{L}(C_2) \subset \mathcal{L}(C_1) \subset \mathcal{L}(C_0)$$

Vsakemu razredu gramatik pripada razred akceptorjev, ki sprejemajo jezike v tem razredu. Na torišču programskih jezikov se bomo omejili na kontekstno proste gramatike.

Zraven opisanega gramatičnega pristopa definicije jezika obstajajo še seveda druge formalne metode.

2.3 Meta opisni jezik BNF

Za opis sintaktične strukture programskega jezika uporabljamo obliko BNF. Meta spremenljivke oz. sintaktični razredi se nahajajo med znakoma < in >. Tako se izognemo nesporazumu in dvoumnostim med neterminalnimi in terminalnimi simboli. V današnjem

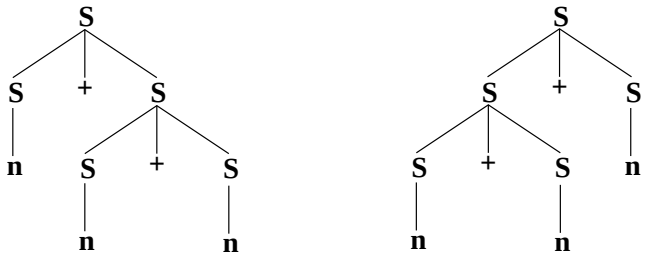
času je na področju programskih jezikov BNF izrazito primarni opisni mehanizem. Produkcijo v BNF notaciji zapišemo kot $< produkcija >::= \alpha_1|\alpha_2|\ldots\alpha_n$, kjer znak | pomeni "ali". Razširitev metajezika BNF je EBNF (Extended BNF), ki vpelje dodatna meta simbola {} za iteracijo in [] za opcijo.

2.4 Sintaktična drevesa

Sintaktična drevesa so pomemben predstavitveni konstrukt za razumevanje sintaktične strukture stavka. Drevo je podatkovna struktura, ki nazorno prikazuje relacije med posameznimi deli stavka. Koren sintaktičnega drevesa predstavlja vedno začetni simbol gramatike jezika. Listi drevesa, torej zunanja vozlišča, so sinonim za terminalne simbole. Neterminalni simboli so opisani z notranjimi vozlišči. Vsak neterminalni simbol ima eno vejo ali več, od katerih lahko vsaka predstavlja svoje poddrevo. Sinovi nekega vozlišča v drevesu se v gramatiki nahajajo na desni strani produkcije. Vzemimo za primer preprosto gramatiko, ki ima dve produkciji $S \rightarrow S + S$ in $S \rightarrow n$, kjer je n terminalni simbol. Poskušajmo izpeljati stavek $n + n + n$. Možno dobljeno zaporedje je:

$$S \Rightarrow S + S \Rightarrow S + S + S \Rightarrow n + S + S \Rightarrow n + n + S \Rightarrow n + n + n.$$

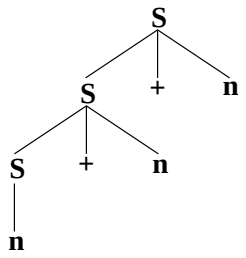
Zraven izpeljanega pa obstaja še eno zaporedje, saj lahko kateregakoli izmed S-ov v drugem koraku zapišemo kot $S + S$. Drevesi za obe izpeljavi sta prikazani na sliki 1.



Slika 1: Dvoumna gramatika povzroči dve različni sintaktični drevesi za izpeljavo stavka $n + n + n$.

Za isti stavek imamo torej dve različni izpeljavi, ki generirata dve različni drevesi. Ta pojav v sintaktični definiciji programskega jezika ni zaželjen in nemalokrat povzroča

velike težave. Če namreč za isto zaporedje vhodnih simbolov obstaja več kot eno drevo izpeljave, rečemo, da je takšno zaporedje dvoumno. Gramatika, ki generira vsaj eno dvoumno zaporedje, je tudi sama dvoumna. Dvoumnosti je iz gramatike potrebno odpraviti, če želimo eksaktno določljiv generiran kod. Težavo zgoraj povzroča operator +, kateremu nismo eksplicitno definirali asociativnosti. Če bi privzeli npr. levo asociativnost, do dvoumnosti ne bi prišlo. Če produkcijska pravila zapišemo nekoliko drugače, se dvoumnosti izognemo tudi brez uporabe asociativnosti (slika 2).



Slika 2: Alternativni zapis gornje gramatike $S \to S + n$ in $S \to n$ ni dvoumen, kar ima za posledico samo eno drevo izpeljave za stavek $n + n + n$.

Poudariti je treba, da dvoumne gramatike ne generirajo nujno dvoumnih jezikov. Veliko različnih gramatik, tako dvoumnih kot nedvoumnih, lahko namreč generira isti jezik. Obstajajo pa jeziki za katere ni moč najti nedvoumnih gramatik. Lahko bi načrtovali množico zadostnih pogojev, ki bi preverili ali je neka gramatika nedvoumna ali ne. Sicer pa v splošnem algoritem, ki bi na vhodu prejel kontekstno prosto gramatiko in v končnem času določil njeno dvoumnost ali nedvoumnost, ne obstaja.

2.5 Leksikalni nivo

Prva faza analize izvornega zaporedja simbolov je leksikalna analiza, ki poišče najprimitivnejše jezikovne dele–leksikalne simbole. Med te se uvrščajo imena identifikatorjev, znakovni nizi, številke, operatorji in rezervirane besede. Leksikalno analizo programa opravlja pregledovalnik (scanner), ki predstavlja vmesnik med izvornim programom in sintaktičnim analizatorjem tj. razpoznavalnikom. Pregledovalnik z vhoda bere znak

za znakom in jih združuje v leksikalne simbole (tokens), ki jih v sintaktični analizi uporablja razpoznavalnik. Leksikalna analiza se običajno opravi v prvem prehodu, ločeno od sintaktične. Najugodnejši način je ta, da razpoznavalnik od pregledovalnika eksplicitno zahteva naslednji simbol, ko ga potrebuje. Tega načina se poslužuje tudi avtomatski generator pregledovalnikov Yacc (Yet Another Compiler Compiler) [57], ki ga bomo podrobneje opisali v nadaljevanju. Za realizacijo pregledovalnikov navadno uporabljamo deterministične končne avtomate. Končni avtomat ali končni akceptor je mehanizem, ki se glede na trenutni vhod ustrezno premika v eno izmed možnih stanj. Operacija končnega avtomata se začne v začetnem stanju. Če se po prebranem zaporedju znakov avtomat nahaja v končnem stanju, je vhodni niz sprejet. Za grafično predstavitev končnega avtomata se uporablja diagram prehajanja stanj. Primer takšnega diagrama je na sliki 3. Deterministični končni avtomat definiramo kot petorček:

$$A = (K, V_T, M, S, Z),$$

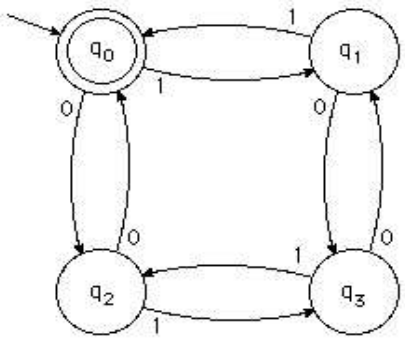
kjer je K končna neprazna množica stanj, V_T vhodna abeceda avtomata, M preslikava $K \times V_T \rightarrow K$, $S \in K$ začetno stanje avtomata in $Z \subseteq K$ neprazna množica končnih stanj. Deterministični končni avtomat je ob zagonu v začetnem stanju S, kjer prične z branjem vhodnega niza znakov. Tranzicija stanj je določena s funkcijo M, ki je definirana kot $M(Q, t) = R$, kjer sta Q in R stanji avtomata, t pa znak vhodne abecede. Takšna definicija funkcije pomeni, da avtomat ob vhodnem znaku t preide iz stanja Q v stanje R. Funkcijo preslikave dopolnimo z naslednjima definicijama:

$$M(Q, \epsilon) = Q \qquad \forall Q \in K$$

$$M(Q, Tt) = M(M(Q, T), t) \qquad \forall T \in V_t \wedge t \in {V_T}^*$$

Prva definicija pove, da se avtomat ne more spremeniti brez vhodnega znaka. Druga definicija omogoča rekurzivno aplikacijo preslikave, kadar imamo na vhodu besedo in ne zgolj en sam znak. Avtomat $A = (K, V_T, M, S, Z)$ stavek t sprejme, če je $M(S, t) = P$, pri čemer je $t \in {V_T}^*$ in $P \in Z$. To pomeni, da se mora avtomat po vseh prebranih znakih iz t nahajati v enem izmed končnih stanj. Množica vseh $t \in {V_T}^*$, ki jih avtomat A sprejme, je definirana z $\mathcal{L}(A)$:

$$\mathcal{L}(A) = \{\ t \mid M(S, t) \in Z \quad \wedge \quad t \in {V_T}^* \}$$



Slika 3: Primer determinističnega končnega avtomata s štirimi stanji, kjer je Q_0 hkrati začetno in končno stanje. Primer prehajanja stanj avtomata za vhodni niz 1 0 0 1 1 1 0 0 je q_0 1 q_1 0 q_3 0 q_1 1 q_0 1 q_1 1 q_0 0 q_2 0 q_0.

Poleg deterministični končnih avtomatov obstajajo tudi nedeterministični. Ti so zelo podobni determinističnim, razlikujejo se samo po tem, da lahko ob istem vhodnem znaku iz enega stanja preidejo v več drugih. Nedeterministični avtomat definiramo na naslednji način:

$$A = (K, V_T, M, S, Z),$$

kjer je K končna neprazna množica stanj, V_T vhodna abeceda avtomata, M preslikava v podmnožice $K \times V_T \rightarrow 2^K$, $S \in K$ začetno stanje avtomata in $Z \subseteq K$ neprazna množica končnih stanj. Preslikava M lahko slika tudi v prazno množico stanj, kar zapišemo $M(Q,t) = \{P_1, P_2, \ldots, P_n\}$. Tranzicija iz stanja Q v eno izmed stanj $P_1, P_2, \ldots, P_n$ se zgodi, kadar se prebere vhodni znak t. Funkcijo preslikave razširimo z naslednjimi definicijami:

$$\begin{array}{lcll} M(Q,\epsilon) & = & \{Q\} & Q \in K \\ M(Q,Tt) & = & \bigcup_{P \in M(Q,T)} M(P,t) & T \in V_T \quad \wedge \quad t \in {V_T}^* \\ M(\{Q_1, Q_2, \ldots, Q_n\}, x) & = & \bigcup_{i=1}^{n} M(Q_i, x) & x \in {V_T}^* \end{array}$$

Prva definicija je enaka kot pri deterministični različici, avtomat se ob praznem vhodu ne spremeni. V drugi definiciji je $M(Q,Tt)$ množica vseh možnih stanj, v katerih se lahko nahaja avtomat po prebranem nizu T_t. Tretja enačba razširi funkcijo preslikave v $2^k \times {V_T}^*$ in jo definira kot unijo vseh množic, ki jih dobimo z aplikacijo posameznega

stanja na vhod x. Pravimo, da nedeterministični avtomat vhodni niz x sprejme, če je po prebranem nizu x vsaj eno izmed dosegljivih stanj (iz začetnega stanja) končno stanje. x je torej v avtomatu $F = (K, V_T, M, S, Z)$ sprejet, če za neko stanje $A \in M(S, x)$ velja, da je $A \in Z$.

Dejanska implementacija deterministčnega končnega avtomata ni pretirano zahtevna. Stanja avtomata se običajno realizirajo kot tabela, prehodi med posameznimi stanji pa s skoki. Ker je kljub relativno preprosti implementaciji v vhodnem programskem kodu veliko število različnih simbolov, je proces izdelave takšnega pregledovalnika s konceptom avtomata, zelo zamudno opravilo. Zato bomo mi raje uporabili eno izmed orodij za avtomatsko generiranje pregledovalnikov. Zelo znano orodje, ki je namenjeno prav temu, se imenuje Flex (Fast lexical annalyzer generator)[57].

Flex na vhodu sprejme specifikacije regularnih izrazov, ki jih želimo razpoznavati in nato generira pregledovalnik v jeziku C++. Da bi razumeli vhodne specifikacije, si je treba najprej podrobneje pogledati, kaj so regularni izrazi.

Regularni izraz je opis vzorca z uporabo meta jezika. Regularni izrazi v osnovi uporabljajo tri operatorje: konkatenacijo, alternacijo in zaprtje. Če dva izraza e_1 in e_2 generirata jezika L_1 in L_2, potem je operator konkatenacije (združevanja) definiran kot $e_1e_2 = \{xy \mid x \in L_1 \ \wedge \ y \in L_2\}$. Alternacija, katero označimo z znakoma | ali +, je unija obeh jezikov: $e_1|e_2 = \{x \mid x \in L_1 \ \vee \ x \in L_2\}$. Zaprtje, ki ga predstavimo z { }, opisuje ponovitve izrazov nič ali večkrat. To zapišemo kot $\{e_1\} = \{x \mid x \in {L_1}^*\}$, kjer je ${L_1}^* = \bigcup_{i=0}^{\infty} {L_1}^i$. Regularne izraze sestavljamo z naslednjimi pravili:

- ϕ je regularni izraz, ki označuje prazno množico,
- ϵ je regularni izraz, ki označuje jezik, sestavljen samo iz praznih besed, torej $\{\epsilon\}$,
- $a \in V_T$ je regularni izraz, ki opisuje jezik z enim samim simbolom a, torej $\{a\}$,
- če sta e_1 in e_2 regularna izraza, ki označujeta jezika L_1 in L_2, potem velja:
  - $(e_1)|(e_2)$ je regularni izraz unije $L_1 \cup L_2$,

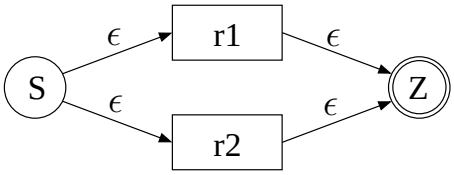
  - $(e_1)(e_2)$ je regularni izraz konkatenacije L_1L_2 in
  - $\{e_1\}$ je regularni izraz ponovitve ${L_1}^*$.

Regularne izraze, ki jih bomo uporabili za konkreten opis specifikacij generatorja pregledovalnika, temeljijo na zgoraj podanih formalnih definicijah, vendar so potrebni nekaterih dopolnil. Meta znaki, ki opisujejo praktičen zapis regularnih izrazov, so naslednji:

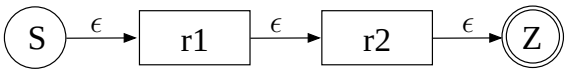
- . se ujema z natanko enim poljubnim znakom, razen znakom za konec vrstice (`\n`),
- * se ujema z nič ali več ponovitvami predhodnega izraza,
- `[ ]` razred znakov, ki se ujema s katerimkoli znakom v oklepajih. Če je prvi znak v oklepaju ^ , se pomen negira. Uporablja se lahko tudi znak - za območje (npr. `[0-9A-Z]`),
- ^ se ujema z začetkom vrstice,
- $ se ujema s koncem vrstice,
- {} pove, kolikokrat se lahko predhodni izraz ujema, Z{1,5} npr. pomeni eno, dve, tri, štiri ali pet ponovitev znaka Z,
- \ se uporablja kot izhodni znak (escape sequence) za opis meta znakov,
- + se ujema z enim ali več ponovitvami predhodnega izraza,
- ? se ujema z nič ali eno pojavitvijo predhodnega izraza,
- | pomeni alternacijo podanih regularnih izrazov,
- "..." interpretira vsebino med narekovaji dobesedno,
- () se uporablja za grupiranje posameznih regularnih izrazov v nov izraz.

Realno število bi z zgoraj definiranimi pravili opisali kot:
`([0-9]+)([0-9]*\.[0-9]+)`

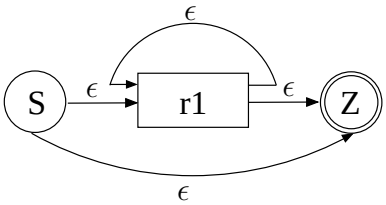
Jedro pregledovalnika (leksikalnega analizatorja) ni nič drugega kot končni (deterministični) avtomat generiran iz regularnih izrazov. Regularni izrazi in končni avtomati so namreč ekvivalentni. Pretvorbo si lahko na enostaven način zamislimo z uporabo prehodnih diagramov. Prehodni diagrami z ϵ prehodom za vse tri osnovne operacije so prikazani na slikah 4, 5 in 6.



Slika 4: Diagram prehajanja stanj za regularni izraz $r_1 \mid r_2$.



Slika 5: Diagram prehajanja stanj za regularni izraz $r_1 r_2$.



Slika 6: Diagram prehajanja stanj za regularni izraz $\{r_1\}$.

2.6 Sintaktični nivo

Primarno opravilo analize programa na sintaktičnem nivoju je ugotavljanje pravilnosti strukture simbolov vhodnega programa. Del prevajalnika, ki opravlja sintaktično anal-

izo se imenuje razpoznavalnik (parser) ali sintaktični analizator. Konkretno, z implementacijskega stališča, ločimo razpoznavanje od zgoraj navzdol (top-down) in razpoznavanje od spodaj navzgor (bottom-up). Razpoznavalnik od zgoraj navzdol karakteriziramo z idejo, da začnemo s ciljnim simbolom gramatike in poskušamo zgraditi zaporedje terminalnih simbolov, ki ustrezajo vhodnemu zaporedju. Razpoznavalnik od spodaj navzgor pa nasprotno izhaja iz listov drevesa in skuša priti do korenskega simbola.

2.6.1 Razpoznavalni pristopi od zgoraj navzdol

V splošnem poznamo tri strategije za tovrstno razpoznavo vhodnega niza simbolov. Najpreprostejša je zagotovo nasilna (brute-force) strategija. Druga metoda je rekurzivno navzdolnje razpoznavanje (recursive descent), ki ne dovoljuje mehanizma vračanja. Tretjo metodo klasificiramo kot razpoznavanje od zgoraj navzdol z omejenim vračanjem.

Tehnika razpoznavanja z brute-force pristopom uporablja mehanizem vračanja na ta način, da sproti gradi poddrevesa, dokler ne pride do želenega zaporedja terminalnih simbolov. Hitro lahko uvidimo zelo slabo lastnost tega pristopa. Če namreč želimo razpoznati niz simbolov, ki ne pripada jeziku, je potrebno najprej zgraditi vse možne kombinacije dreves, preden lahko z gotovostjo rečemo, da zaporedje ni del jezika. Metoda v principu deluje tako, da se najprej aplicira prva produkcija danega neterminala, ki ga želimo razviti. Nato se v tem razvitem neterminalu izbere naslednji nabolj levi neterminal nakar se ponovno aplicira njegova prva produkcija. Ta korak apliciranja produkcije se ponavlja za vse sledeče izbrane neterminale. Proces se ustavi, ko ni več neterminalnih simbolov, ali kadar pride do nepravilnega razvitja neterminala. V prvem primeru je zaporedje uspešno razpoznano, v drugem pa se postopek reducira z razveljavitvijo zadnje aplicirane produkcije in izbiro naslednje. Če produkcij, ki bi jih lahko aplicirali, ni več, se to razvitje (ki je povzročilo napako) nadomesti s samim neterminalom. To razveljavljanje se nadaljuje, dokler lahko apliciramo produkcije. Ko pridemo do začetnega simbola in nimamo več produkcij, ki bi jih lahko aplicirali, lahko rečemo, da vhodnega zaporedja simbolov ni mogoče razpoznati, ker očitno ni del jezika. Potrebno je pojasniti, da opisan pristop ne deluje za sleherno kontekstno

prosto gramatiko. Če imamo kontekstno prosto gramatiko $V = (V_N, V_T, S, \Phi)$, potem za neterminal X rečemo, da je levo rekurziven, če velja $X \Rightarrow^+ X\alpha$ za neko besedo $\alpha \in V^*$. V kolikor gramatika vsebuje vsaj en levo rekurziven neterminal, potem je tudi sama levo rekurzivna. Na takšnih gramatikah, če produkcijskih pravil ne predstavimo drugače, pristop brute-force v splošnem ne deluje. Ker so gramatike te vrste v definicijah realnih jezikov zelo pogoste, se zgoraj opisana metoda razpoznavanja ne uporablja prav veliko.

Pri rekurzivnem navzdolnjem razpoznavanju vpeljemo omejitev, ki ne dovoljuje vračanja. Kot prejšnja metoda, tudi ta ne more razpoznati vseh vrst kontekstno prostih gramatik. Razpoznava nekaterih gramatik namreč ne more biti izvedena brez vračanja. V rekurzivnem navzdoljnem razpoznavalniku zaporedje produkcijskih aplikacij realiziramo z zaporedjem klicev funkcij. Vsak neterminal je torej predstavljen s funkcijo, ki običajno vrača binarno vrednost `true` ali `false`, s katero pove, ali lahko uspešno razpozna zaporedje neterminala.
Razpoznavanje z omejenim vračanjem je podobno brute-force pristopu z manjšimi spremembami, zato ga posebej ne obravnavamo.

Pri razpoznavalnikih brez vračanja se pojavi problem, katero produkcijo danega neterminala aplicirati. Levo razpoznavne gramatike (imenovane LL(k)) odpravljajo to dilemo. Če imamo že razpoznano zaporedje vhodnih simbolov in najbolj levi terminal, lahko z množico naslednjih k simbolov še nerazpoznanega vhoda, izberemo pravilno produkcijo. LL(k) gramatike lahko torej razpoznamo tako, da preprosto pogledamo naslednji simbol. Preprosta gramatika LL(1) je kontekstno prosta gramatika brez ϵ pravil, tako da za vsako alternativo $A \in V_N$ velja, da se začne z drugačnim terminalnim simbolom. To zapišemo:

$$A \rightarrow a_1\alpha_1|a_2\alpha_2|\ldots|a_m\alpha_m, \quad a_i \neq a_j \; \forall i \neq j \quad \wedge \quad a_i \in V_T, \quad 1 \leq i, j \leq m$$

Tako zapisano gramatiko imenujemo tudi s-gramatika [10]. S-gramatike generirajo s-jezike. Preproste LL(1) gramatike posplošimo v LL(1) gramatike z odstranitvijo zahteve, da mora biti najbolj levi simbol na desni strani produkcijskega pravila terminal.

Še vedno pa obstaja omejitev, ki ne dovoljuje ϵ pravil. LL(1) gramatiko formalno definiramo s pomočjo FIRST in FOLLOW množic. Če imamo besedo $\alpha \in V^+$, je množica terminalnih simbolov, ki so najbolj levo izpeljivi iz α, podana z enačbo:

$$FIRST(\alpha) = \{\ w \mid \alpha \Rightarrow^* w \ldots \quad \wedge w \in V_T\}$$

Zdaj lahko rečemo, da je gramatika brez ϵ pravil LL(1) gramatika, če so za vse produkcije oblike $A \rightarrow \alpha_1|\alpha_1|\ldots|\alpha_n$ množice $FIRST(\alpha_1)$, $FIRST(\alpha_2)$,..., $FIRST(\alpha_n)$ paroma disjunktne, kar zapišemo:

$$FIRST(\alpha_i)\ \cap\ FIRST(\alpha_j)\ =\ \emptyset \quad , \quad i \neq j$$

LL(1) gramatike lahko posplošimo in razširimo z uvedbo ϵ produkcijskih pravil. Funkcijo FIRST razširimo za obvladovanje ϵ pravil:

$$FIRST(\alpha) = \{\ w \mid \alpha \Rightarrow^* w \ldots \quad \wedge \quad |w| \leq 1 \wedge w \in {V_T}^*\}$$

Ta definicija nam omogoča besede dolžine 0 in 1. Za gramatiko LL(1) vpeljemo nov pogoj, da mora za vsak par pravil $A \rightarrow \alpha$ in $A \rightarrow \beta$ v Φ za nek neterminal A, veljati:

$$FIRST(\alpha \circ FOLLOW(A))\ \cap\ FIRST(\beta \circ FOLLOW(A)) = \emptyset$$

Množica FOLLOW vrne množico terminalnih simbolov, ki neposredno sledijo določenemu neterminalu. Ekvivalentna a preprostejša definicija LL(1) je, da za vsa pravila oblike $A \rightarrow \alpha_1|\alpha_1|\ldots|\alpha_n$, velja:

$$FIRST(\alpha_i)\ \cap\ FIRST(\alpha_j)\ =\ \emptyset \quad , \quad \forall i \neq j$$

in, če je $\alpha_i \Rightarrow^* \epsilon$,

$$FIRST(\alpha_j)\ \cap\ FOLLOW(A)\ =\ \emptyset \quad , \quad \forall j \neq i$$

Množici FIRST in FOLLOW se uporabljata za izračun razpoznavalnikovih tabel. Podrobna razlaga računanja obeh množic z uporabo relacij po Warshallowem algoritmu je podana v [56].

2.6.2 Razpoznavalni pristopi od spodaj navzgor

Razpoznavalniki te vrste gradijo drevo izpeljav od listov proti korenu. Za razpoznavo od spodaj navzgor obstaja danes kar nekaj tehnik, ki se zaradi svoje učinkovitosti zelo pogosto uporabljajo. Običajno razpoznavalne algoritme delimo glede na klasifikacijo gramatik, ki so jih sposobni uspešno razpoznati. Večina pristopov uporablja en sam simbol na vhodu za deterministično razpoznavanje preostalega zaporedja. Seveda je potrebno povedati, da obstajajo tudi razpoznavalni algoritmi, ki uporabljajo več kot en simbol za nadaljno odločitev, vendar se ti ne uporabljajo prav veliko, saj njihova implementacija in velikost generiranih razpoznavalnih tabel ne odtehtata prednosti, ki jih dobimo z možnostjo odločanja ob več simbolih. Povrhu tega se velika večina programskih jezikov (kot jih poznamo danes) da razpoznati z uporabo samo enega simbola na vhodu. Najpreprostejši razpoznavalniki od spodaj navzgor so takšni, ki operirajo na podlagi prioritete. V ta razred uvrščamo razpoznavalnike, ki uporabljajo prioriteto operatorjev (operator precedence parsers) in splošno prioritetne (precedence parsers), ki posplošujejo prioriteto na vse simbole gramatike–neterminale in terminale. Formalnih definicij vseh gramatik in prioritetnih relacij, ki se uporabljajo kot teoretska osnova pri konstrukciji razpoznavalnikov, ne bomo posebej izpeljevali. Za nas so zanimivejši razpoznavalniki tipa LR(k), katerih lastnosti podrobneje opisujemo v nadaljevanju.

Deterministični razpoznavalniki LR(k) se uporabljajo za razpoznavanje jezikovnih gramatik tipa LR(k), med katere uvrščamo vse nedvoumne kontekstno proste gramatike, vključno z že omenjenimi LL(k) in prioritetnimi gramatikami. LR(k) razpoznavalnik bere vhod od leve proti desni. Vse odločitve se sprejemajo na osnovi trenutne vsebine sklada in k naslednjih simbolov. Dobra lastnost LR(k) razpoznavalnika je možnost odkritja napake že v zelo zgodnjih fazah razpoznave. Eno izmed slabših dejstev pa je zahtevnost v primeru večjega števila simbolov, torej ko je $k > 1$. Posebna zvrst teh razpoznavalnikov so LR(0) različice, ki ne potrebujejo nobenega simbola vnaprej, da bi uspešno razpoznali jezik. LR(0) razpoznavalik za neko vhodno zaporedje zgradi njegovo najbolj desno izpeljavo v obratnem smislu. Če imamo gramatiko G z začetnim

simbolom S, je najbolj desna izpeljava za simbol x:

$$S \Rightarrow_R \alpha_1 \Rightarrow_R \alpha_2 \ldots \Rightarrow_R \alpha_{m-1} \Rightarrow_R \alpha_m = x$$

Najbolj desna izpeljava ima obliko $\phi Bt \Rightarrow \phi\beta t$, pri čemer je $\alpha_i = \phi Bt$, $B \rightarrow \beta$ pa produkcija, ki jo apliciramo. Ker imamo najbolj desno izpeljavo, mora t biti zaporedje terminalnih simbolov. Gramatiko imenujemo LR(k), če lahko za vsako vhodno besedo v vsakem koraku katerekoli izpeljave β razpoznamo iz zaporedja $\phi\beta$ ob pregledovanju največ prvih k simbolov iz še ne razpoznanega vhodnega niza t. Možna predpona stavčne oblike $\phi\beta t$ je vsaka predpona zaporedja $\phi\beta = w_1 w_2 \ldots w_r$. Zaporedje $w_1 w_2 \ldots w_i$ je torej predpona, če je $1 \leq i \leq r$. Možna predpona stavčne oblike ne more vsebovati simbolov desno od β, tj. simbolov v t.

LR(0) razpoznavalnik za svoje delovanje uporablja sklad dvojčkov, v katerih se na prvem mestu nahaja simbol iz slovarja jezika, na drugem pa stanje v katerega se algoritem premakne ko prebere tak simbol. Na začetku sklad vsebuje le začetno stanje. Ko je razpoznavalnik v fazi branja, se prebrani simbol shrani na vrh sklada, hkrati pa se, glede na pravkar prebrani simbol, opravi tranzicija v novo stanje. Tudi novo stanje se shrani na sklad. Ko razpoznavalnik vstopi v aplikacijsko ali redukcijsko stanje (tega prepoznamo po produkciji $B \rightarrow \beta$), s sklada odstrani vrhnjih $2 * |\beta|$ simbolov. Če je pri tem dosežen ciljni simbol ($B = S$), se vhodni niz sprejme. V primeru, ko do ciljnega simbola še nismo prišli, se glede na reducirano stanje na vrhu sklada, izbere novo stanje. Podrobnosti, kako iz specifične množice produkcijskih pravil skonstruiramo pripadajoči razpoznavalnik, so podane v [56]. Razpoznavalni algoritem za najpreprostejši razred LR gramatik, ki temelji na enem vnaprejšnjem simbolu, se imenuje SLR(1) razpoznavalnik. Močnejša razpoznavalna metoda je LALR(1), ki jo uporablja tudi orodje za avtomatsko generiranje prevajalnikov Yacc.

2.6.3 Obravnava napak

Prvi nivo obravnave napak se nahaja v sintaktični analizi vhodnega programa. Kadar se zazna napaka sintaktične narave, je najpreprostejša rešitev izpis te napake in zaustavitev procesa prevajanja. Žal je to eden izmed primerov, kjer stopnja učinkovitosti

raste sorazmerno z enostavnostjo. Večina prevajalnikov, v imenu robustnosti, uvršča ta pristop med nesprejemljive, saj lahko ob vsakem prevajanju odkrijemo le eno (prvo) napako. Z namenom, da v programu najdemo čimveč sintaktičnih napak (pravkar napisani programi vsebujejo ponavadi veliko napak), bi želeli, da se prevajanje po prvi odkriti napaki nadaljuje. Da bi to dosegli, je potrebno modificirati trenutno stanje prevajalnika na ta način, da bo vhod sprejemljiv. Ker program z napako ne dosega želenega smisla, je v takšnem primeru tudi generiranje objektnega koda povsem odveč. Če ima prevajalnik sposobnost nadaljevanja razpoznave po sintaktični napaki, pravimo da je trdoživ in sposoben okrevanja. Pristopov za takšno okrevanje je več. Eden izmed preprostejših je takšen, ki ob pojavitvi sintaktične napake zbriše vse terminalne simbole do nekega "varnega" simbola, od katerega se lahko (predpostavljeno) varno nadaljuje razpoznava. Problem se pojavi, kadar odstranimo simbole, ki sestavljajo neko sintaktično pravilno strukturo. Posledica so kaskadne napake oz. napake zaradi napak. Omenjeni pristop odkrivanja napak lahko bistveno izboljšamo z uvedbo različnih varnih simbolov v različnih kontekstih [86]. Takšna tehnika se imenuje okrevanje na nivoju programskih fraz. Kadar pride do napake v programskem stavku, se pregledovalnik ne vrne povsem na začetek stavka, temveč samo do simbola, ki bi najverjetneje lahko sledil stavku. Vračanje oz. preskakovanje na ta način lokaliziramo, kar nam omogoča boljši vpogled v pravilne predele stavka. Vendar ima tudi ta pristop določene pomanjkljivosti, predvsem glede ϵ produkcij. Obetavna tehnika zaznavanja sintaktičnih napak je uporaba izjem. Te omogočajo, da namesto obravnave napak za vsak terminal določimo množico kontekstov, v katere se premaknemo ob pojavitvi napake. Če torej pride do napake v izrazu, se lahko pomaknemo v kontekst stavka, v katerem se je izraz pojavil.

Yacc generator prevajalnikov uporablja v ta namen poseben neterminal imenovan `error`. Neterminal se vedno nahaja na desni strani produkcijskega pravila in po dogovoru kot zadnja alternativa. V primeru, da se razpoznavalnik nahaja v neki produkciji, katere zaradi sintaktične nepravilnosti ne more uspešno reducirati in hkrati ne obstaja nobena alternativa, se pomakne v stanje "error", v katerem se nahaja uporabniški kod za obravnavo napake v tej produkciji. Neterminalu error lahko sledi še opcijski (vendar zelo uporaben) simbol, do katerega pregledovalnik preskoči ob pojavitvi napake.

Ob razpoznavanju programskih stavkov zaključenih s podpičjem bi bilo torej smiselno preskočiti do naslednjega podpičja.

2.6.4 Avtomatsko generirani razpoznavalniki orodja Yacc

Orodje prevajalnikov Yacc na vhodu sprejme specifikacije gramatike in na izhodu generira ustrezen razpoznavalnik. Z generiranim razpoznavalnikom je mogoče določiti sintaktično pravilne in nepravilne programe. Yacc praviloma operira skupaj z leksikalnim analizatorjem Flex, ki mu priskrbi leksikalne simbole na podlagi katerih se orodje Yacc odloča o tranzicijah med stanji. Sintaksa, ki se uporablja za podajanje specifikacij gramatike, je precej podobna formalnim BNF zapisom, ki smo jih že omenili. Simbola, ki sta neposredno povezana s podajanjem produkcijskih pravil, sta v orodju Yacc samo dva. Operator : ločuje levo in desno stran produkcije, | pa alternacijo. Preprost jezik izrazov bi torej zapisali takole:

```
expression:     expression '+' mulexp
              | expression '-' mulexp
              | mulexp
              ;

mulexp:         mulexp '*' primary
              | mulexp '/' primary
              | primary
              ;

primary:        '(' expression ')'
              | '-' primary
              | NUMBER
              ;
```

Simbol na levi strani je neterminal, medtem ko se na desni lahko pojavijo neterminali in terminali. Slednje zaradi ločevanja običajno označujemo z velikimi črkami, kot je v našem primeru NUMBER. Terminalne simbole, ki jih vrača leksikalni analizator, lahko

referenciramo z imeni ali neposredno s posameznimi znaki. Vsi simboli imajo lahko poljubne vrednosti, ki jih navedemo z unijo v deklaracijskem delu prevajalnika. Če imamo na desni strani zgolj en simbol, lahko prireditev levi strani varno izpustimo, saj orodje Yacc to stori avtomatsko. Gornji primer smo zapisali brez semantičnih akcij, ki se izvedejo, ko je neka produkcija uspešno reducirana. Yacc za označitev leve strani produkcije uporablja simbol $$. Za referenciranje vrednosti simbolov na desni strani se uporabljajo simboli $n, kjer je n zaporedna številka terminalnega ali neterminalnega simbola. Semantični del prevajalnika dodamo neposredno v blok reducirane produkcije:

```
statement:      expression {
                  cout<<"vrednost izraza je "<<$1;
                }
                ;

expression:     expression '+' mulexp {
                 $$ = $1 + $3;
                }
               | expression '-' mulexp {
                 $$ = $1 - $3;
                }
               | mulexp
               ;

mulexp:         mulexp '*' primary {
                 $$ = $1 * $3;
                }
               | mulexp '/' primary {
                 $$ = $1 / $3;
                }
               | primary
               ;
```

```
primary:        '(' expression ')' {
                 $$ = $2;
                }
              | '-' primary {
                 $$ = -$2;
                }
              | NUMBER {
                 $$ = str2int( $1 );
                }
              ;
```

Opazimo lahko, da smo uporabili samo atribute, ki smo jih pridobili z izračunom. To so pridobljeni oz. sintetizirani atributi. Poleg pridobljenih atributov Yacc omogoča še podedovane atribute, ki smo jih definirali v zgodnejših fazah razpoznave. Povedati je treba, da Yacc omogoča samo "delno" podedovane atribute, saj je pri teh potrebno poznati natančno strukturo sklada za vse akcije od definiranja atributa naprej. To ni vedno preprosto, še manj pa elegantno.

Zraven akcij, ki se izvedejo ob redukciji nekega produkcijskega pravila, je omogočena tudi uporaba "vmesnih" akcij, ki jih lahko umestimo med posamezne simbole (terminalne ali neterminalne) znotraj produkcije. S tem lahko dosežemo izvedbo uporabniškega koda še preden se neka produkcija povsem reducira.

Yacc omogoča eksplicitno prioriteto in asociativnost operatorskih terminalov. Asociativnost določimo v deklaracijskem delu z rezerviranimi direktivami %left, %right in %nonasoc, ki si sledijo po naraščajoči prioriteti. Prioriteta se uporablja, kadar zaradi dvoumnosti gramatike pride do pomakni/prevedi (shift/reduce) konflikta. V takšnem primeru Yacc uporabi tabelo prioritet, v kateri poišče simbol, zaradi katerega je prišlo do konflikta. V kolikor se simbol najde, se konflikt razreši glede na prirejeno prioriteto.

Produkcijska pravila lahko v rekurzivnem smislu posredno ali neposredno referencirajo sama sebe, kar omogoča razpoznavo poljubno dolgih vhodnih zaporedij. Seznam celih števil, ločenih z vejico $(1, 4, 2, \ldots, n)$ bi zapisali na naslednji način:

```
list:       NUMBER
```

```
| list ',' NUMBER
;
```

Iz primera je vidno, da smo uporabili levo rekurzijo, ki jo Yacc zelo dobro rešuje. Analogno bi lahko uporabili desno rekurzijo, vendar bi bila razpoznava le-te zaradi algoritma, ki ga Yacc uporablja, bistveno manj optimalna.

Razpoznavalnik, ki ga Yacc generira, je v osnovi končni avtomat s stanji. Vsako stanje odraža možno pozicijo v delno ali popolno razpoznanih produkcijskih pravilih. Vsakič, ko razpoznavalnik prebere naslednji leksikalni simbol, ki ne izpolni nobenega pravila, ta simbol shrani na svoj interni sklad in opravi tranzicijo v stanje, ki je določeno s tem simbolom. Takšno akcijo imenujemo "pomakni" (shift). Ko je prebran simbol, ki izpolnjuje celotno desno stran nekega pravila, se vsi simboli, ki sestavljajo desno stran, odstranijo s sklada. Na sklad se nato shrani leva stran pravkar reducirane produkcije. Temu sledi tranzicija v novo stanje, ki je določeno z reducirano produkcijo. To akcijo imenujemo "prevedi" (reduce). Ob njeni izvedbi, tj. ob razpoznanem programskem stavku, se izvrši uporabniški programski kod.
Razpoznavalni algoritem je dokaj preprost za splošno razumevanje. Proces si lahko ponazorimo s pojmom kazalca, ki kaže trenutno pozicijo v gramatiki. Označimo ta kazalec s simbolom $\uparrow$. Ko se postopek razpoznave prične, obstaja samo en kazalec – na začetku. Če imamo produkcijo $statement \longrightarrow A\ B\ C$, kjer so A, B in C terminalni simboli, to zapišemo:

$$statement \longrightarrow \uparrow\ A\ B\ C$$

Ko se prebere naslednji terminalni simbol, se kazalec premakne za eno mesto naprej:

$$statement \longrightarrow A\ \uparrow\ B\ C$$

Seveda se morajo v primeru alternativnih pravil ohraniti vse trenutno mogoče pozicije kazalcev, ki jih je lahko več:

$$statement \longrightarrow x \mid y$$

$$x \longrightarrow A\ B\ \uparrow\ C\ D$$

$$y \longrightarrow A\ B\ \uparrow\ E\ F$$

Vse možne alternative se ohranjajo vse dokler se ne prebere terminalni simbol, ki pravilo zaključi oz. reducira. Kazalec se lahko odstrani tudi v primeru, ko se prebere terminal, ki nadaljevanje nekega pravila onemogoča. Ko se v gornji produkciji torej prebere terminal C, drugega kazalca ni potrebno več evidentirati:

$$statement \longrightarrow x \mid y$$

$$x \longrightarrow A\ B\ C \uparrow\ D$$

$$y \longrightarrow A\ B\ E\ F$$

Kadar se neko produkcijsko pravilo reducira ter ob tem obstaja več kot le en kazalec, pride do konflikta. Ker se konflikt zgodi ob operaciji reduciranja, ga imenujemo prevedi/prevedi konflikt (reduce/reduce conflict). Nasprotno lahko pride do pomakni/prevedi konflikta (shift/reduce conflict), kadar se razpoznavalnik nahaja v stanju, kjer bi lahko eno pravilo zaključil, drugo pa nadaljeval:

$$statement \longrightarrow x \mid y\ R$$

$$x \longrightarrow A \uparrow\ R$$

$$y \longrightarrow A \uparrow$$

Prevedi/prevedi konflikti običajno nastanejo zaradi nepravilnosti v gramatiki in se praviloma ne bi smeli pojavljati. Do pomakni/prevedi konfliktnih situacij pa ponavadi pride zaradi samo enega simbola, ki ga Yacc lahko pogleda vnaprej; Yacc namreč generira LALR(1) pregledovalnik. V drugem primeru algoritem orodja Yacc izbere akcijo pomakni, saj se drži pravila najdaljšega ujemanja.

2.6.5 Drugi generatorji razpoznavalnikov

V času tega pisanja obstaja razmeroma veliko število bolj ali manj sofisticiranih avtomatskih generatorjev razpoznavalnikov. Večina jih generira izvorni kod v jeziku C,

modernejši pa tudi v Javi. Nesmiselno bi bilo podrobno opisovati vsakega izmed njih. Povzemimo na kratko možnosti, ki jih nudi eden izmed njih–generator LISA.

LISA (Language Implementation System based on Attribute grammars) je splošno interaktivno okolje za načrtovanje programskih jezikov. Osnovna in bistvena naloga orodja je, da iz formalnih specifikacij jezika generira jezikovno specifično okolje, ki sestoji iz prevajalnika in interpreterja. LISA poleg avtomatskega generiranja prevajalnikov omogoča tudi inkrementalno načrtovanje programskega jezika v vizualnem smislu. Podprti so vsi nivoji razvoja jezika: leksikalni, sintaktični in semantični. Svojstvenost v tem orodju je ta, da za inkrementalni razvoj jezika vpeljuje koncept večkratnega dedovanja, ki je značilen za objektne in objektno usmerjene jezike. Večkratno dedovanje atributnih gramatik je strukturalna organizacija gramatike, kjer le-ta nekatere lastnosti (jezikovne specifikacije) deduje od starševskih gramatik, katerim lahko doda povsem svoje lastnosti ali pa spremeni obstoječe [72]. LISA omogoča, da pristop dedovanja apliciramo tako na leksikalnem, kot sintaktičnem in semantičnem nivoju jezikovnih specifikacij. Podrobnosti o konceptu večkratnega dedovanja atributnih gramatik in implementaciji lahko bralec najde v [73]. Dodaten mehanizem za opis generičnosti gramatik, ki ga LISA omogoča, je šablona (template). Šablona je polimorfna abstrakcija semantičnega pravila, parametriziranega z atributi, katere lahko uporabimo v produkcijskih pravilih z različnimi terminalnimi in neterminalnimi simboli. Ko se šablona instancira, se generirajo konkretne semantične akcije. Osnovna ideja dedovanja atributnih gramatik in uporabe šablon je v inkrementalnem načrtovanju jezika, ki spet dalje pomeni modularno strukturiranost in krajši razvojni čas [71].

Avtorji zatrjujejo, da je LISA zasnovana modularno, kar omogoča (sočasno) uporabo različnih pregledovalnikov in evaluatorjev [74]. Tako pregledovalnik kot evaluator komunicirata z leksikalnimi simboli, ki predstavljajo osnovne entitete v leksikalni analizi. Sintaktično drevo, ki ga zgradi pregledovalnik, je sestavljeno iz leksikalnih simbolov. Ob procesu evaluacije oz. vrednotenja drevesa, se v vozlišča vstavijo atributi, ki pretvorijo sintaktično drevo v semantično. Vhod se ovrednoti z obiski vozlišč, v katerih se izvršijo semantične akcije. Moduli za pregledovalnik, razpoznavalnik in evaluator za generiranje izhodnega koda uporabljajo šablone, kar omogoča precejšnjo mero prilagodljivosti. Vhod v generator je lahko tekstovni ali vizualni. V prvem

primeru specifikacije napišemo v preprostem domenskem jeziku, ki omogoča večkratno dedovanje atributnih gramatik in uporabo šablon. Takšne specifikacije sestojijo iz leksikalnih, sintaktičnih in semanitčnih pravil. Če je vhod vizualen, se uporabljajo končni avtomati in sintaktični ter semantični diagrami. V obeh primerih se opis jezika pretvori v monolitne specifikacije. Nato sledi generiranje prevajalnika, ki je sestavljen iz leksikalnega in sintaktičnega analizatorja ter evaluatorja. LISA iz formalnih specifikacij generira celo urejevalnik posebej prirejen za ta jezik. Primarne lastnosti sistema LISA so torej inkrementalni razvoj jezika s konceptom dedovanja atributnih gramatik, poenostavitev načrtovanja splošnih (generic) lastnosti z uporabo šablon ter vizualno orientirana predstavitev posameznih faz v prevajalniku.

2.7 Semantični nivo in uporaba atributnih gramatik

Sintaktični analizator omogoča razpoznavo konstruktov programskega jezika. Povedano drugače, na sintaktičnem nivoju določimo ali je neko zaporedje vhodnih simbolov pravilno ali ne. Pregledovalnik za jezik C++ v sintaktični analizi zazna, da je programski stavek `int i;` dejansko del jezika, `int int i;` pa ne zadošča nobeni produkciji in je zato napačen. Čeprav lahko določamo pravilnost in napačnost programskih fraz, pa ne moremo nič povedati o pomenu teh fraz. Vemo torej, da je `int i;` pravilen stavek, toda kakšen je njegov pomen?

Pomen stavkov določamo na semantičnem nivoju – v okviru semantične analize. Semantične ali pomenske informacije dobimo tako, da gramatiki, s katero je določena sintaktična struktura programa, dodamo atribute. Atributna gramatika (attributed grammar) je torej gramatika, katere simboli so obogateni z atributi, ki nosijo semantično informacijo. Atributi v prvi meri določajo izvrševanje semantičnih akcij. Atributi se prenašajo po vejah sintaktičnih dreves od listov proti korenu in obratno, in na nek način omogočajo komunikacijo med posameznimi semantičnimi akcijami. Izvedba ene semantične akcije je tako povečini odvisna od več atributov hkrati. Običajno govorimo o podedovanih (inherited) in pridobljenih (synthesised) atributih. Podedovani atributi so tisti, katerih vrednosti so podane neposredno s predhodnimi prireditvami atributov predhodnim produkcijam. Podedovani atributi se prenašajo po drevesu navzdol.

Pridobljeni ali sintetični atributi pa so tisti, katerih vrednost se izračuna na podlagi neke funkcije, ki sprejme podedovane atribute kot parametre. Pridobljeni atributi se prenašajo po drevesu navzgor. Značilno za njih je, da se v večini primerov "pridobijo" v terminalnih simbolih, tj. listih drevesa izpeljav.

Kljub temu, da lahko vsak atribut ovrednotimo (izračunamo njegovo vrednost) ni nujno da to lahko storimo kadarkoli. Preden lahko atribut pravilno ovrednotimo, mora ta imeti izračunane vse argumetne, na katerih njegova vrednost temelji. Povrhu tega lahko med atributi obstajajo ciklične odvisnosti, pri katerih vrednosti dveh ali več atributov zavisijo druga od druge. Ovrednotenje atributne gramatike je realizirano z obhodi semantičnega drevesa.

Z ozirom na zahtevnost ovrednotenja semantičnega drevesa klasificiramo atributne gramatike v razred S, razred L, neciklične ter absolutno neciklične gramatike [103].

Formalno definiramo atributno gramatiko kot kontekstno prosto gramatiko, pri kateri vsakemu produkcijskemu pravilu $A \in V_N$ priredimo množico semantičnih funkcij. Terminalnim simbolom $\alpha \in V_T$ priredimo množici podedovanih $S(\alpha)$ in pridobljenih atributov $I(\alpha)$, pri čemer velja $S(\alpha) \cap I(\alpha) = \emptyset$. Atribute gramatike definiramo na sledeč način:

$$\begin{aligned} A(\alpha) &= S(\alpha) \cup I(\alpha) \\ S &= \bigcup_{\alpha \in V_N} S(\alpha) \\ I &= \bigcup_{\alpha \in V_N} I(\alpha) \\ A &= I \cup S \end{aligned}$$

Prva definicija pove množico vseh atributov neterminala α, druga in tretja množici vseh pridobljenih in podedovanih atributov in zadnja množico vseh atributov gramatike.

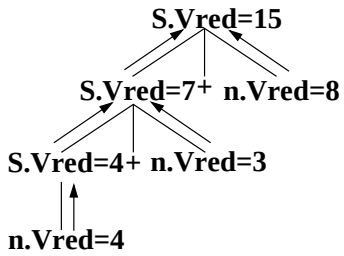
Atributi gramatike so povsem abstraktne podatkovne entitete, s katerimi lahko predstavimo povsem poljubne konkretne podatkovne strukture. Tako lahko atribut v listu drevesa, tj. v neterminalnem simbolu, predstavlja neko vrednost, atribut v neterminalu pa že celo drevo. Kadar je nek programski stavek uspešno razpoznan kot element jezika, mu pomen oz. njegovo interpretacijo določimo z ovrednotenjem njegovih atributov. V osnovi je semantična funkcija preslikava vrednosti množice atributov v

vrednost nekega novega atributa, kar lahko zapišemo $\alpha_a = f(a_1, a_2, \ldots, a_n)$, pri čemer α_a predstavlja vrednost atributa neterminalnega simbola α v odvisnosti od atributov $a_1, a_2, \ldots, a_n \in A(\alpha)$.

Že zgoraj smo omenili, da atributne gramatike delimo v več razredov. Oglejmo si sedaj to klasifikacijo malo podrobneje. Gramatika razreda S je vsaka atributna gramatika, ki uporablja izključno pridobljene atribute. Takšni atributi se, kot že rečeno, prenašajo po semantičnem drevesu od listov proti korenu, ali splošneje, od spodaj navzgor. Atributne gramatike razreda S zaradi tega veljajo za najpreprostejše. Izkaže se, da so takšne gramatike v praktični realizaciji jezikov precej pogoste. Ker je vsak atribut odvisen samo od pridobljenih atributov, ki so ob rekurzivnem ovrednotenju drevesnega vozlišča že izračunani, lahko atributno gramatiko te vrste ovrednotimo z enim samim obhodom semantičnega drevesa.

Atributno gramatiko uvrščamo v razred L, če vrednost poljubnega podedovanega atributa neterminala α_k zavisi izključno od vrednosti atributov (neterminalov), ki se nahajajo na levi strani, torej atributi ob $\alpha_0, \alpha_1, \ldots, \alpha_{k-1}$. Tudi gramatiko razreda L ovrednotimo z enim obhodom semantičnega drevesa.

Zadnja klasifikacija atributnih gramatik nas pripelje do necikličnih in absolutno necikličnih gramatik. Te gramatike omogočajo uporabo podedovanih atributov, katerih vrednosti so lahko odvisne tudi od atributov neterminalov, ki stojijo desno od neterminala α_k. Takšni atributi otežujejo evaluacijo semantičnega drevesa, saj je najprej potrebno izračunati soodvisnosti med atributi in šele nato opraviti pravilen obhod drevesa.



Slika 7: Semantično drevo s pridobljenimi atributi, ki se prenašajo po drevesu navzgor. Primer prikazuje ovrednotenje semantičnega drevesa z atributom $Vred$ za izraz 4+3+8.

Semantičnim funkcijam v takšnih gramatikah dodamo množico vseh atributov, ki se nahajajo na desni strani produkcijskega pravila. Ker lahko ima produkcija več semantičnih funkcij, je potrebno na nek način opisati odvisnosti med posameznimi atributi teh semantičnih funkcij. Te relacije opisuje usmerjen graf odvisnosti atributov, v katerem atribute predstavimo z vozlišči, odvisnosti pa z usmerjenimi povezavami. V kolikor obstaja v grafu odvisnosti za neko produkcijo cikel, pravimo da je atributna gramatika ciklična. Pri splošnih (poljubnih) stavkih jezika lahko odvisnosti med posameznimi atributi trajajo skozi več nivojev. V tem primeru uporabimo razširjene grafe odvisnosti, ki se določijo rekurzivno iz že izračunanih grafov odvisnosti. Cikličnosti atributne gramatike lahko sedaj dodamo absolutnost, če tudi v razširjenem grafu odvisnosti ne obstaja cikel.

Avtomatski generator razpoznavalnikov Yacc je zasnovan tako, da omogoča učinkovito uporabo atributnih gramatik vrste S, torej takšnih, ki v procesu ovrednotenja uporabljajo samo pridobljene atribute. Yacc omogoča tudi podedovane atribute, vendar je pri uporabi teh potrebno natančno poznavanje strukture pregledovalnikovega sklada. Ker lahko atributi v semantičnem drevesu zavzamejo poljubno množico vrednosti, Yacc omogoča definiranje produkcijskih pravil, ki vračajo uporabniško definiran tip ločene unije. Produkcijska pravila gramatike lahko torej operirajo z atributi tipov, ki so navedeni v tej uniji.

2.7.1 Denotacijska semantika

Denotacijska semantika je metodologija, temelječa na matematičnih osnovah, za določanje pomena programa [108]. V denotacijski semantiki pomen vsakega programskega konstrukta imenujemo denotacija. Pomen vsakega programskega konstrukta je določen s semantičnimi funkcijami. Prednost formaliziranja semantike z matematičnim modeliranjem se izkaže v možnosti razumevanja semantike programskega jezika kot celote, brez potrebe, da bi program dejansko izvršili na računalniku. Dodatna prednost uporabe denotacijske semantike je v sklepanju o programih, s čimer lahko dokažemo ekvivalenco in druge lastnosti. Karakteristična značilnost denotacijske semantike je v tem, da pripiše pomen vsakemu sintaktičnemu konstruktu programskega jezika in s

tem tudi programu kot celoti. Zaradi matematičnih osnov se je denotacijska semantika prvotno imenovala matematična semantika. Glede na osnovni cilj bi lahko denotacijsko semantiko poimenovali tudi pomenska semantika.

Denotacijsko semantiko uporabljamo za določanje osnovnih konceptov programskih jezikov, kot so okolje in povezave, pomnilnik, abstrakcije in parametri ter primitivni in sestavljeni podatkovni tipi. Pomen vsakega jezikovnega konstrukta predstavimo z določeno matematično entiteto, ki jo imenujemo denotacija programske fraze. Semantične funkcije programskega jezike opredelimo kot preslikave, ki slikajo programske fraze v pripadajoče denotacije.

Pomen vsake programske fraze je določen z vrednostjo v neki domeni. Za vsak razred fraz P določimo domeno denotacij tega razreda D in vpeljemo koncept semantične funkcije, ki preslika vsako frazo v razredu P v pripadajočo denotacijo v domeni D. To zapišemo $f : P \rightarrow D$. Semantična funkcija f je določena z množico semantičnih enačb za vsako obliko fraze v razredu P. Za zapis semantičnih funkcij se običajno uporablja zapis **let** izraz **in** izraz, s katerim vpeljujemo nove spremenljivke in matematične funkcije:

```
let myfunc(n) =
  if n <= 1  then 1
  else n * myfunc(n-1)
in myfunc(5)
```

Funkcija $myfunc$ ima preslikavo $N \rightarrow N$, če predpostavimo, da slika iz naravnega v naravno število. Z uporabo funkcijske notacije lambda računa lahko zapišemo tudi anonimne funkcije kot npr. $\lambda(a).\lambda(b).a + b$. V denotacijski semantiki uporabljamo domene zelo pogosto. Domeno lahko preprosto razumemo kot množico vrednosti, ki predstavljajo elemente te domene. V denotacijski semantiki uporabljamo domene primitivnih tipov (primitivne domene), katerih elementi so primitivne oz. enostavne vrednosti, ki jih ne moremo dalje razgraditi. Katere primitivne tipe uporabljamo, je povsem odvisno od področja uporabe denotacijske semantike. Načelno velja, da so primitivni tipi denotacijske semantike pogojeni z dejanskimi podatkovnimi tipi obravnavanega programskega jezika. Nad primitivnimi tipi nadalje definiramo domeno kartezičnega

produkta, ki ima za elemente urejene pare, ločene unije ter funkcijske in domene zaporedij.

Kadar denotacijsko semantiko uporabljamo za modeliranje izvajanja imperativnih jezikov, je potrebno zagotoviti izvajanje izrazov s t.i. stranskimi učinki. Stranski učinki v imperativnih jezikih pomenijo, da se zraven dejanskega ovrednotenja spremenijo tudi vrednosti spremenljivk, ki se v evaluaciji izraza referencirajo. Ker se spremenljivke navadno nahajajo v pomnilniku, je v denotacijsko semantiko potrebno vpeljati model pomnilnika. Pomnilnik lahko zelo abstraktno modeliramo kot zbirko pomnilniških lokacij s stanjem "prosto", "nedefinirano" in "definirano". Model pomnilnika opremimo še s funkcijami za alokacijo in sprostitev ter funkcijami za osvežitev in prenos s specificirane lokacije.

Pomemben koncept imperativnih programskih jezikov so deklaracije. Deklaracija vzpostavi povezavo med identifikatorjem in neko programsko entiteto. Programske entitete, ki jih lahko povežemo z identifikatorji, imenujemo povezljive vrednosti. Vsaka takšna povezava ima določen doseg. Dosege v večini imperativnih programskih jezikov določamo z različnimi oblikami blokov. Izvajanje programa poteka z vrednotenjem (sekvenčnim ali vzporednim) posameznih izrazov v določenem okolju. Treba se je zavedati, da lahko imamo iste identifikatorje povezane z različnimi entitetami v različnih okoljih. Katera povezava se bo izbrala v danem trenutku, je torej odvisno od trenutnega okolja. Okolje, podobno kot pomnilnik, modeliramo z določenim naborom funkcij. Običajno vpeljemo funkcije za formiranje praznega okolja, ustvarjanje povezav, konkatenacijo in prekrivanje okolij ter funkcijo, ki enoumno poišče določeno povezavo iz nekega okolja. Za določevanje semantike pravega imperativnega jezika potrebujemo tako model pomnilnika kot model okolja.

Za učinkovito razumevanje semantike večine jezikov je potrebno modelirati tudi abstrakcije. Abstrakcija je vrednost, ki je določena z nekim zaporedjem izračunov. Takšen izračun imenujemo abstrakcija zato, ker se vrednost preprosto izračuna s klicem abstrakcije, pri čemer dejanskega poteka izračuna ne vidimo. V imperativnih jezikih običajno uporabljamo funkcijske abstrakcije, ki vračajo neko vrednost in jih zato klasificiramo kot abstrakcije nad izrazi. Nasprotno pa so proceduralne abstrakcije abstrahiranje nad ukazi, saj je telo takšne abstrakcije eden ali več ukazov.

Zraven primitivnih tipov zelo pogosto uporabljamo še sestavljene podatkovne tipe, katerih modeliranje je bolj zapleteno. Vrednost sestavljene spremenljivke je namreč sama sestavljena iz več komponent. Dodaten problem pa nastane, ker je sestavljena spremenljivka lahko spremenjena na način, pri katerem spremenimo samo določene komponente, ostale pa ostanejo nespremenjene. Model pomnilnika je tem težavam ustrezno potrebno spremeniti.

2.7.2 Algebraična semantika

Algebraična semantika se uporablja za specifikacijo abstraktnih podatkovnih tipov. Ker vrednosti skupaj z operacijami predstavljajo algebro, je abstraktne podatkovne tipe najbolj naravno predstaviti prav s to matematično strukturo. Algebraična semantika uporablja specifikacijo abstraktnega tipa, teorijo, določeno s to specifikacijo, in algebre, ki zadoščajo takšni teoriji. Algebra za opis abstraktnega podatkovnega tipa je definirana kot trojček:

$$A =< C, c, f >,$$

kjer je C nosilna množica, c množica konstant in f množica funkcij. Algebro seštevanja in množenja naravnih števil torej zapišemo kot $< \{0, 1, 2, \ldots\}, +, * >$, kjer nosilna množica predstavlja vsa naravna števila $0, 1, \ldots$, $+$ in $*$ pa sta binarni funkciji, ki operirata nad elementi nosilne množice.

Specifikacijo abstraktnega podatkovnega tipa zgradimo iz signature in aksiomov, pri čemer signatura določa tipe, ki jih specificiramo, operacijske simbole in njihove funkcionalnosti. Aksiomi so logični stavki, ki določajo obnašanje operacij. Aksiomi so preprosto pogojne ali brezpogojne enačbe, ki opisujejo pravila, po katerih se vršijo definirane operacije. Abstraktni podatkovni tip, ki predstavlja tabelo, bi lahko torej opisali na sledeč način:

type `Tabela` **is**

operations:

```
nova(tabela) : Tabela
element(tabela,indeks) : Vrednost
vstavi(tabela,indeks,vrednost) : Tabela
```

axioms:

```
element(nova(_),_) = error
element(vstavi(T,i,v),j) =
   if i=j  then v  else
  element(T,j)
```

end

Algebraično semantiko uporabljamo tako za definiranje primitivnih kot tudi sestavljenih podatkovnih tipov. Sestavljene tipe imenujemo tudi parametrizirane ali generične tipe. Najpogostejši predstavniki sestavljenih tipov so urejeni pari, seznami, polja in preslikave. Podrobnejšo zasnovo algebraične semantike najde bralec v [108].

2.7.3 Akcijska semantika

Formalne metode semantičnih specifikacij so običajno osnovane na konceptih in pristopih, ki so precej oddaljeni od klasičnih operacijskih idej, s pomočjo katerih je pogojeno razumevanje programskih jezikov. Programski koncepti, kot so tok izvajanja, shranjevanje vrednosti in povezovanje, so v sklopu denotacijske semantike sicer natančno specificirani, vendar na matematičen in za računalnik neprimeren način. Akcijska (operacijska) semantika uvede za specifikacijo teh pojmov operacije abstraktnega stroja. Akcijska semantika je bila razvita z namenom, da bi semantične specifikacije programskih jezikov postale bolj berljive in sprejemljive. Semantika programskega jezika je v primeru akcijske semantike določena z akcijami ali operacijami, v katerih lahko vidimo neposredno povezavo s klasičnimi operacijskimi koncepti programskih jezikov. Tako imamo primitivne operacije, ki skrbijo za shranjevanje vrednosti, izdelavo povezav med identifikatorji in povezljivimi vrednostmi itd. Operacije lahko med seboj povezujemo in kombiniramo na različne načine. Ker so specifikacije, podane na ta način, zapisane v naravnem jeziku, so zelo dobro berljive, preproste in razumljive za širši krog ljudi. Zapisane so na modularen način, kar omogoča njihovo ponovno uporabo in preprosto upravljanje s spremembami, ki se pojavijo s spremembami v programskem jeziku. Akcijsko semantiko programskega jezika je treba razumeti na dveh abstraktnostnih stopnjah. Na vrhnji stopnji se nahaja specifikacija jezika z akcijami, na spodnji

stopnji pa so specificirane akcije same.

Akcija je operacija, ki se lahko ob uporabi podatkov iz drugih akcij izvede. Ko se akcija izvrši, se na izhod pošlje rezultat. Če se med izvajanjem pojavi napaka, se akcija zaključi nenormalno, drugače je izvršitev normalna. Seveda je povsem mogoče, da se akcija sploh ne zaključi. Za opis semantike uporabljamo primitivne in sestavljene akcije. Akcije preprosto poimenujemo s smiselnimi imeni, ki opisujejo njihovo nalogo. Primerka primitivnih akcij sta *zakljuci*, ki pomeni takojšnjo uspešno zaključitev in *napaka*, ki zaključi akcijo kot neuspešno. Sestavljene akcije zgradimo z uporabo akcijskih kombinatorjev. Naloga akcijskega kombinatorja je združitev dveh ali več akcij v kompleksnejšo akcijo in določitev vrstnega reda izvajanja podakcij. Primera akcijskih kombinatorjev sta izbira med dvema akcijama $A_1 or A_2$ in zaporedna izvršitev dveh akcij $A_1 and A_2$.

Podatke v akcijah predstavimo z abstraktnimi podatkovnimi tipi, ki jih lahko specificiramo z algebraično semantiko. Še več, celo akcije same lahko opišemo z abstraktnimi tipi. Iz poglavja o algebraični semantiki vemo, da je abstraktni podatkovni tip določen z množico vrednosti in naborom operacij, ki se nad temi vrednostmi izvajajo. V akcijski semantiki so operacije torej predstavljene kot operacije abstraktnega tipa, elementi tega tipa pa so akcije same. Pri modeliranju semantike programskega jezika je potrebno določiti vse potrebne skupine akcij, ki zadoščajo vsem programskim konceptom. Za imperativni jezik bo torej potrebno specificirati akcijske skupine za izraze, shrambo in deklaracije z upravljanjem povezav. V jezikih, ki omogočajo uporabo funkcij in procedur, je potrebna uvedba akcij za izdelavo funkcijskih in proceduralnih abstrakcij.

2.8 Povzetek

Bistvo in osnovna ideja uvodnega poglavja sta bila razumljiva in predvsem zgoščena predstavitev temeljnih konceptov teorije, ki stoji za načrtovanjem programskih jezikov. Principi in opisana teoretska dognanja zagotovo ne predstavljajo poglobljenega vpogleda v teorijo programskih jezikov, saj je ta za tovrstno diskusijo preobsežen in prekompleksen. Podrobnosti in dodatne razlage lahko najdemo v omenjeni literaturi.

Uvod smo pričeli z razumljivima definicijama jezika in gramatike, s katero jezik opišemo

na formalen način. Gramatike smo klasificirali v štiri razrede glede na omejitve v množici produkcijskih pravil. Nato smo podali predstavitveni metodi z metaopisnim jezikom BNF in sintaktičnimi drevesi. Uvodni del smo zatem konceptualno razdelili na leksikalni, sintaktični in semantični nivo, kot si sledijo v procesu implementacije jezika. Ob opisu leksikalnega nivoja smo se osredotočili na formalne opise končnih determinističnih avtomatov, ki se uporabljajo kot osnova v algoritmih leksikalnih analizatorjev. V okviru opisa leksikalne analize smo podali tudi idejo regularnih izrazov in poudarili njihovo uporabnost v specifikacijah, ki jih avtomatski generatorji pregledovalnikov sprejmejo na vhodu. Leksikalnemu nivoju je logično sledil sintaktični, kjer smo najprej podali nekaj osnovnih algoritmičnih pristopov razpoznavanja. Začeli smo z razpoznavanjem od zgoraj navzdol in nato nadaljevali s pristopi od spodaj navzgor. Pri obeh pristopih smo opisali osnovne principe delovanja razpoznavalnikov in morebitne težave, ki se lahko ob tem pojavijo. Kar zadeva avtomatske generatorje razpoznavalnikov, smo se omejili na orodje Yacc, ki generira razpoznavalnike razreda LALR(1). Odločitev je bila logična, saj bomo to orodje uporabili tudi v nadaljevanju. Zadnji del uvodnega poglavja smo zaključili z opisom semantičnih konceptov v načrtovanju jezikov. V okviru tega dela smo podali nekaj najosnovnejših principov, ki se v semantični analizi najpogosteje uporabljajo. Atributno gramatiko smo dobili na način, da smo gramatiko jezika obogatili z atributi, ki prenašajo semantično informacijo.

Opis jezikov na semantičnem nivoju smo zaključili z osnovnimi koncepti denotacijske, algebraične in akcijske semantike. Ob vsaki smo predočili njene bistvene prednosti, cilje in področje uporabe. S tem smo ugotovili, da je denotacijska semantika, ki temelji na matematičnih osnovah, primerna za napovedovanje obnašanja programov ter sklepanje o njihovih lastnostih. Algebraična semantika se skoraj izključno uporablja v domeni formalnih opisov abstraktnih podatkovnih tipov, medtem ko je akcijska semantika z uvedbo operacij abstraktnega stroja primerna za algoritmični in s tem razumljivejši opis semantike programskih jezikov.

3 Koncepti in razvoj objektno usmerjenega jezika

Pomen objektno usmerjenega pristopa k razvoju programskih sistemov ni v avtokratsko naravnani in danes že malodane despotski ideologiji trendovskega apliciranja principov "objektizacije" na manj in bolj primerna terišča, temveč leži drugje. Počelo vpeljevanja objektnega načela je najti v abstrahiranju nad modeliranjem strukture in interakcije. Abstraktne entitete, kot slike te strukture, niso nič drugega kot umišljenina idealnega modela realnosti, katerega interakcija preseže tisto v modelu. Objektizacija, kot jo interpretiramo na tem mestu, je interdisciplinarno drevo in nima nič skupnega z objektnim načrtovanjem, ki ga razumemo kot zgolj vejo v programskem spektru računalništva. Čeprav je drugo precej šibka podmnožica prvega, se zdi, da se je princip abstrahiranja z objekti najvidneje uveljavil prav na tem področju. Objektni vzorec načrtovanja, upodabljanja in abstrahiranja je tehnika konceptualnega modeliranja s principi poustvarjanja, kopiranja. Idealno gre zgolj za prevoj ustvarjalčevega gledišča, ob praktičnih aplikacijah pa običajno vodi do precejšnjih deformacij in alternacij.
Takšna razmišljanja nas vodijo v preučevanje smisla načel in principov, ki se pojavljajo v navezi z objektnim modeliranjem. Snovanja "objektno naravnanih" oz. "objektno usmerjenih" konceptov (v povezavi s sodobnimi spoznanji objektnega načrtovanja) se lahko lotimo z različnimi teoretsko orientiranimi in pragmatičnimi pristopi. Ker nas zraven povsem praktičnih vidikov načrtovanja in razvoja programskega jezika zanimajo tudi teoretični pogledi, bomo z namenom karseda celostne obdelave teme, v dobrem upu združili oboje.

V pričujočem poglavju želimo podati temeljne koncepte, s katerimi se soočamo ob problematiki načrtovanja objektno usmerjenih programskih jezikov. V opisih se bomo osredotočili in zaradi obsega tudi omejili na diskusijo osnovnih aspektov statično tipiziranih, razrednih, objektno usmerjenih programskih jezikov. Pomembno je, da vse komponentne programskega jezika razumemo kot interaktivno celoto, ki jezik na nek način identificira in mu določa atribute. Posebno pozornost bomo namenili sistemom tipov, ki predstavljajo osnovo za teoretiziranje in formalne opise objektno usmerjenih jezikov. Dovršeno razumevanje osnov sistemov tipov objektne usmerjenosti ni pomem-

bno zgolj za formalne zapise jezikov, temveč ima za posledico tudi bolj prilagodljive, skalabilne in izrazno močnejše programske jezike. Formalen opis je za objektno usmerjene jezike še posebej pomemben, saj je semantika teh jezikov precej zapletena že, če se omejimo le na mehanizme dedovanja in podtipiziranja. Sistemi tipov so v statično tipiziranih objektno usmerjenih jezikih še bolj ključnega pomena, kot pri dinamičnih (dinamično tipiziranih) jezikih, saj prav oni nosijo odgovornost za postavljanje ločnice med fleksibilnostjo in statično varnostjo. Cilj preudarnega načrtovanja "dobrega" objektno usmerjenega jezika, je omogočiti kar največjo izrazno moč (fleksibilnost sistema tipov) ob maksimalni varnosti uporabe tipov. Želimo torej največjo stopnjo dinamičnosti v statičnem tipiziranju. To lahko dosežemo samo z bogatim sistemom tipov, ki ni restriktiven glede izrazne moči in pogojuje minimalno uporabo dinamičnih pretvorb tipov. V kolikor smo se omejili na statično tipiziranje, mora biti takšen sistem tipov tudi preverljiv v času prevajanja. Da bi zagotovili varnost sistema tipov, je potrebno za tak sistem ustrezno dokazati, da nima napak, tj. da ne more priti do s tipom povezane napake v času izvajanja.

Ker lahko v jeziku implementiramo samo kanonične konstrukte, bo zajeten del diskusije namenjen tudi kompozicijskim mehanizmom, pri čemer se bomo posebej zavzeto usmerili k mehanizmom dedovanja in podtipiziranja. Dedovanje, kot inkrementalni modifikacijski mehanizem, je namreč primarni kompozicijski (strukturalni) mehanizem v objektno usmerjenih jezikih in ga s tega stališča razumemo kot "potreben" pogoj za objektno orientiranost. Namembnost dedovanja je zelo široka, saj omogoča praktično brezmejno uporabo faktorizacije in s tem ponovne uporabe programskih komponent. Pokazali bomo, da je dedovanje tesno povezano s podtipiziranjem in ga bo zato potrebno smiselno umestiti v sistem tipov programskega jezika. Proces dedovanja bomo predstavili v navezi s koncepti specializacije, tipiziranja in podatkovnega strukturiranja. S primerjalno študijo se bomo lotili opisa različnih vrst dedovanja in z njimi povezane problematike, ki jo bomo predstavili predvsem v luči zagotavljanja konformance tipov znotraj razredne hierarhije. Spoznali bomo, da je ohranjevanje konsistence tipiziranja objektov temeljna zahteva za pravilno delovanje programa. Kljub osredotočitvi na razredne objektno usmerjene jezike, bomo opravili tudi primerjavo z bolj dinamično naravnanimi tehnikami dedovanja, kot je delegacija.

V sklopu dedovanja bomo predstavili tudi koncept polimorfnega obnašanja objektov in funkcij. Tudi polimorfizem razumemo kot enega izmed ključnih mehanizmov objektno usmerjenih jezikov, saj je ravno s tem omogočeno abstrahiranje nad obnašanjem ter generalizacija in specializacija obnašanja. Polimorfizma se bomo sprva lotili v povezavi z dedovanjem, nato pa zlagoma prešli na generične abstrakcije, katerih se lotevamo s parametričnim polimorfizmom. Ker prav polimorfizem in dedovanje bistveno vplivata na ohranjanje varnosti tipov, se bomo zagotavljanja varnosti lotili nič prej kot po obdelanih osnovah obojega (dedovanja in polimorfizma). Podobno kot pri dedovanju, bomo tudi pri polimorfizmu obravnavali več različnih tipov.
Ker nas zanima predvsem konkretna implementacija jezika, bomo večino jezikovnih konceptov, ki že imajo dobro razdelano teoretično ozadje, predstavili tudi s povsem praktičnega, programerskega stališča, ki je zanimiv predvsem za načrtovalca jezika. V pričujočem poglavju se bomo seznanili z različnimi koncepti, tudi izključujočimi, posebno pozornost pa bomo usmerili k tistim, katere bomo dejansko implementirali v jeziku. Navkljub temu, da poglavje ni namenjeno dejanski implementaciji jezika, si bomo nekatere koncepte nazorno predočili tudi z implementacijskega gledišča. V osnovi nam bo besedilo tega poglavja služilo za izbiro in povezavo konceptov objektno usmerjenega jezika, ki ga bomo implementirali v naslednjem poglavju.

3.1 O objektih in razredih

Objektni model nas s svojo neproceduralno naravo navdušuje nad radikalnostjo sprememb, ki jih je doprinesel v nekdaj tradicionalni razvojni proces programskih sistemov. Objekti, kot jih razumemo v našem kontekstu, predstavljajo dogodke in entitete realnega sveta, med katerimi poteka interakcija. S programskega vidika objekti kapsulirajo stanje in vzorec obnašanja. Stanje je običajno predstavljeno z instančnimi spremenljivkami, obnašanje pa določeno z metodami, ki manipulirajo stanje. V objektnem modelu računamo z objekti, končni rezultat računanja pa predstavljajo stanja vseh sodelujočih objektov. Strukturalno lahko na objekt gledamo kot na skupek podatkov in metod. Objekt si lažje predočimo, če poznamo koncept abstraktnega podatkovnega tipa, ki ni nič drugega kot množica podatkov in procedur, ki operirajo nad

temi podatki. Razlika je na prvi pogled zelo zamegljena, vendar se je treba zavedati, da so abstraktni podatkovni tipi veliko preprostejši, saj ne poznajo pomena enkapsulacije s skrivanjem podatkov. Abstraktni podatkovni tipi prav tako ne poznajo ideje dedovanja, specializacije in konformance tipov. Podatke znotraj abstraktnega podatkovnega tipa lahko manipuliramo direktno ali posredno preko pripadajočih procedur, medtem ko lahko pri nekaterih objektno usmerjenih jezikih podatke spreminjamo izključno preko procedur. Pomembna je torej možnost skrivanja podatkov, do katerih želimo preprečiti dostop zunaj objekta. Ker objekti med seboj učinkujejo s pošiljanjem sporočil, lahko funkcionalno na objekt gledamo kot na abstraktni stroj, ki zna odgovarjati na določena sporočila. Sporočilo preprosto določa invokacijo specifične metode nad objektom. Množico sporočil, na katera zna objekt odgovoriti, imenujemo sporočilni vmesnik ali *protokol* objekta. Programski jezik Eiffel za označitev protokola uporablja *pogodbo*, medtem ko Java isto stvar imenuje kar vmesnik. V objektnem modelu procedure oz. funkcije, ki operirajo nad (enkapsuliranimi) podatki objekta, imenujemo metode. Objekt je abstraktna entiteta, ki jo ustvarimo spontano, brez posebne šablone, ali pa jo ustvarimo po nekem predpisu, tj. z uporabo neke šablone oz. načrta. V prvem primeru govorimo o objektnih jezikih, v drugem pa o objektno usmerjenih jezikih. Objektni jeziki poznajo torej samo objekte, katere ustvarjamo kot primerke in jih po potrebi kloniramo. Kloniranje je temeljna lastnost objektnih jezikov, saj omogoča inkrementalne (redkeje dekrementalne) spremembe že obstoječih objektov in s tem dedovanje.

Posebna veja objektnih jezikov so prototipni jeziki (prototype languages), v katerih ustvarjamo prototipne objekte in iz teh, s kloniranjem, povsem nove objekte. Objektni jeziki so povečini dinamični s stališča, da omogočajo dinamično spreminjanje instančnih spremenljivk. Zanimivejša lastnost je možnost dinamičnega spreminjanja metod, ki omogoča nadomeščanje obstoječe metode s povsem novo različico. Razlika med instančnimi spremenljivkami in metodami se tako praktično izniči. Klonirani objekti sicer lahko dinamično spreminjajo svoje stanje in metode, vendar njihova struktura ostane enaka. Da bi lahko dodali nove lastnosti, je potrebno uporabiti princip dedovanja. Le-ta se v objektnih jezikih razlikuje od klasičnega razrednega dedovanja iz preprostega razloga–ni razredov. Metode za izvedbo dodajanja novih lastnosti in

spreminjanja strukture objektov bomo podali v podpoglavju o dedovanju.

Veliko filozofsko naravnanih diskusij in razglabljanj obstaja o tem, kateri jeziki so "bolj" objektno usmerjeni [99]. Dejstvo je, da so razredni in objektni jeziki zelo podobni vsaj glede abstrahiranja in dinamičnih mehanizmov izbire metod. Objektni jeziki so zaradi pomanjkanja razredov preprostejši, saj imajo običajno manj sintaktičnih konstruktov za modeliranje objektov. Po drugi strani pa morajo objekti v teh jezikih prevzeti vlogo razredov in objektov in so v tem oziru semantično kompleksnejši od razrednih jezikov. K temu veliko prispeva tudi zgoraj opisana možnost spreminjanja tako instančnih spremenljivk kot metod. Kakršnekoli so že razlike, oboje lahko imenujemo "objektno naravnane", saj tako eni kot drugi omogočajo objekte. Prvi jezik, katerega lahko imamo za čisto objektno usmerjenega je po vsej verjetnosti Smalltalk [49], saj je prav ta, kot prvi uvedel popolno objektno abstrakcijo nad primitivnimi in kompleksnimi vrednostmi. Terminologija objektne usmerjenosti oz. objektne orientiranosti se je še iz časov Simule 67 [85, 60] oprijela predvsem razrednih jezikov, zato jo bomo v takšnem kontekstu uporabljali tudi mi.
Večina programskih jezikov, ki omogočajo objekte, je vendarle objektno usmerjenih. Ti jeziki za ustvarjanje objektov uporabljajo šablone, ki jih običajno imenujemo *razredi*. Razred je načrt oz. definicija objekta. Objekt za svojo instanciacijo (ustvaritev) potrebuje razred, ki predpisuje njegovo obnašanje in protokol. Ustvarjanje objektov poteka z uporabo operatorja *new*, ki v pomnilniku ustvari objekt želenega tipa in vrne referenco na ta objekt. Ker razredi omogočajo klasifikacijo, lahko rečemo, da je prav klasifikacija tista lastnost, ki ločuje objekte od objektno usmerjenih programskih jezikov. Razred je zelo podoben tipu, saj njegovo ime uporabljamo na vseh mestih, kjer uporabljamo ime tipa. Vendar medtem, ko je tip lahko povsem zaključen, je razred nepopoln, saj ga lahko spričo mehanizmov dedovanja spremenimo, izpopolnimo in specializiramo do neke "višje" oblike. Razred ima razširljivo implementacijo in vmesniško shemo. Kljub temu za razrede pravimo, da realizirajo neke konkretne podatkovne tipe. Praviloma bi morali ločevati med tipom in razredom, saj tip razumemo samo kot protokol, s katerim zagotavljamo konformanco, medtem ko razred, določen s svojim protokolom, vsebuje tudi dejansko implementacijo metod. Tip je torej nekaj bolj abstraktnega kot razred;

nanj gledamo kot na abstrakcijo, ki predstavlja množico vrednosti in operacij, ki jih lahko na tej množici apliciramo [22]. Razred je opisni mehanizem objektov, saj na nek način določa *družino* objektov, tj. objektov, ki si delijo značilnosti in imajo podobno naravo obnašanja. Razred tretiramo monomorfno, če predstavlja povsem konkreten tip za instance ustvarjene iz tega razreda. Nasprotno, pa je lahko razred tudi polimorfna (mnogolična) opisna šablona, če določa zgornjo (osnovno) mejo v hierarhični strukturi neke heterogene družine objektov. Slednje je dosegljivo z uporabo dedovanja. Programski jezik Beta [21, 81, 68] združuje razrede z metodami v t.i. vzorec (pattern), ki je samostojen jezikovni konstrukt. Beta v vzorcih omogoča tudi virtualne razrede ali virtualne vzorce, ki jih bomo podrobneje omenili v podpoglavju o polimorfizmu. Objektno usmerjeni jeziki, ki temeljijo na razredih, dedovanje dosežejo z grajenjem razrednih hierarhij. Razred lahko ima podrazred, ki ga imenujemo otrok. Razred, iz katerega nek razred izpeljemo, imenujemo starš ali starševski razred. Razred, ki se v razredni hierarhiji nahaja na nekem višjem nivoju, kot starševski razred, imenujemo predhodnik. Analogno poimenujemo razred na nižjem nivoju, kot je otrok, naslednik. V okviru tovrstne terminologije bi lahko starša in otroka poimenovali *neposredni* predhodnik in naslednik. V jezikih, ki omogočajo večkratno dedovanje, lahko ima razred več neposrednih predhodnikov, iz katerih deduje lastnosti.

Java ločuje med tipi in konkretnimi razredi in za (striktne) tipe uporablja vmesnike. Vendar se ta ločnica povsem zabriše, saj lahko na mestih, kjer pričakujemo tip, uporabimo tudi ime razreda. Razred kot specifikacija ni v takšnem primeru torej nič drugačen od razreda, na katerega gledamo z implementacijskega stališča. Java zahteva, da pripadnost oz. konformanco razreda nekemu vmesniku, določimo eksplicitno. Ker vmesnik natančno določa metode, ki se nad objektom lahko izvedejo, ga imenujemo tudi *objektni tip*. Objektni tip je običajno čisto abstrakten opis objekta in vsebuje samo nabor predpisanih metod in njihove signature v splošni obliki:

$$metoda_n \quad : \quad parameter_1 \times parameter_2 \times \ldots \times parameter_m \rightarrow tip$$

Objektni tip ne predpisuje instančnih spremenljivk, kar pomeni, da objekti s povsem različnimi stanji zadoščajo istemu objektnemu tipu, če le definirajo vsaj vse metode,

ki jih ta predpisuje.

Objekt je torej entiteta, ki predstavlja instanco konkretnega podatkovnega tipa in obstaja v času izvajanja. Vmesnik oz. protokol, ki ga objekt razume, pa lahko potemtakem razumemo kot abstraktni tip. Popolnoma definiran razred, torej takšen z implementiranimi metodami in protokolom, pa lahko imenujemo konkreten tip. Čeprav se tega povečini ne zavedamo, je večina objektov definiranih rekurzivno. Metode objekta lahko namreč invocirajo (kličejo) druge metode tega istega objekta, ob čemer mora biti objekt sposoben referencirati samega sebe, da bi lahko izbral primerno metodo [88].

Modeliranje "objektnih" družinskih dreves s hierarhičnim strukturiranjem razredov vnaša precejšnje komplikacije v zagotavljanje kompatibilnosti med posameznimi nivoji (generacijami) znotraj dreves. Če namreč želimo zagotoviti določeno mero varnosti tipiziranja v hierarhiji, je potrebno obstoječe relacije med posameznimi nivoji ohraniti. Pri tem mislimo relacije, vezane na signature metod v posameznih razredih hierarhije. Signatura je določena s tipi formalnih parametrov in tipom vračanja. Seveda je relacije, v katerih se ohranja konsistenca signature, smiselna samo pri preobteženih metodah. V nadaljevanju bomo spoznali spremembe tipiziranja, ki veliko prispevajo k izrazni moči jezika, a hkrati ohranjajo varnost signatur. Hierarhična zgradba objektov je posledica konstruiranja razredov z uporabo dedovanja, kar lahko poteka zelo dolgo. Proces izpopolnjevanja, dograjevanja in izboljšav je evolucija družine objektov, ki je pogojena s spremembami v razredni hierarhiji. Hierarhično strukturiranje je koncept, preko katerega bomo vpeljali vključitveni polimorfizem [109], ki ga zaradi relacije tipov v hierarhiji imenujemo tudi polimorfizem podtipa.

Ker je definicija objektov rekurzivne narave, morajo objekti imeti nek mehanizem samoreferenciranja. V objektno usmerjenih jezikih je to običajno izvedeno z uporabo reference *self* ali *this*, ki označuje trenutni objekt, nad katerim se izvaja metoda. Treba je vedeti, da lahko self vsebuje katerikoli objekt v objektni hierarhiji. Ta opazka nam da misliti, da je objekt, predstavljen z razredno hierarhijo, en sam celovit objekt, ali pa je sestavljen iz objektov vseh razredov. Self je dinamičen s tega stališča, da se spreminja v času izvajanja. Njegovo uporabo zasledimo tako v objektnih, kot objektno usmerjenih jezikih.

3.1.1 Objektna abstrakcija

Eden izmed najpomembnejših konceptov, ki so z objektno usmeritvijo doprinesli najbolj temeljite in radikalne spremembe v načrtovanje programskih sistemov, je modeliranje s pomočjo abstrakcije. Abstrahiranje je tisti načrtovalni vzorec, brez katerega si ni mogoče zamisliti sodobnega pristopa k razvoju kompleksne programske opreme. Abstrahiranje v najširšem pomenu omogoča inkrementalno načrtovanje od splošnega h konkretnemu. Objektno usmerjeni programski jeziki s svojimi koncepti ne samo omogočajo, ampak celo spodbujajo načrtovanje z uporabo abstrakcij.
Abstrakcija je tisto, kar obstaja pred specifično izvedbo, je torej preambula dejanski implementaciji. Vendar na abstrakcijo v okviru objektno usmerjenih jezikov gledamo bolj konkretno. Z objektno abstrakcijo definiramo behavioralni model, katerega realizira vsak pripadajoči objekt. Pripadajoči objekt je tisti, ki abstrakcijo konkretizira na način, da poda implementacijo vseh metod, ki jih abstrakcija predpisuje. V razrednih objektno usmerjenih jezikih so abstrakcije formirane z uporabo razredov; abstrakcijo v razrednem jeziku imenujemo preprosto abstraktni razred. Razred je nepopolno definiran, odprt ali abstrakten, če vsebuje vsaj eno abstraktno metodo. Abstraktna metoda je predstavljena samo s svojo definicijo in ne vsebuje konkretne implementacije. Abstraktna metoda poda torej samo signaturo, na način, kot smo ga prikazali zgoraj v povezavi z objektnim tipom. Jezik C++ abstraktne metode označuje z incializacijo na vrednost 0. Java za isti namen uporablja rezervirano besedo `abstract`. Abstraktna metoda je implicitno virtualna, saj se njena konkretna implementacija nahaja v enem izmed podrazredov. Navadno velja, da abstraktnega razreda ni moč instancirati, saj je njegova definicija nepopolna in njegov model obnašanja nedodelan. Programski jezik Java za razrede, v katerih so vse metode abstraktne, vpelje vmesnike, v katerih so metode privzeto abstraktne in jih ni možno implementirati:

```
interface A{
  public void print();
  public void draw();
}
```

Isto stvar lahko dosežemo s čistim abstraktnim razredom, kar pomeni, da Java za izdelavo objektne abstrakcije nima kanoničnega konstrukta. V načrtovanju našega

jezika se bomo poskušali držati kanoničnosti vseh programskih konstruktov, ki jih bo jezik omogočal.

```
abstract class A{
  public abstract void print();
  public abstract void draw();
}
```

C++ za opis čiste abstrakcije nima posebnega konstrukta, ampak uporablja razred, v katerem so lahko vse operacije definirane abstraktno. Razred je v C++ pravzaprav edinstveni in edini konstrukt, s katerim se izvaja definicija abstraktnih in konkretnih ter implementacija konkretnih tipov. Posledično se dedovanje uporablja kot metodologija ponovne uporabljivosti in hkrati služi kot edini mehanizem za modeliranje podtipov. Relacije med podtipi so identične relacijam med razredi. Uporaba razreda v tako široke namene povzroča precejšnje omejevanje izrazne moči samega jezika in omejuje fleksibilnost dedovanja pri ponovni uporabljivosti. Razred, iz katerega bi želeli dedovati zgolj zaradi uporabe obstoječe funkcionalnosti, ni nujno tudi smiselni nadtip novega razreda. Še več, behavioralno je lahko celo podtip.

Ker je abstrakcija tipa abstrahiranje nad nečim neobstoječim, še neimplementiranim, je smiselno za tovrstno abstrakcijo vpeljati poseben konstrukt. Konstrukt, ki nima povsem nič opraviti z implementacijo in konkretizacijo. C++ semantika tipiziranja ne loči med tipom in razredom ter abstrakcijo in implementacijo, kar je zelo omejujoče in neugodno. Tip, ki je semantično množica vrednosti, naj bo ločen od razreda, ki se naj uporablja samo in zgolj v implementacijske namene. V kolikor želimo omogočiti definicijo delnih abstrakcij oz. nedodelanih razredov, to trditev prikrojimo, da naj obstaja mehanizem ločevanja tipov od konkretizacij. Dodatna slabost združevanja tipa in implementacije v jeziku C++ se še očitneje pokaže v tem, da je tudi razred abstrakcije instanciran v času ustvarjanja objekta, čeprav, v kolikor predstavlja čisto abstrakcijo, sploh nima nobene funkcionalnosti. Tako se povsem po nepotrebnem ustvarja objekt brez implementacije. Javanski vmesniki omogočajo konstruiranje celih hierarhij tipov in podtipov, ki nimajo prav nič opraviti s samo realizacijo teh tipov. Težnja po takšnem ločevanju je še bolj opazna pri modeliranju polimorfnih tipov in podtipov. C++ zaradi opisanih omejitev seveda ne omogoča polimorfizma nad tipi ampak zgolj polimorfizem

nad konkretnimi implementacijami. Poudariti velja, da je ta omejitev prisotna samo pri vključitvenem polimorfizmu in ne npr. pri parametričnem polimorfizmu. Kakorkoli že, omenjena slabost je ugodno vplivala na kasnejše jezike, predvsem Javo, kjer je ta omejitev odpravljena. Tudi za C++ so se pojavile nekatere (nestandardne) rešitve, predvsem gre omeniti t.i. signature tipov [16, 15], ki so po svoji namembnosti zelo podobne tistim v ML-u [67, 77], razredom tipov [30] v Haskell-u [24] in definicijskim modulom v Moduli-2. Kljub temu, da je tovrstna razširitev jezika mogoča samo preko C-jevskega mehanizma predprocesiranja, pa vendarle predstavlja enega izmed boljših pristopov k razdvojevanju tipiziranja in razredov.

V okviru podatkovne abstrakcije pa lahko govorimo še o enem konceptu abstrakcije. To je enkapsulacija s skrivanjem. Gledano s strani evolucije programske opreme, je podatkovna abstrakcija nadvse uporaben način modularnega programiranja, še posebno s strani ponovne uporabnosti [18]. Omenili smo že, da je obnašanje abstraktnega podatkovnega tipa celovito definirano z množico abstraktnih operacij. Kako in na kakšen način te operacije dejansko delujejo ter v kakšni obliki je predstavljen abstraktni tip, za entitete zunaj tipa ni pomembno. Potemtakem lahko strukturo in implementacijske podrobnosti operacij povsem brez posledic skrijemo pred zunanjostjo. V postopku klasičnega enkapsuliranja uporabljamo tudi skrivanje na način, kot ga najdemo v mnogih praktično uporabnih objektno usmerjenih jezikih. Strog princip podatkovne kapsulacije zahteva, da je celotno stanje abstraktnega podatkovnega tipa oz. objekta zakrito in navzven nedostopno. Edini način, s katerim lahko izvajamo manipulacijo objektovega stanja, je uporaba metod (abstraktnih operacij), ki so del objekta samega in imajo zato tudi poln dostop do objektovega stanja. Ena izmed pomanjkljivosti Simule 67, kot prvega objektno usmerjenega jezika, je bila prav nezmožnost skrivanja stanja (instančnih spremenljivk) in metod. Zakrivanje je torej abstraktnostni aspekt, ki enkapsulaciji omogoča še višjo stopnjo minimizacije soodvisnosti med posameznimi moduli, saj se funkcionalnost le-teh ne more vezati na (skrito) strukturo odvisnih komponent. S takšnim pristopom prisilimo uporabo vmesnikov, katerih cilj je zagotoviti neodvisnost med implementacijami in strukturami programskih modulov.

3.2 Dedovanje

Če so tipi organizacijski princip nekih omejitev, je dedovanje proces postavljanja relacij med njimi. Relacije ne vzpostavljajo samo povezav med tipi, ampak omogočajo tudi faktorizacijo le-teh po abstraktnostnih nivojih. Splošno prepričanje, da je dedovanje primarni in osnovni kompozicijski mehanizem objektno naravnanih jezikov, zagotovo velja, saj se ga poslužujejo tako objektni kot objektno usmerjeni jeziki. Dedovanje s svojim principom odprtosti za razširjanje [1] teoretično omogoča neskončno faktorizacijo in posledično ponovno uporabnost programskega koda. Idealizirano bi bila evolucija programskih sistemov, načrtovanih z objektnimi paradigmami, zgolj inkrementalno spreminjanje funkcionalnosti, ki dejansko je atribut funkcije dedovanja. Pri večini jezikov na proces dedovanja gledamo kot na specializacijo neke že dodelane ali povsem abstraktne funkcionalnosti. To je torej mehanizem, s pomočjo katerega abstrakcije reificiramo, torej na nek način "usnovimo" in jim damo konkreten pomen. Čeprav je specializacijo relativno enostavno izdelati, sam postopek še zdaleč ni tako preprost. Specializacija namreč zraven funkcionalnosti zadeva tudi obnašanje objekta. Slednje je zelo težko učinkovito nadzorovati, saj na splošnejšem nivoju ne moremo natančno napovedati, kakšno bo obnašanje nižjih, konkretnejših nivojev. Obnašanje lahko do neke mere specificiramo z restrikcijo pri uporabi tipov, ki zahteva, da se na konkretnih ravneh uporabljajo samo bolj specifični tipi, kot so na nivoju abstrakcij. Specializacija tipov z globino omejuje splošnost in prinaša vedno striktnejše omejitve funkcionalnosti. Logika dedovanja je sprva (v zgodnjih letih objektnih jezikov) izhajala iz možnosti modeliranja konceptov neke aplikacijske domene. Dedovanje je bilo strukturalno sredstvo za konceptualno modeliranje oz. v modernejšem "odtenku", sredstvo za konceptualno specializacijo.

Dedovanje je verjetno najpomembnejša funkcija objektne orientiranosti, saj omogoča abstrahiranje in konkretizacijo, specializacijo z inkrementalnimi spremembami in nenazadnje tudi ponovno uporabljivost programskega koda. Hkrati z navedenimi prednostmi pa dedovanje vnaša tudi veliko komplikacij, ki se najprej pokažejo predvsem v semantiki jezika, kasneje pa v načrtovanju in nadzorovanju funkcionalnosti. Slednje je še posebej očitno pri razvoju kompleksnih sistemov, v katerih obstaja veliko odvisnosti

med posameznimi moduli (razredi), kjer je abstrakcija ključnega pomena. Vse prednosti dedovanja vplivajo na njene slabosti. Problemi, ki se pojavljajo pri dedovanju, niso povezani s samim konceptom, temveč s tem, da je kocept povezan in uporabljen za vse omenjene, tudi nezdružljive in izključujoče, funkcionalnosti.
Poleg klientov, ki nad objekti izvajajo metode, dedovanje prinaša še kliente, ki dedujejo iz starševskih razredov in s tem, zaradi odvisnosti, zategujejo morebitne spremembe starševskega obnašanja. Če se za trenutek omejimo na razrede in razredno dedovanje, lahko rečemo, da je vmesnik razreda enako pomemben za objekte tega razreda in njegove podrazrede. Vmesnik namreč med razredom in njegovimi podrazredi vzpostavlja nek protokol (pogodbo), ki določa in omejuje spremembe, dovoljene v podrazredih. Takšno omejevanje ima vpliv na celotno hierarhijo podrazredov. Zaradi učinkovitosti ima večina jezikov dodatne mehanizme dostopa do objektovega stanja. Restrikcija dostopa do stanja v podrazredih je običajno precej blažja, kot pri dostopu zunaj objekta. Java in C++ za dostop do starševskih instančnih spremenljivk v podrazredih uporabljata modifikator dostopa `protected`. Direkten dostop do starševskega stanja na ta način izniči bistveno prednost enkapsulacije in onemogoča vzdrževanje kompatibilnosti med razredom in iz njega izpeljanimi podrazredi. Načrtovanje s takšnimi pristopi zahteva sodelovanje izvajalcev nadrazreda in vseh podrazredov, kar vedno ni mogoče zagotoviti, še posebej če je na voljo samo prevedeni binarni kod. Boljša rešitev s strani ohranjanja konsistence bi bila uporaba vmesnika, ki definira izključno metode za dostop do stanja in ne vključuje instančnih spremenljivk. S takšnim pristopom bi ohranili vso možnost preimenovanja, dodajanja, odstranjevanja in skratka splošnega spreminjanja instančnih spremenljivk starševskega razreda brez vpliva na funkcionalnost izpeljanih razredov. Java in Smalltalk imata za invokacijo metode nadrazreda `super`, C++ pa uporablja resolucijski mehanizem imena nadrazreda. Klic metode specifičnega nadrazreda pomeni statično povezavo, pri kateri ni potrebno opraviti preverjanja v času izvajanja. To je razlog, zakaj je invokacija tako povezanih metod hitrejša [93]. Kljub temu, da se uporaba metod za dostop do stanj predhodnikov zdi primernejša v okviru ohranjanja kompatibilnosti, se lahko tudi tukaj pojavijo težave. Podrazredi, ki specializirajo ali pa zgolj uporabljajo obstoječo hierarhijo nadrazredov, se lahko vežejo na metode celotne hierarhije in tako ustvarijo odvisnosti med seboj in

nadrazredi na različnih globinah hierarhije. Spremembe na višjih ravneh takšne hierarhije niso več trivialne.

Dedovanje je povečini uporabljeno za izvedbo specializacije, katero po splošnem prepričanju sprejemamo kot omejevanje funkcionalnosti in dodajanje nekih bolj specifičnih, konkretnih lastnosti. Povsem sprejemljiva je tudi specializacija, v okviru katere nekatere lastnosti, ki so morda zaradi obsežnosti problema bile predvidene na abstraktnem nivoju, na konkretnem nivoju niso več aktualne in bi jih zato lahko odstranili. To bi zahtevalo mehanizem dedovanja, ki bi omogočal izločanje metod predhodnikov nekega razreda. Težava nastane pri statičnem zagotavljanju varnosti, saj vse metode definirane v nekem razredu niso nujno prisotne v vseh podrazredih. Po drugi strani pa bi takšno dedovanje lahko brez posebnih komplikacij implementirali v dinamičnih (dinamično tipiziranih) jezikih. Primer takšnega jezika je npr. CommonObjects [92] , objetkna razširitev jezika CommonLisp [94]. V jezikih, ki ne podpirajo izločanja operacij na ta način, lahko to dosežemo z eksplicitnim redefiniranjem metode, ki jo želimo izločiti. Implementiramo jo kot skrito (privatno) ali pa tako, da ne naredi nič.

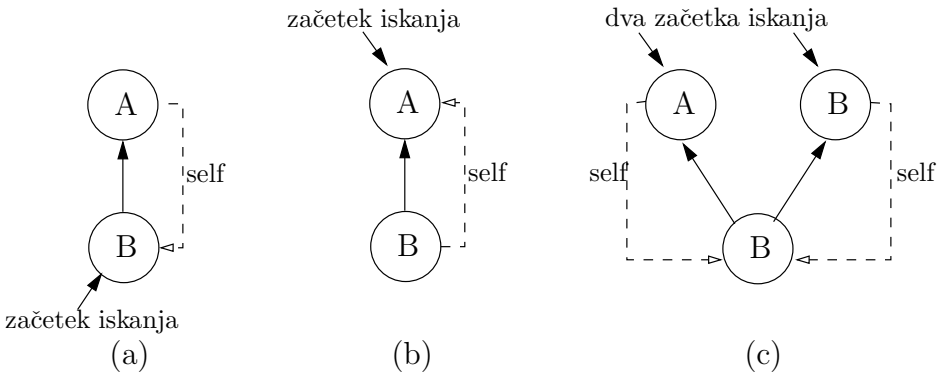
Dedovanje je v večini objektno usmerjenih jezikov tesno povezano s tipiziranjem. Razredna hierarhija je hkrati hierarhija tipov. Kadar dedovanje razredov povezujemo s tipi, mislimo običajno na podtipiziranje. S podtipiziranjem določamo pravila, s katerimi preverjamo konformanco med tipi oz. pripadnost tipa nekemu drugemu tipu. Pravila podtipiziranja so v statično tipiziranih jezikih izjemnega pomena, saj preverjajo pravilnost pretvorb med tipi že v času prevajanja. V podpoglavju o objektni abstrakciji smo omenili, da C++ ne podpira ločenega modeliranja razredov in tipov, temveč ta koncepta združuje. Uporaba dedovanja nad razredi za konstrukcijo hierarhije tipov vnaša v hierarhijo soodvisnosti. V primeru, da enega izmed nadrazredov zamenja nek drug razred, je konformanca hierarhije tipov porušena. Smotrno je, da se za grajenje hierarhije razredov in tipov uporablja dedovanje. Vendar naj bo dedovanje tipov osnovano na specializaciji obnašanja. Tipi so namreč čisto abstraktni koncept, kjer je ponovna uporabljivost povsem nesmiselna in odveč. C++ z združevanjem obojega povzroča nekompatibilnost med tipi in obnašanjem.

3.2.1 Vrste dedovanja

Klasifikacija dedovanja ni enolična, saj jo lahko vršimo po različnih kriterijih. Ker je dedovanje karakteristični atribut objektno usmerjenih jezikov, je njegovo klasificiranje še toliko pomembnejše, ne glede ali nanj gledamo bodisi kot na strukturalni ali modelirni mehanizem. Že na ravni jezikov lahko dedovanje razvrstimo v dedovanje nad razredi in dedovanje nad objekti. V razrednih objektno usmerjenih jezikih se dedovanje odslikava zgolj v razredni hierarhiji in nima skupne točke z instanco. Dejanski objekti – instancirani razredi se ne zavedajo svojega "družinskega drevesa", svojih prednikov in naslednikov. Instanca je nerazslojena homogenizirana entiteta, ki operira kot celota z enim modelom obnašanja in enovito funkcionalnostjo. Dedovanje nad razredi imenujemo razredno dedovanje. V objektnih in prototipnih jezikih nimamo mehanizma razredne kategorizacije, kar pomeni, da mora dedovanje biti realizirano na drugačen način. Objekti ali primerki, ki jih ustvarjamo s kloniranjem iz obstoječih objektov, imajo običajno možnost individualne modifikacije. Sprememba v razredu se odraža v spremembi sleherne instance tega razreda, medtem ko je sprememba v objektu povsem individualne narave, saj spremeni samo dejanski objekt oz. prototip. Prototipni jeziki torej omogočajo inkrementalne ali dekrementalne spremembe samo na nivoju posameznih eksistenc (objektov). Objektni jeziki realizirajo dedovanje s pomočjo delegacije ali kopiranja. Delegacijski mehanizem omogoča modifikacijo objekta s konceptom delegiranja. Kadarkoli objekt prejme sporočilo, na katerega ne zna odgovoriti, to sporočilo delegira v obravnavo svojim staršem. Postopek se ponavlja, dokler sporočilo ne prispe do objekta, ki nanj zna odgovoriti. Če takšnega objekta ni, pride do napake v delegaciji. Dedovanje s kopiranjem, kjer se vsi starševski atributi prekopirajo v nov objekt, kateremu dinamično dodamo nove metode, je konceptualno preprostejše od delegiranja, vendar zaradi kopiranja tudi počasnejše. Dedovanje s kopiranjem včasih imenujemo tudi dedovanje s konkatenacijo, kar je pogostejša terminologija v razrednih jezikih. Dedovanje s konkatenacijo je realizirano npr. v jeziku Beta, medtem ko najznamenitejši izmed prototipnih jezikov–Self [101, 29] dedovanje implementira z delegacijo. Dejstvo je, da je delegacija, kot mehanizem temelječ na pošiljanju sporočil, primernejša strategija za objektne jezike, medtem ko je konkate-

nacija oz. kopiranje primarna domena v razrednih jezikih. Obstajajo pa tudi nekatere rešitve za implementacijo delegacijskih mehanizmov v razrednih in statičnih jezikih [102, 44, 61]. Drug kriterij, po katerem lahko klasificiramo dedovanje, je začetek iskanja primerne metode. Ko pride do invokacije metode, je potrebno to metodo izvesti. Da se metoda lahko izvede, jo je potrebno najprej poiskati. Ena možnost je, da z iskanjem pričnemo v razredu (objektu), nad katerim metodo invociramo. Iskanje je uspešno, če najdemo tako metodo, katere signatura se ujema z metodo, podano v klicu. V kolikor metode ne najdemo, se iskanje premakne v nadrazred, kjer je postopek ekvivalenten. Takšen pristop uporabljata statično tipizirana jezika C++ in Java ter Smalltalk. Ker je rezultirajoča metoda odvisna od definicij, ki se nahajajo na najnižjem nivoju, tj. nivo zadnjega naslednika v hierarhiji, takšno dedovanje imenujemo nasledniško dedovanje [100]. Nasprotna shema poizvedovanja po metodi je starševska, kjer se iskalni proces prične na najvišjem nivoju hierarhije – v staršu. Takšen pristop uporablja jezik Beta, s čimer ohranja behavioralni model hieararhije, saj nasledniški razredi ne morejo nikoli povsem redefinirati starševskih metod. Starševski princip dedovanja je primeren za jezike, ki omogočajo samo enkratno dedovanje, torej kjer imamo vselej kvečjemu en starševski razred. Večkratno dedovanje takšno shemo zakomplicira, saj se lahko iskanje v tem primeru začne v več kot enem nadrazredu. Spoj starševskega in večkratnega dedovanja bi zahteval resolucijo s strani programerja. Primerjavo nasledniškega in starševskega dedovanja prikazuje slika 8.

Ko je metoda najdena in invocirana, z iskanjem običajno končamo. S stališča ohranjanja behavioralnega modela objekta to ni vedno zaželena lastnost. Funkcionalnost nekega modela, ki je realiziran kot hierarhija razredov ali objektov, je namreč lahko izvedena inkrementalno. Takšen način inkrementalne specializacije se kaže v tem, da vsaka, v podrazredu redefinirana metoda, na nek način izboljša obstoječo funkcionalnost z nekim novim znanjem, dodatkom. Če bi hoteli ohraniti celovito, globalno funkcionalnost, bi morali zagotoviti izvršitev vseh konkretnih metod v hierarhiji, ki ustrezajo klicani. V večini programskih jezikov lahko do tega pridemo z eksplicitno invokacijo istoimenske, bolj splošne starševske metode. Java in Smalltalk za dostop do metode starševskega razreda uporabljata super. C++, ki omogoča večkratno de-



Slika 8: Iskanje metode se pri nasledniškem dedovanju (a) prične v zadnjem razredu hierarhije, pri starševskem (b) pa v osnovnem (najvišjem) razredu. Konflikt v točki pričetka iskanja metode pri večkratnem starševskem dedovanju (c).

dovanje, to doseže z imensko resolucijo. V teh jezikih dedovanje, ki ohranja funkcionalnost na opisan način, simuliramo tako, da v vsaki metodi izvedemo eksplicitno izvedbo starševske metode. Če klic starševske metode postavimo na začetek metode, se nasledniško dedovanje prevede v logično starševsko, po drugi strani pa nasledniško dedovanje dosežemo z invokacijo starševske metode na strogem koncu metode podrazreda:

```
class A {
  public void do() {
   // operacije
  }
}

class B  extends A {
  public void do() {
    super.do();
   // operacije
  }
}

class C  extends B {
```

```
class A {
  public void do() {
   // operacije
  }
}

 class B  extends A {
 public void do() {
    // operacije
    super.do();
  }
 }

 class C  extends B {
```

```
    public void do() {
      super.do();
     // operacije
    }
  }
```

```
    public void do() {
      // operacije
      super.do();
    }
  }
```

Dedovanje pri katerem se implicitno izvede samo prva najdena metoda, se imenuje asimetrično dedovanje. Programski jezik Beta ne izvede samo prve najdene metode, ampak vse, ki ustrezajo signaturi. Takšno dedovanje imenujemo sestavljeno ali kompozitno, ker je sestavljeno iz invokacij večih metod.

Večina jezikov implementira dedovanje tako, da fiksira hierarhično strukturo nekega modela. To pomeni, da se struktura v času izvajanja ne more spremeniti, kar ima velik vpliv na statično preverjanje tipov v času prevajanja. Če bi lahko hierarhično zgradbo objekta v času izvajanja spremenili, torej če bi lahko dodali, spremenili ali izvzeli nek razred na poljubnem mestu v hierarhiji, bi morali vpeljati dinamično dedovanje. Zaradi nezmožnosti preverjanja napak v času prevajanja, pride takšno dedovanje v poštev samo za dinamične prototipne jezike. Dinamično dedovanje je realizirano z delegacijo, v kateri so reference na nadrazred običajno predstavljene s posebnimi režami, ki lahko svojo vrednost spreminjajo. Če je takšna reža, ki vsebuje referenco na nadrazred, dosegljiva v nekem podrazredu, lahko spremenimo celoten nadrazred. Nevarnost takšnega početja je takoj očitna, saj se s spremembami v nadrazredu poruši kompatibilnost strukture in funkcionalnosti v celotni hierarhiji pod spremenjenim razredom.

C++ in Java pri dedovanju privzameta, da se dedujejo vsi atributi in metode, ki so navedeni v nadrazredu. Nekega atributa torej ni mogoče izključiti iz množice, katero dedujemo. Nekateri jeziki([97]) omogočajo t.i. selektivno dedovanje, pri katerem se atributi, ki se bodo dedovali, eksplicitno navedejo. Selekcija atributov, ki se bodo prenesli v podrazred, lahko povzroča težave pri tipiziranju. Zaradi možnosti izključevanja nekaterih lastnosti, ki so že prisotne v nadrazredih, podrazred nujno več ne zadošča vmesniku, ki ga je postavil nadrazred. V podpoglavju o podtipih in tipiziranju bomo videli, da sistem tipov, ki omogoča alternacijo nadvmesnika (protokola starša), ni statično varen. Prav zaradi tega je takšna dedovalna shema primerna za dinamične

jezike.

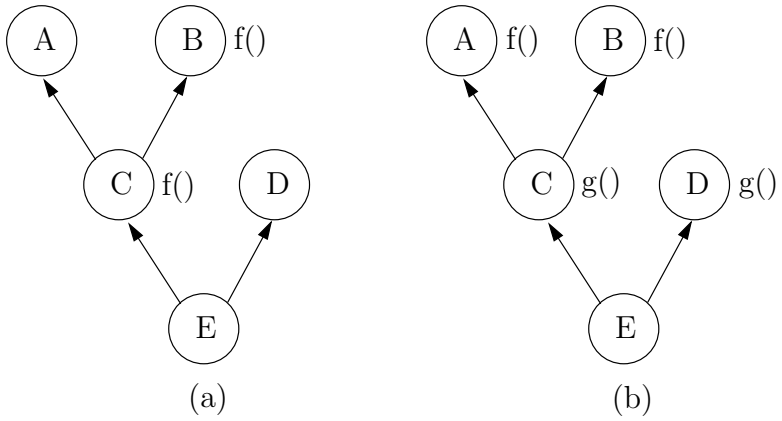
Dedovanje, ki omogoča najbolj splošno uporabo in funkcionalnost je verjetno t.i. mešano dedovanje (mixin inheritance) [45]. Pokazano je bilo, da lahko mešano dedovanje vključuje tudi funkcionalnosti drugih vrst dedovanja in ga zato tretiramo kot neko bolj osnovno oz. elementarno vrsto dedovanja [19]. Mešano dedovanje ima pomembno vlogo v pragmatičnem oziru, saj izboljšuje ponovno uporabljivost obstoječe funkcionalnosti [91, 89], hkrati pa zanj obstaja tudi že precej dodelana teorija [13, 12, 11]. Zaradi praktične komponente je bil koncept mešanega dedovanja implementiran v nekaterih jezikih, ki ga sicer v osnovi ne podpirajo [90].
Osnovni princip mešanega dedovanja je preprost. Inkrementalna modifikacija funkcionalnosti se ne nahaja v izpeljanem delu, temveč je predstavljena samostojno. V primeru razrednega jezika je inkrementalni del realiziran z lastnim razredom s pomočjo katerega izdelamo razred z novo, dedovano funkcionalnostjo. Takšen razred bi lahko imenovali vključitveni ali inkrementalni razred. Sintaktično gledano je takšen razred povsem enak klasičnemu razredu, razlika je le v semantiki. Vključitvenega razreda ne moremo instancirati, saj sam po sebi ne definira neke zaključene funkcije. Ker je takšen razred predstavljen kot neodvisna celota in ker nima starševskega razreda, ni v strukturalnem smislu vezan na nobeno partikularno mesto v hierarhiji in je potemtakem lahko uporabljen večkrat in na različnih mestih, kjer je inkrement njegove funkcionalnosti pač potreben. Na ta način se ob dedovanju izognemo vključevanju nepotrebnih atributov in povečamo možnost ponovne uporabljivosti. Ker vključitvenega razreda ni mogoče instancirati, a je hkrati uporabljen kot podrazred, ga imenujemo tudi abstraktni podrazred. Mešano dedovanje je lahko izvedeno enkratno ali večkratno, pri čemer je slednje primerno za sestavljanje inkrementalnih delov. Pri vključevanju večih abstraktnih podrazredov ti služijo za izdelavo kombinacij funkcionalnosti in njihovo združitev v podrazredu. Čeprav ima mešano dedovanje precej prednosti, povzroča tudi težave. Te se najvidneje kažejo v uporabi konstrukta razreda za realizacijo konceptov z bistveno različnimi nalogami – osnovnih in vključitvenih razredov. Zaradi tega se zmanjša razumljivost konceptualnega modela hierarhije. Koncept mešanega dedovanja je v nekaterih jezikih apliciran na celotne module. Takšne module imenujemo

mešane module (mixin modules) [41, 53]. Mešani moduli so pomembni, ker omogočajo parametrizacijo rekurzivnih definicij tudi izven posameznih modulov [33]. Pravtako je na mešan modul (razred) možno aplicirati koncept generalizacije [9].

3.2.2 Večkratno dedovanje

Sposobnost večkratnega dedovanja ima velik vpliv na konceptualno modeliranje realnih problemov z objektno usmerjenimi koncepti. Večkratno dedovanje poenostavlja modeliranje s kombinacijo obstoječih funkcionalnosti in izboljšuje stopnjo ponovne uporabljivosti programskega koda. Spričo omenjenih prednosti je tovrstni princip dedovanja dobil pomembno vlogo tako s teoretičnega kot praktičnega gledišča. Ne glede na očitne pozitivne lastnosti, se je uveljavilo v razmeroma majhnem številu jezikov. Vzroke za to gre iskati v relativno zapletenem semaničnem modelu [27, 8, 32] in sami implementaciji. Ker večina sodobnih objektno usmerjenih jezikov, med njimi tudi Java in C#, večkratnega dedovanja ne podpirajo, je bila izdelana vrsta raziskovalnih študij za razvoj alternativnih pristopov večkratnemu dedovanju [102, 17]. Večkratno dedovanje zahteva časovno neugodno analizo tako v času prevajanja kot izvajanja. Težave v prevajalnem času so semantične narave in se pojavijo predvsem zaradi dvoumnosti metod in atributov. Enkratno dedovanje takšnih problemov nima, saj se navadno izbere metoda, ki se najde prva. Torej tudi v primeru vertikalnega prekrivanja je resolucija izvedljiva nedvoumno, ne glede ali gre za nasledniški ali starševski model dedovanja. Pri večkratnem dedovanju lahko obstaja več različnih poti od točke referenciranja metode do njene definicije. Tovrstne definicije se prekrivajo horizontalno (slika 9). Resolucija takšnega referenciranja je dvoumna in mora biti razrešena s strani programerja. Shemo večkratnega dedovanja običajno predstavimo z usmerjenim acikličnim grafom (directed acyclic graph), v katerem so posamezni razredi predstavljeni kot vozlišča, relacije med njimi pa z usmerjenimi povezavami.

V splošnem obstajata pri večkratnem dedovanju dva pristopa za razreševanje konfliktov, ki nastanejo pri referenciranju metod in atributov. Prvi pristop izpeljuje semantiko neposredno iz grafa dedovanja, kar pomeni, da graf ostane nespremenjen. Operacije oz. metode se po grafu dedujejo tako dolgo, dokler niso redefinirane v nekem podrazredu.



Slika 9: Prekrivanje definicij metod f() in g() pri večkratnem dedovanju. (a) vertikalno in (b) horizontalno prekrivanje.

Če razred deduje operacije z identično signaturo iz večih staršev (različne poti grafa), je potrebno konflikt razrešiti. Eden izmed načinov je preimenovanje konfliktnih operacij v podrazredu. Preimenovana operacija lahko invocira operacije starševskih razredov. Tehniko preimenovanja uporablja Eiffel. V večini jezikov takšen konflikt rezultira v napaki pri prevajanju, Smalltalk pa ustvari operacijo, ki ob invokaciji signalizira napako. Glede zagotavljanja kompatibilnosti razredne hierarhije se večkratno dedovanje sooča s podobnimi problemi kot enkratno. Ker je dedovanje integralni del zunanjega vmesnika, je potrebno zagotoviti nespremenjeno strukturo hierarhije. Če namreč v nekem trenutku v povsem pravilni, nekonfliktni hierarhiji zamenjamo nadrazred z drugo implementacijo, lahko v podrazredih pride do nepredvidenih konfliktov. Ker grafa dedovanja ne spreminjamo, temu pristopu pravimo tudi neurejeno dedovanje.
Drug pristop razreševanja konfliktnih situacij je vezan na linearizacijo grafa dedovanja. Princip je zelo preprost. Graf dedovanja se najprej linearizira v verigo, kjer je vsak razred v grafu predstavljen kot člen. Ko je graf lineariziran, ga lahko obhodimo na enak način, kot pri enkratnem dedovanju. Pri tem je seveda potrebno ohraniti vrstni red razredov v originalni hieararhiji. Težava se pojavi, kadar nek razred preslikamo v nadrazred razreda, ki je pred linearizacijo imel drug nadrazred. Dejanski starš tako torej ni nujno več enak, kot pred linearizacijo. Na nek način je pač potrebno razvrstiti starševske razrede. Pri konfliktnem referenciranju neke metode se konflikt sicer razreši,

vendar je izbira metode odvisna od ureditve neposrednih nadrazredov. Drug problem uvidimo pri komunikaciji razreda z njegovim neposrednim predhodnikom. Če se v ta namen uporablja resolucija tipa super, kot v jezikih Java in Smalltalk, se semantika komunikacije spremeni. Imenska resolucija, kot jo uporablja C++, je v takšnem primeru bolj ugodna.
V splošnem lahko rečemo, da je razreševanje konfliktov lažje pri lineariziranih grafih, saj se konfliktne situacije preprosto izničijo. Horizontalnega prekrivanja ni več. Vendar pa je pristop, ki za razreševanje uporablja graf dedovanja v njegovi primarni obliki, bolj eksakten, saj ne dopušča, da bi prišlo do zakritih, nenačrtovanih invokacij.

Večina objektno usmerjenih jezikov je implementiranih tako, da se selekcija metode izvede na podlagi prejemnika sporočila. Takšnen pristop imenujemo enkratno razpošiljanje (single dispatch). Nekateri jeziki, predvsem dinamično tipizirani, omogočajo, da se selekcija metode v času izvajanja opravi na podlagi tipa prejemnika, sporočila in parametrov. To so jeziki z večkratnim razpošiljanjem (multiple dispatch). Invokacija z večkratnim razpošiljanjem je bolj striktna, saj je metoda definirana za natančno določene argumente. Ker lahko metode istočasno pripadajo različnim razredom, se imenujejo multimetode. Kljub temu, da večkratno razpošiljanje v določenem pogledu krepi izrazno moč, takšni jeziki niso posebej priljubljeni, saj kršijo nekatere temeljne prvine objektne orientiranosti. Pri klasičnih jezikih z enkratnim razpošiljanjem se operacije definirajo skupaj z abstraktnim podatkovnim tipom, ki s tega stališča predstavlja enkapsulacijo podatkov in funkcionalnosti. Ker se multimetode razpošiljajo z različnimi argumenti, niso vezane na posamezen abstraktni podatkovni tip, s čimer kršijo pravila enkapsulacije. Večkratno razpošiljanje je bolj kot objektno usmerjenim jezikom bliže proceduralnim in funkcijskim. Najbolj znan jezik s konceptom multimetod je CLOS. Čeprav večina jezikov ne podpira multimetodnega pristopa, se predvsem v zadnjem času pojavljajo ogrodja, ki to kljub omejitvam originalnega jezika omogočajo. Implementacija takšnega ogrodja za Javo je navedena v [43].

3.2.3 Virtualni mehanizmi

Najmočnejši modelirni mehanizem objektne usmerjenosti je abstrahiranje v opisu in funkcionalnosti. V času podajanja abstraktnega opisa specifičnosti niso znane. Te se dodelajo kasneje, večkrat v povsem drugih razvojnih okoljih in z drugačnimi nameni. Obstoječa funkcionalnost pa se ne veže na konkretne izvedbe, temveč na abstrakcijo. V razrednih jezikih je potrebno na nek način zagotoviti konsistenco med razredi, ki se zanašajo na signaturo abstraktnih razredov in njihovimi dolgoročnimi konkretizacijami. To je še posebej pomembno v prevedenih jezikih, kjer abstraktni razredi niso nujno na vpogled – zanašati se gre zgolj na njihov zunanji vmesnik. Tehnika, ki preko abstraktnih metod omogoča transparentno invokacijo njihovih konkretiziranih različic, se imenuje pozno povezovanje [36]. Za jezik, ki omogoča pozno povezovanje, pravimo, da vsebuje virtualne mehanizme. Pozno povezovanje pomeni, da natančni naslovi metod niso določeni v času prevajanja, temveč se dobijo svoje prave vrednosti šele ob instanciranju konkretnega razreda. Ker abstraktne metode, vsaj v okviru razrednih jezikov, svoj smisel dobijo v konketnih definicijah v podrazredu, je virtualni mehanizem tesno povezan z dedovanjem, saj brez njega pravzaprav ne bi obstajal. Zaradi svoje nepogrešljivosti je virtualni mehanizem postal eden izmed temeljnih konceptov objektno usmerjenih jezikov.

Virtualno invokacijo lažje razumemo v povezavi s polimorfnimi funkcijami. Polimorfne ali mnogolične funkcije oz. metode so tiste, katerih obnašanje ni vedno enako. Polimorfne funkcije imajo enako ime, njihova funkcionalnost pa je odvisna od konteksta v katerem se izvajajo. Nazoren primer polimorfizma je funkcija za seštevanje dveh entitet. Takšna funkcija tipično operira nad argumenti celih, realnih in celo znakovnih tipov. Kadar govorimo o virtualnih mehanizmih, ki operirajo s polimorfnimi funkcijami, mislimo na vključitveni polimorfizem, ki pride do izraza predvsem pri dedovanju. Koncept parametričnega polimorfizma je do neke mere podoben vključitvenemu, vendar se njegova uporaba pokaže drugje. Čeprav bi si želeli statično tipiziran jezik, virtualni mehanizem zaradi svojih temeljev v dedovanju zahteva tudi dinamično preverjanje tipov. Virtualni klici metod se lahko izvajajo z enkratnim ali večkratnim dedovanjem, pri čemer slednje povzroča precej težav, saj struktura objektov v tem primeru ni več triv-

ialna.

Resolucijo virtualnega klica metode v objektno usmerjenih jezikih imenujemo razpošiljanje sporočila. Razpošiljanje predstavlja funkcijo, ki prejme ime sporočila in prejemnika. Ime sporočila in prejemnik predstavljata selektor na podlagi katerega se pokliče metoda, ki ustreza temu paru. Za iskanje ustrezne metode se najpogosteje uporabljajo tabele razpošiljanja (dispatch tables). Te so lahko generirane že v času prevajanja ali pa med samim izvajanjem. Slednje je primerno za dinamično tipizirane jezike. Dinamični prototipni jeziki za iskanje metode uporabljajo enake pristope kot so tisti za resolucijo metod pri dedovanju. Seveda je dejansko iskanje metode v času izvajanja precej počasnejše kot direktna invokacija preko vnaprej generiranih tabel. Nekateri jeziki zato uporabljajo različne tehnike predpomnjenja, ki iskanje metod precej pohitrijo [106].

Statične tehnike podatkovne strukture za iskanje metode izračunajo vnaprej v času prevajanja ali povezovanja. S tem se čas, potreben za resolucijo metode v času izvajanja krepko zmanjša. Ker naslovi metod v času prevajanja niso znani, se metode referencirajo z indeksi. Dejanski naslov posamezne metode se določi šele v času povezovanja ali izvajanja in se shrani na eno izmed lokacij v tabeli razpošiljanja. Ko se metoda pokliče, se izvajanje prenese na naslov, ki se nahaja v tej tabeli na indeksu metode. Tabela razpošiljanja je potrebna samo za dinamično povezane metode, statične metode se povežejo že v času prevajanja. Najpreprostejša izvedba klica dinamične metode je z uporabo tabele, katero naslavljamo s prejemnikom in selektorjem (indeksom). Slabost takšne predstavitve je visoka prostorska zahtevnost tabele. Če imamo $r \in [1..n]$ razredov in $s \in [1..m]$ selektorjev je velikost tabele $O(r * s)$. Ker je večina sporočil definirana le na nekaterih izmed razredov r, je tabela zelo razpršena, kar povzroča slabo prostorsko izkoriščenost. Ne glede na hitrost dostopa do naslova metode, ki je enaka za statično in dinamično tipiziranje, se takšne tabele ne uporabljajo prav pogosto.

Najpogostejša tehnika za resolucijo virtualnih metod uporablja virtualne tabele razpošiljanja (virtual dispatch tables), ki so bile prvič predstavljene v Simuli. Danes ta pristop uparablja večina statično tipiziranih jezikov, tudi Java in C++ [95]. Virtualna tabela določa selektorje metod povsem lokalno, znotraj enega razreda. V primeru enkratnega dedovanja se selektorji označujejo zaporedno od najvišjega do najnižjega razreda v hierarhiji. V razredu, ki razume m sporočil, so torej selektorji označeni z

$0..m$. Vsak razred ima prirejeno svojo tabelo razpošiljanja velikosti m. Ker se selektorji sporočil referencirajo s številkami, morajo vsi podrazredi za sleherno dedovano metodo uporabljati enak selektor. Postopek razpošiljanja najprej naloži prejemnikovo tabelo razpošiljanja, katere indeksi referencirajo dejanske pomnilniške naslove metod.
Večkratno dedovanje otežuje številčenje selektorjev. Ker lahko posamezen razred deduje iz večih staršev hkrati, se lahko različnim sporočilom priredi enak selektor, kar je napačno. Ta problem se razreši tako, da se za posamezen razred uvede več virtualnih tabel. C++ uporablja originalno tabelo za razred, ki deduje in prvi razred, iz katerega deduje, za vse ostale razrede pa se generirajo dodatne virtualne tabele. Šibka stran virtualnih tabel je njihova odvisnost od statičnega tipiziranja. Brez poznavanja množice sporočil, ki jih objekt lahko sprejme, ne moremo enoumno določiti številk selektorjev za vsa možna sporočila. To je problem dinamično tipiziranih jezikov. Metod, kot so barvanje selektorjev, odmik vrstic in nekaterih drugih v pričujočem kontekstu ne bomo posebej obravnavali.

3.3 Pomen tipov

Kakorkoli že gledamo na tip, kot na abstrakcijo opisa ali na vmesniški mehanizem, je to koncept, s katerim želimo zagotoviti varnost izvajanja operacij. Mehanizem, ki usklajuje in preverja smiselnost operacij nad določenimi tipi, se imenuje sistem tipov. Ker vsak tip nima definiranih vseh operacij, tip razumemo kot koncept za omejevanje funkcionalnosti. Zelo verjetno bi bilo nespametno deliti znakovni niz s številom 2. Prav z restrikcijo operacij, dobijo objekti svoje značilke, preko katerih komunicirajo z drugimi objekti. Jeziki, ki funkcionalnosti objektov ne ločujejo z njihovimi tipi, so šibko tipizirani. Šibka tipiziranost pomeni logično nekonsistenco v interakciji med posameznimi objekti, saj nimamo nikakršnega načina, s katerim bi zagotovili pravilnost operacij med temi objekti. Ne samo razred, tudi tip, kot organizacijski princip, služi za klasifikacijo objektov. Osnovna razlika je le, da opravlja razred klasifikacijo na nivoju abstraktnih podatkovnih tipov, obnašanja in na strukturalnem nivoju, medtem ko tip klasificira funkcionalnosti objektov v času izvajanja, interakcije. Z dobro načrtovano strukturo tipov omogočimo minimizacijo in lokalizacijo odvisnosti med posameznimi

abstrakcijami. Ne potrebujemo natančne strukture objekta ali njegovih implementacijskih nadrobnosti, dovolj je, da poznamo tip tega objekta, saj nam samo ta zagotavlja natančno specifikacijo objektovih lastnosti.

3.3.1 Varno tipiziranje in sistemi tipov

Osnovna naloga sistema tipov je torej zagotavljanje varnosti na način, da se nesmiselne (napačne) operacije preprečijo že v času prevajanja in do njih v času izvajanja sploh ne more priti. To velja v primeru statično tipiziranih jezikov, kjer se legalnost dejansko preveri v prevajalnem času. Sleherni izraz ima prirejen tip, ki v operacijah nad temi izrazi igra ključno vlogo. Z zagotovitvijo kompatibilnosti med tipi lahko preprečimo napake v času izvajanja, vendar s stališča izraznosti in svobode (statični) sistem tipov pomeni precejšno omejitev, ki se včasih izkaže za nedopustno. Po drugi strani pa preverba v času prevajanja izboljša učinkovitost sistema tipov, saj ima ta bistveno preprostejšo opravilo ob izvajanju. Nasprotno pa dinamično tipizirani jeziki, kot sta npr. Lisp in Smalltalk, preverbo tipov opravijo neposredno pred samo izvedbo operacije. To pomeni, da lahko v času izvajanja pride do napake. Kjer za zagotovitev pravilnosti operacij nad različnimi tipi nimamo dovolj informacij, uporabljamo sklepanje o tipih (type inference), s pomočjo katerih lahko pravilni tip izpeljemo. Sklepanje o tipu je primarni sistem tipov v jeziku ML [40]. Sistem sklepanja je v objektno usmerjenih jezikih lahko zelo kompleksen, saj mora poleg klasičnih razširitvenih in zožitvenih kriterijev nad integralnimi tipi dosledno upoštevati tudi hierarhično organiziranost abstraktnih tipov, t.j. razredov ali objektov. Razumljivo je, da so statično tipizirani jeziki s tega stališča, pa tudi zaradi možnosti izvajanja optimizacije, hitrejši, kot dinamično tipizirani. Jezikom, v katerih je vsaka vrednost določena s pripadajočim tipom, pravimo močno tipizirani jeziki.

Glede na omenjeno, lahko rečemo, da so statično tipizirani jeziki varnejši in bolj učinkoviti, dinamično tipizirani pa bolj prilagodljivi in običajno tudi izrazno močnejši, saj je prireditev tipa vsaki entiteti že v času prevajanja včasih preveč omejujoča. Večina modernih jezikov je statično tipiziranih, med katere spadata tudi Java in C++. Kljub varnosti statičnega sistema tipov teh jezikov, ostaja vprašanje pretvorb tipov po razrednih hierarhijah odprto. Dejstvo je, da je poleg statičnega preverjanja v času prevajanja, na nekaterih mestih potrebno tudi preverjanje v času izvajanja. Striktno statično tip-

iziranje zato razrahljamo z zahtevo, da mora sistem tipov za vsak izraz zagotaviti skladnost tipa, čeprav sam tip v času prevajanja še morda ni znan. Java in C++ ta problem rešujeta z dinamičnimi pretvorbami tipov (type casts). Tako lahko pride do pretvorbene napake tudi v izvajalnem času. V sklopu tega dela bomo največ poudarka namenili statično oz. hibridno tipiziranim jezikom.

Hibridno tipizirani jeziki so statično tipizirani z dodatnimi mehanizmi preverjanja v času izvajanja. Za poizvedovanje o tipih programskih entitet v času izvajanja uporabljajo jeziki različne konstrukte. Javi v ta namen služi konstrukt `instanceof`, s katerim se preveri dejanski tip entitete. Operator instanceof vrne logično vrednost true, če je entiteta instanca tipa, podanega kot argument. V kolikor bi se preverba tipa z omenjenim operatorjem izvedla ob vsaki pretvorbi, bi Java bila povsem statično varna. O varnosti javanskega sistema tipov, statičnosti in dinamičnosti je bilo napisanih precej razprav [38, 37, 39], ki z bolj in manj formalnimi pristopi dokazujejo svoje izsledke. C++ ima za ta namen operator `typeid`, za varno (in smiselno) pretvorbo tipov po razredni hierarhiji pa operator `dynamic_cast`. Vendar kljub konstruktom, ki omogočajo testiranje dejanskih instanc razredov in varne pretvorbe, jezika C++ še vedno ne moremo imeti za varnega, saj je uporaba omenjenih mehanizmov povsem manualna. Če bi na nek način lahko zagotovili, da bi C++ ob vsaki pretvorbi polimorfnega tipa opravil preverbo z operatorjem dynamic_cast in ob napaki ustrezno ukrepal, bi tudi C++ lahko imeli za varen jezik.

Opozoriti velja, da je preverjanje tipov potrebno samo pri razširjanju v podtip. Ena izmed temeljnih lastnosti pri podtipiziranju je ta, da lahko podtip stoji na vsakem mestu, kjer pričakujemo nadtip. Podtip ima polno funkcionalnost svojega nadtipa, obratno pa seveda nujno ne velja. Prikažimo to na primeru dveh javanskih razredov, A in B, pri čemer je B podtip A:

```
class A {
  ....
}
class B  extends A {
  ....
}
```

Pretvorba podtipa B v njegov nadtip A je torej povsem varna, saj podtip B zagotavlja

vsaj vse operacije nadtipa A:

```
A a = new B();
```

Pri pretvorbi nadtipa A v podtip B pa potrebujemo dodatno informacijo, saj nadtip A, če ni dejanski tip B, ne vsebuje funkcionalnosti podtipa B. Ugotoviti je torej potrebno, ali ima entiteta statičnega tipa A, dejanski ali dinamični tip B:

```
B b = (B)a;
```

Takšno preverjanje imenujemo dinamično preverjanje tipa (dynamic type check). Namesto prevajalnika, ki preverjanje opravi implicitno, bi lahko tip entitete preverili tudi sami - z operatorjem instanceof. Če bi z `A a = new A()` namesto objekta tipa B, kot zgoraj, ustvarili objekt tipa A, preverba tipa `(B)a` ne bi uspela in prišlo bi do izjeme. Če je slabost operatorja instanceof eksplicitna uporaba, je njegova prednost v preprečitvi izjemne situacije. Programski jezik eifell isto stvar stori implicitno, saj za preverjanje pretvorbe uporablja t.i. obratno prireditev (reverse assignment), ki v primeru nepravilnega tipa priredi prazno vrednost `void`. Obratno prireditev bi za naš primer zapisali `b ?= a`. Statično tipizirani programski jeziki, kot sta Java in C++, zaradi svojega statičnega sistema tipov, potrebujejo eksplicitne pretvorbene mehanizme izključno za izhod iz statičnega tipiziranja, v katerem tipov dinamičnih objektov ni mogoče enoumno klasificirati. Tako java kot C++ imata dvonivojski sistem tipov s statično komponento, ki je aktivna v času prevajanja in dinamično, ki opravlja preverjanja v času izvajanja. Ko se bomo podrobneje seznanili z nekaterimi koncepti objektno usmerjenih jezikov, bomo spoznali, da je dinamično preverjanje v močno tipiziranih statičnih programskih jezikih neizogibno, saj večina teh mehanizmov (abstrakcija, mnogoličnost) temelji prav na dinamični naravi statično tipiziranih objektov. Dinamični jeziki, kot je Scheme, se soočajo z enakimi problemi tipiziranja, le da je dejansko preverjanje tipov zakasnjeno do časa neposredno pred izvajanjem. Dinamični jeziki opravijo vsako preverbo, tudi takšno, ki bi jo lahko povsem dokončno opravili med prevajanjem, pred izvršitvijo dejanske operacije.

Sistem tipov, striktno statični, povsem dinamični ali hibridni zagotavlja varnost pretvorb, izvajanja operacij in interakcije med objekti, katerih tipe lahko nedvoumno določimo šele ob samem izvajanju. Sistem tipov je osnovno vezivo med vsemi koncepti, ki jih

nek programski jezik omogoča, saj so z njim povezane uporaba, omejitve in dejanska implementacija konceptov.

Jeziki, ki povsem temeljijo na preverbi tipov v času izvajanja in v katerih ni prejemnika sporočila, kot ga poznamo v objektnih in objektno usmerjenih jezikih, se imenujejo multimetodni jeziki (multi-method languages). Sintaksa invokacije metod multimetodnih jezikov je zelo podobna proceduralnim, le da je tukaj dejansko telo procedure odvisno od tipov podanih parametrov v času izvajanja. Kriterij za izbiro najprimernejše metode, je seveda čisto specifičen in se razlikuje od jezika do jezika. Na multimetodne jezike lahko gledamo torej kot na povsem dinamične različice proceduralnih jezikov.

3.3.2 Podtipi

Podtip dobi svojo veljavo prav v kontekstu objektno usmerjenih jezikov. Podtipi so tesno povezani s komponentnim načrtovanjem z uporabo dedovanja. Kot tipi, so tudi podtipi sredstvo za modularizacijo abstrakcij, vendar na še bolj razdrobljenem nivoju. S podtipi opisujemo razmerja in asociacije med posameznimi, bolj oddaljenimi tipi. Če je tip opisan z razredom, potem podrazred dobi vlogo podtipa. Čeprav je to povečini smiselno in upravičeno, je vloga (pod)razreda in podtipa povsem druga. Podtipi obstajajo ločeno od razredov in niti ni nujno, da hierarhija razredov smiselno opisuje pripadajočo hierarhijo tipov. Koncept podtipa je vpeljal polimorfno obnašanje objektov, saj lahko isti objekt v različnih kontekstih operira na različne načine. Ne glede na način opisa podtipa, ta vselej predstavlja nek prikrojen, bolj omejen zunanji vmesnik, kateremu zadošča v svoji specifikaciji. Edina razlika med tipom in podtipom je v tem, da ima podtip nujno svoj nadtip, medtem ko za tip to ni obvezujoče. Podtip torej sam zase ne more obstajati. Lahko pa se zgodi, da je v navezi z več kot enim nadtipom hkrati. V tem primeru podtip ne samo specializira obnašanja nadtipov, temveč združi njihove vmesnike. Takšno združevanje je možno v Javi in $C^{\#}$ z večkratnim dedovanjem vmesnikov.

Relacijo podtipa formalno zapišemo kot $P <: T$, kjer P predstavlja podtip tipa T. Relacija podtipa zagotavlja pravilnost njegove uporabe, saj je podtip P lahko uporabljen na vsakem mestu, kjer pričakujemo tip T. Vrednost tipa P lahko torej

predstavlja tudi tip T. Tipu T pravimo tudi supertip tipa P. Ker ima lahko ena vrednost več tipov hkrati, govorimo o polimorfizmu podtipov. Podtip lahko sprejemamo kot tip, ki vključuje vmesnik svojega nadtipa, katerega še dodatno specializira z lastnimi definicijami. S tega stališča govorimo tudi o vključitvenem polimorfizmu. Jezik običajno podpira strukturalno ali eksplicitno tipiziranje. Slednje je prisotno v Javi, saj je treba relacijo med posameznimi tipi navesti eksplicitno.

Podtip torej predstavlja vmesniški nivo svojega nadtipa. Če želimo varen sistem tipov, mora podtip, kjer je uporabljen na mestu nadtipa, zadoščati celotni definiciji tega nadtipa. Podtip lahko torej specificira nekatere nove funkcionalnosti, ne sme pa obstoječih odstraniti. Funkcionalnost, ki je definirana v okolju pričakovanja tipa, se zanaša samo vmesnik tega istega tipa in ne tipov, ki so izpeljani iz njega.

Relacija podtipa vpliva na vsak izraz oz. vsako tipizirano frazo jezika. Če ima programska fraza tip P, ima pri zgoraj zapisanemu pogoju $P <: T$ tudi tip T. To pravilo subsumpcije omogoča, da lahko sistem tipov že na statični ravni preveri eksaktnost programske fraze v kontekstu, kjer je pričakovan nadtip njenega lastnega tipa. Tipiziranje z vpeljavo relacij med tipi in njihovimi podtipi izboljšuje prilagodljivost in razširljivost pravilno tipiziranih izrazov, saj lahko njihove vrednosti uvrstimo v celo hierarhijo tipov. Nadomestitev nadtipa z njegovim podtipom v izrazu nima skoraj nikakršnega vpliva na implementacijski nivo jezika. Večina objektnih jezikov objekte opisuje kot reference, ki so vedno enake dolžine (dolžina pomnilniškega naslova), ne glede na strukturalno naravo posameznega objekta. Pri "vstavitvi" izraza tipa podtipa v kontekst, kjer se pričakuje nadtip, je potrebno preveriti samo logično pravilnost podtipa.

V večini tipiziranih programskih jezikov preverjanje tipa temelji na relaciji kompatibilnosti med posameznimi tipi. Najvišja stopnja kompatibilnosti je ekvivalenca tipov $P \equiv T$, ki pomeni, da lahko na mesto tipa T postavimo tudi tip P in obratno. Relacija podtipa je torej striktnejša kot ekvivalenca. Pravilo podtipa nazorno zapišemo s pravilom:

$$\frac{\tau' <: \tau \qquad e : \tau'}{e : \tau}$$

Pravilo podtipa pravi, da lahko za vsak podtip τ' tipa τ in izraz e tipa τ' sklepamo, da ima izraz e tudi tip τ. Pravilo imenujemo tudi enosmerno substitucijsko pravilo za

tipe v izrazih, saj predpisuje relacijo substitucije, ki velja med tipi.

Najpreprostejši neobjektni tip, na katerega lahko apliciramo relacijo podtipa, je zapis (record). Zapis je zelo podoben kartezičnemu produktu. Vrednost zapisa je predstavljena z dvojčkom, katerega komponenti sta oznaka in njena vrednost. Zapis je fleksibilnejši od splošne n-terice, saj vrstni red elementov zapisa nima pomena. Posamezni element zapisa je identificiran z njegovo oznako. Zapis lahko zapišemo kot $(o_1 : \tau_1,\ o_2 : \tau_2,\ \ldots, o_n : \tau_n)$, pri čemer so $o_1, \ldots, o_n$ različne oznake, $\tau_1, \ldots, \tau_n$ pa pripadajoči tipi. Glavna operacija nad zapisom je selekcija posamezne komponente zapisa $Sel(o_i)$. Če označimo zapis z δ in določimo njegov podtip δ', mora le-ta ohraniti operacijo selekcije. Tip τ_i' vsake (istoimenske) komponente o_i v δ' mora biti podtip tipa τ_i komponente o_i v δ. Pravilo relacije podtipa $\delta' <: \delta$ zapišemo na sledeč način:

$$\frac{\tau_1' <: \tau_1 \ \ldots \ \tau_n' <: \tau_n}{(o_1 : \tau_1',\ \ldots, o_n : \tau_n', o_{n+1} : \tau_{n+1},\ \ldots, o_{n+m} : \tau_{n+m}) <: (o_1 : \tau_1,\ \ldots, o_n : \tau_n)}$$

Podtip δ' zapisu δ je vsak tak zapis, ki ima dodatne komponente $(o_{n+1},\ \ldots, o_{n+m})$, ali istoimenske komponente tipov, ki so podtipi tipom komponent v δ. Primera podtipa zapisa sta:

$$(o_1 : Integer,\ o_2 : Integer) <: (o_2 : Integer)$$

in

$$(o_1 : (o_1 : Integer,\ o_2 : Integer),\ o_2 : Integer) <: (o_1 : (o_1 : Integer))$$

Prvi primer podtipa zapisa imenujemo tudi podtip v *širino*, drugega pa podtip v širino in *globino*. Relacijo podtipa za zapis lahko zapišemo bolj zgoščeno:

$$(o_i : \tau_i)_{1 \le i \le n} \ <:\ (o_j : \upsilon_j)_{1 \le j \le k}, \quad k \le n \quad \wedge \quad \tau_i <: \upsilon_i, \quad 1 \le i \le k$$

Takšna formulacija podtipa je primerna samo za zapis, v katerem je edina operacija selekcija komponente $Sel(o_i)$. V kolikor bi želeli zapis, v katerem bi lahko komponente spreminjali, bi bilo v relaciji treba uporabiti dodatne informacije. Kadar komponente

zapisa samo izbiramo imamo opraviti s tipom vrednosti. Če želimo komponento spremeniti, pa potrebujemo tip reference, saj je vsaka komponenta zapisa shranjena na neki pomnilniški lokaciji. Tip spremenljivke s označimo z $ref(s)$. Pravilo podtipa za zapise, v katerih lahko posamezne komponente spreminjamo, zapišemo:

$$(o_i : ref(\tau_i))_{1\leq i\leq n} \ <: \ (o_j : ref(\upsilon_j))_{1\leq j\leq k}, \quad k \leq n \quad \wedge \quad \tau_i \equiv \upsilon_i, \quad 1 \leq i \leq k,$$

kjer relacija $\tau_i \equiv \upsilon_i$ označuje vzajemnost oz. ekvivalenco podtipov $\tau_i <: \upsilon_i$ in $\upsilon_i <: \tau_i$. Podtip spremenljivega zapisa lahko torej doda nove komponente, obstoječe pa morajo te imeti ekvivalentne tipe.

Izjemno pomembna v okviru objektnega sistema tipov je formulacija funkcijskega tipa. Ta ima širok vpliv na izrazno moč sistema tipov, saj pogojuje spremembe, ki jih le-ta dovoljuje v redefiniciji metod. Funkcijski tip zapišemo $\tau_a \rightarrow \tau_v$, kjer τ_a predstavlja tip argumenta in τ_v tip vrednosti, ki jo funkcija vrača. Tip argumenta funkcije je domena, tip vračanja pa zaloga vrednosti. Operator $\rightarrow$ imenujemo funkcijski operator ali konstruktor tipa. Že omenjena funkcija selekcije komponente $Sel(o_i)$ zapisa $(o_1 : \tau_1, \ \ldots, o_n : \tau_n)$, ima torej funkcijski tip $\mathbf{String} \rightarrow \tau_i$. Funkcijski podtip tipa $\tau_a \rightarrow \tau_v$ lahko po pravilu subsumpcije uporabimo kjerkoli pričakujemo njegov nadtip. Relacijo funkcijskega podtipa zapišemo $\tau'_a \rightarrow \tau'_v <: \tau_a \rightarrow \tau_v$. Problem funkcijskega podtipa je v njegovi navidez nelogični interpretaciji. Če privzamemo, da je $\mathbf{N} <: \mathbf{R}$, bi funkcijo $\mathbf{R} \rightarrow \mathbf{N}$, ki slika iz realnega v naravno število, zlahka imeli za podfunkcijo $\mathbf{R} \rightarrow \mathbf{R}$. Vendar pravilo podtipa zahteva, da lahko vrednost podtipa vstavimo na vsako mesto, kjer pričakujemo nadtip. Če torej na nekem mestu pričakujemo funkcijo $\mathbf{R} \rightarrow \mathbf{R}$, bi lahko na tem mestu poklicali funkcijo $\mathbf{R} \rightarrow \mathbf{N}$. Če pričakujemo, da funkcija vrne realno vrednost, njen podtip pa vrne celo število, je to veljavno, saj vsako celo število uvrstimo tudi v množico realnih števil. Drugače je pri argumentih, saj "nadfunkcija" pričakuje argument tipa $\mathbf{R}$. Funkcijo pokličemo z argumentom realnega tipa, vendar če na njeno mesto vstavimo funkcijo, ki pričakuje naravni tip, bo to pomenilo napako, saj je množica $\mathbf{R}$ močnejša od $\mathbf{N}$. Tip argumenta podfunkcije mora potemtakem biti nadtip argumenta nadfunkcije. Pravilo funkcijskega podtipa zapišemo:

$$\frac{\tau_a <: \tau'_a \qquad \tau'_v <: \tau_v}{\tau'_a \rightarrow \tau'_v \ <: \ \tau_a \rightarrow \tau_v}$$

Funkcijski operator je monoton v drugem argumentu ($\tau'_v <: \tau_v$) in antimonoton v prvem ($\tau_a <: \tau'_a$). Monotonost pomeni kovariantno spremembo, antimonotonost pa kontravariantno. Tip vračanja podfunkcije je torej lahko podtip vračanja nadfunkcije, medtem ko je pri argumentih ravno obratno. Tip argumenta podfunkcije mora biti enak ali nadtip tipa argumenta nadfunkcije.

Na polje lahko gledamo kot na funkcijski operator $\rightarrow$, ki slika iz tipa indeksa v tip elementov polja. Privzemimo homogena polja. Referenciranje elementa polja P[i] v funkcijski notaciji zapišemo $\tau \rightarrow \upsilon$, kjer je τ tip indeksa i in υ tip elementov polja P. Pravilo podtipa je podobno funkcijskemu podtipu:

$$\frac{\upsilon' <: \upsilon \qquad \tau <: \tau'}{\upsilon'[\tau'] <: \upsilon[\tau]}$$

Tako zapisano pravilo ne vključuje polj, v katerih lahko vrednosti spreminjamo. Da bi lahko vrednosti spreminjali, morata biti tipa elementov ekvivalentna:

$$\frac{\upsilon' \equiv \upsilon \qquad \tau <: \tau'}{\upsilon'[\tau'] <: \upsilon[\tau]}$$

Indeks polja se spreminja kontravariantno, medtem ko je tip elementa fiksiran. S stališča variance tipov je relacija podtipa polja s spremenljivimi vrednostmi podobna zapisom s spremenljivimi komponentami. V obeh imamo t.i. neposredno referenciranje. Zato morajo tipi komponent oz. elementov biti invariantni. Java omogoča kovariantne spremembe v tipih elementov polja $\upsilon'[\,] <: \upsilon[\,], \quad \upsilon' <: \upsilon$. S tem je onemogočena statična varnost zaradi česar se mora Java zanašati na dinamično preverjanje, kadarkoli se izvaja prirejanje elementu polja. Takšno tipiziranje je sicer "nepravilno", vendar omogoča večjo fleksibilnost sistema tipov, saj lahko ista funkcionalnost v okviru ene metode operira nad polji različnih tipov. Sistem tipov, ki omogoča generične abstrakcije, takšno tipiziranje odpravi.

Večina objektno usmerjenih jezikov relacijo podtipa objektov povezuje s podtipi pripadajočih razredov. Relacija podtipa razredov $\tau' <: \tau$ pogojuje relacijo objektov $\mathcal{O}_{\tau'} <: \mathcal{O}_\tau$, kjer $\mathcal{O}_r$ predstavlja objektni tip, tj. instanco $\mathcal{O}$ razreda r. Takšna naveza razreda in objekta ni nujna. Razred kot mehanizem klasifikacije lahko, podobno kot

objekt, obravnavamo povsem zase. Edina povezava med tako predstavljenima razredom in objektom je vmesnik, kateremu razred služi kot definicijski mehanizem, objekt pa implementira predpisano funkcionalnost. Striktna objektna naravnanost zahteva, da je edina operacija, ki se lahko izvaja nad objektom, pošiljanje sporočila. Operacija pošiljanja sporočila je zelo podobna selekciji komponente zapisa. Objekt lahko torej predstavimo kot konstanten zapis. Komponente zapisa nadomestijo metode, za katere velja relacija funkcijskega podtipa. Zapišimo objektni tip z naborom definiranih metod kot $\mathcal{O}\{m_i : M_i\}$, kjer m_i predstavlja metodo in M_i njen funkcijski tip. Razred objekta ni pomemben, zato ga izpustimo. Relacijo objektnega podtipa zapišemo na sledeč način:

$$\mathcal{O}\{m_i : M_i{}'\}_{1\leq i\leq n} <: \mathcal{O}\{m_j : M_j\}_{1\leq j\leq k}, \quad k \leq n \quad \wedge \quad M_i{}' <: M_i, \quad 1 \leq i \leq k$$

Ker je M funkcijski tip, lahko za konkretni primer i zapišemo $M_i' = \tau_a' \to \tau_v'$ in $M_i = \tau_a \to \tau_v$. Ker velja $M_i' <: M_i$, mora po pravilu za podtip funkcijskih tipov veljati tudi $\tau_a <: \tau_a'$ in $\tau_v' <: \tau_v$. Objektni tip $\mathcal{O}'$ je podtip $\mathcal{O}$, če $\mathcal{O}'$ vsebuje vse metode iz $\mathcal{O}$, za katere velja monotonost tipov zalog vrednosti in antimonotonost funkcijskih domen. Po pravilu podtipa za zapise, lahko $\mathcal{O}'$ definira tudi svoje metode. Monotonost zaloge vrednosti ima pomembno praktično uporabo pri modeliranju razredov oz. objetov z dedovanjem. Antimonotonost argumentov sicer omogoča najvišjo stopnjo fleksibilnosti statično varnega sistema tipov, vendar nima večje veljave v praksi. Jezik C++ zato omogoča samo prvo. Java in C# ne dopuščata nikakršnih sprememb v tipih redefiniranih metod, saj imata invarianten sistem tipov. Invariantni sistemi tipov so zelo šibki in s tega stališča niso zaželeni.

Potrebno je določiti še relacijo podtipa za razrede. Očitno je, da je razredni tip smiseln samo za jezike, ki za klasifikacijo in opis abstraktnih podatkovnih tipov uporabljajo razrede. Razredni tip vključuje informacije o tipih instančnih spremenljivk in metod. Omejitve relacije razrednega tipa so podobne tistim iz relacije objektnega tipa. Da bi določili, katere metode in instančne spremenljivke lahko v podrazred dodamo, moramo poznati tiste iz nadrazreda. To velja za sistem tipov, za katerega je zahtevana statična varnost; dinamični prototipni jeziki so tukaj veliko bolj fleksibilni. Razredni tip opišemo

kot $\mathcal{C}\{I, S\}$, kjer je I zapis, katerega komponente predstavljajo instančne spremenljivke. Z zapisom S predstavimo metode. Če so vse instančne spremenljivke kapsulirane znotraj razreda s privatnimi specifikatorji dostopa (so nevidne navzven), je tip objekta, ki nastane kot instanca razreda $\mathcal{C}\{I, S\}$ podan z objektnim tipom $\mathcal{O}\{S\}$. Skrivanje instančnih spremenljivk ima torej tudi formalno podlago. Da bi lahko zapisali relacijo, ki pove kateri razredni tip je podtip nekega drugega razrednega tipa, je potrebno preučiti, katere operacije se lahko izvajajo nad razredom.

Ker je razred v prvi vrsti konstrukt za specifikacijo funkcionalnosti objektov, je najpomembnejša operacija nad razredom ustvarjanje objekta. Ker objekt nastane z instanco razreda, to operacijo zapišemo $Inst(\mathcal{C})$. Druga operacija je formiranje podrazreda z dodajanjem novih metod ali redefinicijo že obstoječih v nekem razredu. Razredni podtip tipa $\mathcal{C}\{I, S\}$ je $\mathcal{C}'\{I', S'\}$. Očitno je relacija razrednega podtipa pogojena z relacijama med I in I' ter S in S'. Če naj bo $\mathcal{C}' <: \mathcal{C}'$ in če $Inst(\mathcal{C}')$ ustvari objektni tip S', potem mora ta objektni tip biti podtip S. Relacija podtipa za razredni tip potemtakem zahteva $S' <: S$. Če velja, da ima razred $\mathcal{C}'$ tip $\{I', S'\}$, $\mathcal{C}$ tip $\{I, S\}$ in je $\mathcal{C}'$ podtip $\mathcal{C}$, potem lahko po pravilu subsumpcije $\mathcal{C}$ nadomestimo s $\mathcal{C}'$. Vsaka metoda m iz $\mathcal{C}$ je lahko v $\mathcal{C}'$ redefinirana kot m' ob pogoju, da ima isto signaturo v S in S'. Drug pogoj je, da se v S' nahaja natanko toliko metod kot v S, saj drugače ne moremo zagotoviti kompatibilnosti med $\mathcal{C}$ in razredi izpeljanimi iz $\mathcal{C}'$. Zahtevamo torej $S \equiv S'$. Relacijo zapišemo:

$$\mathcal{C}'\{I', S'\} \ <: \ \mathcal{C}\{I, S\}, \quad I \equiv I' \quad \wedge \quad S \equiv S'$$

Zapisana relacija podtipa razrednih tipov je preprosta, saj ne vključuje konceptov skrivanja instančnih spremenljivk I in metod S. Specifikatorji dostopa nekoliko zakomplicirajo relacijo, saj je treba upoštevati še vidnost in/ali nevidnost posameznih spremenljivk in metod v podrazredih. Privatni specifikator dostopa (*private*) v C++ in Javi pomeni, da je instančna spremenljivka ali metoda dostopna samo znotraj razreda. Privatni specifikator povzroči, da se metoda ali spremenljivka odstrani iz seznama, ki ga je potrebno upoštevati ob dedovanju. Razred, ki poleg privatnih in javnih specifikatorjev vsebuje še zaščitene (*protected*), lahko razširimo s $\mathcal{C}\{I_p, S_p, S_z, S_j\}$, kjer z I_p predstavimo privatne instančne spremenljivke, s S_p privatne metode, s S_z zaščitene in s

S_j metode z javnim dostopom. Zapisa za privatne spremenljivke in metode I_p ter S_p iz relacije podtipa odpadeta, saj dostop do njih v podrazredu ni mogoč. S tem odpadejo tudi semantična pravila za preverjanje pravilnosti dostopa do metod in spremenljivk zunaj razreda in v podrazredih.

3.4 Polimorfni sistemi tipov

Koncept polimorfizma dodatno razširja moč sistema tipov, saj omogoča klasifikacijo tipov na različnih hierarhičnih ravneh. Polimorfizem si najlažje predstavljamo z operacijo. Operacija je polimorfna, če lahko uniformno operira nad različnimi tipi. To lahko dosežemo z "ad-hoc" polimorfizmom, kjer se funkcionalnost operacije prilagodi glede na tip. Takšna rešitev je neelegantna in ne omogoča neskončnih množic tipov. Boljši metodi sta parametrični [5, 82] in vključitveni polimorfizem.

Dedovanje in z njim podtipiziranje vpeljujeta vključitveni polimorfizem. Vsak podtip nekega tipa ima tudi tip vseh svojih nadtipov. To je določeno z relacijo podtipa. Funkcionalnost, ki je definirana na nekem tipu τ, lahko povsem transparentno operira nad podtipi $\tau'_1, \ldots, \tau'_n$, če so slednji podtip tipa τ. Vključitveni polimorfizem izkoriščajo pravzaprav vsi objektno usmerjeni in objektni jeziki, saj je njegova implementacija trivialna. Tudi ločevanje tipa od razreda izrablja vključitveni polimorfizem, le da so v tem primeru tipi čiste abstrakcije.

Vključitveni polimorfizem pa ni vedno učinkovit. Predvsem se to izkaže v statično tipiziranih jezikih, kjer je zaželjeno, da se kar največji del tipiziranja preveri že v času prevajanja. Operiranje s podatkovnimi strukturami, ki vsebujejo nadtipe oz. bolj splošne tipe, je zamudno, saj je potrebno pri pretvorbi v dejanski tip izvesti dinamične preverbe. Podatkovne strukture, ki operirajo nad nekim abstraktnim tipom, lahko vsebujejo vsak njegov podtip. To ni vedno zaželjeno, še najmanj pa, kadar imamo opravka z uniformno podatkovno strukturo, v kateri pričakujemo, da bodo vsi elementi istega tipa. Ta problem se pojavlja v Javi.

Drug koncept je parametrični polimorfizem ali mehanizem generikov (generics). Parametrični polimorfizem je dobro uveljavljen jezikovni koncept [47], katerega prednosti pred dinamičnimi pristopi se pokažejo v varnosti, izraznosti in učinkovitosti. Višja stopnja

varnosti izhaja iz tega, da je večino napak tipiziranja mogoče zavrniti že v času prevajanja. Ob določitvi tipa parametra neki parametrizirani podatkovni strukturi, se lahko preveri, ali ta dejanski tip zares ustreza zahtevam abstraktnega tipa. Sistem tipov, ki omogoča parametriziranje tipov je tudi čistejši, saj se izognemo nepotrebnim in večkrat nepreglednim pretvorbam med abstraktnimi in konkretnimi tipi. Največja prednost parametriziranih tipov pa je v njihovi učinkovitosti. Abstraktni tip se namreč v času instanciranja nadomesti s konkretnim, ki je podan kot parameter. Pri obdelavi konkretnega tipa v času izvajanja ni več nobenih pretvorb.

Parametriziran tip lahko razumemo kot funkcijo iz tipa v tip. Funkcija, ki realizira parametriziran razred, prejme konkreten tip, s katerim se vrši instanciranje in vrne razred, ki je prirejen temu tipu. Če parametriziran razred označimo s $\mathcal{C}^p$, lahko funkcijo preslikave s parametrom τ zapišemo na sledeč način:

$$f_t : \quad \tau \rightarrow \mathcal{C}^\tau$$

Funkcija preslikave tipa lahko seveda operira nad argumentom, ki je sam funkcija preslikave tipa:

$$f_t : (\quad \tau \rightarrow \mathcal{C}^\tau \quad) \rightarrow \mathcal{C}^{\tau'} \qquad \tau' \equiv \tau \rightarrow \mathcal{C}^\tau$$

Težave parametriziranih tipov se pojavijo v implementaciji, še posebej, kadar gre za preveden ali polpreveden jezik. V času prevajanja parametriziranega razreda, parametra ni mogoče določiti. To pomeni, da tudi ne poznamo njegove velikosti. Velikosti entitet je pri prevajanju treba poznati, saj so od njih odvisni naslovi in struktura prevedenega koda. Ta problem lahko odpravimo, če imajo vse vrednosti enako strukturo. V ta razred sodijo jeziki, v katerih je vsaka vrednost predstavljena uniformno kot objekt. V drugih se to rešuje drugače. C++ uporablja vstavljanje, kjer se v času prevajanja abstraktni tip nadomesti z dejanskim. Slabost tega pristopa je v tem, da je potreben izvorni kod parametriziranega razreda, kar ni vedno mogoče. Jezik $C^\#$ omogoča parametriziranje tudi nad neuniformnimi tipi [59, 111]. Če parametrizacija tipov v jeziku ni neposredno podprta, jo lahko simuliramo s koncepti, ki so v jezik že vgrajeni. Najpreprostejši, vendar najmanj fleksibilen način je predprocesiranje vhoda. Drug način je uporaba refleksijskih mehanizmov [105]. Seveda je to možno v jezikih,

ki omogočajo neko stopnjo refleksije, npr. Java.
Včasih parametriziran tip ni dovolj. Želimo, da je tip omejen na neko domeno oz. da zadošča nekemu drugemu tipu. To nam omogoča še boljši nadzor nad definiranjem generične funkcionalnosti. Tak tip parametričnega polimorfizma se imenuje omejen polimorfizem (bounded polymorphism). Vendar ima tudi omejen polimorfizem svoje slabosti. Te se izkažejo pri t.i. binarnih metodah [65], ki prejmejo vrednost tipa lastnega razreda. Primer dobro znane binarne metode je `equals` v javi. Ker ima Java invarianten sistem tipov, metoda `equals` vedno operira nad argumentom tipa `Object`. Zaradi teh težav je nastala posplošena oblika parametričnega polimorfizma imenovana F omejen polimorfizem (F-bounded polymorphism) [25]. Ta oblika polimorfizma ne omogoča samo omejevanje abstraktnega tipa na nek drug tip, temveč tudi parametriziranje tega drugega tipa. Ta koncept je bil zaradi svoje izraznosti implementiran tudi kot razširitev Jave [20, 48, 7].

3.5 Formalne metode za opis sistemov tipiziranja

Večina programskih jezikov nima definiranih formalnih pravil za statično tipiziranje programov. Pravila so povečini definirana posredno v sami strukturi jezika in prevajalnika. Formalna specifikacija pravil nudi možnost natančnega modeliranja programskih fraz ter dokazovanje in preverjanje pravilnosti tipiziranja že v času prevajanja. Formalne metode tipiziranja slonijo na tipiziranem računu lambda ($\lambda - calculus$) [28, 26]. Pravila tipiziranja podamo induktivno na podlagi produkcij kontekstno proste gramatike, ki generira jezik.
Tako kot tipiziran lambda račun tudi pravila tipiziranja zahtevajo neposredno informacijo o tipih prostih spremenljivk znotraj izrazov in o pomenu posameznega tipa. Spremenljivke predstavimo kot identifikatorje povezljivih vrednosti znotraj nekega okolja. Okolje takšnih spremenljivk označimo s Π in naj pomeni končno množico asociacij (parov) med identifikatorji in tipiziranimi izrazi. Asociacijo zapišemo kot $s : T$, kjer je s edinstven identifikator v okolju Π in T njegov tip. Če velja relacija $s : T \in \Pi$, zapišemo $\Pi(s) = T$. Da bi bila pravila tipiziranja razširljiva, je potrebno dopustiti

možnost definiranja novih tipov. Ker tip pomeni asociacijo med identifikatorjem tipa in njegovo definicijo, se lahko identifikator tipa v pravilu nadomesti z njegovo definicijo. Definicijo tipa zapišemo z relacijo $t = T$, kjer je t identifikator tipa in T izraz tipa. Sistem omejevanja tipov (type cnonstraint system) označimo s $\mathcal{S}$ in definiramo na sledeč način:

- prazna množica $\emptyset$ je sistem omejevanja tipov,
- če je $\mathcal{S}$ sistem omejevanja tipov in t identifikator tipa, ki še ni v $\mathcal{S}$ ali T, potem je tudi $\mathcal{S} \cup \{t = T\}$ sistem omejevanja tipov.

Pri preverjanju tipov nekega programskega konstrukta mora sistem $\mathcal{S}$ vsebovati definicije vseh tipov, ki se pojavijo v kontekstu tega konstrukta. Sistem $\mathcal{S}$ je dinamičen, saj omogoča odstranjevanje definicij tipov, ko ti prenehajo obstajati. Če želimo, da sistem tipov ne bo omejen na monomorfne tipe, bo potrebno vključiti tudi polimorfne definicije tipov. Zaradi preprostosti definicije tipov ne bodo več vključevale samo konkretne definicije, temveč tudi sklepanje o podtipih.
$\mathcal{S}$ posplošimo v funkcijo $\mathcal{S}(T)$, ki vrača izraz tipa na način, da da se vsi identifikatorji tipa v T nadomestijo s svojimi definicijami v $\mathcal{S}$. Za sistem tipov, ki vključuje konstantne in referenčne tipe, razredne in objektne ter funkcijske tipe, funkcijo $\mathcal{S}(T)$ za izraz tipa T definiramo na sledeč način:

- če je t identifikator tipa, je $\mathcal{S}(t) = \begin{cases} \mathcal{S}(T) & , \{t = T\} \in \mathcal{S} \\ t & , \text{drugače} \end{cases}$,
- če je c konstanten tip, je $\mathcal{S}(c) = c$,
- $\mathcal{S}(T_1 \times \ldots \times T_n \rightarrow T_{n+1}) = \mathcal{S}(T_1) \times \ldots \times \mathcal{S}(T_n) \rightarrow \mathcal{S}(T_{n+1})$,
- $\mathcal{S}(ref(T)) = ref(\mathcal{S}(T))$,
- $\mathcal{S}(\mathcal{O}(M)) = \mathcal{O}\ \mathcal{S}(M)$,
- $\mathcal{S}(\mathcal{C}(I, S)) = \mathcal{C}(\mathcal{S}(I), \mathcal{S}(S))$.

Končnost funkcije $\mathcal{S}(T)$ zagotavlja druga točka, ki služi kot izhodni pogoj rekurzivnih definicij tipa. Če so vse proste spremenljivke v T vsebovane v domeni $\mathcal{S}$, potem bo

$\mathcal{S}$ izraz tipa brez prostih spremenljivk. Rekurzivna definicija je pomembna zaradi uporabe tipov, ki še niso nujno definirani na mestu uporabe.
Pravila tipiziranja obdelujejo programske fraze, kot si te sledijo v programu in dodajajo informacije o tipih novih identifikatorjev v $\mathcal{S}$.
Pri tipiziranju izrazov imamo opraviti z neko oceno ali razlago (judgement). Če presodimo razlago oblike $\mathcal{S}, \Pi \vdash D \diamond \mathcal{S}', \Pi'$, potem to pomeni, da tipiziranje deklaracije D, ob predpostavkah iz $\mathcal{S}$ in Π, obogati $\mathcal{S}$ v $\mathcal{S}'$ in okolje Π v Π'. Vidimo, da obdelava deklaracije torej spreminja sistem tipov in okolje. Pravila tipiziranja zapišemo kot množico hipotez in sklepov:

$$\frac{\mathcal{S}, \Pi \vdash D_1 \diamond \mathcal{S}_1, \Pi_1, \ldots, \mathcal{S}_{n-1}, \Pi_{n-1} \vdash D_n \diamond \mathcal{S}_n, \Pi_n}{\mathcal{S}, \Pi \vdash D \diamond \mathcal{S}_n, \Pi_n}$$

Sklep je zapisan na mestu imenovalca, hipoteze pa na mestu števca. Izrazi, ki se pojavljajo v hipotezah so povečini podizrazi tistih, ki jih zapišemo na mestu sklepov. Pravila tipiziranja beremo od spodaj navzgor in od leve proti desni. Gornje pravilo preberemo "tipiziranje deklaracije D omogoča, ob predpostavkah iz $\mathcal{S}$ in Π, bogatejše sklepanje o tipih $\mathcal{S}_n, \Pi_n$, če tipiziranje deklaracije iz $\mathcal{S}, \Pi$ da $\mathcal{S}_1, \Pi_1$, ... in $\mathcal{S}_{n-1}, \Pi_{n-1}$ da $\mathcal{S}_n, \Pi_n$". Na tak način se nato zapišejo pravila za vse možne deklaracije, ki jih jezik omogoča.
Tipiziranje navadnih izrazov je podobno. Pravila zapišemo v obliki:

$$\mathcal{S}, \Pi \vdash izraz : T$$

Ob predpostavkah iz $\mathcal{S}$ in Π, sklepamo, da ima $izraz$ tip T. Da bi natančno definirali sistem tipov jezika, je treba zapisati pravila za vse izraze, ki jih jezik omogoča. Pravila tipiziranja je treba zapisati pazljivo, saj predstavljajo osnovo za formalen opis sistema tipov. Formalen opis sistema tipov zavisi od jezikovnih konstruktov. Upoštevati je potrebno polimorfnost tipov, kar ni vedno enostavno, dedovanje, tipiziranje s `self`-om in drugo. Ker je poudarek našega dela na načrtovanju in implementaciji jezika, formalnega opisa sistema tipov zaenkrat ne bomo izdelali.

4 Jezik Z_0 in njegova implementacija

Načrtovanje programskega jezika je zahteven in kompleksen proces, ki zahteva ekspertizo s področja poznavanja jezikovnih modelov, konceptov in implementacijskih postopkov. Če to velja za proceduralne jezike, se pri objektno usmerjenih, objektnih in funkcijskih jezikih stvar še bolj zakomplicira. Načrtovanje novega jezika je vselej izziv, čeprav je zares "novega" v današnji množici raznovrstnih jezikov težko doseči. Kljub temu obstajajo postopki, pristopi in koncepti, ki lahko imajo radikalen vpliv na jezik. Še posebej se to da doseči z združevanjem konceptov, ki so do nekega trenutka veljali za izključujoče. Pogosto se združujejo koncepti, ki so značilni za določen tip jezikov. Tako lahko imamo funkcijske koncepte, kot so funkcije višjega reda, v jezikih z imperativnim programskim vzorcem [79, 80]. Seveda obstajajo tudi obratne variante – objektni koncepti v deklarativnih jezikih.

Sintaktična struktura in semantični model zavisita od namena jezika. Najkompleksnejši semantični model imajo običajno splošno namenski imperativni jeziki, ki so bili načrtovani za zelo široko domeno uporabe, medtem ko funkcijski jeziki veljajo za preprostejše in hkrati bolj izrazne. Eden izmed ciljev našega jezika je vsekakor izrazna moč. Ta mora biti čim večja, vendar mora biti dosegljiva na preprost in razumljiv način. Pomembna je torej sintaksa, pri kateri se zgledujemo po Javi. Predvsem zato, ker je Java moderen, uveljavljen in čist objektno usmerjen jezik, kakršnega si želimo tudi sami. Sintaksa je načrtovana konsistentno, preprosto in z rezerviranimi besedami, kjer so potrebne. Čistost sintakse prispeva k čistosti in razumljivosti celotnega jezika. Pravtako preprosto je zasnovan objektni model jezika. Enotna hierarhija s skupnim najvišjim razredom. Ubrali smo pristop čiste objektizacije in unificirali predstavitev vseh entitet z objektom. Izvajalno okolje je tako le množica aktivnih in pasivnih objektov. S pomočjo potencialnega meta nivoja jezika lahko takšno okolje nadzorujemo na nivoju podrobnih detajlov in ga manipuliramo. Unificirali smo tudi mehanizem dostopa do stanja objekta. Ker je metoda splošnejši pojem od instančne spremenljivke, smo slednje izvzeli in jih nadomestili z metodami. Jezik zato potrebuje mehanizem dinamičnega spreminjanja metode [6, 4].

Ker želimo učinkovito izvajanje, je jezik statično tipiziran. Toda statični sistem tipov

ima znane omejitve [34, 110], zato ga je potrebno razširiti in obogatiti z dodatnimi (dinamičnimi) mehanizmi tipiziranja. Dinamično preverjanje tipov je sestavni del vsakega sodobnega objektno usmerjenega jezika, saj lahko tipe v času prevajanja določimo le parcialno. Prednost statičnega tipiziranja je, da čeprav nepopolno, lahko v času prevajanja vendarle omejimo dinamični tip na neko vnaprej znano domeno. V čistih dinamičnih jezikih nimamo možnosti niti delne klasifikacije, razen morda to, da lahko z gotovostjo trdimo, da bo objekt vsaj tipa Object.

Jezik temelji na razredih in ker je razred edini klasifikacijski in definicijski mehanizem za podatke, je abstrakcija nad podatki vedno realizirana preko razreda. Razred je najprimitivnejša enota za abstrahiranjem nad podatkom.

Čeprav lahko naš jezik klasificiramo kot imperativni objektno usmerjeni jezik, vsebuje nekatere lastnosti, ki so tipično v domeni deklarativnih, funkcijskih jezikov. Ena izmed teh je uporaba zaprtij, kot struktur s kodom in pripadajočim okoljem. Zaprtja se uporabljajo v funkcijskih jezikih, ki omogočajo parcialno aplikacijo funkcij in funkcije višjega reda. Ker smo v našem jeziku omogočili dinamično spreminjanje metod in njihovo manipulacijo kot s prvorazrednimi vrednostmi, je smiselno uporabljati funkcije, ki prejemajo ali vračajo druge funkcije. Okolje funkcije je, podobno kot okolje ostalih jezikovnih konstruktov, predstavljeno kot zaprtje, ki se lahko prenaša naokrog preko parametra ali kako drugače. Zaprtja so mehanizem, ki nam omogočajo čisto objektizacijo konstruktov in njihovo manipulacijo na način prvorazrednih vrednosti.

Odločitev, ki vpliva na dobršen del implementacije jezika, je, da ta ni interpretiran, ampak preveden. V ta namen smo zasnovali abstraktno virtualno arhitekturo, za katero se bo izvorni kod prevedel. Prevajanje pomeni dodaten izziv, saj je treba na nek način prevesti vse jezikovne konstrukte, zagotoviti dinamiko znotraj statičnega sistema tipov in razrešiti vse ostale dinamične mehanizme. Zato je večina dinamičnih jezikov preprosto interpretirana. Prevajamo v vmesno obliko, ki ima značilnosti realne arhitekture. Tako bomo lahko kasneje prevajalnik ustrezno modificirali in prevajali za realno ciljno arhitekturo.

Število in raznolikost konceptov, ki so implementirani, je seveda omejeno s časom in zahtevnostjo. Zagotovo pa lahko trdimo, da bodo med mehanizmi, ki bodo implementirani v nadgradnji jezika parametrični polimorfizem, večkratno dedovanje, definiranje

operatorjev in meta arhitektura. Slednja bo jeziku omogočila izjemne razsežnosti in samozavedanje, ki v okviru statičnega jezika niso mogoče.

4.1 Splošno o jeziku

Jezik Z_0 je statično, močno tipiziran, objektno usmerjen programski jezik. Atributa objektne usmerjenosti, ki ju jezik vključuje, sta še dedovanje in pozno povezovanje metod. Jezik za razliko od večine modernih industrijskih jezikov omogoča čist objektni model, kar pomeni, da se ravnamo po principu "vse je objekt". Objekti niso zgolj sestavljeni kompleksni tipi, kot je to pravilo v C++ in Javi, ampak tudi vse vgrajene entitete, ki so integralni del samega jezika. S tem mislimo podatkovne entitete predstavljene s primitivnimi tipi in vse ostale jezikovne konstrukte, katere je smiselno predstaviti kot zaključene objektizirane celote. Med te prištevamo jezikovne konstrukte kot so zanke, nekatere stavke in bloke. Čisti objektni model prinaša nekatere prednosti, ki jih drugače ne bi imeli. Posebej gre izpostaviti unifikacijo vseh entitet, možnost aplikacije parametričnega polimorfizma na vse entitete in nenazadnje boljšo zasnovo za teoretično modeliranje jezikovne semantike. Podoben princip uporablja jezik Smalltalk [64], ki je pravtako čist objektno usmerjen jezik, vendar dinamično tipiziran. Jezik Z_0 je statično tipiziran, s čimer smo omogočili večjo varnost sistema tipov, saj se večina tipov in njihovih pretvorb lahko preveri že v času prevajanja. Jezik Z_0 smo implementirali na način, da je karseda učinkovit, izrazno močan in hkrati statično in močno tipiziran. Cilj je bil torej poiskati kompromisne rešitve med statičnim sistemom tipov in splošno učinkovitostjo jezika. Učinkovito izvajanje programskega koda jezika narekuje, da je le-ta preveden v obliko, primerno za izvedbo na nekem ciljnem procesorju. Če to ni mogoče, je priporočljivo programski kod prevesti v vmesno obliko, ki je primerna za izvedbo na specifičnem virtualnem stroju. Kompleksna dinamika večine dinamičnih jezikov zahteva izjemno zapleten proces prevajanja, kar je razlog, da je večina teh jezikov interpretirana in s tem precej neučinkovita. Jezik Z_0 je navkljub svojim dinamičnim značilnostim preveden v vmesno obliko, ki je primerna za izvedbo na arhitekturi Z. Prednost prevajanja v vmesno abstraktno obliko je tudi v tem, da je vmesni kod precej preprosteje prevesti v povsem konkreten kod za neko ciljno

arhitekturo. Direktno prevajanje kompleksnega in izrazno močnega jezika za konkretno arhitekturo je namreč zelo zahtevno opravilo, saj je semantični prepad med obema nivojema zelo širok. Glede na statični sistem tipov se na prvi pogled dozdeva, da kakšnih posebnih dinamičnih atributov v jeziku ne moremo implementirati. Vendar statika sistema tipov ne pomeni nič drugega, kot zmožnost preverbe tipov v času prevajanja. Dinamika polimorfnega sistema tipov v času izvajanja ostaja neokrnjena.
Med načrtovane, vendar še ne implementirane jezikovne koncepte sodi tudi čista abstrakcija tipov. Iz palete modernih objektno usmerjenih jezikov vemo, da je ločevanje med implementacijo neke funkcionalnosti in čisto abstraktno predstavitvijo te funkcionalnosti izjemno pomembno. Razred kot klasifikacijski konstrukt naj bo ločen od tipa, ki je povsem abstrakten pojem. Java in $C^{\#}$ za čisto abstrakcijo tipa uporabljata vmesnike. Jezik Z_0 bo za abstrahiranje nad tipom vključeval konstrukt `type`.

Ena izmed primarnih lastnosti objektno usmerjenega jezika je zmožnost dedovanja funkcionalnosti obstoječih razredov. Čeprav je dedovanje uporabljeno zelo pogosto, so pravila za njegovo uporabo večkrat zelo zapletena. Ne glede za kakšno vrsto dedovanja gre, morajo biti pravila, s katerimi se mehanizem dedovanja uporablja, čista in preprosta. C++ je primer jezika z zelo kompleksnimi in nejasnimi pravili, ki otežujejo tako razumevanje samega hierarhičnega modela kot tudi implementacijo mehanizma dedovanja. Java ima precej preprostejša pravila, saj ne uporablja eksplicitnih mehanizmov za (ne)virtualno dedovanje niti nima posebnih pravil glede dostopa. Vse te prednosti smo poskušali udejaniti v našem jeziku. Dedovanje je preprosto vendar ohranja učinkovitost in izrazno moč v modeliranju problema.
Naslednja lastnost, ki jo zahtevamo, je kanoničnost jezikovnih konstruktov. Vsak konstrukt, ki ga jezik podpira, mora biti definiran na enoumem in razumljiv način. Konstrukt mora imeti natančno specificirano nalogo in namen, od katerega ne sme odstopati. Izrazna moč posameznega konstrukta je subjektivna stvar, zato je posebej ne obravnavamo. Ko govorimo o kanoničnosti konstruktov, mislimo na lastnost elementarnosti, ki konstrukte klasificira kot nekaj osnovnega, nedeljivega. Elementarnost ali ortogonalnost konstruktov pomeni, da za vsako stvar obstaja samo en specifičen konstrukt. Objektno usmerjen jezik $C^{\#}$ nima ortogonalnega konstrukta kar zadeva

abstrahiranje nad podatki, saj lahko le-to izvedemo z uporabo konstruktov razreda ali strukture. Ker si v našem jeziku tega ne želimo, smo ga načrtovali tako, da je za vsako nalogo namenjen zgolj en konstrukt. To poenostavlja semantiko jezika in prinaša večjo čistost sintaktične strukture. Edina slaba stran takšne kanonizacije jezika je kompleksnejša in dolgotrajnejša faza načrtovanja.

S komercialnega stališča je hitro učenje programskega jezika zelo pomembno. Ker čas učenja jezika pada z njegovo preprostostjo, si želimo, da bi jezik bil razumljiv in karseda nezahteven. Seveda je to velikokrat izključujoče, saj moderni objektno usmerjeni jeziki zahtevajo tako splošnost kot učinkovitost. Na preprostost jezika najbolj vpliva njegova sintaksa. Ta mora biti dovolj nazorna, nezapletena in lahko berljiva. Kompleksnost sintakse povzroča kompleksnost celotnega jezika. Da bi zagotovili preprost sintaktični model jezika, smo najprej zasnovali abstraktno sintakso, ki služi za osnovo konkretnejšemu zapisu v razpoznavalniku. Sintaksa jezika mora biti načrtovana na samem začetku, saj je sintaktična analiza ena izmed prvih faz načrtovanja jezika. Na sintakso ne vplivata samo splošnost in izrazna moč, temveč tudi tip jezika - imperativnost oz. deklarativnost. Slednji so sintaktično precej preprosteši od imperativnih. Ker je Z_0 jezik imperativne narave, je njegova sintaktična struktura kompleksnejša kot tista pri funkcijskih jezikih. Kljub temu je sintaksa zasnovana preprosto in čisto. Zaradi popularnosti je sintaksa Z_0 zelo podobna tisti, ki jo ima Java. Pa ne samo zato. Java ima namreč sloves preprostega in berljivega objektno usmerjenega jezika, kar si zagotovo želimo tudi mi.

Specifikacije regularnih izrazov za pregledovalnik smo napisali v notaciji kot jih razume generator pregledovalnikov Flex. Ta vhod razdeli na terminale, ki jih uporabljamo za sintaktično analizo programa. Slednjo smo opravili z generatorjem razpoznavalnikov Yacc. Izhod iz razpoznavalnika je s semantičnimi informacijami obogateno sintaktično drevo, ki opisuje strukturo vhodnega programa. Sintaktično drevo nato služi za vhod prevajalniku, ki generira zložni kod.

4.1.1 Jezikovni konstrukti

Med konstrukte jezika štejemo samo tiste entitete, ki stojijo same zase kot zaključene celote. Na najbolj splošni ravni obstajata dve vrsti konstruktov. Prvi so tisti, ki

omogočajo abstrahiranje nad podatki in njihovo kapsulacijo, drugi pa funkcionalnost jezika. Ker gre za objektno usmerjen jezik osnovan na razredih, je primarni konstrukt za abstrakcijo podatkov seveda razred. Razredni konstrukt ima zelo splošno uporabo, saj omogoča abstrakcijo v širšem pomenu. Abstrahiranje v okviru razreda se izvaja skozi klasifikacijo, kapsuliranje in skrivanje.

Ker si želimo ortogonalnosti konstruktov, smo za abstrakcijo nad podatki namenili izključno razred. Razred služi kot abstracijski in klasifikacijski mehanizem, povrhu pa še kot mehanizem tipiziranja. Ker je razred edini tovrstni konstrukt, je smiselno, da mu namenimo tudi vlogo tipa. Če je razred lahko tudi tip, pa to ne pomeni, da je razred edini mehanizem tipiziranja. Ločevanje med konceptoma tipa in razreda je zelo pomembno, saj je tip nekaj povsem abstraktnega, medtem ko je razred mehanizem za implementacijo. Relacija, ki je smiselna med dvema razredoma, bodisi s stališča logične povezanosti ali deljenja obstoječe funkcionalnosti, ni nujno smiselna tudi med pripadajočima tipoma. Ne glede na to, da lahko razred opravlja funkcijo tipa, je vpeljava dveh različnih mehanizmov pametna odločitev. Jezik vključuje konstrukt `class`, v prihodnosti pa bo vključeval še `type`.

Ker je jezik Z_0 imperativne narave, je glavna operacija manipulacija s spremenljivkami. Če čisti funkcijski jeziki ne poznajo stranskih učinkov, je uporaba le-teh v imperativnih (proceduralnih in objektno usmerjenih) zelo pogosta. Prav zaradi tega imajo praktično vsi tovrstni jeziki možnost konstruiranja zank in stavkov, ki se izvedejo samo ob določenih pogojih. Jezik Z_0 ima konstrukte za implementacijo zank `while`, `do while` in `for`. Zanke imenujemo iterativni stavki (iterative statements). Vsi omenjeni iterativni stavki so dobro znani in uveljavljeni, saj so že lep čas prisotni v jeziku C++ in Java. Čisti objektni programski model narekuje, da so vse entitete jezika predstavljene kot objekt, torej tudi zanke. Implementacija iteracijskih konstruktov mora biti izvedena zelo pazljivo, saj bo v obratnem primeru trpela učinkovitost celotnega izvajalnega procesa. Sleherna zanka je sestavljena iz pogoja in programskega koda, ki se ob izpolnjenem pogoju izvede. Zaradi implementacijskih razlogov, ki bodo opisani kasneje, imajo zanke programski kod vedno umeščen znotraj bloka. Tako lahko vse omenjene zanke predstavimo na povsem uniformen način kot objekt, ki vsebuje logični pogoj in blok. Zanke imajo definirane samo osnovne operacije, ki so smiselne nad zankami.

Vsak imperativni programski jezik vsebuje nek mehanizem za nadzor toka izvajanja programa. Logične vejitve smo implementirali s stavkom `if`. Stavek ima podobno obliko kot v Javi, le da naš, podobno kot pri zankah, vedno operira nad blokom. Ker gre za logično vejitev, je pogoj stavka vedno logična vrednost `true` ali `false`.

Konstrukt, v katerega lahko vstavimo programske stavke, je blok. Blok je pravzaprav konstrukt za izvedbo najprimitivnejše in najmanjše funkcionalnosti. Zaradi tega je tudi metoda po svojem bistvu samo blok. Vsak blok, tudi tisti, ki definira metodo, ima lahko poljubno število vgnezdenih blokov. V večini objektno usmerjenih jezikov ima blok povsem sintaktični pomen. Njegova deklaracija pomeni grupiranje oz. združevanje stavkov, ki so na tak ali drugačen način povezani. Deklaracija spremenljivk znotraj bloka na zunanje deklaracije nima vpliva, saj sploh ne obstaja. Na ta način dosežemo skrivanje in lokalizacijo deklaracij, ki se nanašajo samo na del programskega koda. Skratka, blok je sintaktična struktura, ki se pri prevajanju izgubi, tako da v času izvajanja več ne obstaja.

Bloki v našem jeziku sintaktični pomen obdržijo v popolnosti. Zelo pomembna, dodatna lastnost blokov v Z_0 pa je, da se njihov koncept ohranja ves čas, tudi ob izvajanju. Tudi bloki so predstavljeni kot primitivni objekti, nad katerimi lahko izvajamo definirane operacije. Blok se ustvari dinamično na mestu, kjer je definiran. Objektna predstavitev nam omogoča, da z blokom manipuliramo kot s spremenljivko tipa `Block`. Ker je smiselno, da blok referencira spremenljivke definirane v starševskih okoljih, mora blok imeti način dostopa do teh spremenljivk. Iz tega razloga v blok vključimo okolje bloku lastnih spremenljivk, ki so definirane znotraj bloka kot tudi vsa okolja starševskih blokov, katerih spremenljivke se referencirajo. Takšna implementacija predstavlja kompromis med prostorsko in časovno učinkovitostjo. Blok, ki vsebuje okolje vrednosti in programski kod, se tradicionalno imenuje zaprtje (closure) [14, 87]. Zaprtja se zelo pogosto uporabljajo v interpretiranih funkcijskih jezikih.

Zaradi omenjene predstavitve so vsi konstrukti prvorazredne vrednosti, kar pomeni, da lahko z njimi manipuliramo kot bi to počeli s spremenljivkami ali literali. Prvorazredne vrednosti lahko ustvarjamo, shranjujemo, prenašamo in nad njimi invociramo metode. Nad vsemi entitetami, ki sestojijo iz enega ali več blokov, obstajajo omejitve. Ker so tako zanke kot bloki shranljive vrednosti, pomeni, da lahko ubežijo okolju, v katerem

so bile deklarirane. Implementacija blokov, ki ohranja statično tipiziranje jezika, je torej zahtevna.

Tudi metoda je samo blok. Razlika med obema je v tem, da blok ni poimenovan, na metodo pa se lahko sklicujemo z njenim imenom. Blok je v osnovi torej anonimen, metoda pa ne. Metoda je entiteta, ki jo sestavlja vsaj en blok. Ker se bloki ustvarjajo dinamično, se to odraža tudi v metodah. Metode so, podobno kot bloki in zanke, prvorazredne vrednosti, kar pomeni, da jih lahko manipuliramo na enak način. Metoda je samo bolj formalna oblika bloka, ki lahko sprejme tudi parametre in vrača vrednost določenega tipa. Metode se lahko v času izvajanja dinamično spreminjale, pri čemer je treba zagotoviti statično varnost. Ker je metoda natančno določena s svojo signaturo, je ta njihova najpomembnejša lastnost, vsaj kar zadeva sistem tipov. Ker razredi nimajo lastnosti specificiranih v obliki instančnih spremenljivk, kot je to primer v C++ ali Javi, je dinamika metod še toliko pomembnejša. Dinamično spreminjanje metod je edini mehanizem, s katerim bo mogoča alternacija objektovega stanja. Dinamično spreminjanje metod dopuščajo nekateri dinamično tipizirani objektni jeziki. Mi smo mehanizem prenesli pod okrilje statičnega sistema tipov.

Spreminjanje metod nima samo praktičnega pomena, ampak tudi globlje teoretično ozadje. Objekt si lažje predočimo kot zbirko metod. Na instančno spremenljivko lahko gledamo samo kot na okleščen koncept metode. Instančna spremenljivka je preprosto metoda, ki ne uporablja implicitnega parametra `self` ali `this`. Tudi skrivanje objektovega stanja lahko posplošimo na metode, saj imajo tudi te specifikatorje dostopa. Enačenje instančnih spremenljivk z metodami omogoča trivialno pretvorbo med njimi, saj lahko pasivne podatke aktiviramo v kalkulacijo in obratno. Unifikacija instančnih spremenljivk in metod pomeni preprostost, saj je struktura objekta identična znotraj in zunaj njega. Tudi lastne metode objekta spreminjajo objekt samo preko invokacij. Ločevanje metod od spremenljivk pomeni dva različna načina manipulacije objekta. Eksplicitna manipulacija stanja poteka preko instančnih spremenljivk, implicitna pa z invokacijo metod.

Unifikacija ponuja preprostejši formalni model objektnega jezika. Čeprav zahteva sposobnost dinamičnega spreminjanja metod, nudi vrsto drugih prednosti (zgoščenost zapisa, preprostost,...). Spreminjanje instančnih spremenljivk se preprosto preslika v

spreminjanje metod. Spreminjanje metode lahko služi kot osnova za obliko dinamičnega dedovanja, saj omogoča dinamiko na nivoju metod. Povrhu tega pa je dinamika metod statično preverljiva, saj za spreminjanje metod zahtevamo enaka pravila kot pri dedovanju. Na ta način lahko simuliramo dinamično dedovanje, ki v splošnem ni statično varno [98] in hkrati zagotovimo konformanco tipov.

4.1.2 Objektni model

Objektni model je skupek lastnosti, ki določajo objektno usmerjenost jezika Z_0. Ena izmed lastnosti je uniformna predstavitev jezikovnih entitet. Ta zahteva, da so vse entitete, podatkovne ali kontrolne, predstavljene na identičen način. Jezik Z_0 temu zadošča, saj je vsak podatek in vsak konstrukt predstavljen z objektom pripadajočega tipa. S tem ne zagotovimo samo unifikacije predstavitve, temveč tudi unificiran dostop in manipulacijo vseh entitet. Lastnosti objektnega modela jezika so postavljene tako, da je koncepte jezika mogoče učinkovito implementirati z omejitvijo statičnega tipiziranja. Nesmiselno bi bilo v statično tipiziran jezik vključiti npr. koncept dinamičnega dedovanja.

Referenciranje vsakega objekta je realizirano preko reference, ki je, konkretno gledano, samo kazalec na pomnilniško lokacijo, kjer se objekt prične. Vse operacije nad objekti operirajo nad referencami. S tem se precej poenostavita sam model in implementacija. Ker gre za statično tipiziran jezik, je tip objekta znan že v času prevajanja. Kljub temu so v času izvajanja potrebne dinamične preverbe. Če je pretvorba tipa navzgor (narrowing) trivialna, obratno (widening) ne velja. Objekt v jeziku Z_0 imenujemo instanca razreda, iz katerega se objekt ustvari. Osnovni razred celotne hierarhije je `Object`, s katerim hierarhijo sklenemo. Vsi podatki so shranjeni v objektih, ne glede na to ali gre za primitivne ali uporabniško definirane vrednosti.

Edina operacija nad objektom je invokacija metode. To zagotovimo s tem, da onemogočimo instančne spremenljivke. Razred ima samo metode. Med te prištevamo funkcijske in proceduralne abstrakcije. Vsakršna manipulacija objektovega stanja se izvršuje preko invokacije ene izmed metod. Invokacijo metode lahko razumemo kot odgovor na sporočilo objektu, pri čemer ime sporočila sovpada z imenom metode. Metoda, ki predstavlja neko objektovo lastnost ne izvaja manipulacije z instančno spremenljivko, am-

pak odraža stanje neposredno. To pomeni, da je sama del stanja. Objektovo stanje je potemtakem popolno definirano z razrednimi metodami. S tem je obrazložena potreba po konceptu dinamičnega spreminjanja metod. Metode tako niso več vezane na družino objektov istega tipa, temveč opisujejo stanje vsakega objekta posebej. Vse kar ostaja enako za vse objekte istega tipa je programski kod in funkcionalnost metod. Zakaj smo se odločili za tak pristop? Posplošitev objektovih lastnosti v metode ima svoje korenine v teoriji objektnega računa [2, 3]. Unifikacija metod in atributov pomeni čistejši objektni model jezika, saj zagotovimo, da je edina operacija nad objektom res invokacija. Manipulacija je na ta način bolj poenotena, saj se stanje objekta manipulira izključno preko metod. Formalni opisi objektnega računa so čistejši kadar imamo opraviti samo z instančnimi spremenljivkami ali samo z metodami, vendar ne oboje hkrati. Ker so metode bolj splošne od instančnih spremenljivk, slednje povišamo v metode. Obratno seveda ni mogoče.

Podoben objektni model načemu ima jezik Smalltalk. Objekti primitivnih tipov (Integer, String) so v Smalltalk-u konstantni (immutable). To pomeni, da se njihova vrednost nastavi samo enkrat in ostane nespremenjena celoten življenski cikel objekta. Nezmožnost spreminjanja objektove vrednosti ima velik vpliv na splošno performanso izvajanja. Namesto spremembe vrednosti, se konstantni objekti ustvarjajo na novo. Ustvarjanje objekta je časovno zahtevna operacija, zato bi se ji bilo bolje izogniti, če je to le mogoče. Primitivne objekte smo implementirali tako, da jih je mogoče ustvarjati na novo, klonirati in spreminjati vrednost. S tem se izognemo nepotrebnemu poustvarjanju objektov, ki bi se sicer pojavili kot rezultati nekih kalkulacij. Seveda je spreminjanje vrednosti primitivnih objektov mogoče samo pri t.i. levih vrednostih (l-value), torej takšnih, ki lahko stojijo na levi strani izrazov.

Z objektnim modelom jezika je pogojeno tudi konceptualno modeliranje. Tukaj gre izpostaviti predvsem koncepta abstrakcije in specializacije. Slednja ima svoje korenine v dedovanju, zato je le-to eden izmed najpomembnejših konceptov jezika. Več o dedovanju v Z_0 bomo opisali v nadaljnjih podpoglavjih.

Čistost objektnega modela omogoča veliko izrazno moč jezika, saj sta z njo unificirana tako dostop kot struktura. Koncept "vse je objekt" je nadvse dobrodošel, saj lahko celotno okolje programa preslikamo v množico pasivnih ali aktivnih (med katerimi ob-

staja interakcija) objektov. Ker tudi kontrolne strukture, torej bloke, zanke in ostale sintaktične strukture, predstavimo z objekti, jih lahko med seboj poljubno povezujemo in sestavljamo skoraj poljubne uporabniško definirane strukture.

4.1.3 Sistem tipov

Jezik Z_0 je statično tipiziran programski jezik, kar pomeni, da se preverjanje tipov vrši v času prevajanja. Ker gre za objektno usmerjen programski jezik, je potrebno nekatere pretvorbe preverjati tudi v času izvajanja. Jezik Z_0 je močno tipiziran, saj ima vsaka programska fraza jezika predpisan tip. Za statičen sistem tipov se odločamo, ker ta omogoča večjo učinkovitost kar zadeva izvajanje. Dinamičen sistem bi sicer bil fleksibilnejši in izrazno močnejši, vendar tudi počasnejši. Ker je jezik (polovično) preveden in ne interpretiran, kot večina dinamičnih jezikov, je naša odločitev smotrna.
Sistem tipov predstavlja bistvo jezika, saj na nek način povezuje vse pomembne koncepte jezika. Dinamičnen sistem tipov je najmočnejša oblika tipiziranja, ki si jo lahko v močno tipiziranem jeziku privoščimo. Tak sistem ima zelo visoko izrazno moč. Drugače je s statičnim tipiziranjem. Ta velikokrat pomeni določeno stopnjo omejitev, kar postane še posebej očitno, ko tipov ne moremo natančno določiti v času prevajanja. Zaradi tega smo v jezik Z_0 vključili več različnih mehanizmov tipiziranja.
Implementirali smo klasično dinamično tipiziranje, ki se pojavi kot stranski učinek specializacije ali razširjanja funkcionalnosti z uporabo dedovanja. Povsod kjer v času prevajanja ne bo mogoče natančno določiti dejanskega tipa, bo treba vstaviti dinamično preverbo tipa. Preverba se bo izvedla eksplicitno z mehanizmom pretvorbe tipa ali implicitno, ki jo bo generiral prevajalnik.
Ker želimo fleksibilen sistem tipov, ki ga lahko umestimo v statično tipiziranje, je bilo treba implementirati za to posebej namenjene konstrukte. Eden izmed teh je tip `Self`, ki predstavlja tip vrednosti `self`. Tip `Self` je povsem dinamičen tip, s katerim je določena dejanska instanca razreda. Tip `Self` omogoča varno tipiziranje izrazov, ki se nanašajo na dejansko instanco in ne na razred, katerega predpostavljamo. Tako se izognemo nepotrebnim in večkrat neelegantnim pretvorbam tipov, ki se pogosto pojavljajo v skoraj vseh sodobnih (Java, C++, $C^{\#}$) statično tipiziranih jezikih. `Self` osnovnim razredom hierarhije daje možnost referenciranja objektov izpeljanih razre-

dov. Koncept je implementiran v jezikih Eiffel [75], Polytoil [23] in Sather [96, 78]. Velika večina modernih objektno usmerjenih jezikov ima invariantne sisteme tipov. To pomeni, da je signaturo metod pri njihovi redefiniciji potrebno ohraniti. Včasih je to precejšnja omejitev, saj je narava koncepta specializacije navadno takšna, da specializirana metoda operira nad tipi, ki so specializacija tipov metode v starševskem razredu. V poglavju o funkcijskem podtipu smo izpeljali relacijo podtipa z namenom ugotoviti kompatibilno signaturo podmetode. Najmočnejši statični sistem tipov je takšen, ki omogoča kovariantne spremembe v tipu vračanja metode in kontravariantne spremembne v argumentih. Ker smo implementirali tip `Self`, bomo dosegli omejene kovariantne spremembe tudi v tipih argumentov. Tip `Self` lahko namreč stoji na mestu vračanja ali argumenta. Implementirali smo torej sistem tipov s tremi preprostimi pravili: kovarianca zaloge vrednosti, kontravarianca domene in invarianca tipa `Self`.

4.1.4 Primitivni tipi

Primitivni tipi so vsi tisti, ki so v jezik vgrajeni in jih ni mogoče spreminjati. Poraja se vprašanje, katere tipe je sploh smiselno vgraditi in jih napraviti del jezika. Tipična kriterija sta frekvenca uporabe in strukturalna kompleksnost tipa. Če je frekvenca uporabe tipa dovolj visoka, je integracija tipa v jezik upravičena. Podobno velja za kompleksnost. Vgrajeni naj bodo samo zelo preprosti in splošni, pogosto uporabljeni tipi. V jeziku smo implementirali dobro znane primitivne tipe, katere najdemo tudi v drugih jezikih. Primitivne tipe razdelimo v dve skupini, številske in znakovne. Med celoštevilske tipe uvrščamo:

- `Integer`, ki predstavlja 32 bitna predznačena cela števila,
- `Short`, ki predstavlja 16 bitna predznačena cela števila in
- `Byte` za 8 bitna predznačena cela števila.

Številska tipa za realna števila (števila s plavajočo vejico) sta:

- `Double`, ki predstavlja 64 bitna realna števila in
- `Float`, s katerim predstavimo 32 bitna realna števila.

Predstavitev realnih števil zadošča standardu enojne in dvojne natančnosti IEEE 754. Poleg številskih ali integralnih tipov se zelo pogosto uporabljajo tudi znakovni nizi. V ta namen bomo implementirali tipa:

- `String`, ki predstavlja znakovni niz poljubne dolžine in
- `Character` za posamezne znake.

Tip, ki zavzame logični vrednosti 0 in 1 se imenuje `Boolean`. Uporablja se v logičnih operacijah, katerih vrednosti temeljijo na vrednostih `true` in `false`.
Vsak primitivni tip ima definirane operacije, ki so primerne in dovolj pogoste, da se jih izplača implementirati kot instrukcije virtualnega stroja. Operacije številskih tipov so torej osnovne računske operacije, kot so seštevanje, odštevanje, množenje in deljenje, ter pretvorbene operacije za pretvorbe med posameznimi primitivnimi tipi. Tip `String` definira pretvorbene in operacije za združevanje ter iskanje po nizu.

Ker imamo opravka z uniformnim objektnim modelom, potrebujemo še tip `Object`, ki predstavlja najvišji razred v hierarhiji. Razred `Object` ni osnovni razred samo uporabniško definiranim sestavljenim razredom, temveč tudi primitivnim. Uniformni objektni model ima svoje prednosti ne samo v "enosti" družinske hierarhije, temveč tudi v uniformni klasifikaciji vseh entitet v programu. Uniformna klasifikacija primitivnih in kompleksnih tipov pomeni izjemno preprosto aplikacijo mehanizmov vključitvenega in parametričnega polimorfizma. Slednji predstavlja precejšen problem v jeziku C++, saj primitivni in kompleksni tipi niso unificirani kot objekt. Vsaka primitivna podatkovna entiteta je objekt v enakem pomenu, kot pri sestavljenih tipih. Princip čiste objektizacije omogoča torej široko uporabo parametričnega polimorfizma, skupen nadrazred pa pomeni "poceni" pridobljen vključitveni polimorfizem.
Ker bi bilo ustvarjanje objekta primitivnega tipa z operatorjem `new` neugodno, se le-ta ustvarja implicitno ob deklaraciji spremenljivke in prirejanju nove vrednosti. Jezik Java ima za vsak vgrajen primitivni tip definiran tudi razred (wrapper) za ta tip, čeprav primitivni tipi niso realizirani kot objekti. Vsak objekt primitivnega tipa v Javi je konstanten, ker se po instanciranju njegova vrednost ne more spreminjati. Vrednost

primitivnega tipa v jeziku Z_0 ni konstanta, saj se lahko med izvajanjem poljubno spreminja, ne da bi se moral ustvariti nov objekt. V jeziku, v katerem so tudi primitivne vrednosti predstavljene kot objekti, je to velika prednost, saj je operacija instanciranja razreda časovno in prostorsko zelo zahtevna. Še posebej se to izkaže pri primitivnih tipih, saj je frekvenca uporabe teh običajno zelo visoka.
Primitivni tipi so referenčni tipi. To pomeni, da se njihova vrednost prenaša enako kot vrednost kompleksnega, uporabniško definiranega tipa. Dostop do dejanske vrednosti primitivnega objekta se zaradi učinkovitosti ne vrši preko metode, temveč direktno. Vrednost je shranjena za tabelo metod in meta podatki znotraj objekta (slika 10). Poudarimo, da z "direktnim dostopom" do vrednosti primitivnega tipa ne mislimo klasičen dostop oblike `stevilo.vrednost`. Direkten dostop se vrši znotraj virtualnih instrukcij. Dostop na jezikovni ravni je povsem transparenten in ima obliko npr. `stevilo = 5`.

Slika 10: Dejanska vrednost primitivnega objekta je shranjena za vsemi skupnimi podatki objekta, ki so enaki kot pri kompleksnih objektih.

Čisti objektni princip zahteva, da je edina operacija nad objektom invokacija metode. Ker je vsaka primitivna vrednost predstavljena kot objekt, so tudi operacije nad primitivnimi vrednostmi samo metode. Izvedba operacije torej zahteva invokacijo objektove metode. Danes vemo, da je invokacija precej zahtevna operacija, saj terja vpogled v virtualno tabelo, kalkulacijo naslova in nazadnje prenos izvajanja na izbrano metodo. Izvedba na videz preprostih operacij nad primitivnimi vrednostmi se na ta način precej zakomplicira, predvsem s stališča učinkovitosti. Tudi najpreprostejše operacije, kot so seštevanje, odštevanje in množenje, se prevedejo v eksplicitne invokacije pripadajočih metod. Da bi to očitno neučinkovitost premostili, operacij nad primitivnimi tipi nismo implementirali z invokacijo temveč z virtualnimi instrukcijami. Vsaka operacija, definirana nad primitivnim tipom ima svoj ekvivalent v instrukciji virtualnega stroja. Pred-

nost takšne implementacije je očitna, saj je izvedba ene, četudi virtualne, instrukcije neprimerno hitrejša kot invokacija metode, ki mora realizirati identično funkcionalnost kot instrukcija. Ker jezik omogoča dedovanje in pozno povezovanje, v času prevajanja ni vedno mogoče enoumno določiti, katera metoda se bo v času izvajanja dejansko izvedla. S tem nastane težava, saj operacij nad primitivnimi tipi ne moremo preprosto prevesti v virtualne instrukcije, saj bi v času izvanja lahko prišlo do nekonsistence. Do te pride, v kolikor bi želeli neko operacijo, ki je definirana nad primitivnim tipom, prevesti v virtualno instrukcijo ob pogoju, da razred primitivnega tipa ni zaključen (sealed). To pomeni, da lahko ta razred dedujemo in razširimo njegovo funkcionalnost v podrazredu. Na mestu, kjer pričakujemo operacijo primitivnega tipa, v času prevajanja ne moremo določiti virtualne instrukcije, saj bo v času izvajanja na tem mestu lahko vrednost osnovnega ali izpeljanega tipa. Pravilo subsumpcije je namreč eno izmed temeljnih in ga moramo nujno ohraniti. Takšno operacijo je potrebno prevesti v invokacijo metode. Česar pa si ne želimo. Najpreprostejša rešitev, v okviru katere bi ohranili čisti objektni model in učinkovitost izvajanja je ta, da razrede primitivni tipov preprosto zapečatimo. Preprečimo dedovanje na način, pri katerem bi lahko razred primitivnega tipa bil osnovni razred nekega drugega razreda. Nezmožnost dedovanja iz razreda primitivnega tipa se morda zdi omejujoča, vendar omogoča precej preprostejšo in predvsem učinkovitejšo realizacijo funkcionalnosti primitivnega tipa. Poseben primer je tip `Object`, ki pravtako spada med primitivne tipe, vendar pa lahko iz njega dedujemo. Izvedba metode nad objektom tega tipa bo zato vedno prevedena v virtualno invokacijo, saj bo dejanski `Object` skoraj vedno objekt nekega podtipa.
Pretvorba objekta primitivnega tipa v `Object` ne predstavlja posebne težave. Operacije nad objektom so vedno izvedene preko klica virtualnih metod. Za izvedbo operacije nad dejanskim, pozno povezanim objektom pa je tako ali tako potrebna inverzna pretvorba v originalni tip. Smisel virtualne invokacije je v dinamični povezanosti metode in dejanskega objekta, tj. tistega ki je bil ustvarjen in ni nujno omejen s svojim statičnim tipom. Ker je virtualni klic metode torej dinamično vezan, ne moremo vseh operacij nad primitivnim zapečatenim tipom prevesti v virtualne instrukcije. Če imamo metodo `print` definirano v razredu `Object` in njegovim podrazredom `Integer`, je potrebno metodo nad osnovnim objektom izvesti kot invokacijo (b), nad izpeljanim pa kot virtualno instrukcijo (a).

```
Integer a = 5;
```

```
a.print;                  // (a)
(a  as Object).print;     // (b)
```

Izvedba metode nad objektom tipa `Object` se vedno prevede v invokacijo, ne glede ali je dejanski objekt na tem mestu res `Object` ali kaj drugega. V času prevajanja imamo na voljo namreč samo statični sistem tipov. To pomeni, da morajo tudi primitivni tipi imeti možnost izvedbe svojih operacij preko virtualne invokacije. Zaradi optimizacije smo v virtualno tabelo umestili samo metode, ki so že definirane v razredu `Object`.

4.2 Zapis sintaktičnih pravil

Sintaktično strukturo jezika bomo pričeli pri vrhu. Osnovni konstrukt za abstrahiranje podatkov je razred. Razred sestoji samo iz metod. BNF zapis sintakse jezika lahko v specifikacije orodju Yacc zapišemo skoraj brez sprememb. Produkcija za definicijo razreda je sledeča:

```
ClassDefinition:   CLASS  IDENTIFIER ClassExtensionOpt
                  '{' FieldDeclarations '}'
```

Konvencija poimenovanja je takšna, da terminalne simbole pišemo z velikimi črkami, produkcije pa z veliko začetnico.

Seznam nadrazredov in tipov zapišemo s sledečimi produkcijami:

```
ClassExtensionOpt:  // epsilon
                    | ClassExtensions
                    ;

ClassExtensions:    ClassExtension
                    | ClassExtensions ClassExtension
                    ;


ClassExtension:      INHERITS ClassList
                    |  IMPLEMENTS ClassList
```

```
                    ;

ClassList:    QualifiedIdentifier
              | ClassList ',' QualifiedIdentifier
              ;
```

Produkcija `FieldDeclarations` predstavlja eno ali več deklaracij metod in ima sledečo obliko:

```
FieldDeclarations:   FieldDeclaration
                   | FieldDeclarations FieldDeclaration
                   ;

FieldDeclaration: FunctionDeclaration
                   | ProcedureDeclaration
                   ;
```

Ker želimo ločevati med procedurami in funkcijami že na sintaktičnem nivoju, bomo produkciji zapisali ločeno. Razlika med obema je, da funkcija vedno vrača neko vrednost, procedura pa ne. Sintaksa funkcije je naslednja:

```
FunctionDeclaration: MethodAccessModifier IDENTIFIER MethodParametersOpt
                     ':' Type
                     MethodBody;
```

Iz zapisa produkcije je razvidna sintaksa definicije funkcije. Procedura je enaka, le brez tipa vračanja:

```
ProcedureDeclaration: MethodAccessModifier IDENTIFIER MethodParametersOpt
                      MethodBody;
```

Tako funkcija kot procedura lahko imata poljubno dolg seznam formalnih parametrov. Seznam parametrov je opcijski:

```
MethodParametersOpt:
                     | '(' MethodParameterList ')'
                     ;
```

Deklaracija formalnega parametra je podobna deklaraciji spremenljivke. Sestavljata jo tip parametra in njegovo ime. Ker ima seznam lahko poljubno dolžino, produkcijo zapišemo levo rekurzivno:

```
MethodParameterList:   Type IDENTIFIER
                     | MethodParameterList ',' Type IDENTIFIER
                     ;
```

Metodi pripada še modifikator dostopa, ki je lahko:

```
MethodAccessModifier:        PUBLIC
                           | PROTECTED
                           | PRIVATE
                           | RESTRICTED
                           ;
```

Bistvo metode se nahaja v njenem bloku, ki je sestavljen iz lokalnih deklaracij in stavkov:

```
MethodBody: Block;


Block:   '{' LocalsOrStatementsOpt '}';
```

Ker je deklaracija lahko ena, več ali nobena, produkcijo spet zapišemo rekurzivno:

```
LocalsOrStatementsOpt:
                      | LocalsOrStatements
                      ;


LocalsOrStatements:     LocalsOrStatement
                      | LocalsOrStatements LocalsOrStatement
                      ;
```

Ker lahko imamo v bloku lokalne deklaracije ali stavke, to zapišemo:

```
LocalOrStatement:   Type LocalVariableDeclarations Semicolon
                  | Statement
                  ;
```

Deklaracija spremenljivk lahko hkrati imenuje eno ali več spremenljivk istega tipa:

```
LocalVariableDeclarations:   LocalVariableDeclaration
                           | LocalVariableDeclarations ','
                             LocalVariableDeclaration
                           ;
```

Lokalna spremenljivka je deklarirana brez ali z inicializacijskim izrazom na desni strani:

```
LocalVariableDeclaration:    IDENTIFIER
                           | IDENTIFIER '=' Expression
                           ;
```

Ostanejo še stavki. Ločevali bomo med izrazi in stavkom `return`. Vsi stavki razen tega bodo namreč predstavljeni kot objekt. Stavek `return` ne bo prvorazredna vrednost:

```
Statement:    Expression Semicolon
            | ReturnStatement
            ;
```

Stavek `return` lahko vrača neko konkretno vrednost ali pa preprosto zapusti izvajanje procedure:

```
ReturnStatement:    RETURN Expression Semicolon
                  | RETURN Semicolon
                  ;
```

Zaradi čistejšega zapisa sintakse jezika bomo stavke, ki predstavljajo kontrolne konstrukte, zapisali posebej:

```
OtherStatement:   WhileStatement
                | DoWhileStatement
                | ForStatement
                | IfStatement
                | Block
                ;
```

Če bomo v prihodnosti želeli kak stavek dodati, ga bomo zapisali na tem mestu. Iteracijska stavka `while` in `dowhile` sestavimo takole:

```
WhileStatement:  WHILE Block Block;

DoWhileStatement:  DO Block  WHILE Block;
```

Stavek `for` je sestavljen iz več blokov:

```
ForStatement:  FOR Block Block Block Block;
```

Stavek `if` ima tri različice:

```
IfStatement:     IF Block  THEN Block
              |  IF Block  THEN Block  ELSE Block
              |  IF Block  THEN Block  ELSE IfStatement
              ;
```

Z zadnjo alternativo dosežemo lepši in berljivejši zapis večnivojskih vejitev.

Tip entitete je lahko eden izmed vgrajenih tipov ali uporabniško definiran:

```
Type:     PredefinedType
        | QualifiedIdentifier
        ;
```

Predefinirani tipi so sledeči:

```
PredefinedType:      BYTE
                   | CHAR
                   | SHORT
                   | INT
                   | LONG
                   | FLOAT
                   | DOUBLE
                   | STRING
                   | VOID
                   | BOOLEAN
                   ;
```

Izrazi so stavki, ki se najpogosteje uporabljajo. Zaradi izrazne moči jezika smo sintakso jezika zasnovali tako, da je vse razen stavka `return` izraz. Produkcije za izraze pričnemo z operatorjem, ki veže najšibkeje, to je prireditev. Zapise produkcij bomo nadaljevali tako, da bomo upoštevali prioriteto operatorjev, ki v njih nastopajo:

```
Expression:   AssignmentExpression;


AssignmentExpression:   LogicalExpression
                      | IDENTIFIER '='  AssignmentExpression
                      | METHOD UpdateReference '=' AssignmentExpression
                      ;

LogicalExpression:   BooleanExpression
                   | LogicalExpression '&&' BooleanExpression
                   | LogicalExpression '||' BooleanExpression
                   ;


BooleanExpression:    AdditiveExpression
                    | BooleanExpression '<'  AdditiveExpression
                    | BooleanExpression '>'  AdditiveExpression
                    | BooleanExpression '<='  AdditiveExpression
                    | BooleanExpression '>='  AdditiveExpression
                    | BooleanExpression '=='  AdditiveExpression
                    | BooleanExpression '!='  AdditiveExpression
                    ;
```

Produkcijo za spreminjanje metode zapišemo:

```
UpdateReference:   FieldReference
                 | MethodReferenceWithArgs
                 ;
```

Produkcijo za logični izraz bi lahko zapisali s samo eno alternativo, vendar je ta zapis

bolj nazoren.

Sledijo izrazi za operacije nad številskimi tipi:

```
AdditiveExpression:   MultiplicativeExpression
                    | AdditiveExpression '+'  MultiplicativeExpression
                    | AdditiveExpression '-'  MultiplicativeExpression
                    ;

MultiplicativeExpression:   PowerExpression
                          | MultiplicativeExpression '*' PowerExpression
                          | MultiplicativeExpression '/' PowerExpression
                          ;

PowerExpression:    CastExpression
                  | PowerExpression '^' CastExpression
                  ;
```

Najvišjo prioriteto vrednotenja imajo izrazi za pretvorbo tipov in primitivni izrazi:

```
CastExpression:     PrimitiveExpression
                  | CastExpression  AS Type
                  | '(' Expression ')'
                  ;
```

Primitivni izrazi so tisti, ki se ovrednotijo najprej. Mednje spadajo literali, referenciranje spremenljivk in konstruktov ter invokacija metod:

```
PrimitiveExpression:   Literal
                     | Literal '.' PartialReference
                     | MethodReferenceWithArgs
                     | FieldReference
                     | NewExpression
                     | '(' Expression  ')'  '.' PartialReference
                     | OtherStatement
                     ;
```

S takšnim zapisom omogočimo, da se konstrukti, ki smo jih zgoraj definirali v produkciji `OtherStatement`, obravnavajo kot prvorazredne vrednosti. Referenciranje metod s parametri smo ločili od referenciranja tistih, ki ne prejmejo nobenega parametra. S tem postane sintaksa jezika bolj zgoščena, saj pri metodah brez argumentov ni potrebno pisati praznih oklepajev ( ). Sintaksa primitivnega izraza omogoča invokacijo metod neposredno nad literali in izrazi:

```
PartialReference:  FieldReference
                 | MethodReferenceWithArgs
                 ;
```

Dostop do metode brez argumentov je preprost:

```
FieldReference:     FieldAccess;


FieldAccess:   QualifiedIdentifier
             | MethodAccess '.' QualifiedIdentifier
             ;
```

Dostop do metode z argumenti pa je podoben:

```
MethodAccess:  QualifiedIdentifier '(' Arguments ')'
             | MethodAccess '.' QualifiedIdentifier '(' Arguments ')'
             ;
```

Slabost ločevanja metod na podlagi argumentov je daljši zapis sintakse, vendar omogoča preprostejši zapis referenciranja metod.

Argumenti metode so preprosto izrazi. Ker je argument lahko en sam ali več, produkcijo zapišemo rekurzivno:

```
Arguments:   Expression
           | Arguments ','  Expression
           ;
```

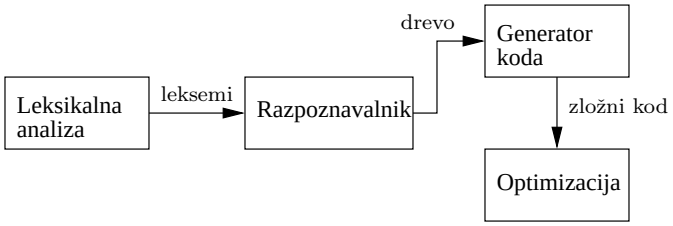
Poseben izraz je ustvarjanje objekta z operatorjem `new`. Sintaksa je sledeča:

```
NewExpression:   NEW Type
               | NEW Type '(' Arguments ')'
               ;
```

Prva alternativa pomeni ustvarjanje objekta s privzetim konstruktorjem, tj. brez argumentov, druga pa omogoča instanciranje s klicem konstruktorja z argumenti.
Zdaj lahko zapišemo osnovne stavke jezika in jih razpoznamo.

4.3 Arhitektura razpoznavalnika

Razpoznavalnik je odgovoren za razpoznavo sintaktične strukture vhodnega vzorca. V najabstraktnejši obliki razpoznavalnik na vhodu prejme program in na izhodu generira sintaktično drevo oz. drevo izpeljav. To drevo nato služi kot vhod prevajalniku, ki program prevede v zložni kod. Celotna struktura prevajalnika z leksikalnim analizatorjem in razpoznavalnikom je prikazana na sliki 11.



Slika 11: Groba struktura prevajalnika.

Seveda bomo strukturo prevajalnika kasneje, ko bodo znani vsi koncepti, zapisali bolj podrobno.

4.3.1 Večprehodna razpoznava

V programskem kodu se na določenem mestu lahko pojavi referenca na metodo ali tip, za katerega na tem mestu še ne obstaja definicija. Tipičen in zelo pogost primer tega je invokacija razredne metode v telesu konstruktorja, kjer je le-ta naveden kot prva metoda razreda:

```
class Student {
```

```
    public Student {
     inicializiraj;    // klic metode
    }
    private inicializiraj {
     ...
    }
   ...
}
```

V konstruktorju izvršimo klic metode `inicializiraj`, ki je navedena za konstruktorjem. Ker je razpoznava sekvenčni proces, v času razpoznavanja telesa konstruktorja v okolju razreda še nimamo podatka o obstoju metode `inicializiraj`. Zaradi tega klica metode ne moremo uspešno prevesti, saj ne poznamo njenega indeksa v virtualno tabelo, niti če bo metoda kasneje v razredu sploh deklarirana. Prva možnost za odpravo te težave je popravljanje za nazaj. Povsod, kjer se pojavi nedoločena referenca, generiramo posebno vozlišče, ki pomeni, da bo ob koncu razpoznave treba preveriti eksaktnost te reference. Vendar izrazi lahko imajo precej kompleksnejšo obliko. Primer takšnega izraza je:

```
vrniStudenta( steviloStudentov-1 ).vrniStarost(‘‘6.6.2004’’).toString.print;
```

Kakšni so rezultati, če metoda `vrniStudenta` sploh ne obstaja? V tem primeru bomo generirali precej kompleksno drevo sestavljenih invokacij povsem zaman. Izkaže se, da takšna strategija ni učinkovita. Zato smo izbrali drug pristop, ki najprej zgradi samo okolje razreda in šele nato prične z razpoznavo programskih fraz. Ker razpoznavalnik potrebuje dva prehoda, ga imenujemo dvopasovni razpoznavalnik (two-pass parser). V drugem prehodu imamo okolje že zgrajeno, zato lahko brez težav ločujemo med pravilnimi in nepravilnimi identifikatorji. Če torej neka metoda v okolju ne obstaja, potem je zagotovo ni in lahko javimo napako.

4.3.2 Drevesa izpeljav

Drevo izpeljave se gradi sproti ob razpoznavi posameznih stavkov jezika. Kadarkoli je nek pravilen vzorec v vhodnem programu razpoznan, se izvede zahtevana akcija. Zgradi se novo drevo dotičnega tipa in se pripne na starševsko drevo, če to obstaja. Na

ta način se drevo izpeljav rekurzivno dograjuje. Drevesa izpeljav smo razbili na nivo metode, kar pomeni, da ima vsaka metoda svoje drevo izpeljav. Drevo se sproti bogati z atributi semantičnih informacij, med katere spadajo imena identifikatorjev, vrednosti literalov in ostalo. Ker bo drevo na koncu potrebno ovrednotiti, ima to svoj tip. Drevo, ki predstavlja npr. iteracijski stavek `for`, poimenujemo `NODE_FOR`. Podobno se drevo za seštevanje imenuje `NODE_PLUS` in ima vedno dve poddrevesi, ki predstavljata levo in desno stran izraza +. Semantične informacije drevesa dobimo iz hierarhične zgradbe okolja, v katerem opravljamo razpoznavo.

Zaradi poenotenja predstavitve in enostavne manipulacije vse vrednosti, ki lahko nastopajo v programih predstavimo kot simbole. Simboli so tako literali, spremenljivke kot tudi metode. Vrednotenje drevesa poteka rekurzivno z vrednotenjem vsakega posameznega vozlišča v drevesu. Če je tip vozlišča preprost, to vozlišče preprosto ovrednotimo. V primeru, ko je tip vozlišča kompleksnejši, se spustimo rekurzivno v ovrednotenje sinov. Predstavitev z drevesi je zelo dobrodošla, saj produkcije v razpoznavalniku med seboj komunicirajo na preprost in enoumen način. Posamezne tipe dreves izpeljav bomo obravnavali sproti.

4.3.3 Okolje

Koncept okolja je zelo pomemben, saj omogoča poizvedovanje po semantičnih informacijah na eleganten in preprost način. Okolje omogoča povezovanje identifikatorjev s programskimi entitetami, med katere prištevamo spremenljivke, literale, metode in tipe. Osnovni operaciji nad okoljem sta izdelava povezave in referenciranje obstoječe povezave. Operacija povezave identifikatorja z entiteto je stranski učinek deklaracije. Okolje preprosto definiramo kot množico povezav identifikatorjev s povezljivimi vrednostmi:

$$\Pi = \{I_1 : t_1, \ldots, I_n : t_n\}$$

S $t_1, \ldots, t_n$ predstavimo izraze tipov in z I imena identifikatorjev. Ker v okolju potrebujemo tudi starševsko okolje, ga dodamo:

$$\Pi_{k+1} = \{\Pi_k;\ I_1 : t_1, \ldots, I_n : t_n\}, \quad k \geq 0\ \wedge\ \Pi_k = \{\ \} \quad \text{za } k = 0$$

Povezljive vrednosti so vse takšne vrednosti, ki jih lahko povežemo z identifikatorjem.

Povečini velja, da so povezljive vrednosti tudi prvorazredne vrednosti. Ker smo sintakso jezika definirali na način, da je skoraj vse izraz in ker je izraz povezljiva vrednost, so potemtakem skoraj vse entitete v našem jeziku povezljive vrednosti.
Poleg povezav, okolje določa še doseg posamezne entitete. Deklaracije, ki izvedejo neko povezavo v okolju, niso dosegljive izven tega okolja. Tako omogočimo lokalizacijo, ki je zelo dobrodošla v jezikih z imperativnim vzorcem. Deklaracije jezika Z_0 definiramo tako, da je njihov vpliv viden samo v vgnezdenih okoljih. Vgnezdeno ali vsebovano okolje zapišemo z relacijo $\Pi_i \sqsubseteq \Pi_k$. Če je neka povezava identifikatorja z povezljivo vrednostjo izvedena v okolju Π_i, bo ta povezava vidna samo v vgnezdenih okoljih Π_v, torej tistih, za katere velja $\Pi_i \sqsubseteq \Pi_v$. S tem se odraža tudi iskanje simbolov, ki se prične v trenutnem okolju in se, če simbol ni najden, nadaljuje v starševskem. Ko iskalni algoritem pride do korenskega okolja in simbol še vedno ni najden, se javi napaka.
Okolje nastane z deklaracijo bloka, ne glede ali blok predstavlja dejanski blok ali implicitnega npr. v stavku `while`:

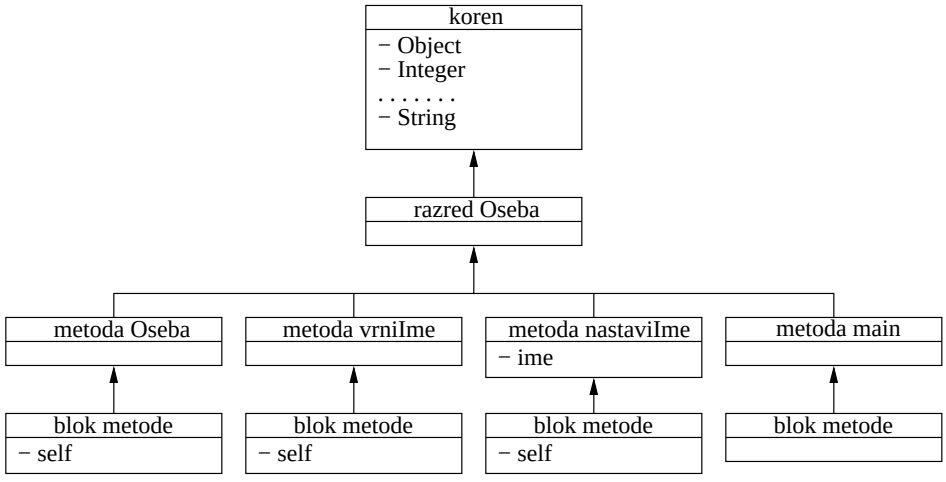
```
...
Integer a = 5;
{
  Integer a = 3;
   while{ a <=3; }
   {
    Integer b = a * 2;
   };
};
```

Deklaracija z istoimenskim identifikatorjem se lahko nahaja samo v različnih okoljih. Ker so bloki in večina stavkov v našem jeziku prvorazredne vrednosti, bo okolje, ki bo obstajalo v času izvajanja, verna kopija tega, ki ga prikazujemo tukaj. Blok je lahko anonimen ali poimenovan. Slednji predstavljajo funkcijske in proceduralne abstrakcije. V splošnem obstajajo tri bločne strukture deklaracij. Jezik Z_0 uporablja vgnezdeno bločno strukturo. Drugi dve sta monolitna in ravna. Monolitna struktura predstavlja celotni program z enotnim blokom. Ravna bločna struktura je sicer izboljšava monolitne, vendar še vedno nezadostna za učinkovito lokalizacijo deklaracij. Bloki jezika Z_0 lahko uporabljajo identifikatorje z istim imenom v različnih kontekstih in deklaracijah.

To pomeni, da lahko identifikatorje prekrivamo.
Katera deklaracija učinkuje nam pove mehanizem povezovanja. Mi bomo za vse deklaracije uporabljali statično povezovanje, ki pomeni, da se kontekst ovrednoti v okolju definicije. Dinamično povezovanje je primerno za dinamično tipizirane in predvsem interpretirane jezike.
Okolje se ustvari na mestu deklaracije:

- razreda,
- metode,
- bloka metode in
- vsakega vgnezdenega bloka

Posebna vrsta okolja je korensko okolje, ki predstavlja globalne deklaracije razredov. V njem se nahajajo razredi vgrajenih tipov, kot sta npr. `Object` in `Integer`, kot tudi razredi eksplicitno vključenih zunanjih razredov. Iskanje identifikatorja v okolju se na ta način prevede v nasledniško iskanje v trenutno aktivnem okolju. Ker se okolja gnezdijo, je smiselna skladovna predstavitev. Okolje v naslednjem podpoglavju definiranega razreda predstavimo kot na sliki 12.



Slika 12: Razčlenjeno okolje razreda `Oseba`.

4.4 Struktura vhodnih programov

S pričujočo sintakso jezika Z_0 lahko zapišemo osnovno ogrodje vsakega programa. Vhod predstavlja definicijo razreda. Ta vsebuje metode, ki so lahko procedure ali funkcije. Razlika med njimi je ta, da imajo funkcije definiran tudi tip vračanja. Zapišimo preprost primer razreda:

```
class Oseba {

    public Oseba {
     ...
    }

    restricted vrniIme : String {
     ...
    }

    restricted nastaviIme(String ime) {
     ...
    }

    public main {
     ...
    }
}
```

Po dogovoru je metoda `main` glavna metoda razreda, ki se, podobno kot v javi, izvrši samodejno. Modifikator dostopa metode ima lahko 4 vrednosti:

- `public`, ki omogoča polni dostop do metode. Polni dostop pomeni invokacijo in spreminjanje metode.
- `restricted`, ki pravtako omogoča invokacijo metode, vendar ne njenega spreminjanja.

- `protected`, ki omejuje dostop zunaj razreda (metode zunaj razreda so nevidne), vendar ohranja vidljivost metod v podrazredih.

- `private`, ki omogoča invokacijo in spreminjanje metod samo znotraj razreda.

Vsaka metoda mora imeti definiran modifikator dostopa. Metode, ki ne prejmejo nikakršnih argumentov, seveda nimajo deklariranih formalnih parametrov. Oklepaje v tem primeru izpustimo in tako pridobimo na čistosti in zgoščenosti zapisa invokacije metode. Funkcije, ki vračajo neko vrednost, imajo tip vračanja naveden za znakom ':'. Glavni del metode predstavlja njen blok, v katerem so navedene deklaracije lokalnih spremenljivk in stavki.

4.5 Deklaracije in lokalne spremenljivke

Z deklaracijo v okolju izdelamo povezavo vrednosti z identifikatorjem. Deklaracije razdelimo na deklaracije primitivnih tipov in deklaracije kompleksnih tipov.
Deklaracija primitivnega tipa je sestavljena iz tipa, spremenljivke in neobvezujočega inicializacijskega izraza. Deklaracijo spremenljivke tipa `Integer` zapišemo na sledeč način:

```
Integer var1, var2 = 2 + 3;
```

Spremenljivka `var1` dobi privzeto vrednost celoštevilčnega tipa, ki je 0, `var2` pa vrednost 5. Izrazi in podizrazi, v katerih nastopajo vrednosti literalov, se ovrednotijo v času prevajanja. Ker so tudi primitivne vrednosti objekti, se na mestu deklaracije ustvari objekt tipa `Integer`. Objekti primitivnih vrednosti se torej ustvarjajo implicitno brez uporabe operatorja `new`.
Objekt kompleksnega tipa se ustvari eksplicitno z operatorjem `new`. Če spremenljivko kompleksnega tipa ne inicializiramo dobi vrednost `nil`, ki pomeni ničelno referenco. Takšna referenca ne kaže nikamor in nad njo ne moremo invocirati metod. Deklaracija spremenljivke zgoraj deklariranega tipa `Oseba` je sledeča:

```
Oseba oseba =  new Oseba;
```

Instanciranje z operatorjem `new` se lahko izvaja tudi anonimno:

```
....
dodajOsebo(  new Oseba );
....
```

Učinek deklaracije je viden samo v okolju, v katerem je bila deklaracija generirana in v vseh podokoljih tega okolja. V istem okolju lahko obstaja samo en identifikator z enakim imenom. Identifikatorje spremenljivk lahko prekrijemo v podokoljih.

4.6 Literali

Literali so najpreprostejši izrazi, saj jih sestavlja samo ena primitivna vrednost številskega ali znakovnega tipa. Glede na primitivne tipe imamo v jeziku naslednje literale:

- cela števila,
- realna števila,
- logični vrednosti `true` in `false`,
- ničelno referenco `nil`,
- znak in
- znakovni niz.

Ker literali predstavljajo konstantne vrednosti primitivnih tipov, so tudi sami objekti. Zato lahko nad njimi izvajamo invokacijo metod:

```
....
''Dober dan.''.print;
....
```

4.7 Osnovni aritmetični izrazi

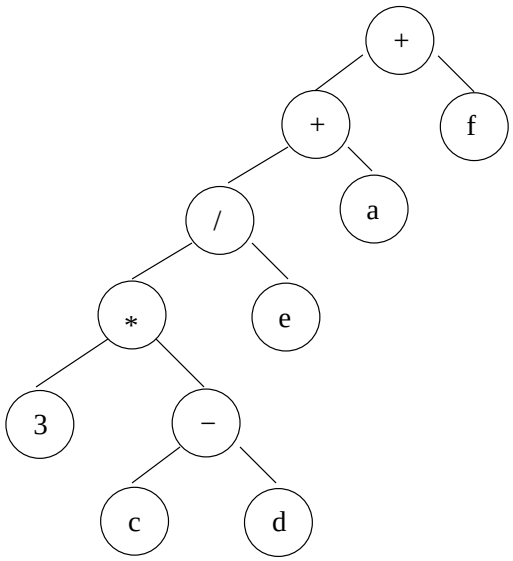
Aritmetične izraze delimo na celoštevilčne in tiste z realnimi števili. S stališča implementacije med njimi ni posebne razlike, saj med obojimi obstajajo implicitne pretvorbe. Aritmetični izrazi omogočajo:

- seštevanje,
- odštevanje,
- množenje,
- deljenje,
- prefiksno in postfiksno seštevanje,
- prefiksno in postfiksno odštevanje in
- vse osnovne operacije nad isto entiteto (+=, -=, ...).

Aritmetični izraz

```
a + 3 * (c - d) / e + f
```

predstavimo s sintaktičnim drevesom na sliki 13.



Slika 13: Drevo izpeljav za izraz `a + 3 * (c - d) / e + f`.

Vsako vozlišče v sintaktičnem drevesu ima svoj tip. Za računske operacije definiramo vozlišča `NODE_PLUS`, `NODE_MINUS`, `NODE_MULTIPLY`, `NODE_DIVIDE`, `NODE_MODULO`, `NODE_POSTINC`, `NODE_POSTDEC`, `NODE_PREINC`, `NODE_PREDEC`.

Edina operacija, ki je definirana na primitivnem tipu `String` je seštevanje (`NODE_PLUS`). Prištevanje znakovnemu nizu je dovoljeno s strani vsake primitivne vrednosti, razen kontrolnih entitet. Tako lahko zapišemo:

```
....
(''Star si '' + getStarost + '' let.'').print;
....
```

Da bi lahko nizu prišteli vrednost poljubnega tipa, mora tip definirati pretvorbni operator, ki vrača tip `String`.

4.8 Logični izrazi

Logični izrazi operirajo nad logičnima vrednostima `true` in `false`. Logično vrednost predstavimo s tipom `Boolean`, katerega zaloga vrednosti je:

$$Boolean = \{true,\ false\}$$

Tip `Boolean` je poseben tip in nima nič skupnega s celoštevilčnimi tipi kot je to primer v jeziku C++. S strogo ločitvijo logičnih vrednosti od številskih omogočimo večjo čistost izrazov in boljše razumevanje programov. `Boolean` je sicer implementiran tako, da je njegova vrednost mapirana na celoštevilčni vrednosti 0 in 1, vendar na jezikovnem nivoju ta predstavitev ni vidna.
Logični operatorji so trije: in (`&&`), ali (`||`) in ne (`!`). S temi tremi lahko sestavljamo poljubne logične izraze, katerih členi so logične vrednosti. Izraz je lahko logičen tudi, če se v njem pojavljajo druge vrednosti, npr. literali in spremenljivke številskih tipov:

```
(a + 3 < b) && ( b > c)
```

Logični izrazi se uporabljajo v pogojnih blokih zank, ki jih bomo predstavili kasneje.

4.9 Invokacija metod

Invokacija metode je dosežena z navedbo identifikatorja metode. V sintaktičnem zapisu ločujemo med metodami, ki prejmejo argumente in tistimi brez argumentov. Invokacijo slednjih zapišemo brez oklepajev:

```
....
izpisiDatum;
....
```

Produkcija, ki obravnava referenciranje lokalnih spremenljivk in invokacije brez argumentov, je v sintaksi jezika definirana pod `FieldReference`. V kolikor je referenca metode nesestavljena, je prejemnik objekt, v katerem se trenutno nahajamo. Invokacija metode, ki prejme seznam argumentov, je določena s produkcijo `MethodReference-WithArgs`. Metodo lahko seveda invociramo tudi nad eksplicitno podanimi referencami objektov:

```
....
Student student(‘‘Tone Baraba’’);
student.vrniPriimek.upcase( true ).print;
....
```

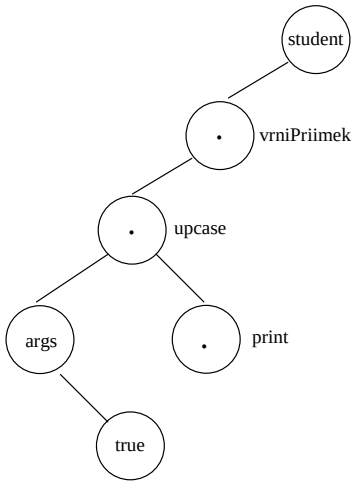
Implicitno podana referenca `self` se lahko izpusti ali navede, učinek je identičen:

```
....
 self.izpisiDatum;
izpisiDatum;
....
```

Vozlišče drevesa, ki predstavlja invokacijo, poimenujemo `NODE_METHODCALL`. Drevo izpeljave za gornji sestavljeni klic je prikazan na sliki 14.

4.10 Bloki

Blok je jezikovni konstrukt, v katerem zapišemo programski kod. Obstajata dva tipa blokov. Blok metode definira celotno metodo, lokalni blok pa se nahaja v bloku metode. Lokalni blok je anonimen, medtem ko je blok metode poimenovan. Bloki so lahko vgnezdeni.
Vsak blok ima svoje deklaracije, ki učinkujejo samo v njegovem dosegu. Ker jezik omogoča čisto objektno abstrakcijo, tudi bloke predstavimo kot objekte. Predstavitev blokov kot prvorazrednih vrednosti pomeni, da jih lahko manipuliramo enako kot npr. spremenljivke ali literale. Blok se lahko izvede na mestu, kjer je deklariran ali kasneje.



Slika 14: Drevo izpeljav za sestavljen klic metode `student.vrniPriimek.upcase( true ).print`.

Zaradi slednjega mora blok vsebovati okolje referenc, s katerimi operira. Blok lahko namreč ubeži svojemu lokalnemu okolju. Lokalne spremenljivke starševskega okolja v času izvedbe bloka ne obstajajo nujno. Metoda, iz katere je blok prišel kot vrednost, se je namreč lahko že končala. Starševske spremenljivke, ki so v bloku referencirane nimajo več svoje lokacije v okvirju. Zaradi optimizacije, bloku dodamo samo spremenljivke, ki jih dejansko referencira in ne celotnih starševskih okolij. Zato bloki, ki lahko ubežijo, ne smejo spreminjati starševskih referenc. Seveda pa lahko spreminjajo objekt. Če želimo spreminjati objekt z operatorjem =, mora razred imeti definiran operator prireditve.
Druga stvar je vračanje. Kaj se zgodi, če blok vrne vrednost, medtem ko se je metoda njegovega okolja že zaključila? Takšno vračanje je s stališča statične varnosti potrebno preprečiti. Bloki, ki lahko ubežijo, ne smejo vračati.
Povsem statično varni so bloki, ki se izvedejo neposredno ob deklaraciji. Takšni bloki ne ubežijo. Sintaksa deklaracije je preprosta:

```
Student student(‘‘Tone Baraba’’);
....
{
  ....
  student.vrniPriimek.print;
```

```
  ....
};
....
```

Ker je blok prvorazredna vrednost, ga lahko shranimo v spremenljivko ali ga podamo kot parameter metodi. Če metoda `izvedi` prejme argument tipa `Block`, lahko zapišemo:

```
izvedi( { student.vrniPriimek.print; } );
```

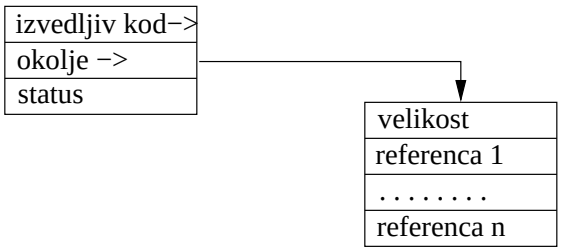
Blok lahko izvedemo zakasnjeno in ne na mestu njegove deklaracije z metodo `exec`:

```
Student student(‘‘Tone Baraba’’);
....
Block blok = {
  ....
  student.vrniPriimek.print;
  ....
};
student.nastaviPriimek(‘‘Nekdo drug’’);
blok.exec;
```

Priimek, ki ga bo blok izpisal ne bo “Tone Baraba” ampak “Nekdo drug”, ker je referenca originalna in ne kopija. Na ta način dosežemo konsistenco med vrednostmi starševskih spremenljivk v blokih, ki se lahko nahajajo kjerkoli v pomnilniškem prostoru. Blok je sestavljen iz izvedljivega koda in lokalnega okolja spremenljivk, ki so v bloku uporabljene. Poleg tega ima blok še status, ki pove na kakšnen način se je izvajanje bloka končalo. Strukturo objekta tipa `Block` prikazuje slika 15. Na tem mestu poudarimo, da v strukturi bloka in vseh nadaljnjih konstruktov prikazujemo samo podatke, ki so relevantni. Vsak objekt vsebuje seveda še informacije, ki so skupne vsem objektom – meta kazalec, razred in deskriptorje.

4.11 Iterativni stavki

Iterativni stavki so tisti, ki omogočajo iteracijo. Med te prištevamo zanke `While`, `DoWhile` in `For`. Vsaka zanka je sestavljena vsaj iz enega pogoja in bloka, ki se ob tem



Slika 15: Struktura bloka vključuje naslov izvedljivega koda, kazalec na okolje in izhodni status. Okolje, ki zajema seznam referenc, je predstavljeno kot samostojen meta objekt.

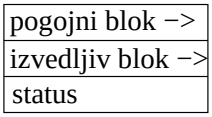
pogoju izvrši. Ker je zanka prvorazredna vrednost, ima identične značilnosti kot blok. Modularna predstavitev iteracijskih in vejitvenih konstruktov ima širši pomen. Čista objekta predstavitev omogoča enolično manipulacijo, modularnost pa boljšo razdrobljenost objektov v času izvajanja. Programsko okolje v času izvajanja je zbirka objektov. Če je vsak konstrukt predstavljen kot zbirka objektov, dobimo nad njim zelo natančno kontrolo, katero lahko s pridom izrabljamo. Zankam lahko na ta način povsem dinamično v času izvajanja manipuliramo bloke – lahko jih zamenjujemo ali spreminjamo. Ker je sistem statično tipiziran, ne more priti do kakršnihkoli napak.

4.11.1 Stavek `While`

Stavek `While` je sestavljen iz logičnega pogoja in bloka izvršitve. Zaporedje izvedbe je takšno, da se najprej izvrši pogoj in šele nato glavni blok. To pomeni, da se telo `While` nujno ne izvrši. Če pogoj ni izpolnjen, se zanka zaključi. Primer zanke `While` je:

```
while{ stevec <= 10; }
{
  stevec.print;
  stevec = stevec + 1;
};
```

Zanka `While` je sestavljena iz dveh ločenih blokov, od katerih mora prvi biti pogojni, drugi pa poljuben izvedljiv blok. Objekt tipa `While` je prikazan na sliki 16.



Slika 16: Struktura `While` vključuje referenci pogojnega in izvršnega bloka ter izhodni status.

4.11.2 Stavek `DoWhile`

Stavek `DoWhile` je zelo podoben `While`, le da je zaporedje izvedbe takšno, da se najprej izvrši glavni blok in šele nato pogoj. Telo se tako izvrši vsaj enkrat. Če pogoj ni izpolnjen, se zanka zaključi. Primer zanke `DoWhile` je:

```
do {
 stevec.print;
 stevec = stevec + 1;
} while{ stevec <= 10; };
```

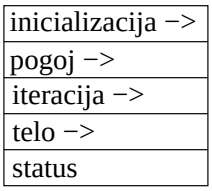
Zanka `DoWhile` je sestavljena iz dveh ločenih blokov, od katerih mora prvi biti telo zanke, drugi pa pogojni blok. Struktura objekta tipa `DoWhile` je enaka `While` zato je posebej ne prikazujemo.

4.11.3 Stavek `For`

Stavek `For` sestoji iz štirih ločenih blokov. Prvi je inicializacijski, drugi je pogojni, tretji se izvrši ob koncu vsake iteracije in četrti je telo zanke. Inicializacijski blok je poljuben blok, ki lahko vsebuje veliko več kot samo inicializacije v zanki uporabljenih spremenljivk. Pogojni blok vsebuje logični pogoj in je enak tistima v zankah `While` in `DoWhile`. Blok, ki se izvede po vsaki iteraciji, je pravtako povsem poljuben blok, ki lahko vsebuje karkoli. Zadnji blok predstavlja telo zanke. Zanko `For` zapišemo:

```
Integer a;
 for {''zacetek''.print; a=2;} {a < 20;} {a = a + 1;}
 {
   a.print;
 };
```

Struktura objekta `For` je prikazana na sliki 17.



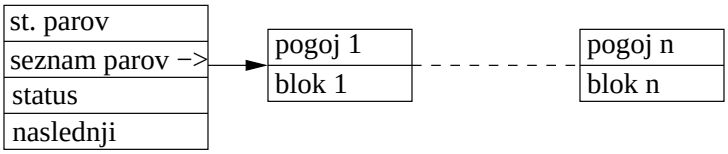
Slika 17: Objekt `For` vsebuje reference na vse štiri bloke in izhodni status zanke.

4.12 Pogojne vejitve in stavek `If`

Ker je jezik imperativne narave, so pogojne vejitve osnova za spreminjanje toka izvajanja. Pogojno vejitev realiziramo s stavkom `If`. Ta ima lahko tri oblike. Prva je preprosta in vključuje samo en pogoj in en blok. Druga oblika ima vejo `else`, tretja pa omogoča nadaljne `If` vejitve. Telo se izvede samo ob izpolnjenem pogoju. Če obstaja alternativna veja, se izvede ne glede na to, ali je to veja `else` ali povsem nova `If` vejitev. Stavek `If` lahko zapišemo na primeru izračuna fakultete:

```
public fakt(Integer s) : Integer
{
   if { s <= 1; }  then
     {  return 1; };
   return s * fakt(s - 1);
}
```

Če želimo celotno vejitev predstaviti kot objekt, je treba v objekt vključiti tudi vse alternativne vejitve, ki se v stavku pojavijo. Stavek `If` ima namreč lahko nedoločeno število vejitev `if .. then .. else if .. then ...` Struktura objekta je zato predstavljena podobno kot pri bloku. Vsako vejitev lahko predstavimo kot par (pogoj, blok). Stavek `If` tako vsebuje seznam takšnih parov. Struktura objekta se nahaja na sliki 18.



Slika 18: Objekt `If` vsebuje število parov, dinamičen seznam parov (pogoj, blok) in dve statusni spremenljivki.

4.13 Spreminjanje metod

V uvodnem poglavju smo že omenili, da instančnih spremenljivk ni. Spremenljivke so podane samo znotraj metod in so zato vezane na posamezno metodo. Ker se metode od instance do instance razlikujejo, lahko govorimo o instančnih metodah. Potemtakem bi lahko tudi spremenljivke označili kot instančne, vendar implicitno podane. Stanje objekta se odraža samo v metodah. Takšen pristop je bolj abstrakten, kot če bi bilo stanje določeno eksplicitno z definicijo instančnih spremenljivk. Zaradi abstraktnejšega pristopa k definiranju objektovega stanja potrebujemo mehanizem, ki omogoča manipulacijo stanja, kot če bi bilo to podano s spremenljivkami. Mehanizem, ki nam to omogoča, je dinamično spreminjanje metod. Vsaki metodi razreda lahko, pod določenimi pogoji, spremenimo funkcionalnost. Ker morajo spremembe biti statično varne, nastanejo omejitve pri signaturi metode. Ta mora ostati nespremenjena. Pravzaprav to sploh ni tako resna omejitev, če pomislimo, da se obnašanje metode naj ne bi bistveno spremenilo. Metodo spremenimo tako, da navedemo njeno ime in novo definicijo. Zgoraj definirano metodo za izračun fakultete omejimo samo za vrednosti med 1 in 100:

```
method fakt(Integer s) =
{
   if { s > 100; }  then
        {  return 0; };

   if { s <= 1; }  then
        {  return 1; };
   return s * fakt(s - 1);
}
```

Sprememba metode je vidna v naslednjem klicu. Poudariti velja, da ima takšna sprememba funkcionalnosti bistveno prednost pred spremembo znotraj same metode. Dodatna funkcionalnost znotraj telesa metode se izvede namreč ob vsakem klicu, medtem ko če metodo spremenimo v celoti, se izvajanju dodatne funkcionalnosti izognemo. Prikažimo to na primeru ovrednotenja vozlišča drevesa, ki predstavlja binarno operacijo. Konstruktor razreda `ParseTree` prejme dve celoštevilični vrednosti in operacijo vozlišča. Metoda `evaluate` ovrednoti vozlišče glede na operacijo. Razred definiramo takole:

```
class ParseTree {

   public ParseTree(Integer a, Integer b, Character op)
   {
      if{ op == '-'; }  then {
           method evaluate = {
               return a - b;
            }
     } else if{ op == '+'; }  then {
           method evaluate = {
               return a + b;
            }
     } else if{ op == '*'; }  then {
           method evaluate = {
               return a * b;
            }
     } else {
           method evaluate = {
               return a / b;
            }
     }
     ;
   }

   public evaluate : Integer
```

```
    {
       return 0;
    }
 }
```

Metodo `evaluate` smo nastavili samo enkrat v konstruktorju, kjer smo opravili vse potrebne primerjave. Ne glede na to, kolikokrat jo pokličemo, se bo vedno izvedla na začetku nastavljena metoda brez kakršnihkoli primerjav znotraj metode. Na ta način smo bistveno pohitrili izvedbo metode `evaluate`.
Novo telo metode mora ustrezati podani signaturi. To pomeni, da v primeru procedure v telesu ne sme biti stavka `return` z vrednostjo. Namesto da navedemo novo definicijo, lahko metodo nastavimo na neko že definirano metodo kot na primer:

```
method fakt(Integer s) = drugObjekt.fakt(Integer s);
```

Na spreminjanje metod lahko gledamo kot na omejeno različico dinamičnega dedovanja. Dinamično dedovanje omogoča, da v času izvajanja spremenimo nadrazred. Sprememba nadrazreda efektivno pomeni spremembo metod definiranih v nadrazredu. V našem primeru lahko metodo nadrazreda pravtako spremenimo, le da mora nova metoda zadoščati stari signaturi. Na ta način dobimo sicer omejen mehanizem dinamičnega dedovanja, vendar je statično varen.

4.13.1 Spreminjanje metod tujih objektov

Spreminjanje metode nad lastnim objektom ni pretirano zahtevno za implementacijo. Težave nastopijo pri spreminjanju metode, ki je definirana nad nekim drugim objektom:

```
class Drugi {
   ......
    public Drugi(ParseTree tree)
    {
       method tree.evaluate = {
              error;
              return -1;
            };
```

```
    }
    public error
    {
       ‘‘Ni implementirano!’’.print;
    }
  ......
}
```

Ko v objektu `tree` pokličemo metodo `evaluate`, pokličemo tudi metodo `error`. Problem nastane, ker metodi pripadata različnim objektom. Klic `tree.evaluate` zahteva referenco sprejemnika na skladu. Toda katera referenca bo na skladu, `tree` ali objekt tipa `Drugi`? Prvo referenco potrebujemo, da lahko metodo sploh najdemo, saj je njen naslov zapisan v tabeli metod, do katere pridemo preko vrednosti v virtualni tabeli. Vendar v telesu metode invociramo tudi `error`, ki je definirana nad drugim objektom. Metodi imata v splošnem povsem različne indekse. Rešitev je, da se spremenjena metoda `evaluate` ovrednoti v kontekstu prejemnika tipa `Drugi` in ne `ParseTree`. To je razlog, zakaj potrebujemo še dodatno, dejansko referenco prejemnika. Instrukcija invokacije bo naslov metode poiskala preko originalne reference, na sklad pa bo potisnila referenco dejanskega prejemnika.

4.13.2 Spreminjanje metod z ohranitvijo starševskih okolij

Metodo lahko spremenimo tako, da njeno novo telo vsebuje reference starševskih okolij. Preprost primer tega je sledeča definicija razreda `Stevec`:

```
class Stevec {
    public Stevec { }
    public getValue : Integer
    {
        return 0;
    }
    public increment
    {
       Integer v = getValue;
```

```
        method getValue = {  return v + 1; };
    }
}
```

Razred `Stevec` ima metodi za vračanje trenutne vrednosti števca in povečanje vrednosti za 1. V metodi `increment` spremenimo metodo `getValue` tako, da vrača za 1 povečano staro vrednost. Vendar novo telo metode referencira spremenljivko `v` iz starševsega okolja. Ko se metoda `increment` zaključi, se njen okvir sprosti in lokalnih spremenljivk več ni. Na novo definirano telo metode mora s tega razloga ohraniti originalno referenco spremenljivke `v`. Referenca ostane shranjena v lokalnem okolju metode `getValue`. V okolje se shranijo samo tiste reference, ki so potrebne za ovrednotenje.

4.14 Vključevanje definicij zunanjih razredov

Večkrat se pojavi potreba po nekem zunanjem razredu, katerega funkcionalnost želimo izkoristiti znotraj našega razreda. Tak razred je potrebno vključiti v seznam zunanjih razredov s čimer postane definicija tega razreda vidna. To je pomembno, saj za dejansko uporabo razreda potrebujemo vsaj nabor metod in njegovih nadrazredov. Ko je zunanji razred vključen, ga lahko uporabljamo v deklaracijah in izrazih. Za vključitev razreda smo uvedli rezervirano besedo `use`, kateri sledi seznam razredov. Primer vključitve dveh zunanjih razredov je sledeč:

```
use Oseba, Student;
```

Razredov primitivnih tipov ne vključujemo, saj so njihove definicije prisotne implicitno. Tudi dveh razredov z enakim imenom ni mogoče vključiti. To bi namreč pomenilo semantično dvoumnost programov.
Mehanizem vključevanja je implementiran tako, da je možno vključevanja tudi gnezditi. Če se pri tem pojavijo rekurzivne vključitve, jih je potrebno razrešiti.

4.15 Metode višjega reda

V jeziku Z_0 metod višjega reda še nismo implementirali. Ker ima uporaba takšnih metod praktični pomen, naš jezik pa že omogoča osnovo za njihovo implementacijo, jih bomo prikazali samo okvirno.

Metode višjega reda so metode, ki kot argumente prejmejo druge metode ali vračajo metode. Takšne definicije metod imajo za posledico višjo abstraktnost in včasih tudi razumljivost programov. Metode višjega reda imenujemo tudi "curry" metode. Običajno se v literaturi uporablja izraz "curry" funkcije in ne metode. V jeziku Z_0 smo pač funkcije poenotili v metode. Ideja metod višjega reda temelji na dinamičnem generiranju koda v zaprtjih. Osnova za implementacijo metod kot prvorazrednih vrednosti so zaprtja. Zaprtje v primeru metode vsebuje kod metode in njeno lokalno okolje referenc. Metodo lahko v takšni obliki prenašamo kot prvorazredno vrednost, tj. podamo jo lahko kot argument, shranimo v lokalno spremenljivko ali jo vrnemo kot rezultat funkcije. Metodo višjega reda lahko apliciramo delno ali popolno. Aplikacija metode, ki vrne drugo metodo, je popolna, če apliciramo tudi vrnjeno metodo. Sicer je delna. V nekaterih jezikih obstajajo mehanizmi, ki posredno omogočajo funkcije višjega reda. Aplikacija takšnih funkcij žal ni povsem transparentna. Primer je jezik $C^{\#}$, ki v ta namen uporablja delegatorje. Delegator je v bistvu le varen kazalec na funkcijo, vendar ima precejšnje omejitve. V našem jeziku želimo implementirati metode kot prave prvorazredne vrednosti.

Za predstavitev metod potrebujemo metodni tip. Ta je lahko funkcijski ali proceduralni. Metodni tip ima obliko $(TA_1, \ldots, TA_n) \rightarrow (TR)$, ki pomeni, da funkcija slika iz tipov parametrov v tip rezultata. Metodni tip za metodo `potenca` bi zapisali:

```
(Integer, Double) -> (Double)
```

Spremenljivki metodnega tipa lahko priredimo metodo podobno kot pri spreminjanju metod:

```
(Integer, Double) -> (Double) var;
 method var(Integer n, Double v) =
 {
    // kod metode
 };
```

Metoda, ki vrača drugo metodo, vrača torej metodni tip. Poglejmo to na primeru potence:

```
 class Potenca {
   ....
```

```
    public kvadrat : (Double) -> (Double)
    {
         return  method(Double v) {
             return v * v;
         };
    }
}
```

Metoda `kvadrat` vrne metodo, ki prejme realno število in vrne njegov kvadrat. Metodo apliciramo povsem transparentno:

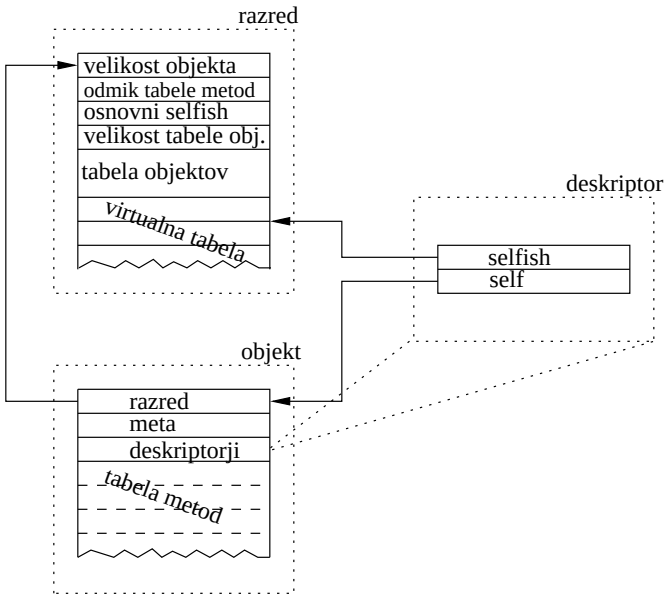
```
kvadrat(4.0).print;
```

Časovna zahtevnost metod višjega reda je enaka naši dinamično spremenjeni metodi z okoljem.

4.16 Dedovanje

4.16.1 Predstavitev objekta

Preden se lotimo dedovanja s konceptualnega vidika, moramo povedati besedo o predstavitvi objektov. Predstavitev objekta je pomembna s stališča učinkovitosti in možnosti manipulacije objektovega stanja. Strukturo objekta prikazujemo na sliki 19.
Vsak objekt vsebuje naslov razreda. Na tem naslovu se nahajajo razredni podatki, ki se uporabljajo pri instanciranju razreda in manipulaciji deskriptorja. Naslovu razreda sledi naslov meta podatkov. Ta je zaenkrat še neuporabljen, saj meta arhitektura v tem trenutku še ni implementirana. Nato sledijo deskriptorji objekta in tabela metod objekta. Ker so metode vezane na instanco razreda, je tabela metod shranjena v objektu. Posamezen vstop v tabelo metod vsebuje pomnilniški naslov metode, kazalec `self` in kazalec okolja. Tabela metod vsebuje vstope za vsako metodo v razredu.
Deskriptor ali opisnik objekta je dvojček, ki vsebuje kazalec na objekt (`self`) in kazalec na virtualno tabelo trenutnega objekta (`selfish`). Vzemimo za primer število tipa `Integer`, ki ga pretvorimo v `Object`. Oba kazalca, `self` in `selfish`, ostaneta nespremenjena in kažeta na začetek virtualne tabele za `Object` znotraj tabele za `Integer`.



Slika 19: Struktura objekta.

Če bi `Integer` dedoval iz več kot enega razreda, bi se kazalec `selfish` spremenil, ker pri večkratnem dedovanju v splošnem ni mogoče zagotoviti istih indeksov redefiniranih metod. V tem primeru bi se za nov pogled na objekt spremenil deskriptor. Deskriptor objekta je torej tisti, ki se s pretvorbami tipov pri večkratnem dedovanju spreminja. To pomeni, da lahko ima isti objekt več različnih deskriptorjev. Naslov deskriptorja objekta se nahaja v lokalni spremenljivki, kadar objekt referenciramo.

4.16.2 Izpeljava iz nadrazreda

Nadrazred v definiciji razreda navedemo z rezervirano besedo `inherits`. Če razred `Student` deduje iz razreda `Oseba`, to zapišemo:

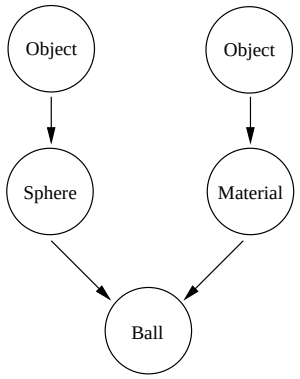
```
class Student  inherits Oseba {
    public Student { ... }
    ....
}
```

Vsak razred implicitno deduje razred `Object`. Object lahko torej navedemo ali ne. Če dedujemo iz nekega uporabniškega razreda, se `Object` ne vključi v seznam neposrednih

predhodnikov. V tem primeru se `Object` vključi kot nadrazred najvišjega uporabniškega razreda. Sintaksa večkratnega dedovanja je preprosta:

```
class Ball  inherits Sphere, Material {
   public Ball { ... }
   ....
}
```

Na tem mestu naj poudarimo preprostost modela dedovanja. Dedovanje je enoumno in nazorno. Ni zapletenih pravil in modifikatorjev, kot v jeziku C++. Dedovanje je vedno "public" in "nevirtualno". Imamo torej dedovanje z replikacijo. V gornjem primeru v objektu `Ball` obstajata dva *objekta* (slika 20). Eden kot osnovni razred `Sphere` in eden iz razreda `Material`. Nasprotje replikaciji je deljeno dedovanje, kjer bi en `Object` nastavili kot nadrazred obeh omenjenih.



Slika 20: Hierarhična zgradba objekta `Ball`.

Obe strategiji imata svoje prednosti. Mi smo se odločili za eno zaradi preprostosti semantičnega modela. Jezik C++ omogoča t.i. "virtualno dedovanje", ki je dejanska različica deljenja razredov. Za deljeno dedovanje se nismo odločili tudi zaradi implementacije. Le-ta je v tem primeru nekoliko zahtevnejša, saj je v času preverjanja potrebno izvajati analizo hierarhije, ki ni vedno preprosta. Poleg tega "razdelitev" razreda med dva ali več podrazredov ni vedno povsem logično in v skladu s pričakovanji. Dedovanje se odraža tako, da je v podrazredu prisotna tudi funkcionalnost nadrazreda. Gledano s semantičnega vidika to pomeni, da imamo v podrazredu na voljo vse metode

nadrazreda, ki so bile deklarirane `public`, `restricted` ali `protected`. Metode deklarirane z modifikatorjem `private` so na voljo samo v pripadajočem razredu in ne tudi v podrazredih. V podrazredu je metode možno dodati ali že obstoječe redefinirati. Ta postopek si bomo pogledali bolj podrobno v sledečem podpoglavju.

4.16.3 Pravila za redefiniranje metod v podrazredu

Redefiniranje metode je bržkone najpomembnejša lastnost mehanizma dedovanja. Na redefiniranje gledamo kot na mehanizem dedovanja, ki omogoča koncept specializacije. Specializacija podaja podrobnejšo in bolj dodelano funkcionalnost, kot je bila na voljo v starševskem razredu. Specializacija kot temeljni koncept dedovanja se najbolj izraža v metodah. Ker v našem jeziku nimamo instančnih spremenljivk, je to še toliko bolj res. Specializacijo izvajamo nad metodami razreda. Če imamo v nadrazredu neko nedodelano metodo, jo v podrazredu specializiramo na način, da bo imela dodatno funkcionalnost in bo hkrati kompatibilna s tisto v nadrazredu. Na prvo zahtevo ne moremo vplivati, saj bi za to potrebovali mehanizme za vrednotenje obnašanja. Naš cilj je zagotoviti kompatibilnost signature metod, saj se lahko le tako zanesemo na statično varnost. Omenili smo že, da večina modernih jezikov ne omogoča kakšnih posebnih sprememb v signaturi redefiniranih metod. Takšno politiko je sicer preprosto implementirati, vendar je neučinkovita in ima velike omejitve. V našem jeziku bomo omogočili spremembe v takšnem obsegu, da bodo te statično preverljive in uporabne za praktične primere. Ker težimo k preprostosti dedovanja, bomo pravila konstruirali karseda preprosto in nazorno. Metoda se tretira kot redefinirana, če:

1. ima enako ime in
2. ima enako število parametrov.

Ti pravili sta osnova za naš model redefiniranja metod. Če katerikoli izmed pogojev ne velja, metoda ne velja za redefinirano. Če je metode redefinirana, potem morata veljati še naslednji dve pravili:

1. tip vračanja je enak ali pa se spreminja kovariantno in
2. tipi formalnih parametrov so enaki ali pa se spreminjajo kontravariantno.

S tem smo omogočili najvišjo stopnjo sprememb, ki je še statično varna. Poseben primer je tip `Self`, ki je invarianten. To pomeni, da mora ostati enak. Invarianca tipa `Self` je nenavadna, saj se ta tip spreminja implicitno in ni nikoli enak. Invarianca na sintaktični ravni torej. V primeru, ko katerikoli izmed pogojev redefinirane metode ne velja, se javi napaka.
Do dvoumnosti lahko pride že pri enkratnem dedovanju. Razred lahko zaradi kakršnihkoli razlogov definira več metod z istim imenom in enakim številom parametrov. Takšna metoda je polimorfna. Na mestu invokacije prevajalnik na podlagi tipov argumentov izbere najprimernejšo različico metode. V kolikor ustrezna metoda ni najdena, se pokliče naslednja najbolj ustrezna metoda. Če je teh več, se javi napaka. Pri redefiniranju metode v podrazredu je potrebna pazljivost, saj metode vedno ni mogoče redefinirati. Poglejmo si sledeč primer. Razred `A` definira polimorfno metodo `dodaj`:

```
class A {
   ....
   public dodaj(Integer a) { ... }
   public dodaj(String a) { ... }
   ....
}
```

Izpeljimo sedaj razred `B` iz tega razreda. Če želimo redefinirati npr. metodo `dodaj(Integer a)`, ne moremo preprosto napisati metode z isto signaturo, ker bi s tem povzročili nepravilno varianco v metodi `dodaj(String a)`. Tip `Integer` je kontravariantno kompatibilen z `Integer`, vendar ne s `String`. Metodo moramo redefinirati na tak način, da bo njena signatura kompatibilna z vsemi obstoječimi metodami nadrazredov. Redefinicijo zapišemo takole:

```
class B  inherits A {
   ....
   public dodaj(Object a) { ... }
   ....
}
```

S tem smo zadostili vsem pravilom redefiniranja. Z eno metodo smo izvedli redefinicijo dveh metod starševskega razreda. To pomeni, da se bosta indeksa obeh metod povezala

na indeks spodaj redefinirane metode. Na prvi pogled se to zdi omejujoče. Toda z globljim razmislekom uvidimo, da s takšnimi pravili ne izgubimo na izrazni moči jezika. Java ima bolj kompleksna pravila za redefinicijo, vendar ne omogoča variance tipov. Mi smo pravila sestavili zelo minimalistično in jih umestili v varianten sistem tipov. S tem pridobimo neprimerno več, kot če bi vse signature fiksirali skozi hierarhijo. Identifikator dostopa se lahko širi. To pomeni, da lahko ima redefinirana metoda enak ali šibkejši dostop.
V primeru večkratnega dedovanja se lahko pravtako pojavijo dvoumnosti. Dva ali več nadrazredov lahko definirajo isto metodo na nezdružljiv način. Takšne metode v podrazredu ne moremo redefinirati, saj bi se v nasprotnem primeru definicije metode izključevale. Prikažimo to na primeru:

```
class A {
   ....
    public dodaj(Integer a) { ... }
   ....
}
class B {
   ....
    public dodaj(String a) { ... }
   ....
}
class C  inherits A, B {
   ....
    public dodaj(Integer a) { ... }  // napaka
   ....
}
```

Razred `C` je sicer kompatibilen z `A`, vendar ne tudi z `B`. Rešitev je metoda, ki bo kompatibilna z obema razredoma. Podobno kot v zgornjem primeru enega razreda.

Pri tipih vračanja velja kovarianca. Pravila za kovariantne spremembe tipov vračanja so enaka kot pri parametrih. Tip vračanja metode mora biti takšen, da ustreza vsem že definiranim metodam v nadrazredih. Tipičen primer, kjer potrebujemo kovariantne

spremembe je metoda, ki ustvari kopijo objekta in jo vrne:

```
class A {
   ....
    public kopiraj : A { ... }
   ....
}
```

Podrazred definiramo na naslednji način:

```
class B  inherits A {
   ....
    public kopiraj : B { ... }
   ....
}
```

Takšna redefinicija metode je statično povsem varna in zelo dobrodošla. Java in podobni jeziki imajo težavo, saj je pri vsaki tovrstni invokaciji potrebna eksplicitna pretvorba tipa, ki je časovno zahtevna. Kasneje bomo spoznali še boljši način kovariantnega tipiziranja.

4.16.4 Iskanje metode

Implementirali smo asimetrično nasledniško dedovanje. To pomeni, da metodo pričnemo iskati v razredu, nad katerim metodo invociramo. Iskanje metode je realizirano rekurzivno. Če ustrezne metode v razredu ne najdemo, se premaknemo v prvi nadrazred in nadaljujemo z iskanjem. Premikanje navzgor poteka, dokler ne pridemo do absolutnega nadrazreda `Object`. V kolikor metoda ni najdena, prevajalnik javi napako. Ker lahko razred deduje iz več nadrazredov hkrati, je treba preiskati vse nadrazrede. Iskanje v nadrazredih se izvaja od leve proti desni. Iskanje po vseh poteh je potrebno tudi, ko je metoda že najdena, saj je lahko metoda dosegljiva po različnih poteh. V primeru, da najdemo dve ali več ustreznih metod, se javi napaka. Klic takšne metode je potrebno natančneje specificirati. To storimo z eksplicitno kvalifikacijo imena, pri katerem poleg metode navedemo tudi ime razreda. Primer konflikta prikažimo na naslednjem primeru:

```
class A {
   ....
    public izpis(String s) { ... }
   ....
}
class B {
   ....
    public izpis(String s) { ... }
   ....
}
class C  inherits A, B {
   ....
    public test
     {
        izpis(‘‘Dober dan!’’);  // napaka
     }
   ....
}
```

V metodi `test` kličemo `izpis`. Težava nastane, ker je metoda `izpis` definirana z identično signaturo v dveh nadrazredih. Iskalni algoritem ne more določiti, katero metodo bi izbral. Zato se javi napaka pri prevajanju. Metodo bi morali specificirati tako, da bi navedli, iz katerega razreda jo želimo poklicati. Če želimo klic metode iz razreda `B`, potem pokličemo:

```
....
public test
  {
     B.izpis(‘‘Dober dan!’’);
  }
....
```

Ko metodo specificiramo s polnim imenom, razrešimo imenski konflikt.
Iskalni algoritem izbere prvo metodo, ki ustreza pogojem. Tukaj se izkaže preprostost

pravil za redefiniranje metod. Prva metoda, ki zadošča signaturi, je zagotovo prava, saj drugače ne bi mogla biti definirana.

4.16.5 Dinamično tipiziranje s `Self`

Izrazna moč statično tipiziranih programskih jezikov je v veliki meri omejena z njihovim sistemom tipov. Večina tovrstnih jezikov ima invariantne sisteme tipov. Tipi instančnih spremenljivk in signature metod se po hierarhiji ne smejo spreminjati. Kovariantne spremembe v tipih vračanja so sicer dobrodošle, vendar vedno ne zadoščajo potrebam. V praksi so kovariantne spremembe velikokrat potrebne tudi v tipih argumentov in instančnih spremenljivk. Eiffel takšne spremembe omogoča, vendar niso statično varne. Z instančnimi spremenljivkami se ne bomo posebej ukvarjali, saj jih jezik Z_0 ne omogoča. Posvetili se bomo metodam.
Implicitna meta spremenljivka `self` ima identično vlogo kot `this` v javi ali C++. Referenca `self` je na voljo v telesu vsake nestatične metode. Kaže na objekt, nad katerim se metoda izvaja. S stališča tipiziranja nas seveda zanima tip te reference. Spremenljivka `self` ima enak tip kot objekti, ki jih ustvarimo iz razreda, v katerem se na `self` sklicujemo. Razumljivo je, da ta tip ni enak, temveč se spreminja po hierarhiji. Definirajmo dva razreda:

```
class A {
   public vrni : A {
    return self;
   }
}

class B inherits A {
   public vrni : B {
    return self;
   }
}
```

V razredu `A` lahko vrnemo `self`, saj metoda vrača `A`. Podobno velja v razredu `B`, kjer je tip spremenljivke `self` enak `B`. Poraja se vprašanje, kako bi ta tip poenotili. V ta namen bomo vpeljali nov tip, ki bo kontekstno odvisen. Ker se naša referenca že

imenuje `self`, poimenujmo ta tip `Self`. Če bi stvari povzeli po Javi, bi tip bržkone poimenovali `This`. Ker ima `self` v vsakem razredu drug tip, je tip `Self` dinamičen. Ne glede na to, ga lahko preverimo v času prevajanja in je zato statično varen. Zdaj postane toliko bolj nerazumljivo dejstvo, da večina statično tipiziranih modernih jezikov ne podpira takšnega tipa, čeprav ima veliko praktično vrednost. Praktično vrednost tipa `Self` bomo demonstrirali na dveh primerih. Prvi primer je metoda `clone`, ki iz objekta nad katerim je invocirana, ustvari kopijo in jo vrne. Tip vračanja te metode je očitno enak razredu, v katerem je metoda definirana. Metoda `clone` je deklarirana v razredu `Object` in vrača `Object` na sledeč način:

```
class Object {
   ...
    public clone : Object {
     return  self;
    }
}
```

Če metode ne redeklariramo v podrazredu, ima še vedno isti tip vračanja. Razred `Sub` deduje iz `Object`:

```
class Sub {
   ...
    public izpis {
      ...
    }
    public main {
     Sub sub =  new Sub;
     sub.clone.izpis;  // napaka
    }
}
```

Metode `izpis` ne moremo klicati nad objektom, ki ga vrne `clone`, čeprav je ta objekt dejansko naš objekt `Sub`. Problem je v statičnem sistemu tipov. Razred `Object` nima metode `izpis`. Da bi lahko klicali to metodo, bi morali vstaviti eksplicitno pretvorbo tipa:

```
(sub.clone as Sub).izpis;
```

Drug način je, da bi metodo `clone` redefinirali v podrazredu `Sub`, ki bi vračala `Sub` in preprosto samo poklicala `clone` iz objekta. Toda to ni zaželjeno, saj bi morali metodo redefinirati samo zaradi spremembe tipa. Boljši pristop je, če metoda vrača `Self`:

```
class Object {
   ...
    public clone :  Self {
     return  self;
    }
}
```

V razredu `Sub` metoda `clone` zdaj vrača tip `Sub`:

```
class Sub {
   ...
    public izpis {
      ...
    }
    public main {
     Sub sub =  new Sub;
     sub.clone.izpis;  // brez napake
    }
}
```

Metodo `izpis` lahko zdaj kličemo brez dinamične pretvorbe tipa. Tip vračanja metode `clone` je kontekstno odvisen, saj je enak tipu objekta, v katerem se izvede invokacija. Tip `Self` omogoča še bolj dinamično tipiziranje kot kovariantne spremembe v tipih vračanja. Zelo dobra lastnost tega tipa je, da je statično varen in se ga da preveriti v času prevajanja.

4.17 Pretvorbe med tipi

Najpreprostejše pretvorbe so tiste med objekti primitivnih tipov. Med te štejemo `Integer, Short, Byte, Character, Double, Float, Boolean` in `String`. Trivialne

pretvorbe so tiste, ki slikajo iz šibkejšega v močnejši tip. Primer takšne pretvorbe je *Short* → *Integer* ali *Float* → *Double*. Pri netrivialnih pretvorbah kot je npr. *Integer* → *Byte* lahko pride do izgube natančnosti. Zaradi prikladnosti in široke uporabe, ki se je izkazala pri javi, je vsak primitiven tip mogoče pretvoriti v tip `String`. Pretvorbe med primitivnimi tipi so zelo hitre, saj se izvajajo neposredno preko vgrajenih virtualnih instrukcij.

Pretvorbe med objekti kompleksnih tipov potekajo drugače. Prva stvar, ki jih loči od pretvorb primitivnih tipov, je zahtevnost. Pretvorbe kompleksnih tipov sicer potekajo pravtako preko instrukcij, vendar je postopek s stališča hitrosti precej bolj neugoden. Pretvorba tipa objekta je zahtevna, ker je najprej potrebno preveriti, ali je ta objekt dejansko tega tipa. Pri pretvarjanju navzgor (upcasting) tj. od bolj specifičnega k bolj splošnemu tipu ni kakšnih težav, ker lahko pretvorbo preverimo že v času prevajanja. Če vemo, da je razred `Student` izpeljan iz razreda `Oseba`, lahko pretvorbo izvršimo brez težav, saj če je objekt tipa `Student` je zagotovo tudi tipa `Oseba`. Pretvorba objekta tipa `Oseba` v objekt tipa `Student` pa ni enostavna, ker `Oseba` ni nujno tudi `Student`. Tukaj je potrebno dinamično preverjanje, ki se izvrši znotraj pretvorbene instrukcije. Vsak razred vsebuje tabelo vseh objektov, ki jih vsebuje. Tabela sestoji iz naslova razreda in odmikov v virtualno in tabelo metod. Tabela objektov vsebuje torej razred konkretnega objekta in vse njegove nadrazrede. Pretvorba tipa je implementirana tako, da se pregleda ta tabela. Če v njej najdemo podatke za razred, v katerega želimo pretvoriti, potem pretvorbo izvršimo. Če takega razreda ni, je pretvorba napačna. Časovna zahtevnost je v pregledovanju tabele.

V jeziku lahko tudi pretvorbo kompleksnega tipa izvedemo implicitno ali eksplicitno. Če pretvarjamo v bolj splošni tip, je implicitna pretvorba vedno dovolj. Primer takšne pretvorbe je:

```
Student student(‘‘Evgen Hugo’’);
Oseba oseba = student;
```

Pretvorbo v originalni tip izvedemo eksplicitno:

```
Student student = oseba  as Student;
```

ali implicitno:

```
Student student = oseba;
```

Eksplicitna pretvorba uporablja rezervirano besedo `as`, ki pove, na kakšen način gledamo na objekt. Menimo, da je takšna sintaksa bolj berljiva od javanske, saj bolj nazorno pove, kaj želimo doseči. Zgornji primer pove, da prirejamo osebo, na katero gledamo kot na študenta.

Ne glede ali gre za pretvorbo navzgor ali navzdol po hierarhiji, je pretvorba vedno izvedena tako, da se ponastavi kazalec `selfish` v deskriptorju objekta, ki vedno kaže na virtualno tabelo za trenutni objekt. Pretvorba tipa objekta ne modificira. To je logično, saj gre ves čas za isti objekt. Spreminja se samo pogled na objekt. Lahko bi tudi rekli, da se spremeni zunanji vmesnik objekta, saj pretvorba pomeni neko omejevanje ali razširjanje vidne funkcionalnosti. V gornjem primeru gledamo objekt enkrat kot osebo in drugič kot študenta, čeprav gre v obeh primerih za konkreten objekt študenta. Da bi na isti objekt istočasno gledali z različnih vidikov, potrebujemo različne deskriptorje.

4.18 Prevajanje

Jezik Z_0 je bil že v osnovi načrtovan za prevajanje v abstraktni zložni kod. V ta namen smo implementirali prevajalnik za virtualno arhitekturo Z. Prevajalnik operira nad sintaktičnim drevesom in okoljem. Odločili smo se, da ima vsaka metoda v razredu svoje sintaktično drevo. Ovrednotenje razreda je potemtakem sestavljeno iz ovrednotenja vseh metod razreda. Za pravilno ovrednotenje potrebujemo poleg drevesa tudi okolje. Strukturo okolja smo opisali v uvodnih poglavjih, zato je tukaj ne navajamo ponovno. Osnovna naloga prevajalnika je, da iz sintaktičnih dreves metod generira zložni kod. Ker simbolične oblike zložnega koda ne potrebujemo, naš prevajalnik generira neposredno objektni kod, ki je primeren za izvedbo na virtualnem stroju.

Sintaktično drevo je sestavljeno iz vozlišč posemeznega tipa. Ločimo med aktivnimi in pasivnimi vozlišči. Aktivna so tista, ki prožijo generiranje ene ali več instrukcij, pasivna pa tista, ki služijo zgolj kot dopolnilo aktivnim. Pasivno vozlišče je npr. vozlišče za predstavitev argumetov metode, ker bodo šele vozlišča pod njim generirala instrukcije za posamezen argument. Stranski učinek ovrednotenja aktivnega vozlišča je ena ali več virtualnih instrukcij. Vrednotenje drevesa se prične v korenu in se rekurzivno nadaljuje do listov. Virtualne instrukcije se shranjujejo v objektni kod okolja metode.

Ker ima vsak blok svoje okolje, se tudi zaprtja prevajajo v svojem okolju. Objektni kod za zaprtje se generira neodvisno od starševskega okolja. To pomeni neodvisno od metode, v kateri se nahaja. Objektni kodi zaprtij se shranijo za objektnim kodom metode. Tak način prevajanja je potreben, če želimo učinkovito implementirati zaprtja kot objekte. V času izvajanja bomo naslove zaprtij potrebovali pri njihovem instanciranju. Če poenostavimo, se sleheren blok prevede neodvisno, torej sam zase. Objektni kod metode se v razredno datoteko zapiše rekurzivno.
Kljub temu, da zbirnika nismo posebej načrtovali, lahko v prevajalniku generiramo tudi simbolično obliko zložnega koda. Ta zbirniški zapis nam služi pri preverjanju pravilnosti in iskanju napak. Omenjamo ga zato, ker bomo na njegovi osnovi prikazali konkretne primere prevedenega koda.

4.18.1 Simbolična oblika prevedenega objektnega koda

Lokalne spremenljivke referenciramo z imeni `lN`, kjer je `N` lokacija v okvirju. Spremenljivka `l1` pomeni prvo lokalno spremenljivko. Poudarimo, da naslovi v času prevajanja niso določeni. V najboljšem primeru obstajajo samo njihove relativne različice.
Najbolj nazoren opis simboličnega koda bomo podali kar s primerom izračuna fibonaccijevega zaporedja. Zapišimo razred:

```
class Fibonacci {
   public Fibonacci { }
   public clen(Integer n) : Integer
   {
       if{ n <=2; }  then
            {
               return 1;
            };
       return clen(n-2) + clen(n-1);
   }
   public main
   {
      Fibonacci fib =  new Fibonacci;
      ( ‘‘vrednost je ‘‘ + fib.clen(6) ).print
```

```
    }
}
```

Metode razreda se prevedejo v naslednji kod, kateremu smo zaradi lažjega razumevanja dodali še komentarje:

public Fibonacci:

```
Core.new_frame 3, 3       // nov okvir
Core.return_1             // izhod iz metode
```

public clen (Integer) : Integer:

```
Core.new_frame 8, 5       // nov okvir
Core.push l4              // trenutni self
Core.push l6              // spremenljivka n
l8.new Block ??, 2        // nov blok
Core.push l4              // trenutni self
l8.new Block ??, 1        // nov blok
l8.new If l6, l7          // nov if
l8.exec                   // izvrsi if
Core.skip_ifclear 0, ??   // test vrnitve
Core.return_2 l1, 1       // vrni
l6.new Integer  0         // ustvari 2
l7.new Integer l5
l7.sub l6                 // n - 2
Core.push l7              // argument
l5.invoke_1_f #3, l6       // fib(n-2)
l7.new Integer  4         // ustvari 1
l8.new Integer l5
l8.sub l7                 // n - 1
Core.push l8              // argument
l5.invoke_1_f #3, l7       // fib(n-1)
l6.add l7                 // +
Core.return_2 l6, 1       // vrni
```

public clen (Integer)_block_1:

```
Core.new_frame 7, 4       // nov okvir
l6.new Integer  0         // ustvari 2
l7.new Boolean            // ustvari Boolean
l5.iflseq , l6, l7        // primerjaj in nastavi
l2.status l7, 1, 2        // nastavi status bloka
Block.exit_1              // izhod
```

public clen (Integer)_block_2:

```
Core.new_frame 5, 3       // nov okvir
l5.new Integer  4         // ustvari 1
Block.return l5           // vrni
```

public main:

```
Core.new_frame 4          // nov okvir
l1.new_object 26 #2       // nova instanca
l2.new String  8          // ustvari 'vrednost je '
l3.new Integer  22        // ustvari 6
Core.push l3              // argument
l2.invoke_1_f #3, l3      // fib.clen(6)
l4.new String l3          // Integer -> String
l2.add l4                 // +
l2.print                  // izpis
Core.free_frame 4         // sprosti okvir
Core.terminate            // konec izvajanja
```

Iz zbirnega koda lahko razločimo, da se zaprtja dejansko prevajajo ločeno. Najprej se prevede zaprtje vsake metode in nato zaprtja, ki se nahajajo znotraj telesa metode. Telo metode `clen` ima v našem primeru dve zaprtji, ki pripadata stavku `if`. Prvo zaprtje predstavlja pogojni blok in drugo telo, ki se izvede ob izpolnjenem pogoju. Epilog zaprtja metode in bloka je podoben. Eksplicitno sproščanje okvirja se izvede samo v metodi `main`, kjer se izvajanje tudi zaključi.

Poglejmo si sedaj primer spreminjanja metode na primeru števca. Razred zapišemo:

```
class Stevec {
   public Stevec { }
   public getValue : Integer
   {
       return 0;
   }
   public increment
   {
      Integer v = getValue;
      method getValue = {  return v + 1; };
   }
   public main
   {
      Stevec stevec =  new Stevec;
       while{ stevec.getValue < 10; }
        {
          ("vrednost=" + stevec.getValue).print;
          stevec.increment;
        };
   }
}
```

Razred se prevede v naslednji kod:

public Stevec:

```
Core.new_frame 3, 3
Core.return_1
```

public getValue : Integer:

```
Core.new_frame 5, 4
l5.new Integer  0
Core.return_1 l5
```

public increment:

```
Core.new_frame 5, 3
l3.invoke_1_f #3, l5
l4.new Integer l5
Core.push l4
Core.setmethod_2 l4, l4, 1, #3, ??
Core.return_1
```

public increment_block_1:

```
Core.new_frame 7, 4
l6.new Integer  4
l7.new Integer l5
l7.add l6
Core.return_1 l7
```

public main:

```
Core.new_frame 4
l1.new_object 21 #2
Core.push l1
Core.pushref l2
l4.new Block ??, 2
Core.push l1
Core.pushref l2
l5.new Block ??, 2
l4.new While l2, l3
l4.exec
Core.skip_ifclear 0, ??
Core.return_1
Core.free_frame 4
Core.terminate
```

public main_block_1:

```
Core.new_frame 8, 4
Core.deref l6, l5
```

```
  l6.invoke_1_f #3, l7
  l6.new Integer 8
  l8.new Boolean
  l7.ifls , l6, l8
  l2.status l8, 1, 2
  Block.exit_1

public main_block_2:
  Core.new_frame 8, 4
  l6.new String  12
  Core.deref l7, l5
  l7.invoke_1_f #3, l8
  l7.new String l8
  l6.add l7
  l6.print
  Core.deref l6, l5
  l6.invoke_1_p #4
  Block.exit_1
```

Iz prevedenega koda lahko vidimo kako poteka nadomestitev metode z novim okoljem v metodi `increment`. Ustvari se novo zaprtje s starševsko spremenljivko, katerega naslov se nato nadomesti v objektovi tabeli metod.

Kot zadnji primer si poglejmo izpeljavo iz dveh nadrazredov:

```
class Bitje {
  public Bitje { }
   public vrsta : String
   {
       return ‘‘bitje’’;
   }
}

class Clovek {
```

```
  public Clovek { }
  public vrsta : String
  {
       return ‘‘clovek’’;
  }
}

class Filozof  inherits Bitje, Clovek {
  public Filozof(String Ime)
  {
       method ime = {  return Ime; };
  }
  public vrsta : String
  {
       return ‘‘filozof ’’ + ime;
  }
  public ime : String
  {
       return ‘‘neznan’’;
  }
}
```

Zaradi preprostosti prevedenih verzij `Clovek` in `Bitje` ne prikazujemo. Razred `Filozof` pa se prevede v naslednji kod:

public Filozof:

```
  Core.new_frame 4, 4
  l3.invoke_s #2, 16  // klic superkonstruktorja
  l3.invoke_s #6, 32  // klic superkonstruktorja
  Core.push l4
  Core.setmethod_2 l4, l4, 1, #10, ??
  Core.return_2
```

public Filozof_block_1:

```
Core.new_frame 5, 4
Core.return_1 l5
```

public vrsta:

```
Core.new_frame 6, 4
l5.new String  0
l4.invoke_1_f #2, l6, 0
l5.add l6
Core.return_1 l5
```

public ime : String:

```
Core.new_frame 5, 4
l5.new String  8
Core.return_1 l5
```

public main:

```
Core.new_frame 2
l2.new String  14
Core.push l2
l2.new_object 18 #0
l1.invoke_1_f #2, l2, 40
l2.print
Core.free_frame 2
Core.terminate
```

V konstruktorju se izvršita dva klica konstruktorjev. Prvi se pokliče nad razredom `Bitje` in drugi nad razredom `Clovek`. Invokacija konstruktorskih metod se poveže statično, saj je konstruktor natančno določen že v času prevajanja.

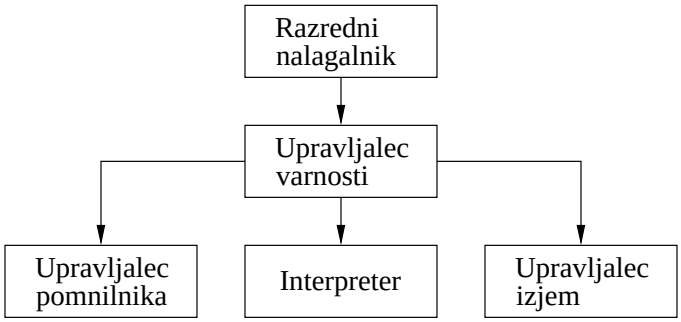
5 Virtualna arhitektura

Populariziran trend razvoja sodobne programske opreme je pridobitev na programski prenosljivosti. To je mogoče s prevajanjem v vmesno obliko, osnovano na definiciji nekega abstraktnega stroja. Ideja virtualnega stroja je že precej stara, vendar je praktični pomen dobila še nedavno, predvsem pod vplivom Jave. Čeprav so si virtualni stroji glede osnovne ideje med seboj zelo podobni, obstajajo razlike. Načrtovanje virtualne arhitekture je zapleten proces, ki zahteva poznavanje stvari s strojnih in jezikovnih vidikov. Da bi se programi učinkovito izvajali, je potrebno načrtovati takšne konstrukte, ki bodo lahko v virtualnem stroju učinkovito implementirani. Veliko jezikov ima izrazno zelo močne konstrukte, vendar je njihova implementacija skrajno zahtevna ali neučinkovita. Ker smo se odločili za prevedeno arhitekturo, bo izvajanje zložnega koda precej hitrejše, kot če bi vsako programsko frazo interpretirali na nivoju jezika. Po drugi strani pa je interpretacijo bistveno lažje implementirati. Nekateri jeziki so tako dinamični, da bi bilo prevajanje v zložni kod sploh neumestno ali nemogoče. Naš cilj so dinamični mehanizmi v okviru statičnega tipiziranja, prevedene arhitekture in varnega dinamičnega povezovanja [35].

Posamezne segmente virtualne arhitekture smo zasnovali tako, da bomo v prihodnosti lahko na preprost način aplicirali nove koncepte. Trend virtualnih arhitektur je v neprestanem večanju njihove funkcionalnosti in dinamike. Funkcionalnost, ki se zahteva, ni več zgolj osnovno izvajanje ali interpretiranje instrukcij, temveč tudi vse bolj splošna opravila. Med te v prvi vrsti uvrščamo interakcijo z operacijskim sistemom in strojno opremo, večnitno in porazdeljeno procesiranje in nekatere dinamične mehanizme, ki jih klasične virtualne arhitekture navadno ne omogočajo. Našo arhitekturo smo pripravili tako, da bo možno implementirati tudi meta nivo [31]. Ta bo igral pomembno vlogo pri kasnejši implementaciji meta konstruktov v jeziku. Druga pomembna stvar je arhitekturna dinamika. Ta se kaže predvsem v arhitekturni razširljivosti in prilagodljivosti v času izvajanja. Primer dinamike v času izvajanje je npr. dinamično generiranje zložnega koda [58, 84, 83]. Koncept dinamičnega prevajanja [62] se je izkazal za zelo uporabnega, saj so v času izvajanja na voljo informacije, ki jih

prevajalnik ob prevajanju še ne pozna. Prilagodljivost se pokaže v tem, da se lahko generira samo kod, ki je dejansko potreben. Drug primer je specializacija metod [70] ali optimizacija [69] v času izvajanja.

V pričujočem poglavju bomo opisali arhitekturo virtualnega stroja "Z", katerega smo implementirali za ciljno arhitekturo x86 [54, 55].
Virtualni stroj služi kot hipotetični računalnik, prirejen za izvajanje programov napisanih v jeziku Z_0. Osrednji del virtualnega stroja bo interpreter virtualnih instrukcij. Pomembno vlogo ima še razredni nalagalnik in upravljalec pomnilnika (slika 21). Ker je virtualni stroj implementiran programsko, se s strojnimi podrobnostmi ne bomo posebej ukvarjali.



Slika 21: Arhitektura virtualnega stroja.

Arhitekturo smo zasnovali modularno, s čimer smo omogočili razširitve in nadgradnjo funkcionalnosti. Še posebej pomembna je modularnost pri interpreterju, saj je zaželjeno, da lahko virtualne instrukcije dodajamo na enostaven način, ne da bi pri tem porušili obstoječo funkcionalnost. Procesno enoto virtualnega stroja smo segmentirali v podenote, od katerih vsaka izvršuje ukaze, za katere je bila načrtovana. V osnovi imamo podenote za realizacijo vgrajenih primitivnih tipov ter enote, ki so zadolžene za sistemske akcije. Sistemska enota se imenuje `Core` in definira instrukcije za ustvarjanje objektov, invokacijo metod, alokacijo okvirjev in druge.
Virtualni stroj se postavlja med izvajajočim se programom in specifičnim operacijskim sistemom ter na ta način doda nov nivo abstrakcije in s tem tudi zaščite med fizičnim

in programskim sklopom. Učinkovita zaščita je izjemnega pomena, saj preprečuje, da bi program izvajal napačne ali nedovoljene operacije in tako povzročil škodo. Pomen varnega izvajanja je dobil večjo veljavo z Javanskim virtualnim strojem [76]. Java je dokazala, da se dajo koncepti varnosti zelo elegantno vpeljati na nivo virtualnega stroja, še posebej če je ta implementiran v višjem programskem jeziku. Visoka stopnja varnosti zahteva pazljivo načrtovanje in implementacijo dovršenih varnostnih mehanizmov, ki pa žal upočasnjujejo izvajanje. Realni procesorji format instrukcije preverijo neposredno pred njeno izvedbo in signalizirajo izjemo, če pride do napake. V programsko implementiranem virtualnem stroju to ni možno, saj je treba instrukcijo preveriti pravtako programsko. S stališča učinkovitosti je takšen pristop nedopusten. Boljša ideja je preverjanje preden se program sploh prične izvajati. Tako lahko preverimo pravilnosti operacijskih kod, pripadajoče operande, tipe itd. S tem precej zmanjšamo možnost pojavitve napake med izvedbo programa. Seveda pa obstajajo logične napake, ki jih tudi na tak način ni mogoče odpraviti.

Pred izvajanjem ne moremo odpraviti tudi vseh napak pri naslavljanju. Ker se objekti ustvarjajo v času izvajanja, se tudi naslovi generirajo takrat. Referenciranje napačnega ali neobstoječega pomnilniškega naslova privede do izjeme. Ker je pravilne naslove zelo težko določati, smo arhitekturo zasnovali tako, da neposrednega naslavljanja pomnilnika ni. Referenciramo lahko samo lokalne naslove znotraj alociranega okvirja. Logični naslovni prostor zunaj okvirja ne obstaja. Prekoračitev referenciranja lokacij je enostavno preverljiva pred izvajanjem.

5.1 Razredna datoteka `.class`

Definicija razreda v izvorni datoteki `.z0` se prevede v razredno datoteko tipa `.class`. Zelo podobno shemo uporablja Java in javanski virtualni stroj. Preveden razred je binarne oblike in vsebuje informacije o nadrazredih in zunanjih razredih ter metodah. Seveda se v datoteki nahaja tudi preveden programski kod za vsako metodo in zaprtje, ki se v definiciji razreda pojavi. Razredi se prevajajo ločeno, kar pomeni, da je v eni datoteki izvršni kod samo enega razreda.

Format datoteke določa razvrstitev razrednih podatkov znotraj datoteke. Datoteka

vsebuje naslednje informacije:

- Glava datoteke, v kateri se nahajajo:
  - `signature` pomeni enoumno signaturo datoteke, s katero preverimo, ali dejansko gre za datoteko našega razrednega tipa.
  - `totalSize` je celotna velikost razredne datoteke v zlogih brez glave.
  - `nMethods` je število metod v tem razredu.
  - `vtableEntries` je število vstopov v virtualno tabelo razreda.
  - `mtableEntries` je število vstopov v tabelo metod razreda.
  - `literalLength` je dolžina literalov v zlogih.
  - `relocLength` je število vstopov v relokacijsko tabelo razreda.
  - `nExternalClasses` je število vstopov v tabelo zunanjih razredov.
  - `mainIndex` je indeks metode `main`.
- Relokacijska tabela, v kateri se nahajajo naslovi koda, ki jih je potrebno prenasloviti ob nalaganju razreda. Dolžina vstopa v relokacijsko tabelo je 4 zloge na 32 bitni arhitekturi.
- Povezovalna tabela, kjer so shranjeni naslovi, na katere je potrebno prenesti naslove zunanjih razredov. Povezovalna tabela se uporablja ob dinamičnem povezovanju razreda z zunanjimi in nadrazredi. Vsak vstop v povezovalno tabelo vsebuje:
  - dolžino imena razreda,
  - število vstopov za ta razred,
  - ime razreda in
  - zaporedje naslovov, ki jih je potrebno prenasloviti.
- Seznam literalov, v katerem se nahajajo vsi literali, ki so kjerkoli v razredu uporabljeni. Vrednosti literalov si sledijo zaporedno, zato njihovih naslovov

ni potrebno shranjevati. Instrukcije, ki referencirajo literale, bodo v procesu relokacije dobile njihove veljavne naslove.

- Razredni podatki, ki so potrebni pri instanciranju razreda. Sem spada velikost objekta in naslovi posameznih tabel.
- Tabela razredov se uporablja za dinamično preverjanje dejanskega objekta. V tej tabeli se nahajajo naslovi vseh nadrazredov skupaj s pripadajočimi odmiki v virtualno tabelo.
- Virtualna tabela.
- Tabela metod.
- Zložni kod razrednih metod.

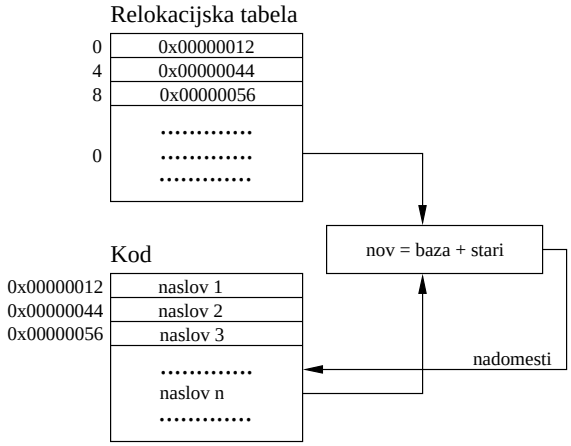
5.2 Nalaganje razreda

Ker je nalaganje razreda zapleten proces, ga razdelimo na več podopravil. Nalaganje se prične z branjem razredne datoteke. Nato se alocira pomnilniški prostor, v katerega se razred shrani. Sledi relokacija koda in tabel ter povezovanje z zunanjimi razredi. V primeru, ko katerikoli izmed zunanjih razredov ni naložen, ga je potrebno najprej naložiti. Nalaganje razreda je zato rekurzivno. Razredni nalagalnik smo zasnovali abstraktno, kar pomeni, da dejanski izvor razreda ni natančno specificiran. Zankrat se razred vedno nalaga iz razredne datoteke, v prihodnosti pa bo mogoče razrede v virtualni stroj nalagati transparentno tudi z drugih lokacij, kot je npr. internet.
Poglejmo si najprej proces relokacije.

5.2.1 Relokacija razreda

Relokacija ali prenaslavljanje pomeni spremeniti naslov. V času prevajanja vseh naslovov ni mogoče določiti, ker v splošnem ne vemo, kam se bo razred naložil v času izvajanja. Virtualne instrukcije za ustvarjanje objektov nekaterih primitivnih tipov (Integer, String,...) operirajo neposredno nad literali. Literal je podan z odmikom znotraj seznama literalov (constant pool), ki se nahaja na začetku razrednih podatkov. Ker

instrukcije operirajo nad naslovom posameznega literala, velikosti literalov ni potrebno posebej voditi. Razred se ob nalaganju naloži na nek pomnilniški naslov. To pomeni, da se literali pričnejo na nekem odmiku od tega naslova. Relokacijo opravimo spomočjo relokacijske tabele, v kateri se nahajajo odmiki v zložni kod, kjer je treba naslove prenasloviti. Relokacijo bi lahko opravili tudi neposredno med izvajanjem. Ta bi se izvedla, ko bi program bil že naložen v pomnilniku. Primer relokacije prikazuje slika 22.



Slika 22: Relokacija ob nalaganju razreda.

Ker imamo opraviti z 32 bitno arhitekturo, so tudi naslovi dolgi 32 bitov ali 4 zloge. To pomeni, da je vsak vnos v relokacijski tabeli dolg prav toliko.
Relokacija je potrebna samo za instrukcije, ki operirajo nad literali. Kljub temu, da se do literalalov dostopa neposredno preko naslova, programer na to nima vpliva, tako da tudi do nepravilnih naslovov ne more priti. Takšna zasnova referenciranja literalov je čistejša, kot če bi se vrednosti literalov nahajale neposredno v samih instrukcijah.

5.2.2 Dinamično povezovanje

Dinamično povezovanje pomeni povezovanje v času izvajanja ali nalaganja. Mi smo implementirali drugo možnost. Dinamično povezovanje je precej učinkovitejše od statičnega. Razredne datoteke so manjše in v primeru sprememb drugih razredov jih ni potrebno znova prevajati. Dovolj je, da se prevede samo spremenjen razred. Dinamično

povezovanje je tudi pomnilniško precej manj zahtevno, saj se že naloženi razredi ne nalagajo ponovno, ampak se delijo. Nalagajo se samo tisti razredi, ki so potrebni. Še boljšo prostorsko učinkovitost dosežemo s čistim oz. lenim dinamičnim nalaganjem (lazy linking). Pri tem se razredi nalagajo takrat, ko so prvič uporabljeni. To pomeni, da je nalaganje nedeterministično in odvisno od programskega toka. Opišimo sedaj proceduro dinamičnega nalaganja, kot smo jo uporabili v našem primeru.

Razred in njegove metode lahko vsebujejo reference na druge razrede. To se zgodi pri sami deklaraciji razreda v primeru dedovanja ali kjerkoli v izrazih. Razred mora imeti reference na vse nadrazrede, da lahko ustvari nadobjekte. Pravtako je razred potreben v deklaraciji, kjer ustvarjamo objekt tega razreda. V kolikor ustvarjamo objekt lastnega razreda imamo podatke že na voljo. Če pa ustvarjamo objekt nekega zunanjega razreda, moramo ta razred imeti že naložen v pomnilniku.

Instanciranje zunanjega razreda se prevede v instrukcijo za instanciranje `new_object`, ki med drugim prejme naslov razreda. Razredni naslovi v vseh tovrstnih instrukcijah se morajo prenastaviti na dejanske naslove, ki so znani šele v času nalaganja. Odmiki znotraj koda, kjer se referencirajo naslovi razredov, so shranjeni v povezovalni tabeli razreda. V tej tabeli so shranjeni vsi razredni naslovi za vse zunanje razrede, ki so referencirani znotraj tega razreda. Strukturo povezovalne tabele prikazujemo na sliki 23.

dolzina, st. vstopov, razred 1, naslov 1, naslov 2, ...
dolzina, st. vstopov, razred 2, naslov 1, naslov 2, ...
. . . . .
dolzina, st. vstopov, razred n, naslov 1, naslov 2, ...

Slika 23: Struktura povezovalne tabele razreda.

Povezovalna tabela je vezana na razred. Ker se razred poveže samo enkrat (ob nalaganju), tabele ni potrebno shraniti v pomnilniku. Povezovalna tabela je v tem zelo podobna relokacijski.

Povezovanje z zunanjim razredom pa ne pomeni samo ponastavitev razrednih naslovov. Ob nalaganju razreda je potrebno dinamično povezati tudi tabelo metod. Razred poleg metod, ki jih definira sam, vsebuje tudi vse metode nadrazredov. Naslove lastnih metod lahko izračunamo v času prevajanja in jih v času nalaganja samo prenaslovimo. Z

naslovi metod iz nadrazredov pa v času prevajanja ne moremo početi prav nič. Naslove metod nadrazredov pridobimo šele v času nalaganja. Zato morajo vsi nadrazredi biti že naloženi. Če to velja, potem tabelo metod nadrazreda preprosto skopiramo v tabelo metod našega razreda. Virtualna tabela ostane nespremenjena.
Povezovalna tabela se uporablja tudi pri povezovanju naslovov v pretvorbenih instrukcijah. Tukaj mislimo na pretvorbene operacije (casts) med objekti primitivnih in kompleksnih tipov. Vsak objekt je med drugim tipa `Object`. To pomeni, da lahko vsak objekt pretvorimo v `Object`. Tovrstna pretvorba je trivialna. Objekt pa je lahko tudi drugega, uporabniškega tipa. Ker v času izvajanja v splošnem ne vemo, ali je nek objekt nekega tipa, je potrebno takšne preverbe vršiti dinamično. Tukaj pridejo v poštev spet povezovalni naslovi. Vsak objekt nekega tipa vsebuje reference na vse nadrazrede, iz katerih lahko v času izvajanja hitro ugotovimo, za kateri tip dejansko gre.

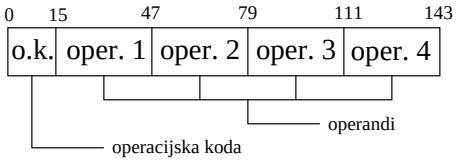
5.3 Zložni kod

Zložni kod je izvedljiv kod za našo virtualno arhitekturo. Poudarimo, da naš zložni kod ni povsem "zložni", saj instrukcije ne temeljijo na enem zlogu, temveč večih. Posamezna instrukcija sestoji iz operacijske kode in opcionalnih operandov. Najmanjša dolžina virtualne instrukcije je torej dolžina operacijske kode, ki znaša dva zloga ali 16 bitov. Operandi so 32 bitni. Dolžina instrukcij je variabilna. Format instrukcije je prikazan na sliki 24.
Naslavljanje je preprosto, saj nimamo ne segmentov kot tudi ne odmikov znotraj posameznih naslavljanj. Pravzaprav imamo samo tri vrste naslavljanja:

- takojšnje, pri katerem operand predstavlja dejansko vrednost,
- lokalno, kjer referenciramo lokacijo znotraj lokalnega okvirja in
- literalno, kjer je operand pomnilniški naslov literala.

Zadnje naslavljanje je v osnovi bolj znano absolutno naslavljanje, kjer je podan pomnilniški naslov, katerega se ob nalaganju prenaslovi. Kar zadeva naslavljanje, format instrukcije spada v družino RISC (Reduced Instruction Set Computer). Ker imamo

opraviti z virtualnim strojem, ki je implementiran programsko, je dekodiranje instrukcije izvedeno drugače kot pri realnem procesorju. Interpreterjev dekodirnik najprej prebere operacijsko kodo in se pomakne na funkcionalnost pripadajoče instrukcije.



Slika 24: Format virtualne instrukcije.

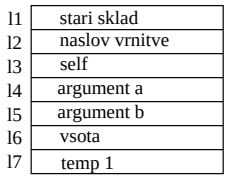
5.4 Lokalne spremenljivke

Lokalne spremenljivke se nahajajo samo znoraj teles metod. Vse lokalne spremenljivke so shranjene v okvirju. Okvir je zelo podoben skladu, le da ima okvir možnost direktnega dostopa do svojih lokacij. Ta lastnost je bistvenega pomena, saj dopušča neprimerno več možnosti za optimizacijo med prevajanjem. Instrukcije Javanskega virtualnega stroja lahko referencirajo elemente samo na vrhu sklada. V našem primeru lahko referenciramo lokacije kjerkoli znotraj okvirja. Lokacija okvirja je pri prevajanju v realni kod za nek procesor sinonim za register. Če ima arhitektura dovolj registrov, lahko praktično vse lokacije preslikamo v registre s čimer krepko pohitrimo izvajanje, saj so operacije nad registri običajno dosti hitrejše kot tiste s pomnilnikom.
Okvir se alocira v prologu metode z neko določeno velikostjo. Velikost okvirja je določena med prevajanjem, ko se izračuna število poimenovanih in začasnih spremenljivk v telesu metode. Sprostitev okvirja se izvrši ob izstopu iz metode ali bloka. Okvir je reflektiran na skladu virtualnega stroja. To pomeni, da instrukcije nad okvirjem in skladom naslavljajo isti pomnilniški prostor. Hkrati pridobimo na preprostosti prevajalnega modela, saj je referenciranje lokalnih spremenljivk in argumentov na skladu poenoteno. Strukturo okvirja najnazorneje prikažemo na primeru metode. Okvir na sliki 25 je alociran za metodo `izpis`, ki je podana na sledeč način:

```
izpis(Integer a, Integer b)
 {
   Integer vsota = a + b;
```

```
  vsota.print;
}
```

Okvir se razlikuje za procedure, funkcije ter statične in virtualne metode. Na prvi lokaciji okvirja se nahaja skladovni števec, kot je bil pred alokacijo okvirja. Temu sledi naslov vrnitve, ki pove, kam se treba vrniti po izvršitvi metode. Naslov vrnitve je vedno instrukcija, ki sledi invokaciji. Na tretji lokaciji se nahaja referenca prejemnika sporočila. To je objekt, nad katerim smo metodo invocirali. Nato sledijo uporabniške spremenljivke, ki se pričnejo s podanimi argumenti. Povsem na koncu okvirja se nahajajo lokalne in začasne spremenljivke.



Slika 25: Struktura okvirja za metodo `izpis`.

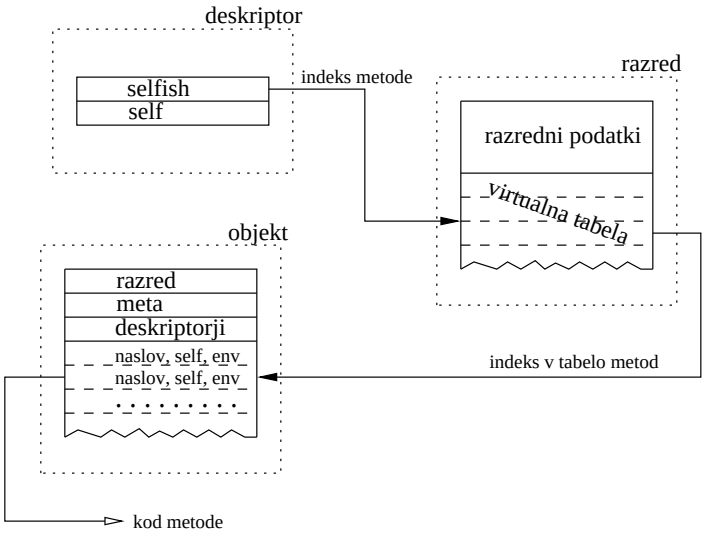
Statične metode ne potrebujejo reference na prejemniški objekt, ker niso vezane na instanco ampak na razred. Okvir funkcij zajema dodatno lokacijo za shranjevanje rezultata v lokalno spremenljivko po izstopu iz funkcije.
Dodatna razlika med skladom in okvirjem je v tem, da okvir raste navzgor. Lokalne spremenljivke referenciramo od `l1`, ki pomeni lokalno spremenljivko na prvi lokaciji, do `ln`, kjer je `n` velikost okvirja.
Okvir je pomemben koncept naše arhitekture, saj je z njegovo pomočjo realizirano okolje, ki igra pomembno vlogo pri blokih in metodah. Vse lokalne spremenljivke, ki se v telesu metode ali bloka referencirajo, morajo biti alocirane v okvirju. Ker se argumenti metodam pošljejo na skladu, je ob alokaciji okvirja potrebno skladovne reference skopirati v okvir. Vsaka metoda in vsak blok skopira vsaj skladovni števec in naslov vrnitve. Instrukciji za alokacijo okvirja sta `Core.new_frame` in `Core.new_frame_copy`. Prva opravi samo alokacijo, druga pa alokacijo in kopiranje. Kopiranje v okvir je pomembno, ker se sklad dinamično spreminja, metode in bloki pa lahko ubežijo svojemu starševskemu okolju. Na strukturo sklada se torej ne moremo zanašati v vsakem

trenutku.
Epilog metode ali bloka okvir sprosti. V metodi se to zgodi z eno izmed `return` instrukcij, v bloku pa z `exit`.

5.5 Invokacija metod

Invokacija metode je temeljna operacija nad objektom. Proces invokacije se prične s pošiljanjem sporočila in konča s klicem metode. Sporočilo, ki prispe do objekta, se najprej preslika v ustrezno metodo. Vsako sporočilo ima namreč pripadajočo metodo. Ker je jezik statično tipiziran, je klic metode vedno veljaven, saj objektu ne moremo poslati sporočila, h kateremu ne bi pripadala metoda. V jeziku Z_0 so vse metode implicitno virtualne, kar pomeni, da gre za pozno povezane klice. Nasprotje temu so statične metode, ki se določijo že v času prevajanja. Ker se invokacija izvrši nad nekim objektom, potrebujemo referenco sprejemnika. Ta se v našem primeru imenuje `self`. Referenca prejemnika se pošlje implicitno.
Vsi argumenti se pošljejo na skladu. Prolog metode jih nato skopira v lokalni prostor, ki se nahaja v okvirju. Po izvedbi metode se sklad počisti. Ker se struktura okvirja razlikuje, ločujemo med procedurami in funkcijami. Instrukciji za virtualno invokacijo teh sta `Core.invoke_1_p` in `Core.invoke_1_f`. Vsaka instrukcija za invokacijo, ne glede ali gre za virtualno ali statično invokacijo, prejme samo indeks metode. V virtualni tabeli se na tem mestu nahaja indeks v tabelo metod, ki je vezana na instanco razreda. Virtualna tabela se deli med vse objekte istega razreda. V tabeli metod se na mestu indeksa nahaja naslov metode, katero želimo invocirati. Tako imamo za vsako invokacijo dvojno referenciranje. Takšna implementacija je bolj zamudna kot npr. tista v C++, vendar je treba zavedati varnosti sistema tipov in dinamičnosti, ki je C++ ne omogoča. Vsak vstop v tabelo metod vsebuje pomnilniški naslov metode, dejanski objekt in okolje metode. Dejanski objekt je referenca na objekt, ki je lastnik metode. Ker jezik omogoča dinamično spreminjanje metode, je invocirana metoda lahko definirana na nekem drugem objektu in ne našem trenutnem sprejemniku. Tako za invokacijo v splošnem potrebujemo dva prejemnika. Enega da metodo sploh najdemo in drugega kot dejanski objekt, nad katerim metodo invociramo. Tretja informacija, ki jo potrebu-

jemo samo v določenih primerih, je referenca okolja. Metoda je bila lahko spremenjena tako, da vsebuje reference iz enega ali več starševskih okolij. Te reference je potrebno shraniti v okolje metode. Celovit postopek invokacije je prikazan na sliki 26. Telo metode se izvede v kontekstu starševskega okolja. Ovrednotenje funkcije ni dinamično v klasičnem pomenu. V izvršnem kodu imamo samo reference in ne identifikatorjev, zato lahko dinamično ovrednotenje izvedemo samo na osnovi referenc in ne identifikatorjev.



Slika 26: Proces invokacije virtualne metode.

V splošnem je invokacija sledeč postopek:

- instrukciji `Core.push` in `Core.pushref` na sklad potisneta argumente od leve proti desni,
- klic se izvrši preko instrukcij `Core.invoke_1_p` ali `Core.invoke_1_f`,
- znotraj instrukcije za klic se na sklad shrani še dejanski prejemnik in naslov vrnitve. Iz tabele metod objekta se prebere naslov metode in programski števec skoči na ta naslov,
- v prologu metode se ustvari okvir, v katerega se skopirajo vrednosti s sklada in star skladovni kazalec,

- prične se izvedba glavnega dela metode.

Vzemimo za primer zgoraj definirano metodo `izpis`, ki prejme dve celi števili ter izpiše njuno vsoto. Telo metode se prevede v naslednji kod, ki ga zapišemo v zbirniku jezika Z_0:

```
Core.new_frame 7, 5   // prolog
l7.new Integer l4
l7.add l5

l6.new Integer l7
l6.print

Core.return_2        // epilog
```

Prva vrstica predstavlja prolog, zadnja pa epilog metode. Če se vrednosti, ki jih želimo podati kot argumenta metode nahajata v lokalnih spremenljivkah `l4` in `l5` ter prejemnik v `l1`, se invokacija te metode prevede v naslednji kod:

```
Core.push l4
Core.push l6
l3.invoke_1_p #2
```

Lokacije v okvirju se z vsako instrukcijo `Core.push` premaknejo. Metode `izpis` se v virtualni tabeli nahaja na lokaciji 2.

Razlika med klasično virtualno in statično invokacijo je v prejemniku. Statične metode ne potrebujejo prejemnika, ker ne operirajo nad stanjem objekta. Zato jih lahko povežemo že v času prevajanja. Kot vsaka virtualna, ima tudi statična metoda svoj indeks. Razlika med obema je ta, da kaže indeks statične metode neposredno v tabelo metod. Statična invokacija je s tega stališča preprostejša, saj potrebuje samo enkratno referenciranje. Z indeksom statične metode pridemo neposredno do pomnilniškega naslova, kjer se nahaja zložni kod metode.

Statično povezovanje je lahko tudi implicitno. Kadarkoli se ime metode "dereferencira" s polnim imenom, se metoda poveže statično, čeprav ni deklarirana kot statična.

5.6 Zaprtja

Zaprtja so osnova dinamike blokov in metod. Zaprtje (closure) je v najpreprostejšem pomenu struktura, ki vsebuje okolje in nek izvršni kod. Okolje sestavljajo spremenljivke in njihove vrednosti. Izvršni kod zaprtja se izvede v kontekstu pripadajočega okolja. Zaprtja so se izkazala za zelo učinkovita pri implementaciji parcialnih funkcij ali funkcij višjega reda (higher order functions). Funkcije višjega reda so takšne funkcije, ki vračajo funkcije kot prvorazredne vrednosti. Ker so funkcije višjega reda domena funkcijskih jezikov, so tudi zaprtja osnovni konstrukti praktično vseh tovrstnih jezikov. Za zaprtja smo se odločili zaradi dinamičnosti konstruktov našega jezika. Tukaj imamo v mislih bloke in metode. Preslikava med blokom in zaprtjem je 1:1. Blok ima svoje okolje in kod, ki se izvrši ob izvršitvi bloka. Zaprtje samo po sebi ni objekt, temveč je del objekta, ki vsebuje referenco nanj. Vse kontrolne strukture jezika, ki smo jih že predstavili, vsebujejo vsaj eno referenco na blok. Blok vsebuje pomnilniški naslov zaprtja.

Zaprtje vsebuje okolje lokalnih spremenljivk, ki jih referencira. Okolje zaprtja je implementirano tako, da preprosto vsebuje seznam referenc. To okolje se aktivira, ko se zaprtje izvrši. Okolje vsebuje vse reference starševskih spremenljivk, ne glede kje se te spremenljivke nahajajo. Starševskih okolij je v splošnem lahko več. Referenca, ki je uporabljena v najbolj vgnezdenem bloku, vmes pa ne, se mora tako prenesti skozi vse bloke. Kljub temu je ta rešitev boljša, kot da bi shranjevali reference na celotna starševska okolja. Dostop do spremenljivke bi ob takšni filozofiji bil precej daljši, kot je zdaj.

Metodo za izvršitev zaprtja smo poimenovali `exec`. Metodo implementirajo vsi razredi, s katerimi predstavimo kontrolne strukture. Invokacija metode `exec` nad blokom opravi sledeče korake:

- pogleda, če okolje zaprtja ni prazno. V primeru ko ni, potisne vse reference okolja na sklad,
- na sklad shrani naslov vrnitve in prejemnika sporočila `exec`, tj. blok,
- inicializira zastavice bloka,

- tok izvajanja se na enak način kot pri invokaciji metode prenese na naslov izvršnega koda bloka.

Izvedba bloka je sinonim za izvedbo zaprtja. Postopek izvedbe je praktično identičen invokaciji procedure ali funkcije.
Vrnitev s stavkom `return` znotraj bloka pomeni vrnitev iz metode, v kateri je blok definiran. Omejimo se zaenkrat na bloke, ki se izvedejo implicitno in ne ubežijo starševskemu okolju. Kadar v bloku pride do vrnitve, se izvajanje bloka zaključi. Neposredno za instrukcijo `exec` je zato treba preveriti, ali se je vrnitev res dogodila. Če je blok izvršil stavek `return`, je treba isti stavek izvršiti tudi v metodi. Zavedati se je treba, da se bloki lahko gnezdijo do poljubnih globin. Če pride do vrnitve v nekem vgnezdenem bloku, je treba to preveriti v bloku, ki je en nivo više. Vračanje se v takem primeru veriži skozi bloke vse do nivoja metode, kjer se izvede `return`. Preverjanje vračanja pride v poštev, kot že rečeno, samo pri blokih, ki se izvedejo takoj. Stavek `if`, ki v bloku telesa vrača vrednost, se prevede v naslednje instrukcije:

```
........
Core.push l4                    // spremenljivka okolja
Core.push l6                    // spremenljivka okolja
l8.new Block ??, 2              // nov pogojni blok
Core.push l4                    // spremenljivka okolja
l8.new Block ??, 1              // blok then
l8.new If l6, l7                // stavek if
l8.exec
Core.skip_ifclear 0, 10         // preverimo ali je blok vrnil
Core.return_2 l1, 1             // vrnemo rezultat iz bloka
.....                           // nadaljujemo z izvajanjem
```

Instrukcija, ki preveri ali je do vrnitve prišlo, je `Core.skip_ifclear`, ki testira, ali je bit 0 sistemskega registra zastavic zbrisan. Z ?? označimo naslove zaprtij, saj ti v času prevajanja niso znani.
Bloki, ki se lahko izvršijo zunaj starševskega okolja, nimajo težav z vračanjem, saj le-to ni dovoljeno.

5.7 Implementacija spreminjanja metod

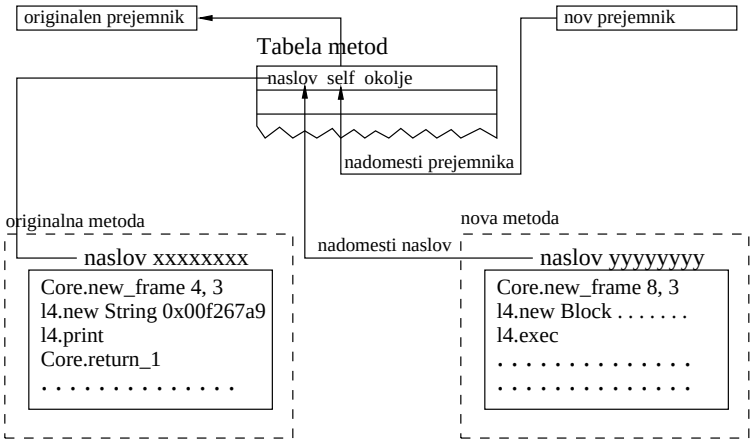
Stanje objekta je določeno izključno z metodami. Da bi lahko stanje spreminjali, potrebujemo torej manipulacijski mehanizem, ki bo znal spreminjati metode objekta. Takšna filozofija zahteva pristop, ki v obstoječih jezikih ni prav dobro znan. Ker metode opisujejo interaktivno (trenutno) stanje objekta, jih je potrebno vezati na instanco. Metode so po klasičnem razumevanju identične za vse instance istega razreda. Toda v našem jeziku imamo možnost, da instanci razreda C metodo M spremenimo v M', medtem ko mora ista metoda druge instance razreda C ostati nespremenjena. Kako torej doseči spreminjanje metod?

Ena možnost je ta, da ima vsak objekt dejansko svoje metode. Žal je taka rešitev s stališča prostorske zahtevnosti povsem nesprejemljiva. Kod posamezne metode lahko ostane nespremenjen. To pomeni, da se funkcionalnost metod lahko deli med različne instance. Vse kar se menjuje, je prejemnik sporočila. Ker je metoda lahko nadomeščena z metodo nekega drugega objekta, mora ta prejemnik biti tisti objekt, nad katerim je metoda definirana. Metoda mora biti torej invocirana nad dejanskim objektom. Da bi metodo našli, pa potrebujemo tudi prejemnika, nad katerim metodo invociramo.

Pri spreminjanju metode postane pomembno tudi okolje. Nov blok metode lahko referencira lokalne spremenljivke in argumente starševskih okolij. Ker so te vrednosti reference, jih je potrebno na nek način prenesti v okolje metode. Ker se reference vrednotijo dinamično, imamo dinamično povezovanje na nivoju referenc. Vrednost reference se ne kopira, temveč se vzame trenutna vrednost. Če ponastavljena metoda M referencira starševsko spremenljivko v, potem bo vrednost te spremenljivke ob izvedbi metode M takšna, kot je neposredno pred klicem in ne kot je bila v času spreminjanja metode. Da bi bila invokacija čimučinkovitejša, se okolje metode ustvari samo, ko je to potrebno. Če blok metode ne referencira starševskih spremenljivk, okolja ni smiselno ustvarjati.

Ker so metode vezane na instanco, bo tudi tabela metod shranjena v objektu. To pomeni nekoliko večjo prostorsko zahtevnost, vendar boljšega načina, ki bi omogočal dinamične spremembe, zaenkrat ne poznamo. Tabela metod vsebuje vstope za vsako metodo v razredu. To vključuje vse nadrazrede. Posamezen vstop v tabelo metod

zajema pomnilniški naslov metode, dejanskega prejemnika in naslov okolja. Sprememba metode pomeni spremembo vnosa v tabeli metod. Naslov se spremeni v naslov metode, ki staro metodo nadomesti. Dejanski prejemnik se ponastavi tako, da kaže na prejemnika nove metode. Končno se ponastavi še naslov okolja, ki kaže na okolje nove metode. V kolikor metoda okolja ne potrebuje, je ta kazalec prazen in se ob invokaciji ne upošteva. Instrukciji za spreminjanje metod sta `Core.setmethod_1` in `Core.setmethod_2`. Postopek spreminjanja metode je prikazan na sliki 27.



Slika 27: Proces spreminjanja metode brez okolja (`Core.setmethod_1`).

Tukaj se izkaže naša predstavitev objekta. Omenili smo že, da so vse metode razen statičnih, virtualne. To pomeni, da se povezujejo v času nalaganja oz. izvajanja. Če bi ubrali preprostejšo predstavitev in bi virtualno tabelo združili v tabelo metod, bi imeli nemalo težav pri spremembi virtualne metode. Sprememba virtualne metode v podrazredu pomeni, da mora biti sprememba vidna tudi ob invokaciji te iste metode nad objektom, ki je nadrazrednega tipa. Torej bi morali v splošnem spremeniti več kot le en naslov, kar pa v času izvajanja ni trivialno.

5.8 Ustvarjanje objektov

V jeziku, kjer so vse entitete predstavljene kot objekti, je ustvarjanje objektov nadvse pomembno opravilo. Ustvarjanje objekta mora biti implementirano zelo učinkovito, saj bo v nasprotnem primeru trpelo celotno izvajanje programa. Objekte razdelimo na

objekte primitivnih tipov in objekte uporabniških oz. splošnih tipov. Zunanja struktura obojih je enaka, vendar se glede implementacije razlikujejo. Ker so primitivni tipi že vgrajeni v jezik, je njihova implementacija že določena. Razrede splošnih tipov implementira programer. Objekti primitivnih tipov se ustvarjajo drugače od splošnih. Objekti primitivnih tipov se ustvarjajo implicitno z navedbo njihove vrednosti. Če se v programu torej pojavi celoštevilčna vrednost 3, to pomeni, da gre za objekt tipa `Integer`, katerega vrednost je 3. Objekt se lahko ustvari neposredno v deklaraciji ali posredno v izrazu kot delni rezultat. Objekti primitivnih tipov se ne ustvarjajo z operatorjem `new`.

Drugače je z objekti splošnih tipov. Le-ti se ustvarjajo izključno z operatorjem `new`, ki vrne referenco na pravkar ustvarjen objekt. Čeprav obstaja razlika na programskem nivoju, pa je implementacija alokacije prostora za objekt identična. Objekt zajema kazalec na razred, meta podatke in tabelo metod. Objekti primitivnih tipov za tabelo metod zajemajo še dodatne vrednosti, vendar te navzven niso vidne. Objekt je referenciran z deskriptorjem, ki vsebuje kazalec nanj in na trenutno virtualno tabelo v razredu. Prostor, ki je potreben za shranjenje objekta v pomnilniku, je alociran s kopice. S kopice se alocirajo objekti tako primitivnih kot splošnih tipov.

Objekti primitivnih tipov se ustvarjajo s konstruktorskimi instrukcijami, ki ustvarijo objekte s privzetimi vrednostmi, kopijami ali povsem specifičnimi vrednostmi, ki so podane kot operandi instrukciji. Specifične vrednosti primitivnih tipov se ustvarjajo preko literalov, ki so shranjeni na začetku razreda. Objekti neprimitivnih tipov se ustvarjajo z instrukcijo `Core.new_object`.

Instrukcija `Core.new_object` najprej alocira prostor za objekt. Velikost objekta se nahaja v razrednih podatkih, katerih naslov se pošlje kot operand instrukciji. Nato se ustvari dejanski objekt, v katerem se nastavi kazalec na razredne podatke in kazalec na meta podatke. Meta podatki trenutno še niso uporabljeni. Svoj pomen bodo dobili v implementaciji meta nivoja jezika. Za ta dva kazalca se skopira začetna tabela metod, ki se pravtako nahaja v razrednih podatkih. Vrednosti dejanskega prejemnika v tabeli metod se nastavijo na pravkar alocirani objekt. Kazalci okolij se nastavijo na 0, ker niso potrebni. Nazadnje se ustvari še deskriptor objekta, preko katerega poteka referenciranje.

Ob vsaki ustvaritvi objekta se izvede eden izmed definiranih konstruktorjev. Z implementacijskega stališča na konstruktor gledamo kot na navadno metodo, ki se izvede

nad objektom. Instrukcija `Core.new_object` zato kot operand prejme tudi indeks konstruktorske metode v virtualni tabeli. Ko je objekt ustvarjen, se implicitno invocira metoda s pripadajočim indeksom. V primeru, ko konstruktor prejme argumente, so ti že na skladu, saj je ustvarjanje objekta enako invokaciji metode, le da tej dodamo operator `new`. Primer instanciranja razreda `Oseba` s konstruktorjem `Oseba(String priimek, String ime)`, je sledeč:

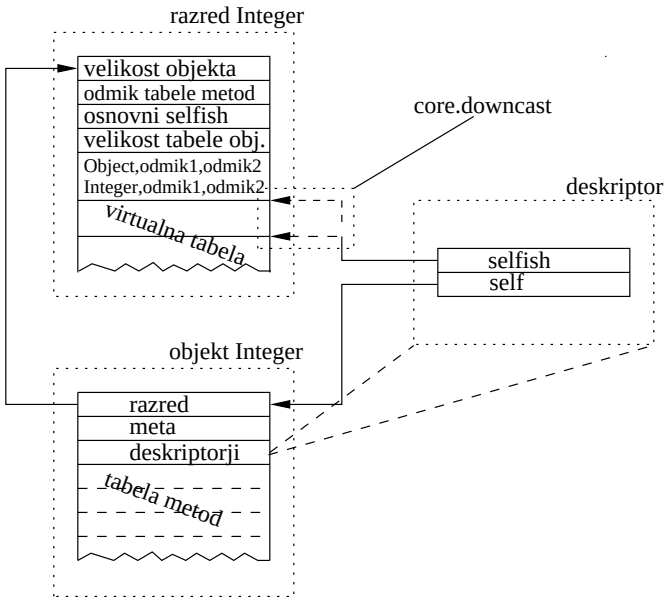
```
Core.push l4  // priimek
Core.push l6  // ime
l3.new_object 96 #0
```

V `l1` (`l3` po instrukcijah `push`) se shrani deskriptor objekta `Oseba`, 96 je naslov razreda `Oseba` in #0 je indeks konstruktorske metode. Poudarimo, da se naslov razreda določi šele v času nalaganja v procesu povezovanja.

Dedovanje ne doprinese večjih zapletljajev pri instanciranju. Ker smo razredno hierarhijo načrtovali s skupnim korenom `Object`, je vsak razred, razen samega Objekta seveda, dedovan iz tega razreda. Zaradi učinkovitosti primitivnih tipov, se instanciranje nadrazredov pri teh ne vrši na enak način kot pri kompleksnih tipih. Instanciranje razreda primitivnega tipa ne povzroči invokacije konstruktorja nadrazreda. Potrebno je samo, da se skopira tabela metod. Takšna posplošitev je možna, dokler v konstruktorju razreda `Object` nimamo eksplicitno definirane funkcionalnosti.
V primeru instanciranja razreda kompleksnega tipa je potrebno instancirati vse nadrazrede. Objekt se ustvari celovito kot en sam objekt konkateniran z vsemi starši. Ker so konstruktorji nadrazredov v splošnem uporabniško definirane metode, je treba zagotoviti invokacijo teh metod. Konstruktorji se kličejo od zgoraj navzdol. To pomeni, da se najprej pokliče konstruktor najvišjega razreda v hierarhiji in nazadnje konstruktor zadnjega razreda. Privzeti konstruktor nadrazreda se pokliče implicitno. Če nadrazred nima definiranega privzetega konstruktorja, je njegov klic potrebno navesti eksplicitno.

5.9 Dinamično preverjanje tipov

Dinamično preverjanje tipov poteka povsod, kjer je tip treba pretvoriti v nek drug tip. Operacija preverjanja je pozitivna, če je tip vsebovan v trenutnem tipu, in negativna, če takega tipa ni. Operacija pretvorbe tipa je upodobljena na sliki 28.



Slika 28: Pretvorba tipa.

Instrukciji za pretvorbo tipa sta `Core.upcast` in `Core.downcast`. Instrukciji preverita, ali se da trenutni tip pretvoriti v želenega in spremenita kazalec na deskriptor objekta, ko je to potrebno. V nasprotnem primeru se javi napaka. Instrukcija `Core.upcast` je funkcionalno preprostejša, saj lahko kazalec na virtualno tabelo spremeni neposredno. V splošnem je pretvorba tipa zahtevna operacija, saj terja vpogled v tabelo objektov. Pretvorba tipa je v jeziku C++ implementirana precej preprosteje kot pri nas. Posledica tega je, da pretvorbe v C++ niso varne in lahko povzročijo nepredvidljive napake. C++ s klasičnim mehanizmom pretvorbe ne more preveriti, ali je pretvorba smiselna in ali je veljavna. Po drugi strani pa obstaja mehanizem dinamične pretvorbe (dynamic cast), ki pa je pravtako zahteven kot naš. Varnost za čas torej.

6 Zaključek

Primarni cilj pričujočega dela je bil načrtovati objektno usmerjen programski jezik, ki bi bil statično varen in hkrati omogočal dinamične mehanizme, kateri so običajno prisotni v interpretiranih dinamično tipiziranih jezikih. Drug cilj je bil jezik tudi učinkovito implementirati. Jezik smo načrtovali in implementirali na način, da lahko v tem trenutku pišemo programe, jih prevajamo in izvajamo na virtualnem stroju, katerega smo implementirali specifično za naš jezik.

Načrtovanja smo se lotili v okviru jezikovnih konceptov, katere smo želeli implementirati. Ker je jezik Z_0 objektno usmerjen programski jezik, smo vzeli nekaj najpopularnejših objektno usmerjenih jezikov in preučili njihove koncepte. Še posebej smo se osredotočili na tiste koncepte, ki so značilni za posamezen jezik. Če je bila objektna usmerjenost skupna nit vsem jezikom, to ne velja za nekatere druge koncepte. Smalltalk nas je navdušil s svojim čistim objektnim modelom, Java s statično tipiziranostjo in prevedenim kodom. Konstrukt `Self` smo povzeli po Eifflu in sintakso jezika spet po Javi. Izbrane koncepte smo združili še z dinamiko spreminjanja metod in jih stkali v lasten jezik. Jezik smo skupaj z izbranimi koncepti načrtovali kot statično tipiziranega, s čimer smo sicer povzročili nekaj omejitev, vendar je jezik varnejši.

Javanski tip sintaktičnega modela smo izbrali zaradi preprostosti, berljivosti in razširjenosti. Javanska sintaksa velja za eno izmed najčistejših v imperativnih objektno usmerjenih jezikih. Javansko sintakso smo torej združili z lastnimi koncepti in jo izoblikovali v sintakso jezika Z_0. Na nekaterih mestih je bilo potrebno stvari dodati, drugje smo jih še bolj poenostavili.

Kar zadeva jezikovne koncepte, ne moremo trditi, da smo načrtovali kaj povsem novega. Kljub temu pa koncepti združeni na naš način so originalni. Originalna je predstavitev objektovega stanja, ki je podano precej bolj abstraktno kot npr. v Javi ali C++. Ker smo stanje objekta unificirali z metodami, le-to ni več podano eksplicitno, temveč postane zakrito. Takšna predstavitev uporabniku razreda omogoča bolj

abstrakten pogled na funkcionalnost razreda. Obstajajo samo lastnosti objekta, do katerih dostopamo preko metod. Tako za predstavitev starosti študenta ni potrebno razmišljati ali je v razredu starost predstavljena s spremenljivko nekega tipa, ampak je dovolj, da ima razred dve metodi, od katerih ena vrne starost, druga pa jo nastavi. Zaradi unifikacije instančnih spremenljivk in metod pridobimo tudi na čistosti manipulacije.
Odločitev za mehanizem spreminjanja metod je bila ključnega pomena. S tem ne pridobimo samo na abstraktnosti in čistosti, temveč tudi veliko povsem pragmatičnih stvari. Ena izmed teh so vsekakor metode višjega reda. Zaradi možnosti manipulacije telesa metode in njene predstavitve kot prvorazredne vrednosti, lahko implementiramo metode, ki prejmejo metode kot argumente ali vračajo nove metode. Koncept metod višjega reda sicer v obstoječi fazi jezika še ni implementiran, vendar računamo na to v bližnji prihodnosti. Prvorazredna predstavitev vseh vrednosti je bilo sicer eno izmed osnovnih vodil v fazi načrtovanja jezika. Za primitivne vrednosti takšna predstavitev velja za nekaj povsem običajnega. Prvorazrednost višjih programskih konstruktov pa, predvsem zaradi implementacijskih zahtevnosti, ni v navadi. Ker tovrstna predstavitev konstruktov doprinese jeziku neprimerno večjo izrazno moč, smo se tega lotili s kar največjim zanosom. Ne glede na zaplete in kompleksnosti ob implementaciji, smo uspeli koncept na nevsiljiv in učinkovit način umestiti v jezik.

Druga lastnost, katero smo pridobili s prvorazrednostjo metod, je dinamična prilagoditev trenutnim zahtevam. Funkcionalnost metode lahko v času izvajanja dogradimo ali jo povsem prilagodimo nekim zatevam, katerih ne moremo predvideti v času prevajanja. Metode tako postanejo preprostejše, bolj specifične in lažje berljive. Ker jih lahko prilagajamo, so tudi bolj učinkovite, saj izvajajo izključno funkcionalnost, ki je potrebna.

Precej časa smo posvetili tipiziranju in sistemu tipov. Zaradi čistega objektnega pristopa nismo imeli posebnih težav pri tipiziranju posameznih entitet. Princip objektizacije smo aplicirali na prav vse jezikovne entitete, od primitivnih vrednosti do kontrolnih struktur. Sistem tipov smo zasnovali z najvišjo stopnjo fleksibilnosti, ki je

sprejemljiva v okviru statičnega tipiziranja. Odločili smo se za varianten sistem tipov, ki dovoljuje statično varne in preverljive spremembe. Če kontravariantne spremembe v tipih parametrov nimajo večje praktične veljave, pa to ne velja za kovarianco v tipih vračanja. Ta je s pragmatičnega stališča izjemnega pomena, kateri se izkaže predvsem v jezikih, ki teh sprememb ne omogočajo. Tipiziranje smo razširili na še bolj dinamično obliko s tipom Self. Tip Self se je izkazal za nepogrešljivega pri nekaterih povsem praktičnih problemih programiranja. Implementirali smo ga predvsem zato, ker ponuja fleksibilnosti dinamičnega tipiziranja, katerih v statično tipiziranih jezikih ne najdemo prav pogosto in je hkrati statično varen. Tip Self dodatno bogati izrazno moč sistema tipov.

Dedovanja smo se lotili s konceptualnega vidika in ga nato implementirali v samem jeziku. Pri tem smo poseben pomen dali pravilom redefiniranja metod, katera smo konstruirali preprosto in nazorno. Model dedovanja smo poenostavili do te mere, da imamo možnost večkratnega dedovanja v sklopu variantnega sistema tipov. Pravila smo minimizirali z razlogom, da ohranimo model dedovanja čist. Primarno vodilo, ki nam je služilo za osnovo dedovanja, je bila izraznost sistema tipov in preprostost pravil. Enkratno dedovanje smo posplošili v večkratno. Implementirali smo dedovanje z replikacijo, kar pomeni, da se objekt osnovnega razreda lahko ustvari večkrat. Tako replikacija kot deljenje imata svoje prednosti. Večkratno dedovanje je sicer doprineslo nekaj dodatnega dela v implementaciji, vendar je koncept tako pomemben, da ga nismo mogli izpustiti.

Predstavitev jezikovnih konstruktov, kot so zanke in stavki z objekti, ima svoje prednosti. Omogočena je enaka manipulacija kot pri vrednostih primitivnih tipov. To pomeni precej večjo izrazno moč jezika, saj lahko s kompleksnimi konstrukti počnemo vse, kar bi počeli z najprimitivnejšimi vrednostmi. S podporo meta arhitekture jezika bomo lahko sestavljali skoraj poljubne, uporabniško definirane konstrukte, ki v jeziku drugače niso implementirani. Čistega referenčnega modela nismo vpeljali samo v uporabiške tipe, temveč tudi v vgrajene konstrukte, kar pomeni, da lahko v času izvajanja vršimo manipulacijo teh konstruktov na trivialen način. Objektizacija konstruktov pa ne pomeni samo čiste predstavitve. Možnosti, ki se ponujajo, so zelo široke.

Zakasnjeno izvajanje, prenos preko parametrov, deljenje po celotnem izvajalnem okolju in še kaj bi se našlo.
Cilj našega dela ni bil samo v načrtovanju jezika, temveč tudi v možnosti praktične aplikacije. Da bi to dosegli, je bilo potrebno načrtovati tudi virtualno arhitekturo, ki je sposobna izvajati izvršni kod prevedenih programov. S tem namenom smo implementirali virtualno arhitekturo, ki izvaja razrede jezika Z_0. Arhitekturo smo zasnovali modularno. Koncipirali smo jo kot skupek razrednega nalagalnika, interpreterja virtualnih instrukcij in upravljalca pomnilnika. Ker je razredni nalagalnik arhitekture načrtovan abstraktno, ga lahko razširimo s funkcionalnostmi, ki omogočajo nalaganje razredov z različnih medijev. Zaradi čimvečje učinkovitosti smo metode razredov primitivnih tipov implementirali neposredno v virtualnem stroju. Metode nad objekti primitivnih tipov se prevajajo direktno v virtualne instrukcije, s čimer je dosežena precejšnja pohitritev v primerjavi s klasično invokacijo.

Enote virtualnega procesorja smo razdelili na enote za delo z vgrajenimi tipi in sistemsko enoto, ki je zadolžena za izvajanje skupnih instrukcij. Med te uvrščamo upravljanje z okvirji, instanciranje razredov ter instrukcije, ki so povezane z invokacijo metod.
Ker smo zahtevali transparenco med objekti primitivnih in kompleksnih tipov, smo zunanji vmesnik obojih implementirali poenoteno. Na ta način lahko posamezne instrukcije operirajo identično nad objekti primitivnih ali kompleksnih tipov. Zagotovo je bila predstavitev objekta ena izmed pomembnejših odločitev. Vse stvari, ki se vežejo na razred, delimo. Vse ostalo je treba vključiti v objekt. Tukaj imamo v mislih predvsem tabelo metod, ki je zaradi dinamičnih mehanizmov jezika, vezana na instanco in ne na razred. S tem smo povečali velikost objekta, vendar ohranili dinamičnost spreminjanja metod objekta. Predstavitev objekta je bila močno pogojena tudi z modelom dedovanja. Ker imamo večkratno dedovanje, referenciranje objekta v splošnem ni več enoumno. Indeksi sorodnih metod v nadrazredih se vedno ne ujemajo, zato potrebujemo originalne virtualne tabele nadrazredov. Ker lahko isti objekt referenciramo skozi različne vmesnike, je bilo treba vključiti deskriptorje objekta, ki se lahko za posamezen objekt spreminjajo.

Večji del virtualnega stroja zajema interpreter. Tudi tega smo načrtovali abstraktno z razlogom, da ga lahko v prihodnosti razširimo ali ga implementiramo na drugačen način ali za drugo ciljno arhitekturo. Implementacija interpreterja se razlikuje od klasičnega procesorjevega cikla prevzemanja in izvrševanja instrukcij. Medtem ko lahko procesor na podlagi operacijske kode prevzame operande v nekaj nanosekundah, je v primeru interpreterja potrebno te brati iz glavnega pomnilnika, kar znatno upočasni izvajanje. Dobra stran interpreterja virtualnih instrukcij je ta, da je še vedno mnogo bolj učinkovit, kot če bi program interpretirali na nivoju jezika.

Stvari, ki niso implementirane in morda bodo v prihodnosti, vključujejo meta arhitekturo, parametrizirane tipe in bolj abstraktno tipiziranje. Meta arhitektura bo omogočala povsem nov spekter možnosti, med katere štejemo dinamično spreminjanje in sestavljanje programov v času izvajanja in refleksijske mehanizme. Refleksija se je izkazala za praktično zelo uporaben jezikovni mehanizem, saj omogoča poizvedovanje o dejanskem stanju objektov. S tem je dosežena precej višja stopnja dinamike, kot v jezikih, kjer se prilagajanje lahko vrši samo v času prevajanja. Jezik smo načrtovali tako, da se bo meta informacije dalo vključiti na preprost način. To se odraža predvsem v strukturi objekta, ki že ima predvideno referenco na meta strukturo. Obstoječih mehanizmov tako ne bo potrebno spreminjati. Kar zadeva sintaktični nivo jezika, bo verjetno potrebno dodati kakšen konstrukt, s katerim bomo manipulirali meta informacije. Z gotovostjo lahko trdimo, da ne bo večjih težav pri implementaciji refleksije brez spreminjanja. Zaradi učinkovitosti bodo refleksijski mehanizmi podprti neposredno v virtualnem stroju. Težave varnostne narave se lahko pojavijo pri refleksiji, ki omogoča spreminjanje, vendar se tudi tukaj najdejo rešitve.

Parametrizacija tipov, z omejitvami ali brez, je dobrodošla, saj ponuja bistveno bolj učinkovito prevajanje generičnih tipov. Generične abstrakcije so osnovni razlog za implementacijo parametriziranih tipov. Generiki omogočajo definicije funkcionalnosti, ki operirajo nad celimi družinami tipov. Jezik smo načrtovali tako, da vključitev parametriziranega tipiziranja ne bi smela predstavljati večjih težav. Parametrični tipi bodo botrovali spremembam na sintaktičnem nivoju jezika. Te spremembe bo potrebno

vpeljati tako, da ne bodo v konfliktu z obstoječimi konstrukti. Ker parametrizacija zadeva predvsem tipe, bo potrebno sistem tipov ponovno preučiti. Ker je jezik Z_0 čisti objektno usmerjen jezik, bo parametrizacija implementirana na nivoju razredov. Na parametriziran razred lahko gledamo kot na funkcijo, ki prejme tip in vrne konkreten razred za ta tip. Čist referenčni model jezika lajša implementacijo parametričnih tipov s stališča upravljanja s pomnilnikom. Vse vrednosti so reference, kar pomeni, da imajo enako velikost. Pomnilniški prostor lahko torej predvidimo vnaprej. Zahteva parametrizacije, kateri želimo zadostiti, je tudi kompatibilnost binarnega nivoja. Šablone v jeziku C++ niso dobro zasnovane, saj ne omogočajo, da bi se parametrizacija ohranila v binarni obliki. Potrebujemo izvorni kod generičnih abstrakcij, kar pa ni vedno ugodno niti možno.
Kar zadeva variante parametričnih sistemov tipov, je pametno imeti klasično parametrizacijo in tisto z omejitvami. Omejevanje generičnih tipov ponuja dodatne prednosti, saj lahko neko družino tipov specificiramo natančneje in vnaprej določimo nabor sporočil, ki jih bodo objekti dejanskega tipa razumeli. Toda ker osnovni polimorfizem z omejitvami ne zajema vseh želenih omejitev nad tipi in se je izkazal za prešibkega, je dobro omejevanje vključiti tudi nad konstrukti abstrakcij tipov. Z_0 bo v kratkem imel konstrukt za definicijo čiste abstrakcije tipa. Omejitve polimorfnega tipa bodo takrat možne tudi nad tem konstruktom.

Upravljanje s pomnilnikom je realizirano samo delno, saj pravega alokacijskega podsistema nismo implementirali. Prostor za objekte se alocira z vnaprej pripravljene kopice. Okvirji se alocirajo v pomnilniškem prostoru, v katerem je realiziran tudi sklad. Pravo upravljanje s pomnilnikom je zahtevno vendar nujno opravilo, če želimo, da bo jezik uspešen tudi v konkretnih aplikacijah. Trenutna alokacija pomnilnika je realizirana tako, da bomo upravljalca pomnilnika lahko vključili transparentno, brez posegov v obstoječe mehanizme virtualnega stroja. Katero strategijo čiščenja bomo uporabili v prihodnosti, ni mogoče natančno določiti. Vsaka ima svoje prednosti in slabosti. Ne glede na to, avtomatsko čiščenje doprinaša veliko k robustnosti in učinkovitosti izvajalnega okolja arhitekture. Ker je jezik Z_0 čisti objektno usmerjen programski jezik, bo avtomatsko čiščenje tesno povezano z objektnim modelom. Edina pot do objekta je

referenca in objekti vsebujejo druge reference. Čiščenje bo torej potekalo rekurzivno. Vsi objekti, ki so še v uporabi, so dostopni iz okvirja. Vsi preostali se lahko odstranijo iz pomnilnika in na njihov prostor dajo drugi.

Ostajajo še operatorji, katerih redefiniranje bi dodatno razširilo izrazno moč jezika. Operatorji niso pravzaprav nič drugega kot metode, katerih ime je pač operator. Tako lahko govorimo o metodi '+' in ne o operatorju za seštevanje. Operatorje bi v splošnem razdelili na prefiksne, postfiksne in pretvorbene. V primeru, da operatorji v prihodnosti res postanejo del jezika, bomo uporabili takšno delitev. V okviru operatorjev se postavlja tudi vprašanje o operatorju invokacije. Kakšne bi bile posledice pri možnosti redefiniranja tega operatorja, pa tudi kaj in koliko bi s tem pridobili na izraznosti jezika.

Omenimo še metodne tipe in metode višjega reda, katerih implementacija je skoraj trivialna z ozirom na zdanjo predstavitev metod. Metode so prvorazredne vrednosti, kar pomeni, da lahko z njimi manipuliramo enako kot s primitivnimi vrednostmi. Metode ali funkcije višjega reda so dobro znane v funkcijskih programskih jezikih, ki prav zaradi njih omogočajo programiranje na bistveno višjem abstraktnem nivoju. Nič ni narobe, če koncept v domeni funkcijskih jezikov prenesemo v okvire objektno usmerjenega jezika imperativne narave. Jezik tako postane izrazno močnejši in omogoča implementacijo nekaterih stvari, ki sicer ne bi bile mogoče ali pa vsaj zelo težavne. Metode višjega reda so časovno in prostorsko zahtevnejše od tistih prvega reda, saj morajo biti predstavljene kot zaprtja z lastnim okoljem referenc. Z metodami višjega reda bi lahko zelo lepo realizirali osnovne koncepte aspektno usmerjenega programiranja [51, 52, 63, 107]. Seveda pa je v ta namen dobro imeti tudi določeno stopnjo refleksije.

Kako bi se jezik obnesel v realnih produkcijskih okoljih, je težko napovedati. Jezik je bil zasnovan kot robusten, splošno namenski jezik, vendar bi bilo za industrijsko rabo potrebno implementirati še nekaj stvari. Predvsem aplikacijski vmesnik, ki je pri konkretnem reševanju problemov nepogrešljiv. Toda ne pozabimo, da je bil jezik načrtovan eksperimentalno.

Literatura

[1] The Open Closed Principle
`http://www.eventhelix.com/realtimemantra/open_closed_principle.htm`.

[2] Martín Abadi. Baby modula-3 and a theory of objects. *Journal of Functional Programming*, 4(2):249–283, 1994.

[3] Martín Abadi and Luca Cardelli. A theory of primitive objects: Second-order systems. In *In Proceedings of the European Symposium on Programming. Lecture Notes in Computer Science*, number 788, pages 1–25. Springer-Verlag, 1994.

[4] Martín Abadi and Luca Cardelli. An imperative object calculus: Basic typing and soundness. In *SIPL '95 - Proc. Second ACM SIGPLAN Workshop on State in Programming Languages*. Technical Report UIUCDCS-R-95-1900, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995.

[5] Martín Abadi, Luca Cardelli, and Pierre-Louis Curien. Formal parametric polymorphism. In *Conf. Record 20th Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL'93, Charleston, SC, USA, 10–13 Jan 1993*, pages 157–170. ACM Press, New York, 1993.

[6] Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An interpretation of objects and object types. In *ACM Symposium on Principles of Programming Languages (POPL), St. Petersburg Beach, Florida*, pages 396–409, 1996.

[7] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 49–65, Atlanta, GA, 1997.

[8] Ole Agesen, Jens Palsberg, and Michael Schwartzbach. Analysis of objects with dynamic and multiple inheritance, 1993.

[9] Eric Allen, Jonathan Bannet, and Robert Cartwright. A first-class approach to genericity. *ACM SIGPLAN Notices*, 38(11):96–114, 2003.

[10] John E. Hopcroft Allen J. Korenjak. Simple deterministic languages. In *Proceedings of the 7th Symposium on Switching and Automata Theory*, IEEE, pages 36–46, 1966.

[11] Davide Ancona and Elena Zucca. An algebra of mixin modules. In *Workshop on Algebraic Development Techniques*, pages 92–106, 1997.

[12] Davide Ancona and Elena Zucca. A theory of mixin modules: Basic and derived operators. *Mathematical Structures in Computer Science*, 8(4):401–446, 1998.

[13] Davide Ancona and Elena Zucca. A theory of mixin modules: Algebraic laws and reduction semantics. Technical Report ISI-TR-99-05, Dipartimento di Informatica e Scienze dell'Informazione, Universita di Genova, 1999.

[14] Andrew W. Appel. *Modern compiler implementation in Java.* Cambridge Universisty Press, 1997.

[15] Gerald Baumgartner and Vincent F. Russo. Signatures: A language extension for improving type abstraction and subtype polymorphism in C++. *Software–Practice & Experience*, 25(8):863–889, Aug 1995.

[16] Gerald Baumgartner and Vincent F. Russo. Implementing signatures for C++. *ACM Transactions on Programming Languages and Systems*, 19(1):153–187, January 1997.

[17] Lorenzo Bettini, Michele Loreti, and Betti Venneri. On multiple inheritance in java, 2002.

[18] Robert Biddle, Angela Martin, and James Noble. No name: just notes on software reuse. *ACM SIGPLAN Notices*, 38(12):76–96, 2003.

[19] Gilad Bracha and William Cook. Mixin-based inheritance. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming*, pages 303–311, Ottawa, Canada, 1990.

[20] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998.

[21] Soren Brandt and Ole Lehrmann Madsen. Object-oriented distributed programming in BETA. In Rachid Guerraoui, Oscar Nierstrasz, and Michel Riveill, editors, *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, volume 791, pages 185–212. Springer-Verlag, 1994.

[22] Kim B. Bruce. *Foundations of Object-Oriented Languages, Types and Semantics*. The MIT Press, Cambridge, Massachusetts, 2002.

[23] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. *Lecture Notes in Computer Science*, 952:27–51, 1995.

[24] Carlos Camarao and Lucília Figueiredo. Class types, 1999.

[25] Peter Canning, William Cook, Walter Hill, John Mitchell, and Walter Olthoff. F-bounded quantification for object-oriented programming. In *In Proceedings of Functional Programming and Computer Architecture*, pages 273–280, 1989.

[26] Luca Cardelli. A polymorphic lambda calculus with type:type. Technical Report 10, Palo Alto, California, 1986.

[27] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, February/March 1988.

[28] Luca Cardelli and John C. Mitchell. Operations on records. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 295–350. The MIT Press, Cambridge, MA, 1994.

[29] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 24, pages 49–70, New York, NY, 1989.

[30] Jian Chen. Class types as sets of classes in object-oriented formal specification languages, 1995.

[31] Shigeru Chiba. A metaobject protocol for c++. *OOPSLA'95*, 1995.

[32] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 24, pages 433–444, New York, NY, 1989.

[33] Karl Crary, Robert Harper, and Sidd Puri. What is a recursive module? In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 50–63, 1999.

[34] Rowan Davies and Frank Pfenning. Practical refinement-type checking, 1997.

[35] Drew Dean. The security of static typing with dynamic linking. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security*, Zürich, Switzerland, 1997.

[36] Karel Driesen. Software and hardware techniques for efficient polymorphic calls. Technical Report TRCS99-24, 15, 1999.

[37] Sophia Drossopoulou and Susan Eisenbach. Java is type safe — probably. *Lecture Notes in Computer Science*, 1241:389–429, 1997.

[38] Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is the java type system sound? *Theory and Practice of Object Systems*, 5(1):3–24, 1999.

[39] Sophia Drossopoulou, Tanya Valkevych, and Susan Eisenbach. Java type soundness revisited, 1997.

[40] Catherine Dubois and Valerie Menissier-morain. Certification of a type inference tool for ML: Damas-milner within coq. *Journal of Automated Reasoning*, 23(3-4):319–346, 1999.

[41] Dominic Duggan and Constantinos Sourelis. Mixin modules. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '96)*, volume 31(6), pages 262–273, 1996.

[42] Anton Eliens. *Principles of Object-Oriented Software Development*. Addison-Wesley Publishing Company, 1994.

[43] R emi Forax, Etienne Duris, and Gilles Roussel. Java multi-method framework, 2000. In International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'00), November 2000.

[44] Erik Ernst. Dynamic inheritance in a statically typed language. *Nordic Journal of Computing*, 6(1):72–92, 1999.

[45] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 171–183, New York, NY, 1998.

[46] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. *ACM SIGPLAN Notices*, 39(1):111–122, 2004.

[47] Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. A comparative study of language support for generic programming. *ACM SIGPLAN Notices*, 38(11):115–134, 2003.

[48] Gilad Bracha, Martin Odersky, David Stoutamire and Philip Wadler. GJ: Extending the java programming language with type parameters. 1998.

[49] Adele Goldberg and David Robson. *Smalltalk-80 The Language and its Implementation*. Addison-Wesley Publishing Company, 1983.

[50] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.

[51] Anurag Mendhekar Chris Maeda Cristina Videira Lopes Jean-Marc Loingtier John Irwin Gregor Kiczales, John Lamping. Aspect-oriented programming. *Proceedings European Conference on Object-Oriented Programming*, 1241:220–242, 1997.

[52] Jim Hugunin Mik Kersten Jeffrey Palm William G. Griswold Gregor Kiczales, Erik Hilsdale. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[53] Tom Hirschowitz and Xavier Leroy. Mixin modules in a call-by-value setting. In *European Symposium on Programming*, pages 6–20, 2002.

[54] Intel. *IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*. Intel Corporation, 2004.

[55] Intel. *The IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*. Intel Corporation, 2004.

[56] Paul G. Sorenson Jean-Paul Tremblay. *The theory and practice of compiler writing*. McGraw-Hill Book Company, 1985.

[57] Doug Brown John R. Levine, Tony Mason. *lex & yacc*. O'Reilly & Associates, Inc., 1991.

[58] Sam Kamin. Routine run-time code generation. *ACM SIGPLAN Notices*, 38(12):44–56, 2003.

[59] Andrew Kennedy and Don Syme. Design and implementation of generics for the .net common language runtime. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, Snowbird, Utah, 2001.

[60] Bjoern Kirkerud. *Object-Oriented Programming with SIMULA*. Addison-Wesley Publishing Co., 1989.

[61] Guenter Kniesel. Implementation of dynamic delegation in strongly typed inheritance-based systems. Technical Report IAI-TR-94-3, 1994.

[62] Seshu Kumar. When and what to compile/optimize in a virtual machine? *ACM SIGPLAN Notices*, 39(3):38–45, 2004.

[63] Donal Lafferty and Vinny Cahill. Language-independent aspect-oriented programming. *ACM SIGPLAN Notices*, 38(11):1–12, 2003.

[64] Wilf R. LaLonde and John R. Pugh. *Inside Smalltalk Volume I.* Prentice-Hall Internation, Inc., 1990.

[65] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Object Group, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[66] Cristina Videira Lopes, Paul Dourish, David H. Lorenz, and Karl Lieberherr. Beyond aop: toward naturalistic programming. *ACM SIGPLAN Notices*, 38(12):34–43, 2003.

[67] Robin Milner, Mads Tofte , Robert Harper , David MacQueen. *The Definition of Standard ML - Revised.* MIT Press, 1997.

[68] Ole Lehrmann Madsen and Birger Moeller-Pedersen. Part objects and their location. Technical report, Aarhus, 1992.

[69] Hidehiko Masuhara and Akinori Yonezawa. A portable approach to dynamic optimization in run-time specialization. *Journal of New Generation Computing*, 20(1):101–124, November 2001.

[70] Hidehiko Masuhara and Akinori Yonezawa. Run-time bytecode specialization. *Lecture Notes in Computer Science*, 2053:138–145, 2001.

[71] Marjan Mernik, Mitja Lenič, Enis Avdičauševič, and Viljem Žumer. A reusable object-oriented approach to formal specications of programming languages, 1998.

[72] Marjan Mernik, Mitja Lenič, Enis Avdičauševič, and Viljem Žumer. The template and multiple inheritance approach into attribute grammars. In *International Conference on Computer Languages*, pages 102–110, 1998.

[73] Marjan Mernik, Mitja Lenič, Enis Avdičauševič, and Viljem Žumer. Multiple Attribute Grammar Inheritance. In D. Parigot and M. Mernik, editors, *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, pages 57–76, Amsterdam, The Netherlands, 1999.

[74] Marjan Mernik, Mitja Lenič, Enis Avdičauševič, and Viljem Žumer. Compiler/interpreter generator system LISA. In *HICSS*, 2000.

[75] Bertrand Meyer. *Object-oriented software construction*. Prentice Hall, 1988.

[76] Jon Meyer and Troy Downing. *Java Virtual Machine*. O'Reilly, 1997.

[77] Stephan A. Missura. Theories = signatures + propositions used as types. In *AISMC*, pages 144–155, 1994.

[78] Stephan Murer, Jerome A. Feldman, Chu-Cheow Lim, and Martina-Maria Seidel. pSather: Layered extensions to an object-oriented language for efficient parallel computation. Technical Report TR-93-028, Berkeley, CA, June 1993.

[79] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 146–159. ACM Press, New York (NY), USA, 1997.

[80] Martin Odersky and Philip Wadler. Leftover curry and reheated pizza: How functional programming nourishes software reuse. In *Fifth International Conference on Software Reuse*, Vancouver, BC, 1998. IEEE.

[81] Birger Moller-Pedersen Ole Lehrmann Madsen and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. ACM Press, 1993.

[82] Gordon Plotkin and Martín Abadi. A logic for parametric polymorphism. In M. Bezem and J. F. Groote, editors, *International Conference on Typed Lambda Calculi and Applications*, number 664, pages 361–375, Utrecht, The Netherlands, 1993.

[83] Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek. 'C and tcc: a language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21(2):324–369, 1999.

[84] Massimiliano A. Poletto. *Language and Compiler Support for Dynamic Code Generation*. PhD thesis, Massachusetts Institute of Technology, September 1999.

[85] Rob Pooley. *An Introduction to Programming in SIMULA*. Blackwell Scientific Publications, Oxford, 1987.

[86] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann Publishers, University of Rochester, 2000.

[87] Daniel Simon and Andy Walter. An implementation of the programming language dml in java, 2000.

[88] Anthony J. H. Simons. Let's agree on the meaning of 'class'. Department of Computer Science, University of Sheffield, 1996.

[89] Yannis Smaragdakis and Don Batory. Implementing layered designs with mixin layers. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 550–570. Springer-Verlag LNCS 1445, 1998.

[90] Yannis Smaragdakis and Don Batory. Mixin-based programming in C++. *Lecture Notes in Computer Science*, 2177:163–178, 2001.

[91] Yannis Smaragdakis and Don S. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *Software Engineering and Methodology*, 11(2):215–255, 2002.

[92] Alan Snyder. Commonobjects: an overview. In *Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming*, volume 21, pages 19–28. ACM Press New York, NY, USA, 1986.

[93] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 21, pages 38–45, New York, NY, 1986.

[94] Guy Steele. Common lisp: The language. *Digital Equipment Corporation*, 1984.

[95] Bjarne Stroustrup. Multiple inheritance for C++. In *Proceedings of the Spring 1987 European Unix Users Group Conference*, Helsinki, 1987.

[96] Clemens Szypersky, Stephen Omohundro, and Stephan Murer. Engineering a programming language: The type and class system of sather. Technical Report TR-93-064, Berkeley, CA, 1993.

[97] Antero Taivalsaari. Kevo – a prototype-based object-oriented language based on concatenation and module operartions. Technical Report DCS–197–1R, June, 1992.

[98] Antero Taivalsaari. Object-oriented programming with models. In *Journal of Object-Oriented Programming*, volume 6, pages 25–32, 1993.

[99] Antero Taivalsaari. Classes vs. prototypes - some philosophical and historical observations, 1996.

[100] Antero Taivalsaari. On the notion of inheritance. In *ACM Computing Surveys*, volume 28, pages 439–477. ACM Press, 1996.

[101] David Ungar and Randall B. Smith. Self: The power of simplicity. *OOPSLA'87*, 4(8):227–242, 1987.

[102] John Viega, Bill Tutt, and Reimer Behrends. Automated delegation is a viable alternative to multiple inheritance in class based languages. Technical Report CS-98-03, 2, 1998.

[103] Marjan Mernik Viljem Žumer. *Programski jeziki, Semantika in vzorci*. Fakulteta za elektrotehniko, računalništvo in informatiko, 1996.

[104] Marjan Mernik Viljem Žumer. *Programski jeziki, sintaksa in koncepti*. Fakulteta za elektrotehniko, računalništvo in informatiko, 2001.

[105] Mirko Viroli and Antonio Natali. Parametric polymorphism in Java: an approach to translation based on reflective features. *ACM SIGPLAN Notices*, 35(10):146–165, 2000.

[106] Jan Vitek and R. Nigel Horspool. Taming message passing: Efficient method look-up for dynamically typed languages. *Lecture Notes in Computer Science*, 821:432+, 1994.

[107] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. *ACM SIGPLAN Notices*, 38(9):127–139, 2003.

[108] David A. Watt. *Programming Langue Syntax and Semantics*. Prentice Hall International, Ltd., 1991.

[109] Luca Cardelli, Peter Wegner. On understanding types, data abstraction,and polymorphism. In *Computing Surveys*, volume 17, pages 471–522. December 1985.

[110] Andrew Wright. Practical soft typing. Technical Report TR94-236, 8, 1998.

[111] Dachuan Yu, Andrew Kennedy, and Don Syme. Formalization of generics for the .net common language runtime. *ACM SIGPLAN Notices*, 39(1):39–51, 2004.

Bibliografija

1.08 Objavljeni znanstveni prispevek na konferenci

1. GREINER, Sašo, BOŠKOVIĆ, Borko, BREST, Janez, ŽUMER, Viljem. Security issues in information systems based on open-source technologies. V: ZAJC, Baldomir (ur.), TKALČIČ, Marko (ur.). The IEEE Region 8 EUROCON 2003 : computer as a tool : 22-24. September 2003, Faculty of Electrical Engineering, University of Ljubljana, Ljubljana, Slovenia : proceedings. Piscataway: IEEE, cop. 2003, vol. B, str. 449-453. [COBISS.SI-ID 8241174]

2. GREINER, Sašo, REBERNAK, Damijan, BOŠKOVIĆ, Borko, BREST, Janez, ŽUMER, Viljem. Web information system based on open source technologies. V: BUDIN, Leo (ur.), LUŽAR - STIFFLER, Vesna (ur.), BEKIĆ, Zoran (ur.), HLJUZ DOBRIĆ, Vesna (ur.). 25th International Conference on Information Technology Interfaces, June 16-19, 2003, Cavtat, Croatia. ITI 2003 : proceedings of the 25th International Conference on Information Technology Interfaces, June 16-19, 2003, Cavtat, Croatia. Zagreb: University of Zagreb, SRCE University Computing Centre, 2003, str. 137-142. [COBISS.SI-ID 7991318]

3. BREST, Janez, ROŠKAR, Silvo, BOŠKOVIĆ, Borko, REBERNAK, Damijan, GREINER, Sašo, KRUŠEC, Robert, ŽUMER, Viljem. Informacijski sistem v proizvodnem procesu. V: ZAJC, Baldomir (ur.). Zbornik dvanajste mednarodne Elektrotehniške in računalniške konference ERK 2003, 25. - 26. september 2003, Ljubljana, Slovenija, (Zbornik ... Elektrotehniške in računalniške konference ERK ..., 1581-4572). Ljubljana: IEEE Region 8, Slovenska sekcija IEEE, 2003, str. 365-368. [COBISS.SI-ID 8229910]

4. GREINER, Sašo, TUTEK, Simon, BREST, Janez, ŽUMER, Viljem. Razvojna paradigma MVC v spletnih aplikacijah. V: ZAJC, Baldomir (ur.). Zbornik dvanajste mednarodne Elektrotehniške in računalniške konference ERK 2003, 25. - 26. september 2003, Ljubljana, Slovenija, (Zbornik ... Elektrotehniške in računalniške konference ERK ..., 1581-4572). Ljubljana: IEEE Region 8, Slovenska sekcija IEEE, 2003, str. 393-396. [COBISS.SI-ID 8232982]

5. BOŠKOVIĆ, Borko, REBERNAK, Damijan, GREINER, Sašo, BREST, Janez, ŽUMER,

Viljem. Nadzorovanje virov računalniškega sistema v operacijskem sistemu Linux. V: ZAJC, Baldomir (ur.). Zbornik dvanajste mednarodne Elektrotehniške in računalniške konference ERK 2003, 25. - 26. september 2003, Ljubljana, Slovenija, (Zbornik ... Elektrotehniške in računalniške konference ERK ..., 1581-4572). Ljubljana: IEEE Region 8, Slovenska sekcija IEEE, 2003, str. 397-400. [COBISS.SI-ID 8233494]

6. GREINER, Sašo, BREST, Janez, ŽUMER, Viljem. Grozdna arhitektura za porazdeljeno izvajanje programskega bremena. V: ZAJC, Baldomir (ur.). Zbornik dvanajste mednarodne Elektrotehniške in računalniške konference ERK 2003, 25. - 26. september 2003, Ljubljana, Slovenija, (Zbornik ... Elektrotehniške in računalniške konference ERK ..., 1581-4572). Ljubljana: IEEE Region 8, Slovenska sekcija IEEE, 2003, str. 417-420. [COBISS.SI-ID 8235030]

7. GREINER, Sašo, TUTEK, Simon, BREST, Janez, ŽUMER, Viljem. Quick adaptation of Web-based information systems with aspect-oriented features. V: LUŽAR - STIFFLER, Vesna (ur.), HLJUZ DOBRIĆ, Vesna (ur.). ITI 2004 : proceedings of the 26th International Conference on Information Technology Interfaces, June 7-10, 2004, Cavtat, (Dubrovnik), Croatia, (IEEE Catalog, No. 04EX794). Zagreb: University of Zagreb, SRCE University Computing Centre, 2004, str. 145-150. [COBISS.SI-ID 8808214]

2.05 Drugo učno gradivo

8. ŽUMER, Viljem, MERNIK, Marjan, BREST, Janez, KRUŠEC, Robert, ČREPINŠEK, Matej, KOSAR, Tomaž, GREINER, Sašo, REBERNAK, Damijan, BOŠKOVIĆ, Borko, TUTEK, Simon. Operacijski sistem LINUX/UNIX : gradivo za nadaljevalni tečaj, (Računalniško opismenjevanje MŠZŠ). Maribor: Laboratorij za računalniške arhitekture in jezike in Center za programske jezike, 2003. 48 f. [COBISS.SI-ID 8160790]

9. ŽUMER, Viljem, MERNIK, Marjan, BREST, Janez, KRUŠEC, Robert, ČREPINŠEK, Matej, KOSAR, Tomaž, GREINER, Sašo, REBERNAK, Damijan, BOŠKOVIĆ, Borko. Operacijski sistem LINUX/UNIX : gradivo za začetni tečaj, (Računalniško opismenjevanje MŠZŠ). Maribor: Laboratorij za računalniške arhitekture in jezike in Center za programske jezike, 2003. 27 f. [COBISS.SI-ID 8160534]

2.11 Diplomsko delo

10. GREINER, Sašo. Arhitektura za objektno orientirane jezike : [diplomsko delo univerzitetnega študijskega programa], (Fakulteta za elektrotehniko, računalništvo in informatiko, Diplomska dela univerzitetnega študija). [Maribor]: [S. Greiner], 2002. IX, 88 f., ilustr. [COBISS.SI-ID 7595030]

2.25 Druge monografije in druga zaključena dela

11. ŽUMER, Viljem, BREST, Janez, MERNIK, Marjan, ČREPINŠEK, Matej, KRUŠEC, Robert, KOSAR, Tomaž, REBERNAK, Damijan, BOŠKOVIČ, Borko, TUTEK, Simon, GERLIČ, Goran, GREINER, Sašo. LINUX in Qt : tečaj za AMES, Brezovica pri Ljubljani. Maribor: Fakulteta za elektrotehniko, računalništvo in informatiko, Laboratorij za računalniške arhitekture in jezike, 2003. 1 mapa (loč. pag.). [COBISS.SI-ID 7801110]

Življenjepis

Rodil sem se v Mariboru dne 13.7.1978. Od 1985 do 1993 sem obiskoval osnovno šolo Borcev za severno mejo v Mariboru. V letih 1993-1997 sem obiskoval srednjo elektro računalniško šolo Maribor. Leta 1997 sem pričel s študijem na fakulteti za elektrotehniko, računalništvo in informatiko Univerze v Mariboru. Univerzitetni študij sem leta 2002 zaključil z uspešnim zagovorm diplome "Arhitektura za objektno orientirane jezike". Istega leta sem se zaposlil kot raziskovalec v laboratoriju za računalniške arhitekture in jezike. Leta 2003 sem prevzel delovno mesto asistenta s področja računalništva. Avgusta leta 2004 sem uspešno zagovarjal magistrsko delo z naslovom "Implementacija dinamičnih konceptov čistega statičnega objektno usmerjenega jezika". Trenutno sem aktiven na področju načrtovanja in implementacije objektno usmerjenih programskih jezikov, dizajna virtualnih računalniških arhitektur ter implementacije informacijskih sistemov.