Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Toni Kocjan

Vgradnja objektno usmerjenih gradnikov v programski jezik PINS

diplomsko delo

visokošolski strokovni študijski program
prve stopnje
računalništvo in informatika

Mentor: doc. dr. Bištjan Slivnik

Ljubljana, 2017

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Besedilo teme diplomskega dela študent prepiše iz študijskega informacijskega sistema, kamor ga je vnesel mentor. V nekaj stavkih bo opisal, kaj pričakuje od kandidatovega diplomskega dela. Kaj so cilji, kakšne metode uporabiti, morda bo zapisal tudi ključno literaturo.

Na tem mestu zapišite, komu se zahvaljujete za izdelavo diplomske naloge. Pazite, da ne boste koga pozabili. Utegnil vam bo zameriti. Temu se da izogniti tako, da celotno zahvalo izpustite.

Svoji dragi Alenčici.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
CA	classification accuracy	klasifikacijska točnost
DBMS	database management system	sistem za upravljanje podatkovnih baz
SVM	support vector machine	metoda podpornih vektorjev

Povzetek

Naslov: Vgradnja objektno usmerjenih gradnikov v programski jezik PINS

Avtor: Toni Kocjan

V diplomskem delu bom predstavil programski jezik Atheris, ki je nastal kot nadgradnja programskega jezika PINS. Prog. jezik PINS, oz. prevajalnik zanj, je bil zgrajen tekom semestra pri predmetu prevajalniki in navidezni stroji. Ker mi je bilo delo na prevajalniku izjemno zanimivo, sem se odločil, da ustvarim svoj programski jezik in sam določim pravila zanj.

V diplomskem delu na kratko predstavim prevajalnike in programske jezike, kaj sploh so in kaj je njihov namen. Opišem kakšne so sodobne prakse pri ravoju prevajalnikov, s kakšnimi probleme se prevajalnik sooča ter kako je zgrajen.

Podrobneje bom obrazložil nadgradnje jezika PINS, s kakšnimi problemi sem se tekom razvoja soočal ter kako sem jih reševal. Pokazal bom nekaj primerov programov v mojem jeziku ter primerjal z drugimi jeziki.

Ključne besede: prevajalnik, programski jezik, sintaksa, semantika, Java, Swift.

Abstract

Title: Diploma thesis sample

Author: Toni Kocjan

This sample document presents an approach to typesetting your BSc thesis using LaTeX. A proper abstract should contain around 100 words which makes this one way too short.

Keywords: compiler, programming language, syntax, semantics, Java, Swift.

Poglavje 1

Uvod

Razvoj prevajalnikov, ter s tem tudi programskih jezikov, je, po mojem mnenju, izjemno pomembna panoga v računalništvu. Programski jezik je medij, preko katerega komuniciramo z računalnikom. Prevajalniki razvijalcem omogočajo, da se med razvojem programske opreme ne rabijo osredotočati na nizkovojske detajle, ampak se lahko posvetijo reševanju praktičnih problemov. Naloga prevajalnika je, da pretvori človeku berljivo kodo v računalniku razumljivo zaporedje strojnih ukazov.

Dandanes lahko za razvoj programske opreme izbiramo med veliko programskih jezikov. Trenutno eni izmed najbolj popularnih so JavaScript, Java, Python in C++, popularnost pa dobivajo tudi novejši jeziki, kot so GoLang, Swift, Kotlin in podobni. [1]

S prevajalniki sem se začel ukvarjati pri predmetu Prevajalniki in Navidezni Stroji (PINS) v drugem letnik na Fakulteti za Računalništvo in Informatiko. Tekom diplomske naloge bom predstavil programski jezik Atheris ter prevajalnik zanj. Osredotočil se bom predvsem na vgradnjo objektno usmerjenih gradnikov v programski jezik.

Poglavje 2

Prevajalniki

2.1 Uvod v prevajalnike in programske jezike

Programski jezik je poseben jezik, ki se uporabljaja za razvoj programske opreme. Programski sistemi, ki poskrbijo, da se izvede pretvorba kode, napisane v programskem jeziku, v računalniku razumljivo obliko, se imenujejo *prevajalniki*.
Nekaj definicij:

1. **Računski model (angl. computational model):** zbirka vrednosti in računskih operacij
2. **Izračun (angl. computation)**: zaporodje operacij nad vrednostjo (ali več vrednosti), ki vrne nek rezultat
3. **Program:** specifikacija izračuna
4. **Programski jezik:** zapis (notacija) za pisanje programov

[2]

Program lahko predstavimo kot funkcijo, pri kateri je rezultat (angl. *output*) funkcija vhodnih parametrov (angl. *input*):

```
rezultat = program(vhodni parametri)
```

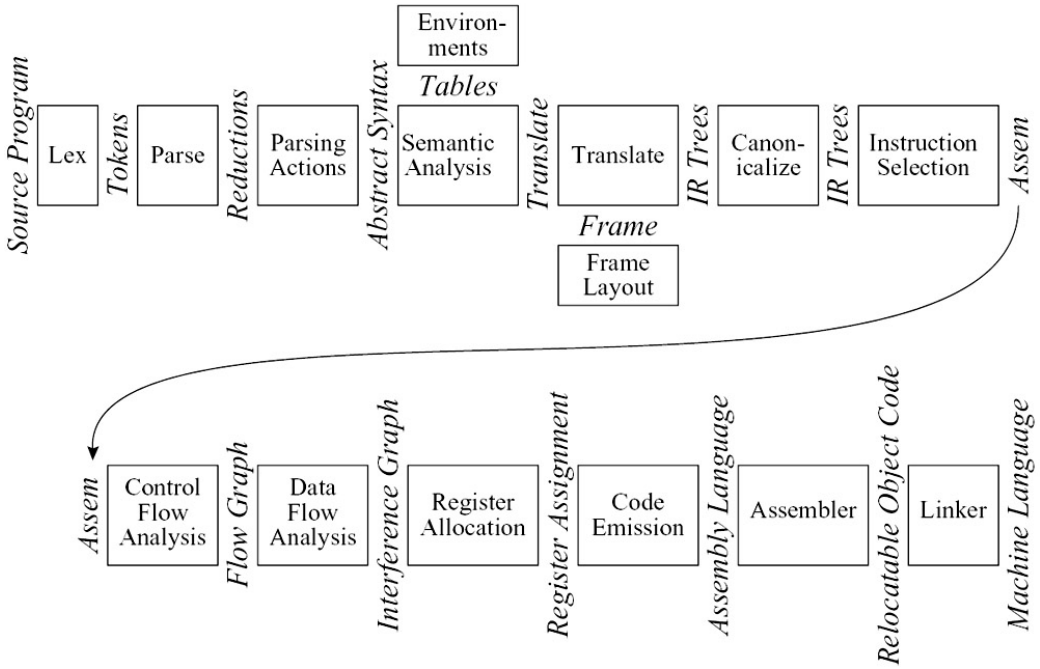
Iz drugega zornega kota si lahko program predstavljamo tudi kot model problemske domene, kjer je instanca izvedbe programa simulacija problema:

```
program = model problemske domene
izvedba programa = simulacija problema
```

[2]

2.2 Zgradba prevajalnika

Sodobni prevajalniki so pogosto organizirani v več posameznih faz, vsaka izmed njih pa operira na različnem nivoju abstrakcije jezika. [3]



Slika 2.1: Faze prevajalnika ter vmesniki, ki jih povezujejo med seboj.

Prevajalnik, da lahko program prevede iz ene oblike v drugo, mora program najprej analizirati, razumeti njegovo strukturo ter pomen, ter ga nato sestaviti nazaj v drugačno obliko.
Analizo programa običajno delimo v naslednje korake:

vrsta žetona	angl.	primeri
ime	identifier	x foo bar thisIsAnIdentifier
rezervirana beseda	keyword	while for if public override
operator	operator	, . && = ==
niz znakov	string	"this is a string"
znak	character	'a' 'x' '@'
celo št.	integer	10 125 082
decimalno št.	real	201.5 3.14 1.2e10

Tabela 2.1: Primeri žetonov v programskem jeziku Java

1. **Leksikalna analiza** (angl. *lexical analysis*)
2. **Sintaksna analiza** (angl. *syntax analysis*)
3. **Semantična analiza** (angl. *semantic analysis*)

[3]

2.2.1 Leksikalna analiza

Tako imenovani leksikalni analizator (modul, ki izvede leksikalno analizo) kot vhod prejme tok znakov (angl. *stream of characters*), kot izhod pa vrne tok v naprej definiranih žetonov (angl. *stream of tokens*). Žeton je običajno zgrajen iz imena, vrednosti (t.i. *lexeme*) ter lokacije v izvorni datoteki. [3]

Leksikalni žetoni:

Žeton (ali simbol) je zaporedje znakov, ki ga interpretiramo kot samostojno enoto v slovnici programskega jezika. [3]
Tabela 2.1 prikazuje nekaj vrst simbolov ter primere.

```
let x: Int
let y: Int
x * y
```

Listing 2.1: Primer programa v programskem jeziku Atheris

Rezultat:

```
[1:1-1:4]       LET:let
[1:5-1:6]       IDENTIFIER:x
[1:6-1:7]       COLON::
[1:8-1:11]      IDENTIFIER:Int
[1:11-1:12]     NEWLINE:\n
[2:1-2:4]       LET:let
[2:5-2:6]       IDENTIFIER:y
[2:6-2:7]       COLON::
[2:8-2:11]      IDENTIFIER:Int
[2:11-2:12]     NEWLINE:\n
[3:1-3:2]       IDENTIFIER:x
[3:3-3:4]       MUL:*
[3:5-3:6]       IDENTIFIER:y
EOF:$
```

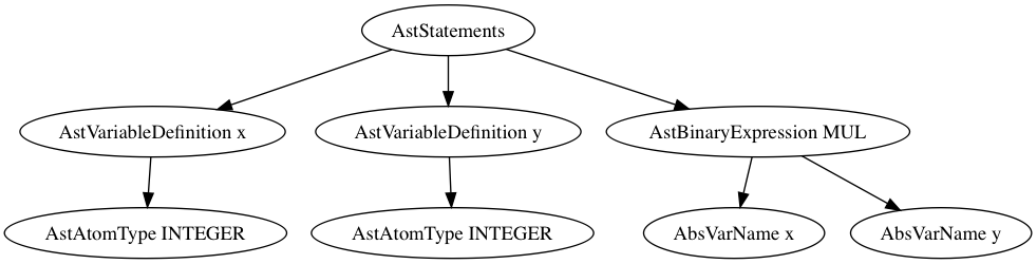
Listing 2.2: Rezultat leksikalne analize za program 2.1

2.2.2 Sintaksna analiza

Druga faza prevajanja je sintaksna analiza (angl. *syntax analysis* ali *parsing*). Naloga te faze je, da zagotovi, da je napisan program slovnično pravilen in v skladu s sintaksnimi pravili. Sintaksni analizator prejme kot vhod tok žetonov, ki ga zgenerira prejšnja faza, rezultat pa je abstraktno sintaksno drevo.

Abstraktno sintaksno drevo (AST) je drevesna podatkovna struktura, ki predstavlja slovnično strukturo programa. Vsako vozlišče drevesa pona-

zarja konstrukt v programski kodi.



Slika 2.2: Abstraktno sintaksno drevo za program 2.1.

Iz slike 2.2 lahko razberemo, da gre za dve definiciji spremenljivk in množenje.

Abstraktno sintaksno drevo je bistvenega pomena, saj nadaljne faze operirajo izključno nad njim.

2.2.3 Semantična analiza

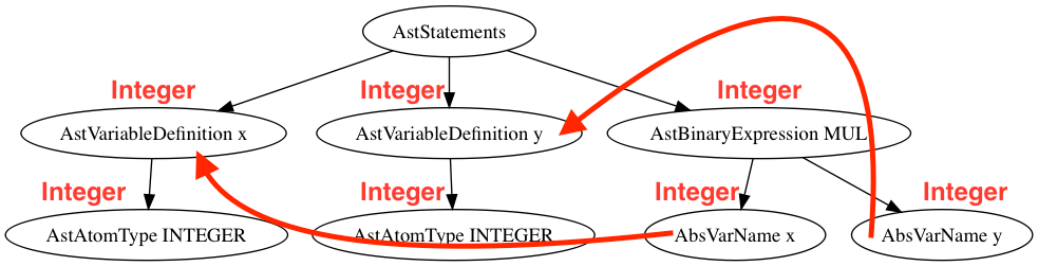
Semantična analiza poveže definicije spremenljivk z njihovimi uporabami ter preveri, ali so vsi izrazi pravilnih podatkovnih tipov. [3]

Običajno semantično analizo razdelimo na dve pod-fazi:

1. **Razreševanje imen:** zagotovi, da za vsako uporabo imena obstaja znotraj trenutnega območja vidnosti definicija z istim imenom, ter uporabo poveže z definicijo

2. **Preverjanje tipov:** vsakemu vozlišču v AST določi podatkovni tip, ter na podlagi postavljenih semantičnih pravil zagotovi, da so vsi izrazi pravilnih tipov

Pri implementaciji semantične analize nam pomaga *simbolna tabela*.



Slika 2.3: Rezultat semantične analize za program 2.1

Simbolna tabela

Simbolna tabela je podatkovna struktura, ki mapira imena v njihove definicije in podatkovne tipe. [3] Ker običajno programi vsebujejo več tisoč unikatnih definicij imen, mora podatkovna struktura omogočati učinkovito poizvedovanje. Iz slike 2.3 lahko razberemo, kaj se med izvajanjem semantične analize zgodi v ozadju: puščice predstavljajo povezave med definicijami in uporabami, evaluacija podatkovnih tipov pa je pri vseh vozliščih *Integer*, razen pri korenu, ki nima tipa oz. je tipa *Void*.

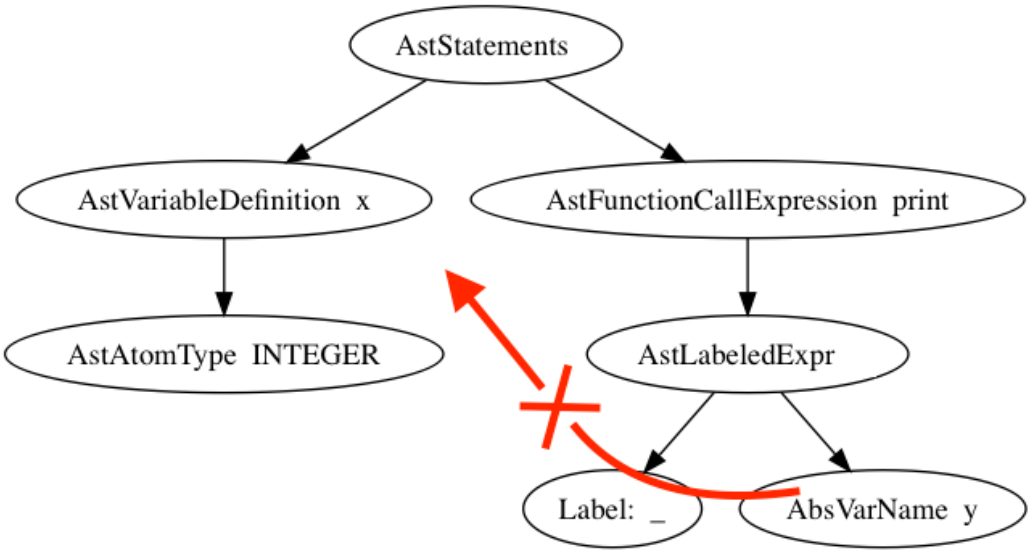
Kot sem omenil, semantična analiza zagotovi, da za vsako uporabo imena obstaja njena definicija, in da so podatkovni tipi pravilni. Sledita dva primera, kjer to ne drži:

```
let x: Int
print(y)
```

Listing 2.3: Primer programa, kjer spremenljivka y ni definirana

```
let x: Int
let s: String
x + s
```

Listing 2.4: Primer programa, kjer je napaka v podatkovnih tipih



Slika 2.4: Napaka v programu 2.3. Spremenljivka y ni definirana.

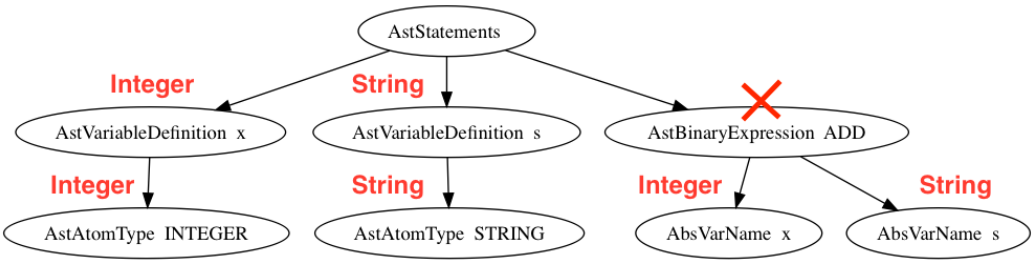
2.2.4 Klicni zapisi

V skoraj vsakem modernem programskem jeziku ima lahko funkcija *lokalne* spremenljivke, ki so kreirane ob vstopu v funkcijo. Hkrati lahko naenkrat obstaja več zapisov iste funkcije, zato je pomembno, da ima vsak zapis lastne instance lokalnih spremenljivk. [3]

V funkciji

```
let x: Int
let y: Int
x * y
```

se za vsak njen klic ustvari nova instanca x, ki jo inicializira klicalec funkcije. Ker je funkcija rekurzivna, živi v pomnilniku naenkrat veliko x-ev. Podobno se ob vsakem vstopu v jedro funkcije ustvari tudi nova instanca y. [3]



Slika 2.5: Napaka v programu 2.4. Seštevanje med podatkovnima tipoma *Integer* in *String* ni dovoljeno.

Sklad

Klicni zapisi funkcij se shranjujejo na sklad. Sklad je v pomnilniku predstavljen kot velika tabela s posebnim registrom imenovanim *stack pointer* oz. kazalec na sklad, ki kaže na konec sklada. Ob vsakem klicu funkcije se sklad poveča za velikost klicnega zapisa klicane funkcije. Podobno se ob vrnitvi funkcije zmanjša za enako vrednost. Prostor v klicnem zapisu namenjen hrambi vhodnih parametrov, lokalnih spremenljivk in ostalih registrov se imenuje *activation record*. [3]

Kazalec na klicni zapis

Predpostavimo da funkcija g kliče funkcijo f. Ob vstopu v f kazalec na sklad (SP) kaže na prvi vhodni argument funkciji f. Nov prostor na skladu je rezerviran tako, da se od SP odšteje velikost klicnega zapisa f. Tako SP sedaj kaže na konec sklada. Stara vrednost SP postane nova vrednost kazalca na klicni zapis (*angl. frame pointer*). V programskih jezikih, kjer je velikost klicnega zapisa za posamezno funkcijo konstantna, je vrednost kazalca na klicni (FP) zapis vedno izračunljiva, zato si je ni potrebno posebaj shranjevati na sklad. FP = SP + velikost sklada. [3]

Prenos parametrov

Standarden način klicanja funkcij je, da klicoča funkcija rezervira prostor na skladu za prenosih njenih izhodnih parametrov (oz. vhodnih parametrov

za klicano funkcijo). [3]

Prostor za njih se običajno nahaja pred SP. Klicana funkcija tako naslov njenega i-tega parametra izračuna z enačbo

```
argumentAddress(i) = FP + (4 * i)
```

Lokalne spremenljivke

Prevajalnik mora na skladu alocirati dovolj prostora za vse lokalne spremenljivke.

Static link

V jezikih, ki podpirajo gnezdenje funkcij, lahko gnezdene funkcije dostopajo do spremenljivk, ki se nahajajo v zunanjih funkcijah. Da lahko gnezdena funkcija dostopa do spremenljivk, ki niso na njenem klicnem zapisu, ji ob klicu poleg ostalih parametrov posredujemo FP funkcije, ki jo neposredno definira. Temu kazalcu rečemo *static link*.

```
func f() {
    var x: Int = 10
    func e() {
        var z: Int = 100
    }
    func g() {
        var y: Int = 20
        func h() {
            print(y, x)
        }
    }
}
```

Listing 2.5: Primer gnezdenih funkcij

Primer 2.5 vsebuje dve gnezdeni funkciji *g* in *h*. Funkcija h lahko dostopa do spremenljivk definiranih v *g* in *f*, ne pa tudi tistih v *e*.

2.2.5 Generiranje vmesne kode

Vmesna koda

Vmesna koda (*angl. intermidiate representation*) je abstraktna predstavitev strojnega jezika in predstavlja ukaze, brez da bi poznali arhitekturo ciljne naprave. Poleg tega je vmesna koda neodvisna od izvornega jezika. [3]

Vmesno predstavitveno drevo

Dobra predstavitev vmesne kode ima naslednje lastnosti:

1. Njegovo generiranje mora biti priročno
2. Generiranje strojne kode v dejansko strojno kodo mora biti priročno za vse ciljne arhitekture
3. Vsak gradnik mora imeti jasen pomen, da so lahko optimizacijske transformacije enostavno implementirane

[3]

Posamezni deli abstraktnega sintaksnega drevesa so lahko kompleksne stvari, na primer zanke, klici funkcij, itd., ki jih ne moremo neposredno mapirati v strojne ukaze. Zato morajo gradniki vmesne kode predstavljati le enostavne operacije, kot so npr. LOAD (preberi vrednost iz pomnilnika), STORE (shrani vrednost v pomnilnik), ADD (seštej dve vrednosti), itd. Tako lahko vsak posamezen del AST prevedemo v ravno pravo zsaporedje ukazov abstraktne vmesne kode. [3]

Poglavje 3

Programski jezik PINS

Programski jezik PINS je učni programski jezik, zato je tudi dokaj preprost. Prevajalnik zanj smo implementiral v sklopu domačih nalog pri predmetu Prevajalniki in navidezni stroji.

3.1 Leksikalna pravila

Programski jezik PINS podpira tri atomarne podatkovne tipe: *integer*, *logical*, *string*, za katere so rezervirane istoimenske besede. Celoštevilske konstante so poljubno predznačeno zaporedje števk, logične konstante so ali *true* ali *false*, znakovne konstante pa so definirane kot poljubno (lahko prazno) zaporedje znakov z ASCII kodami med vključno 32 in 126, ki je obdano z enojnima navednicama (ASCII koda 39); izjema je en sam enojni narekovaj, ki je podvojen.

Imena so definirana kot poljubno zaporedje črtk, številk in podčrtajev, ki se ne začne s številko in ni rezervirana beseda ali kakšna od prej naštetih konstant.

Belo besedilo (*angl. whitespace*) so presledki (ASCII 32), tabulatorji (ASCII 9) in znaka za konec vrstice (ASCII 10 in 13).

Komentarji se začnejo z '#' (ASCII 35) in se raztezajo do konca vrstice.

3.2 Sintaksna pravila

Celotna izvorna koda je sestavljena iz seznama definicij. Vsaka definicija je lahko:

1. Definicija tipa (oz. sklic na tip - *typealias*)
2. Definicija spremenljivke
3. Definicija funkcije

Kot sem že omenil, podpira PINS tri osnovne podatkovne tipe, vendar sintaksa omogoča tudi definicijo tabel (*angl. array*) s fiksno velikostjo.

Definicija funkcije je sestavljena iz imena, seznama parametrov, tipa ki ga funkcija vrača, ter *izraza* oz. jedra funkcije. Zanimivo pri PINSu je to, da izven funkcij ne moremo početi ničesar drugega, kot ustvarjati definicije (podobno kot pri Javi).

Izraz (*angl. expression*) je lahko:

1. Logični izraz
2. Primerjalni izraz
3. Seštevalni izraz
4. Multiplikativni izraz
5. Seštevalni izraz
6. Prefiksni izraz
7. Postfiksni izraz
8. Atomarni izraz

Atomarni izraz se še naprej deli in je lahko:

1. Logična konstanta

2. Celoštevilska konstanta
3. Znakovna konstanta
4. Ime
5. Klic funkcije
6. If stavek
7. If else stavek
8. While stavek
9. Zaporedje izrazov

Vsakemu izrazu lahko sledijo tudi definicije gnezdene znotraj zavitih oklepajev.

3.3 Semantična pravila

Območja vidnosti

Imena so vidna v celotnem območju vidnosti, ne glede na mesto definicije. Izraz *expression* { *WHERE definitions* } ustvari novo vgnezdeno območje vidnosti. To pomeni, da definicije znotraj zavitih oklepajev niso vidne navzven. Tudi definicija funkcije ustvari novo vgnezdeno območje vidnosti, ki se začne za imenom in se razteza do konca funkcije..

Tipiziranost

1. *integer*, *logical*, *string* opisujejo podatkovne tipe INTEGER, LOGICAL in STRING, zaporedoma
2. izraz

```
arr [ n ] type
```

opisuje podatkovni tip ARR(n, type), kjer je n celoštevilska konstanta

Deklaracije

1. Deklaracija tipa

$$typ \quad identifier \quad : \quad type$$

ustvari sklic na podatkovni tip *type* z imenom *identifier*

2. Deklaracija funkcije

$$fun \quad identifier(identifier_1 : type_1, ..., identifier_n : type_n) : type = expression$$

določa funkcijo, ki je tipa

$$type_1 \quad *, \quad ..., \quad * \quad type_n \rightarrow type$$

3. Deklaracija spremenljivke

$$var \quad identifier \quad : \quad type$$

določa spremeljivko tipa *type*

4. Deklaracija parametra

$$identifier \quad : \quad type$$

določa parameter tipa *type*

3.4 Spremembe in razširitve

Sintaksa

Sintaksa je v primerjavi s sintakso jezika PINS popolnoma spremenjena. Pravzaprav je sintaksa skoraj identična programskemu jeziku Swift.

Med drugim nova sintaksa omogoča definiranje kompleksnejših podatkovnih tipov z uporabo razredov, vmesnikov, terk in enumeracij.

Posamezni stavki so med seboj ločeni z novimi vrsticami (Python, Swift), vendar pa prevajalnik omogoča njihovo ločevanje tudi s podpičjem ';' (C++,

Java).

Komentarji se lahko začnejo bodisi z '#' in se raztezajo do konca vrstice (Python, PINS), bodisi z '/*' in končajo z '*/' (Java, C++).

Podprt je tudi *switch* kontrolni stavek s sintakso identične tej od Swifta.

Definicije funkcij ter klici funkcij so spremenjeni; parametri funkcije so sedaj sestavljeni iz labele ter imena parametra (tako kot v Swiftu). Pri klicu funkcije se uprabi labela parametra, znotraj jedra funkcije pa se uporablja ime.

Podatkovni tipi

Podprti atomarni podatkovni tipi so: *integer, double, string, char* in *bool*. Poleg atomarnih tipov pa so podprti tudi sestavljeni podatkovni tipi:

1. Razredi
2. Enumeracije
3. Terke

Poleg naštetih podatkovnih tipov obstaja še podatkovni tip, ki ga ni mogoče eksplicitno uporabljati. To je kazalec (*angl. pointer*). Spremenljivka, ki je tipa *kazalec*, ne hrani dejanske vrednosti, temveč v sebi hrani naslov na neko lokacijo v pomnilniku. Ena izmed prednosti kazalcev je v tem, da lahko do kompleksnejših objektov, ki v pomnilniku zasedejo veliko prostora, dostopamo preko njihovega naslova s kazalcem, namesto kopiranja celotne vsebine (na primer pri pošiljanju objekta kot argument funkciji).

Prevajalnik omogoča tudi t.i. avtomatsko prepoznavanje tipov, oz. angl. *type inference*. To pomeni, da izrazom, kjer prevajalnik lahko implicitno prepozna podatkovni tip, le-tega ni potrebno eksplicitno navajti.

```
let x = 120
let y = "To je niz"
```

Listing 3.1: Primer deklaracij spremenljivk, kjer je njun tip prepoznan avtomastsko

Enumeracije

Enumeracija je sestavljen podatkovni tip, ki vsebuje določeno število konstant. Enumeracije, za razliko od tistih v Javi, omogočajo, da ima vsaka konstanta tudi *surovo vrednost* (*angl. raw value*). Podatkovni tip surovih vrednosti je definiran v naprej in je pri vseh kostantah enak. (Swift)

```
enum Languages {
  case Cpp
  case Java
  case ObjectiveC, Swift
}
```

Listing 3.2: Enumeracija brez surovih vrednosti

Vrednosti surovih vrednosti morajo biti eksplicitno navedene pri vseh konstantah, razen v primeru, ko je podatkovni tip *int* ali *string*. Če je podatkovni tip int in vrednosti niso eksplicitno navedene, se implicitno dodeli zaporedni indeks konstante (začenši z 0).

Če je podatkovni tip string, potem je implicitna vrednost konstante kar njeno ime.

```
enum CarBrands: String {
  case Audi, Renault, VW="Volks Wagen"
  case None="Izbrali niste nobene znamke"
}
```

Listing 3.3: Enumeracija s surovimi vrednostmi tipa String

Terke

Terka je podatkovna struktura sestavljena iz poljubnega števila elementov, ki so lahko različnih podatkovnih tipov. Terke poznajo jeziki, kot sta Python in Swift, medtem ko jih Java in C++ ne poznata.

Terka je definirana kot zaporedje izrazov znotraj oklepajev. Do posameznih elementov terke dostopamo podobno kot dostopamo do elementov v

razredu, s to razliko, da je ime elementa njegov indeks v terki.

```
let x = (10, 5.5)
print(x.0)
print(x.1)
```

Listing 3.4: Terka, sestavljena in dveh vrednosti (Int in Double)

Možno pa je tudi, da pri definiciji tipa terke specificiramo imena elementov, in potem do njih dostopamo preko njih. (Swift)

```
let y = (x: "Lorem", "Ipsum")
print(y.x)
print(y.1)
```

Listing 3.5: Terka, v kateri je en element poimenovan, drugi pa ne

Razredi in vmesniki

Razred je sestavljena podatkovna struktura, ki lahko, za razliko od tabel, v sebi hrani spremenljivke različnih podatkovnih tipov.

Vmesnik *angl. interface* je abstraktna podatkovna struktura, ki definira poljubno število metod, ki jih mora implementirajoči se razred implementirati. Instanc vmesnikov ne moremo kreirati neposredno.

Vsak razred lahko vsebuje poljubno število atributov (spremenljivk) ter poljubno število metod - to so člani rezreda. Atributi in metode so lahko statični, kar pomeni, da niso del posamezne instance razreda, ampak živijo v statični instanci, ki je kreirana avtomatsko. Poleg tega so lahko člani razreda *privatni* ali *javni*. Razlika je v tem, da do privatnih članov ni možno dostopati izven razreda. Metodam lahko dodamo še modifikator *final*, kar prepreči, da bi bila funkcija re-implementirana v dedujočem se razredu. Če želimo, da metoda re-implementira metodo v starševkem razredu, ji moramo dodati modifikator *overriding*.

Dedovanje

Vsak razred lahko deduje največ en razred (lahko tudi nobenega), implemen-

tira pa lahko poljubno število vmesnikov. Identično je dedovanje narejeno tudi v Javi in Swiftu, za razliko od C++, ki omogoča večkratno dedovanje in ne pozna vmesnikov.

Razširitve

Podprte so tudi razširitve nad razredi, ki omogočajo, da obstoječim razredom dodamo nove metode (vzeto iz Swifta). Razširitve so predsvem uporabne takrat, ko želimo razredom, do katerih nimamo dostopa (npr. razredi v zunanjih knjižnicah), dodati uporabne funkcionalnosti. V jezikih, ki tega ne podpirajo, je ponavadi praksa, da se ustvarijo *utilty* razredi (npr. v Javi).

Razširitve omogočajo tudi to, da razredom dodamo implementacijo vmesnikov, čemur se reče *interface extension*.

```
interface Hashable {
        func hash() Int
}

extension Int: Hashable {
        func hash() Int {
                return self % 21632
        }
}
```

Listing 3.6: Primer razširitve tipa Int z vmesnikom Hashable

Poglavje 4

Programski jezik Atheris

4.1 Sintaksa

Celoten program v jeziku PINS je sestavljen iz seznama definicij, definicija pa je lahko definicija spremenljivke, definicija tipa ali definicija funkcije. Sintaksa programskega jezika Atheris se razlikuje v tem, da je program sestavljen iz zaporedja stavkov, stavek pa je lahko *izraz* ali *definicija*. Definicije so predstavljene z naslednjimi konteksno neodvisnimi gramatikami:

1. Spremenljivke

```
var_def -> visibility var identifier
var_def -> visibility var identifier =
   expression
var_def -> visibility var identifier: type
var_def -> visibility var identifier: type =
   expression

visibility -> public
visibility -> private
visibility -> $
```

2. Funkcije

```
func_def -> func identifier ( parameters ) {
   statements }
func_def -> func identifier ( parameters )
type { statements }

parameters -> $
parameters -> paramater
parameters -> paramaters, paramater

parameter -> identifier : type
parameter -> identifier identifier : type
```

3. Enumeracije

```
enum_def -> enum { enum_defs }

enum_defs -> $
enum_defs -> enum_member_def
enum_defs -> enum_defs, enum_member_def

enum_member_def -> case identifier
enum_member_def -> case identifier = expression
   (literal)
enum_member_def -> enum_member_def,
   enum_member_def // todo
```

4. Terke

```
todo
```

5. Razredi

```
class_def -> class { definitions }
```

Sintakse izrazov so skoraj identične, razen izrazov za nadzor toka (angl. *control flow*):

1. If stavek

```
if_expression { statements } if_expression'
if_expression' -> $
if_expression' -> else if { statements }
   if_expression'
if_expression' -> else { statements }
```

2. Switch stavek

```
switch_expression { cases }

cases -> case expressions : statements cases'
cases' -> $
cases' -> cases
cases' -> default : statements
```

3. While stavek

```
while_expression -> while expression {
   statements }
```

4. For stavek

```
for_expression -> for identifier in expression
{ statements }
```

Kot sem že omenil, omogoča sintaksa programskega jezika Atheris, da so posamezni stavki ločeni bodisi z ';' bodisi z novo vrstico. Da sem lahko to implementiral, sem razširil leksikalni analizator z novo vrsto simbola *newline*. V primeru, da je v izvorni kodi več zaporednih novih vrstic, jih leksikalni analizator *požre* in vedno vrne samo en zaporedni *newline* simbol. To naredi

sintaksno analizo enostavnejšo. Ločevanje posameznih stavkov z novo vrstico je tako trivialno, potrebno je samo paziti na primere, kjer so stavki ločeni z obema ločiloma.

4.2 Funkcije

Programski jezik Atheris zahteva, da ob klicu funkcij, poleg imena funkcije, navedemo tudi imena parametrov. Podobno kot to dela tudi Swift.

Ob deklaraciji funkcije lahko opcijsko definiramo poljubno število poimenovanih ter tipiziranih vrednosti, ki jim skupaj rečemo parametri funkcije. Poleg tega lahko opcijsko navedemo še podatkovni tip vrednosti, ki jo bo funkcija vračala (privzeto so funkcije tipa **Void**).

Imena parametrov so sestavljena iz dveh delov: labele argumenta ter imena parametra. Labele argumentov se uporabljajo ob klicu funkcije, medtem ko se imena parametrov uporabljajo v sami implementaciji funkcije. Privzeto so labele iste kot imena.

```
func someFunction(firstParameterName: Int,
   secondParameterName: Int) {
  /* znotraj telesa funkcije, sta firstParameterName
    in secondParameterName referenci na vrednosti
   argumentov za prvi in drugi parameter. */
}
someFunction(firstParameterName: 1,
   secondParameterName: 2)
```

Listing 4.1: Primer definicije in klica funkcije v programskem jeziku Atheris

Določanje label argumentov

Če želimo, da sta labela in ime različne, navedemo labelo argumenta pred imenom:

```
func someFunction(argumentLabel parameterName: Int)
   {
```

```
/* znotraj telesa funkcije, se parameterName
   sklicuje na vrednost argumentza funkciji */
}
someFunction(argumentLabel: 1)
```

Listing 4.2: Labela argumenta in ime parametra se razlikujeta

Podprta je tudi možnost, da se izognemo navajanju imen argumentov pri klicanju funkcij. To storimo tako, da označimo labelo argumenta z '_'.

```
func someFunction(_ firstParameterName: Int, _
   secondParameterName: Int) {
  ///
}
someFunction(1, 2)
```

V ozadju so opisane funkcionalnosti implementirane tako, da se v simbolno tabelo ne shrani samo *ime* funkcije, ampak se celotna definicija pretvori v posebno znakovno predstavitev, ki je sestavljena iz imena ter label argumentov, ki so med sabo ločena z dvopičjem. Funkcije iz zgornjih primerov se pretvorijo v:

```
someFunction(firstParameterName:secondParameterName)
someFunction(argumentLabel)
someFunction(_:_)
```

Listing 4.3: Znakovne predstavitve funkcij

T.i. *overloadanje* funkcij sedaj ni problem, saj čeprav imajo funkcije ista imena, vsako izmed njih predstavimo z drugim imenom, zato jih lahko v simbolni tabeli ustrezno poiščemo. Seveda je potrebno v podobni obliki predstaviti tudi klice funkcij (logika je popolnoma identična).

4.3 Enumeracije

V primeru, da enumeracija nima surovih vrednosti, je semantična analiza dokaj preprosta. Kompleksnejša postane kadar so surove vrednosti prisotne,

saj je potrebno zagotoviti, da so ustreznega podatkovnega tipa. Poleg tega je potrebno določiti implicitne vrednosti, v primeru, da niso eksplicitno navedene. Vendar samo takrat, ko je surova vrednost tipa **Int** ali **String**. Če je katerega koli drugega podatkovnega tipa, prevajalnik vrne napako.

Privzeto so surove vrednosti za **Int** zaporedna števila začenši z 0. V primeru, da je vrednost eksplicitno navedena, je vrednost naslednje surove vrednosti naslednje zaporedno število od prejšnje vrednosti.

```
enum Days: Int {
  case Monday // vrednost = 0
  case Tuesday = 10 // vrednost = 10
  case Wednesday // vrednost = 11
  case Thursday = 100 // vrednost = 100
  case Friday // vrednost = 101
}
```

Listing 4.4: Enumeracija s surovimi vrednostmi tipa Int

Za **String** so privzete surove vrednosti imena članov enumeracije.

```
enum Fruit: String {
  case Apple // privzeta vrednost = "Apple"
  case Orange = "Annoying Orange"
  case Strawberry // privzeta vrednost = "Strawberry
   "
}
```

Listing 4.5: Enumeracija s surovimi vrednostmi tipa String

Do posameznih elementov enumeracije dostopamo z *DOT* operatorjem ('.'), kjer je na levi strani ime enumeracije, na desni pa ime elementa. Podobno dostopamo tudi do surovih vrednosti z uporabo imena *rawValue*. Operator DOT bom podrobneje opisal na razdelku razredov, za zdaj lahko povem, da preveri, ali v sestavljenem podaktovnem tipu, ki se nahaja na levi strani, obstaja član z imenom na desni strani.

```
print(Fruit.Apple.rawValue) // 'Apple'
print(Fruit.Orange.rawValue)  // 'Annoying Orange'
print(Fruit.Strawberry.rawValue)  // 'Strawberry'
```

Listing 4.6: Primer dostopa do elementov enumeracije 4.5

Člane enumeracije lahko prirejamo spremenljivkam, ter nad njimi izvajamo logične operacije primerjanja vrednosti.

```
let x: Languages = Languages.Cpp
if x == Languages.Java {
  print("Java")
}
else {
  print("Some other language")
}
```

Listing 4.7: Izvajanje logike na podlagi vrednosti enumeracije 3.2

4.4 Terke

Terka je zaporedje izrazov ločenih z dvopičjem ':' obdano z oklepaji. Vsakemu izrazu znotraj terke lahko opcijsko določimo tudi ime. V primeru, da ime ni eksplicitno določeno, se za ime uporabi indeks izraza v terki (začenši z 0).

Do elementov prav tako dostopamo z uporabo DOT operatorja, ki zagotovi, da terka vsebuje izraz z željenim imenom.

V pomnilniku so terke predstavljene skoraj identično kot tabele, s to razliko, da se podatkovni tipi elementov med sabo lahko razlikujejo. Odmik posameznega elementa od začetnega naslova terke izračunamo tako, da seštejemo velikost podaktovnih tipov vseh elementov pred tem elementom

4.5 Razredi

Razred je kompleksna podatkovna struktura in je sestavljena iz poljubnega števila poljubnih definicij. Tako lahko znotraj razreda definiramo nove razrede, enumeracije, vmesnike ter seveda funkcije (znotraj razredov jih imenujemo metode) in spremenljivke.

V pomnilniku je instanca razreda (objekt) predstavljena podobno kot terka, t.j. kot tabela z vrednostmi različnih podatkovnih tipov. Zato tudi odmik elementov izračunamo podobno kot pri terkah.

4.5.1 Dostop do članov in nadzor dostopa

Do članov razreda dostopamo preko DOT operatorja. DOT operator je binarni operator in je sestavljen iz dveh izrazov, ki ju povezuje pika '.'.

Običajno se izvaja povezovanje uporab spremenljivk z definicijami v fazi razreševanja imen, vendar to pri sestavljenih podatkovnih strukturah (vključno z enumeracijami in terkami) ni mogoče. Problem je namreč v tem, da se definicija nahaja znotraj **dejanskega podatkovnga tipa**, ki pa ga v fazi razreševanja imen še ne poznamo. Zato se s tem ne ukvarjamo v razreševanju imen, ampak problem prepostimo razreševanju tipov. Razreševanje tipov zato delimo na dve pod-fazi, oz. dva sprehoda po AST. V prvem sprehodu poberemo vse informacije o definiranih podatkovnih tipih ter razredom in ostalim definicijam pripišemo dejanske podatkovne tipe.

Razred (v prevajalniku), ki opisuje podatkovne tipe za razrede (v programu), hrani imena vseh članov razreda, njihove definicije ter podatkovne tipe. Vsebuje tudi pomožne metode, s lahko preverjamo, ali razred vsebuje člana z nekim imenom, pridobimo podatkovni tip člana, odmik člana v razredu, itd.

Sedaj imamo dovolj informacij, da lahko razrešimo imena znotraj DOT operatorja. Na levi se nahaja sestavljen podatkovni tip, za katerega natanko vemo, katere elemente vsebuje, zato ni težko preveriti, če se ime na desni strani nahaja znotraj tipa na desni.

Nadzor dostopa

Nadzor dostopa omejuje dostop do članov razreda izven definicije, če to eksplicitno navedemo z uporabo modifikatorja ***private***. Privzeto so vsi člani ***public***. Programski jezik Atheris trenutno pozna samo public in private modifikatorja; public modifikator omogoča dostop do in spreminjanje vrednosti člana razreda kodi, ki se ne nahaja v razredu, za razliko od modifikatorja private, ko to prepoveduje. Java in C++ poznata še modifikator vidnosti ***protected***, medtem ko jih Swift pozna še nekaj več: ***fileprivate***, ***internal***, ***private(set)***.

4.5.2 Instančne funkcije

Razred ima lahko v sebi definirane funkcije, ki se jim v OOP žargonu pogosto reče *metode*. Metoda se od ostalih funkcij (ki niso definirane v razredu) razlikuje v tem, da vsebujejo impliciten parameter, ki ga prevajalnik vstavi avtomatsko, vrednost parametra pa je kazalec in kaže na objekt, ki funkcijo kliče. V C++ in Javi se ta parameter imenuje ***this***, v Pythonu, Swiftu, in s tem tudi v Atherisu, pa ***self***. V Pythonu je ta razlika, da je parameter potrebno eksplicitno navesti, če ga ne, se funkcija / metoda smatra kot statična (več o statičnih metodah v nadaljevanju).

Prevajalnik mora implicitni *self* parameter avtomatsko vstaviti v vse metode razreda. To stori v prvi fazi semantične analize - razreševanje imen. Podatkovni tip parametra še ni znan, zato se mora nastaviti šele v fazi razreševanja tipov. Če se znotraj metode sklicujemo na self parameter, uporabo ni težko povezati z definicijo, saj je sedaj del funkcije.

V C++, Javi in Swiftu se ni potrebno eksplicitno sklicevati na this / self parameter, da bi dostopali do članov objekta, vendar to zaenkrat še ni podprto v Atherisu.

Pri generiranju vmesne kode mora prevajalnik poskrbeti, da se v parameter prenese naslov na objekt, ki kliče metodo. Naslov objekta je znan, v parameter je potrebno je presesti naslov spremenljivke, ki se nahaja na levi strani DOT operatorja.

4.5.3 Konstruktorji

Konstruktor je statična metoda razreda, njegova naloga pa je kreacija objekta ter inicializacija atributov. Podobno kot instančne metode tudi konstruktorju prevajalnik implicitno vstavi *self* parameter, vendar pa z drugačnim namenom.

Naloga konstruktorja je, da na kopici, t.j. del pomnilnika, kamor shranjujemo dinamično alicirane objekte (za razliko od sklada, kamor shranjujemo statične), alocira prostor kamor bo shranjen objekt. Za dinamično alociranje pomnilnika se uporabi gradnik vmesne kode **ImcMALLOC**. Ta gradnik na kopici rezervira željeno velikost pomnilnika ter vrne naslov.

Pri generiranju vmesne kode prevajalnik doda na začetek kode funkcije MALLOC gradnik, ki bo rezerviral prostor za objekt. Nato bo prevajalnik shranil vrnjen naslov na sklad v parameter *self*, zato da lahko znotraj konstruktorja nastavimo vrednosti ostalih atributov razreda. Na koncu bo ta naslov konstruktor tudi vrnil. Vrnjen naslov je referenca na objekt in jo lahko uporabljamo za nadaljne operacije nad objektom.

Razred ima lahko poljubno število konstruktorjev, ki jih definiramo z uporabo ključne besede ***init***.

```
class Atheris {
    var x: Int
    init() {
        self.x = 0
    }
    init(x: Int) {
        self.x = x
    }
}
```

Listing 4.8: Konsturktorji

Poleg eksplicitnih konstruktorjev prevajalnik zgenerira tudi impliciten *privzeti* konstruktor. Vanj je vstavljena koda za inicializacijo atributov, ki

so inicializirani ob definiciji.

```
class Atheris {
    var x: Int = 10
    var y: Int = 20
}

print(Atheris().x) // 10
print(Atheris().y) // 20
```

Listing 4.9: Implicitni privzeti konstruktor

V primeru, da je v razredu privzeti konstruktor tudi eksplicitno definiran, bo poleg svoje kode vseboval tudi kodo implicitnega.

```
class Atheris {
    var x: Int = 10
    var y: Int = 20
    init() { // privzeti konstruktor
        y = 100
    }
}

print(Atheris().x) // 10
print(Atheris().y) // 100
```

Listing 4.10: Eksplicitni privzeti konstruktor

4.5.4 Statični člani razreda

Statični člani razreda so tisti člani, ki ne živijo znotraj posameznih instanc, ampak živijo v globalni statični instanci razreda. Definiramo jih z uporabo modifikatorja ***static***. Statična instanca razreda je naložena v pomnilnik avtomatsko ob zagonu programa. Do statičnih članov dostopamo preko imena razreda. Ker statične funkcije niso del instance razreda, ne vsebujejo *self* parametra.

```
class Static {
    static func foo() {
        print("I am a static function")
    }
}
Static.foo()
```

Listing 4.11: Klicanje statične funkcije

V času razreševanja tipov so statične definicije razreda shranjene v ločeno podatkovno strukturo od instančnih. Vozlišču AST, ki definira razred (*AstClassDefinition*), v simbolno tabelo ne pripišemo razredni podatkovni tip, ampak *statični / kanonični* tip, ki hrani podatke o statičnih definicijah razreda ter referenco na dejanski instačni podatkovni tip. Ko se sklicujemo na člane statične instance preko DOT operatorja, prevajalnik preveri, ali statični tip vsebuje iskan član.

Razredi lahko vsebujejo tudi enumeracije in gnezdene razrede. Definicije enumeracij in razredov se smatrajo privzeto kot statične, zato do njih dostopamo enako kot do statičnih članov.

```
class Static {
    enum Languages: Int {
        case Cpp, Java
    }
}
let x = Static.Languages.Java
print(x.rawValue) // '1'
```

Listing 4.12: Enumeracija znotraj razreda

Da lahko kreiramo gnezdene razrede, prevajalnik konstruktorje gnezdenih razredov potisne v kanonični tip razreda, sicer ne bi bilo možno do njih dostopati.

```
class Atheris {
    class NestedClass {
```

```
            var x = 20
        }
    }
    print(Atheris.NestedClass().x) // '20'
```

Listing 4.13: Gnezden razred

4.6 Dedovanje in polimorfizem

Dedovanje in polimorfizem, sta, poleg enkapsulacije in abstrakcije, zelo pomembna OOP koncepta. Dedovanje je mehanizem, s katerim objekt pridobi (podeduje) nakatere ali vse lastnosti drugega objekta, medtem ko je polimorfizem sposobnost predstaviti isto funkcionalnost z različnimi implementacijami. Programski jezik Atheris omogoča enkratno dedovanje, tako kot Java in Swift, medtem ko C++ dovoli večkratno.

Tekom preverjanja tipov prevajalnik zagotovi, da je dedovan podatkovni tip razred, saj dedovanje od drugih tipov ni dovoljeno. Prevajalnik za vsako definicijo preveri, ali že obstaja v nadrazredu. V primeru da ja, javi napako. Prekrivanje je dovoljeno samo za definicije označene z modifikatorjem ***override*** in je trenutno mogoče samo za metode.

V pomnilniku se atributi dedovanega razreda nahajajo pred atributi dedujočega se razreda. Enačba za izračun velikosti razreda (koliko prostora porabi objekt v pomnilniku) postane kompleksnejša, saj je potrebno rekurzivno prišteti velikosti nadrazradov. Prav tako postane komleksnejši izračun odmikov atributov od začetnega naslova.

4.6.1 Dynamic Dispatch

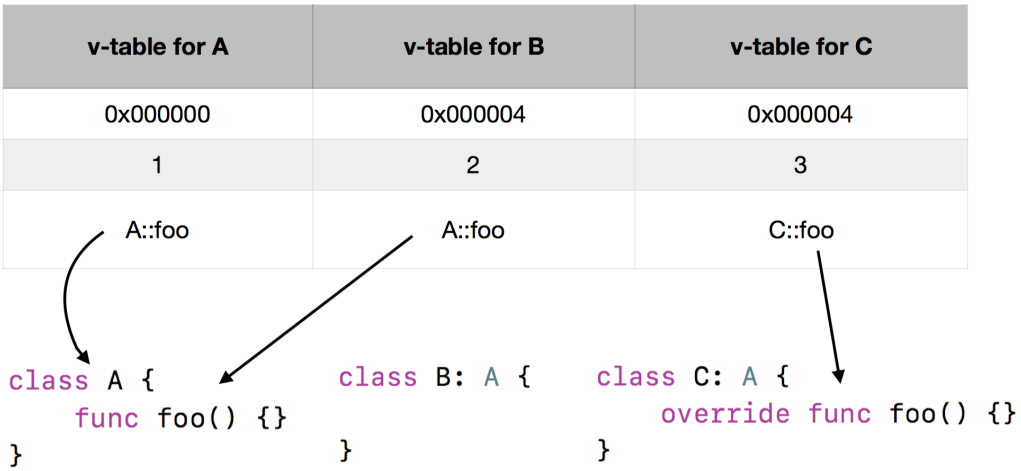
Dynamic dispatch je proces izbira implementacije polimorfizne metode v času **izvajanja programa**. Razlog zakaj proces izbire ni možno opraviti tekom prevajanja, se skriva v tem, da lahko obstaja več različnih implementacij iste metode (overridanje metod). Statično izbiranje (t.j. v času prevajanja) je

možno za statične metode.

Dynamic dispatch je možno implementirati na različne načine. Rešitev v tem prevajalniku temelji na rešitvi, ki jo ima tipično C++, in sicer s podatkovno strukturo ***v-table*** oz. virtualna tabela.

Virtualne tabele

Virtualna tabela je podatkovna stuktura, ki vsebuje kazalce na funkcije implementirane v danem razredu. Prevajalnik zgenerira, in shrani v pomnilnik, za vsak definiran razred unikatno v-tabelo. Vsaka instanca razreda vsebuje, polen njenih spremenljivk, še kazalec na ustrezno tabelo.



Slika 4.1: Virtualne tabele za dane razrede.

Virtualna tabela vsebuje kazalce na naslove, kjer se nahajajo vse nestatične metode, ki so implementirane v razredu. Vsaka tabela vsebuje še kazalec na virtualno tabelo nadrazreda (ali null, če ga nima), ter unikatni identifikator tipa. Identifikator je pozitivno celo število, ki ga prevajalnik dodeli vsakemu tipu.

4.6.2 Realizacija

V času prevajanja imamo na voljo informacijo o tem, katera po vrsti je metoda definirana v razredu, kar se izkaže, da je dovolj. V virtualni tabeli namreč prevajalnik na točno isti indeks, na katerem je motoda definirana, shrani kazalec na implementacijo. Prevajalnik nato zgenerira vmesno kodo, ki izračuna naslov funkcije z naslednjo enačbo:

$$address(method_i) = adress(vtable) + (i * sizeof(void*)) + 8$$

kjer je

$$address(vtable)$$

lokacija tabele v pomnilniku,

$$(i * sizeof(void*)) + 8$$

indeks pomnožen z velikostjo kazalca (običajno 4 byti) plus odmik od začetka tabele zaradi prej opisanih dodatnih podatkov v tabeli, ter

$$address(method_i)$$

naslov metode na indeksu i. Ko interpreter tekom izvajanja kode izračuna naslov metode, njeno kodo naloži iz izračunanega naslova, ter jo izvede.

4.7 Instance of operator

Instance of operator je operator, ki v času izvajanja programa preveri, ali je objekt instanca danega razreda. To naredi z uporabo virtualne tabele. Kot sem omenil, vsaka tabela vsebuje unikatni deskriptor njenega razreda. Preko deskriptorja lahko preverimo, ali se dva podatkovna tipa ujemata.

V programskem jeziku Atheris se operator namesto *instance of* imenuje *is*.

```
var object: A
object = B()
print(object is A) // 'true'
print(object is B) // 'true'
print(object is C) // 'false'
```

Listing 4.14: Uporaba operatorja *is* za razrede iz sheme 4.1

Izračun je malenkost kompleksnejši kot sem opisal zgoraj, saj če želimo preveriti, ali je objekt instanca razreda, ki se lahko po dedovani verigi nahaja daleč navzgor, je potrebno preverjanje izvesti rekurzivno po celotni hiearhiji razredov.

```
todo
```

Listing 4.15: Algoritem za izračun ali je objekt instanca danega razeda

4.8 Pretvarjanje tipov

Mehanizem pretvarjanja iz enega tipa v drug (angl. *type casting*) deluje po zelo podobnem principu kot *as* operator. Razlikuje se samo v tem, da je rezultat operacije, če pretvorba uspe naslov objekta, sicer 'null'. Takemu tipu pretvarjanja rečemo *safe casting*, saj program nadaljuje z izvajanjem tudi če pretvorba ni uspešna. Za razliko od *non-safe castanja*, kjer bi program vrnil *null pointer exception*.

```
    var object: A = B()
    let casted = object as B // pretvorba iz 'A'
v 'B' je uspesna
    if casted == null {
        print("cast failed")
    }
    else {
        print("cast succeeded")
    }
```

Listing 4.16: Pretvorba razrede iz sheme 4.1

4.9 Vmesniki

Vmesnik je abstraken opis akcij, ki jih lahko izvede objekt. Tekom prevajanja prevajalnik zagotovi, da razred vsebuje vse metode v vmesnikih, ki jih želi implementirati. Ker je vmesnik abstraktna podatkovna struktura, ne moremo kreirati instance neposredno, lahko pa spremeljivki priredimo instanco razreda, če razred implementira vmesnik. Vse metode so klicane dinamično, po postoku opisanem v razdelku 4.6.1.

```
    interface I {
        func foo()
    }
    class A: I {
        func foo() {
            print("A's implementation of foo")
        }
    }
    class B: I {
        func foo() {
            print("B's implementation of foo")
```

```
    }
}
var x: I
x = A()
x.foo() // 'A's implementation of foo'
x = B()
x.foo() // 'B's implementation of foo'
```

Listing 4.17: Vmesniki

4.10 Razširitve

Razširitev (angl. *extension*) je mehanizem, ki omogoča dodajanje funkcionalnosti obstoječim razredom.

```
extension A {
    func toString() {
        print("ext::toString()")
    }
}
A().toString()
```

Listing 4.18: Razširitev razreda A

Razred lahko razširimo z metodami, enumeracijami in razredi.

4.10.1 Razširitev z vmesnikom

Razredom lahko preko razširitev dodamo tudi implementacijo vmesnikov, čemur rečemo *interface extension*.

```
class Collection {}
interface Iterable {
    func next() -> Any
}
```

```
extension Collection: Iterable {
    func next() -> Any {
        return nil
    }
}
let iterable: Iterable = Collection()
print(iterable.next())
```

Listing 4.19: Razširitev z vmesnikom

4.11 Type inference

Type inference omogoča prevajalniku, da avtomatsko prepozna podatkovni tip izraza, na podlagi vrednosti, ki so v izrazu. To je posebej uporabno pri definicijah spremeljivk, ki jim pogosto nastavimo vrednost ob sami definiciji.

```
let meaningOfLife = 42
// meaningOfLife je tipa Int
let pi = 3.14159
// pi je tipa Double
let anotherPi = 3 + 0.14159
// anotherPi je tudi tipa Double
```

Listing 4.20: Type inference

Prepoznavanje tipov postane malenkost kompleksnejše pri tabelah, kadar njeni elementi niso istih tipov. Prevajalnik tekom razreševanja tipov vedno poišče najmanjši možen tip, ki je skupen tipom elementov tabele. V primeru, da so v tabeli objekti, ki jim je skupen nadrazred A, bo tudi tabela tipa A. V primeru, da elementi nimajo nobenega skupnega tipa, prevajalnik tabeli dodeli poseben abstrakten tip ***Any***, ki je definiran v standardni knjižnici. *Any* je v programskem jeziku Atheris to, kar je v Javi *Object*, s to razliko, da je *Object* razred, *Any* pa vmesnik. Vsak podatkovni tip je implicitno tipa *Any* .

```
let array = [C(), B(), C(), C()] //
spremenljivka array = tipa ARR ( A )
```

Listing 4.21: Elementi tabele s skupnim nadrazredom

```
let array = [10, "string", A(), 3.14] //
spremenljivka array = tipa ARR ( Any )
```

Listing 4.22: Elementi tabele nimajo skupnega tipa

Poglavje 5

Sklepne ugotovitve

Uporaba LaTeXa in BibTeXa je v okviru Diplomskega seminarja **obvezna**! Izbira LaTeX ali ne LaTeX pri pisanju dejanske diplomske naloge pa je prepuščena dogovoru med vami in vašim mentorjem.

Res je, da so prvi koraki v LaTeXu težavni. Ta dokument naj vam služi kot začetna opora pri hoji. Pri kakršnihkoli nadaljnih vprašanjih ali napakah pa svetujem uporabo Googla, saj je spletnih strani za pomoč pri odpravljanju težav pri uporabi LaTeXa ogromno.

Literatura

[1] Michael Riis Andersen, Thomas Jensen, Pavel Lisouski, Anders Krogh Mortensen, Mikkel Kragh Hansen, Torben Gregersen, and Peter Ahrendt. Kinect depth sensor evaluation for computer vision applications. Technical report, Department of Engineering, Aarhus University, 2012.

[2] A complete description of a programming language includes the computational model, the syntax and semantics of programs, and the pragmatic considerations that shape the language. Dosegljivo: `http://www.emu.edu.tr/aelci/Courses/D-318/D-318-Files/plbook/intro.htm`. [Dostopano: 26.11.2017].

[3] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java, Second Edition.* Cambridge University Press, 2002.

[4] Andreja Balon. Vizualizacija. Diplomska naloga, Fakulteta za elektrotehniko in računalništvo, Univerza v Ljubljani, 1990.

[5] Donald knuth. Dosegljivo: `https://sl.wikipedia.org/wiki/Donald_Knuth`. [Dostopano: 1. 10. 2016].

[6] Donald E Knuth and Peter B Bendix. Simple word problems in universal algebras. In Jörg H. Siekmann and Graham Wrightson, editors, *Automation of Reasoning: Classical papers on computational logic 1957–1966*, pages 342–376. Springer, 1983.

[7] Leslie Lamport. *LaTEX: A Document Preparation System.* Addison-Wesley, 1986.

[8] Oren Patashnik. BibTeXing. Dosegljivo: `http://bibtexml.sourceforge.net/btxdoc.pdf`, 1988. [Dostopano 5. 6. 2016].

[9] PDF/A. Dosegljivo: `http://en.wikipedia.org/wiki/PDF/A`, 2005. [Dostopano: 5. 6. 2016].

[10] Peter Peer and Borut Batagelj. Art—a perfect testbed for computer vision related research. In *Recent Advances in Multimedia Signal Processing and Communications*, pages 611–629. Springer, 2009.

[11] Franc Solina. 15 seconds of fame. *Leonardo*, 37(2):105–110, 2004.

[12] Franc Solina. Light fountain—an interactive art installation. Dosegljivo: `https://youtu.be/CS6x-QwJywg`, 2015. [Dostopano: 9. 10. 2015].

[13] Matjaž Gams (ured.). DIS slovarček, slovar računalniških izrazov, verzija 2.1.70. Dosegljivo: `http://dis-slovarcek.ijs.si`. [Dostopano: 1. 10. 2016].