Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Toni Kocjan Turk

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. todo

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
AST	abstract syntax tree	abstraktno sintaksno drevo
IR	intermidiate representation	vmesna koda
PINS	compilers and virtual machines	prevajalniki in navidezni stroji
SP	stack pointer	kazalec na sklad
FP	frame pointer	kazalec na klicni zapis

Povzetek

Naslov: Vgradnja objektno usmerjenih gradnikov v programski jezik PINS

Avtor: Toni Kocjan Turk

V diplomskem delu je predstavljen programski jezik Atheris, ki je nastal kot nadgradnja programskega jezika PINS. Programski jezik PINS je učni programski jezik, za kategera prevajalnik smo implementirali tekom semestra pri predmetu Prevajalniki in navidezni stroji. Ker mi je bilo delo na prevajalniku zelo zanimivo, in zaradi želje po dodatnem znanju, sem z delom nadaljeval po zaključku semestra in tako implementiral svoj programski jezik.

V diplomskem delu so na kratko predstavljeni programski jeziki in prevajalniki, kaj sploh so in kaj je njihov namen. Opisane so sodobne prakse pri ravoju prevajalnikov, s kakšnimi problemi se prevajalnik sooča ter kako je zgrajen.

Bistvo diplomske naloge je nadgradnja obstoječega prevajalnika z gradniki, ki niso bili del učnega načrta pri predmetu. Posvetil sem se predvsem objektno usmerjenim gradnikom. Izmed njih programski jezik Atheris podpira enumeracije, terke, razrede in vmesnike, poleg tega pa vsebuje tudi povsem spremenjeno sintakso. Tekom naloge so razširitve in njihova implementacija podrobneje opisane.

Delovanje programskega jezika Atheris je preverjeno na testnih primerih, izmerjena pa je tudi hitrost izvajanja in primerjana z osnovnim jezikom PINS ter za referenco tudi s Pythonom.

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

Abstract

Title: Diploma thesis sample

Author: Toni Kocjan Turk

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, Atheris.

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 velikim številom programskih jezikov. Trenutno eni izmed najbolj popularnih so, Java, C in C++, Python in C#, popularnost pa dobivajo tudi novejši jeziki kot so Go-Lang, Swift, Kotlin in podobni [4].

S prevajalniki sem se začel ukvarjati pri predmetu Prevajalniki in navidezni stroji (PINS) v drugem letniku na Fakulteti za Računalništvo in Informatiko. Pri predmetu smo se osredotočili predvsem na enostavne proceduralne jezike, cilj diplomske naloge pa je vgradnja naprednejših gradnikov, ki niso bili del učnega načrta. V diplomski nalogi grobo opišem programske jezike in prevajalnike, v nadaljevanju predstavim programski jezik PINS, podrobneje pa se posvetim programskemu jeziku Atheris.

Poglavje 2

Prevajalniki

2.1 Uvod v programske jezike in prevajalnike

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)**: zaporedje operacij nad vrednostjo (ali več vrednosti), ki vrne nek rezultat

3. **Program:** specifikacija izračuna

4. **Programski jezik:** zapis (notacija) za pisanje programov

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

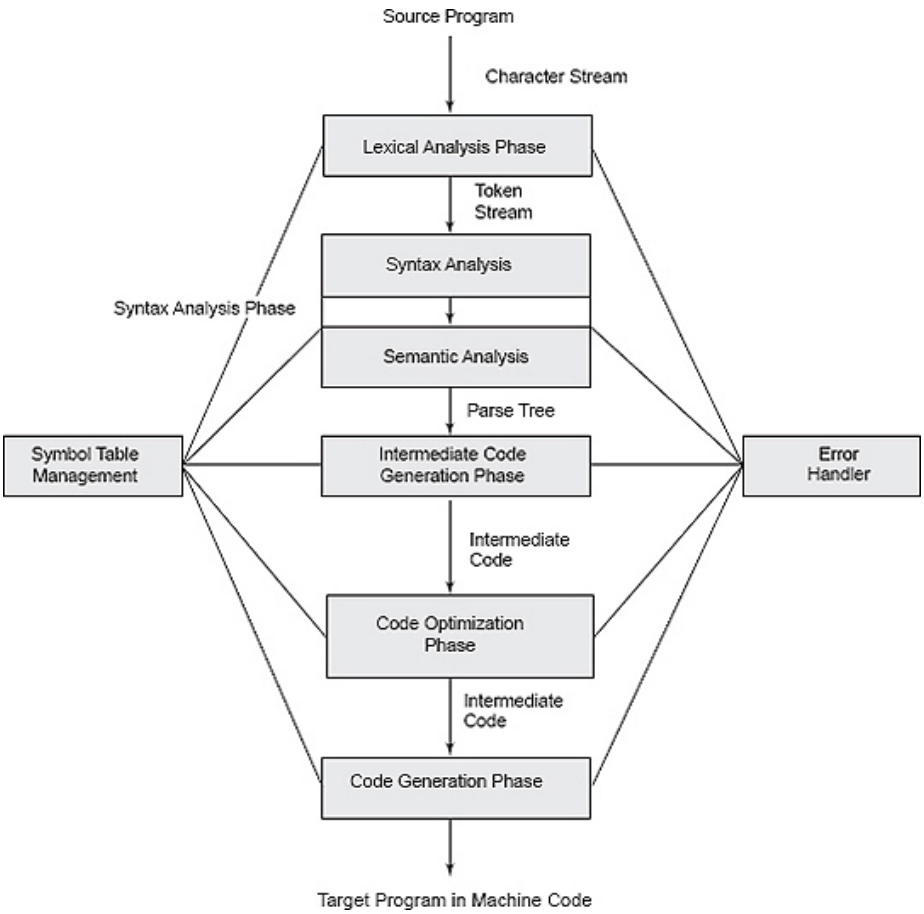
```
rezultat = program(vhodni parametri)
```

Iz drugega zornega kota si lahko program predstavljamo tudi kot model problemske domene, kjer je instanca izvedbe programa simulacija problema [6]:

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

2.2 Zgradba prevajalnika

Sodobni prevajalniki so pogosto organizirani v več posameznih faz, vsaka faza pa operira na drugem nivoju abstrakcije jezika [7].



Slika 2.1: Zgradba prevajalnika [3].

Da lahko prevajalnik program prevede iz ene oblike v drugo, ga najprej analizira, razume njegovo strukturo ter pomen, nato pa ga pretvori v drugačno obliko.
Analizo programa običajno delimo v naslednje korake [7]:

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

vrsta simbola	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 simbolov v programskem jeziku Java.

Po končani analizi sledi sinteza programa v izhodni / ciljni jezik. Sintezo do vmesne kode sestavljata:

1. **Izračun klicnih zapisov** (angl. *frame evaluation*)
2. **Generiranje vmesne kode** (angl. *intermidiate code generation*)

2.2.1 Leksikalna analiza

Leksikalni analizator kot vhod prejme tok znakov, kot izhod pa vrne tok v naprej definiranih simbolov. Simbol je običajno zgrajen iz imena, vrednosti (t.i. *lexeme*) ter lokacije v izvorni datoteki [7].

Leksikalni simboli:

Simbol je zaporedje znakov, ki ga interpretiramo kot samostojno enoto v slovnici programskega jezika [7].
Tabela 2.1 prikazuje nekaj vrst simbolov ter primere.

Sledi preprost primer programa v programskem jeziku Atheris ter rezultat leksikalne analize zanj.

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

Program 2.1: Primer programa v programskem jeziku Atheris.

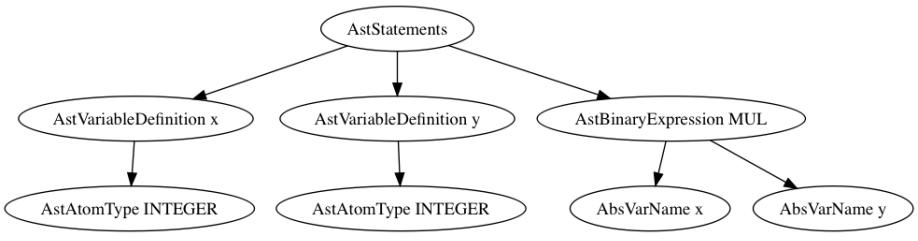
```
[1:1-1:4]       LET:let
[1:5-1:6]       IDENTIFIER:x
[1:6-1:7]       COLON::
[1:8-1:11]      IDENTIFIER:Int
[1:11-1:12]     NEWLINE:\n
[2:1-2:4]       LET:let
[2:5-2:6]       IDENTIFIER:y
[2:6-2:7]       COLON::
[2:8-2:11]      IDENTIFIER:Int
[2:11-2:12]     NEWLINE:\n
[3:1-3:2]       IDENTIFIER:x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

Izpis 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 simbolov, ki ga zgenerira prejšnja faza, rezultat pa je abstraktno sintaksno drevo.

Abstraktno sintaksno drevo (AST) je drevesna podatkovna struktura, ki predstavlja sintaksno strukturo programa. Vsako vozlišče drevesa ponazarja 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 [7].

Običajno izvedbo semantične analize 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.
   Sledi primer, kjer se sklicujemo na nedefinirano spremenljivko:

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

   Program 2.3: Primer programa, kjer spremenljivka y ni definirana.

2. **Preverjanje tipov:** vsakemu vozlišču v AST določi podatkovni tip, ter na podlagi postavljenih semantičnih pravil zagotovi, da so vsi izrazi pravilnih tipov.
   Sledi primer, kjer pride do napake pri razreševanju podatkovnih tipov (operacija '+' ni definirana nad podatkovnima tipoma *String* in *Int*):

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

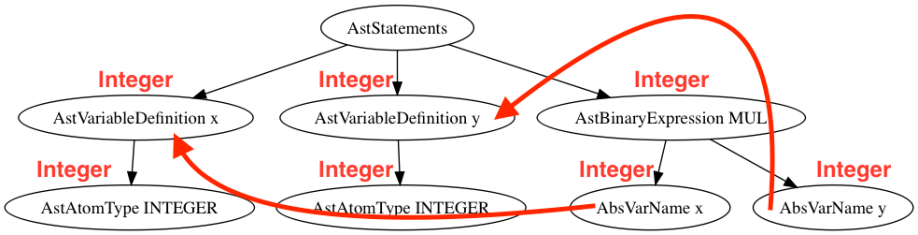
   Program 2.4: Primer programa, kjer je napaka v podatkovnih tipih.

V zahtevnejših prevajalnikih se lahko tekom semantične analize opravi še dosti drugih analiz, npr. analiza inicializiranosti spremenljivk, zaznavanje mrtve kode, eliminacija neuporabljenih izrazov, ipd., vendar se z njimi tekom te diplomske naloge ne ukvarjamo.

Pri implementaciji semantične analize nam pomaga *simbolna tabela*.

Simbolna tabela

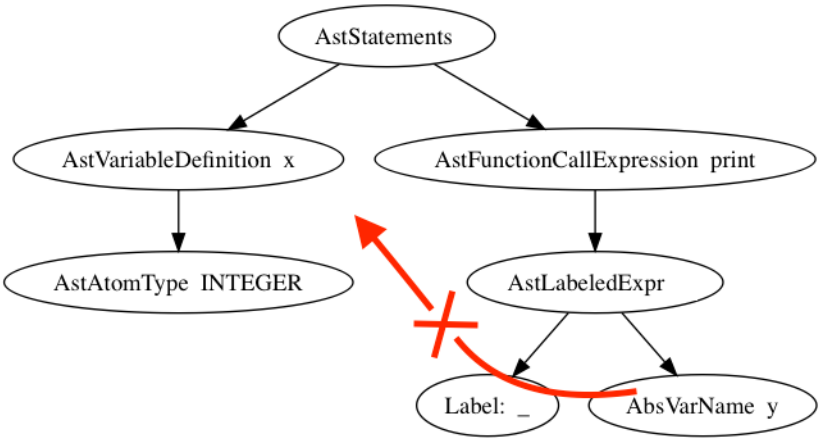
Simbolna tabela je podatkovna struktura, ki preslika imena v njihove definicije in podatkovne tipe [7]. Ker običajno programi vsebujejo več tisoč definicij



Slika 2.3: Rezultat semantične analize za program 2.1.

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 posameznih imen, evaluacija podatkovnih tipov pa je pri vseh vozliščih *Integer*, razen pri korenu, ki nima tipa oz. je tipa *Void*.

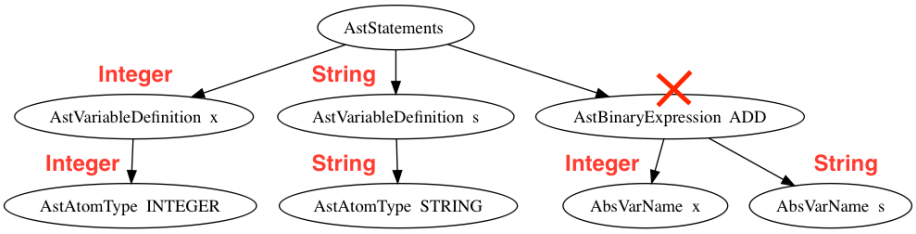
Semantična analiza zagotovi, da za vsako uporabo imena obstaja njena definicija, in da so podatkovni tipi pravilni. Sliki 2.4 in 2.5 prikazujeta semantični napaki.



Slika 2.4: Napaka v programu 2.3; spremenljivka y ni definirana.

2.2.4 Klicni zapisi

Klicni zapis je prostor v pomnilniku, ki vsebuje vse potrebne informacije za izvedbo posameznega klica funkcije. Običajno klicni zapis vsebuje prostor za lokalne spremenljivke, vhodne parametre in register, kamor se shrani izhodna vrednost funkcije.



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

V skoraj vsakem modernem programskem jeziku ima lahko funkcija *lokalne* spremenljivke. Ker lahko hkrati obstaja več klicev iste funkcije je pomembno, da ima vsak klic dostop do lastnih spremenljivk. Ta problem rešujemo s klicnimi zapisi [7].

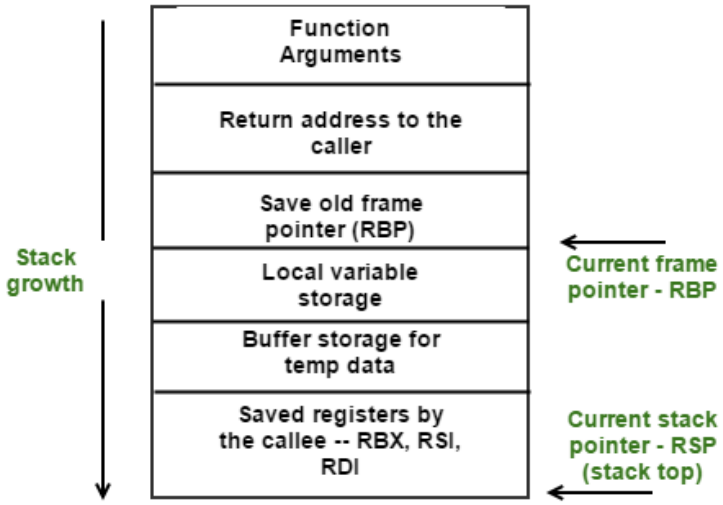
V funkciji

```
func fib(_ n: Int) Int {
    if n < 2 {
        return 1
    }
    else {
        return fib(n - 1) + fib(n - 2)
    }
}
fib(10)
```

se za vsak njen klic ustvari nova instanca spremeljivke n, ki živi na klicnem zapisu funkcije, vrednost pa ji priredimo kot vhodni argument. Ker je funkcija rekurzivna, živi v pomnilniku naenkrat veliko instanc spremenljivke n, vsaka v lastnem klicnem zapisu [7].

Sklad

Klicni zapisi funkcij se shranjujejo na sklad. Sklad je v pomnilniku predstavljen kot velika tabela s posebnim registrom imenovanim *stack pointer*, 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 klicni zapis [7].



Slika 2.6: Klicni zapis [5].

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 [7]:

```
FP = SP + velikost klicnega zapisa
```

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) [7].

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

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

Statična povezava: 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 statična povezava (angl. *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)
        }
    }
}
```

Program 2.5: Primer gnezdenih funkcij.

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

2.2.5 Generiranje vmesne kode

Vmesna koda

Vmesna koda (angl. *intermidiate representation*) je abstraktni približek strojne kode, a brez mnogih podrobnosti posameznih strojnih jezikov. Vmesna koda (IR) predstavlja ukaze, brez da bi poznali arhitekturo ciljne naprave. Poleg tega je vmesna koda neodvisna od izvornega jezika [7].

Dobra predstavitev vmesne kode ima naslednje lastnosti [7]:

1. Njeno generiranje mora biti priročno.
2. Pretvorba vmesne 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.

Poznamo dve vrsti vmesne kode: drevesno in linearno. Za ploščato vmesno kodo se pogosto uporablja t.i. *three-address code* (TAC ali 3AC), kar pomeni, da imajo lahko ukazi take kode največ tri operande.

Pri predmetu smo uporabili drevesno vmesno kodo, zato je tudi vmesna koda zgenerirana s strani prevajalnika za Atheris drevesna.

Posamezni deli abstraktnega sintaksnega drevesa so lahko kompleksne stvari, na primer zanke, klici funkcij, itd., ki jih ne moremo neposredno preslikati 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 [7].

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 jedra funkcij ne moremo početi ničesar drugega, kot ustvarjati definicije (podobno kot pri Javi).

S stališča sintaksne analize je Izraz (*angl. expression*) lahko:

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

Za uspešno izvedbo sintaksne analize atomarni izraz obsega naslednje:

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

Zavedajmo se, da sestavljeni stavki, ki smo jih našteli kot atomarne izraze seveda niso nedeljivi s stališča jezika. Med atomarne izraze so vključeni zaradi upoštevanja prioritet aritmetičnih in logičnih operatorjev. Vsakemu izrazu lahko sledijo tudi definicije gnezdene znotraj zavitih oklepajev.

todo primeri

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*

todo dodaj bubble in fibonacci

Poglavje 4

Programski jezik Atheris

Kljub temu, da prevajalnik za Atheris izhaja iz prevajalnika za PINS, sta si jezika med seboj zelo različna. Razlikujeta se predvsem v sintaksi, ki je v programskem jeziku Atheris zelo podobna tisti, ki jo ima programski jezik Swift.

Slika 4.1: Atheris Chlorechis (western bush viper) [1].

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.

4.0.1 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
4. Vmesniki

Poleg naštetih podatkovnih tipov obstaja še podatkovni tip, ki ga ni mogoče eksplicitno uporabljati. To je kazalec (*angl. pointer*). 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). Spremeljivke, ki jim priredimo objekt, terko ali tabelo, se implicitno smatrajo kot kazalci.

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)
```

4. Terke

```
todo
```

5. Razredi

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

Sintaksa izrazov je skoraj identična PINSu, razen izrazov za nadzor toka (angl. *control flow*):

1. If stavek

   ```
   todo
   ```

2. Switch stavek

   ```
   todo
   ```

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 s ';' bodisi z novo vrstico. Za ta namen je dodana nova vrsta simbolov, ki predstavlja novo vrstico v izvorni kodi (pred tem se je znak za novo vrstico štel kot belo besedilo). 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.

4.2 Funkcije

Programski jezik Atheris zahteva, da ob klicu funkcij, poleg imena funkcije, navedemo tudi imena parametrov. Podobno kot to zahteva 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 funkcije ne vračajo ničesar - *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 znotraj jedra funkcije. Labele so privzeto identične imenom.

```
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)
```

Program 4.1: Primer definicije in klica funkcije.

4.2.1 Določanje label argumentov

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

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

Program 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. Funkcije z istim imenom, a različnimi imeni parametrov, lahko tako shranimo v simbolno tabelo brez težav.

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

Program 4.3: Znakovne predstavitve funkcij.

T.i. *overloadanje* funkcij sedaj ni problem, saj čeprav imajo funkcije ista imena, se na vsako izmed njih sklicujemo preko različne predstavitve, zato jih lahko v simbolni tabeli ustrezno poiščemo. Seveda je potrebno v podobni obliki predstaviti tudi klice funkcij.

4.3 Enumeracije

Enumeracija je podatkovna struktura, ki definira skupen tip skupini povezanih vrednosti, in olajša delo nad njimi. Vsebuje lahko poljubno število vrednosti / konstant, ki jim lahko priredimo t.i. surove vrednosti. Podatkovni tip surovih vrednosti definiriamo v naprej in je pri vseh konstantah enak.

V primeru, da enumeracija vsebuje surove vrednosti, jih je potrebno eksplicitno navesti za vse konstante enumeracije. Izjema so surove vrednosti za podatkovna tipa *Int* in *String*.

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
}
```

Program 4.4: Enumeracija s surovimi vrednostmi tipa Int.

Za *String* so privzete surove vrednosti kar imena konstant.

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

Program 4.5: Enumeracija s surovimi vrednostmi tipa String.

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. V primeru, da

surove vrednosti niso eksplicitno navedene, jih prevajalnik skuša določiti sam. To lahko stori samo v primeru, če so surove vrednosti tipa **Int** ali **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 zagotovi, da se v sestavljenem podaktovnem tipu na levi strani nahaja član z imenom na desni strani (več o operatorju v nadaljevanju).

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

Program 4.6: Primer dostopa do elementov enumeracije 4.5.

Člane enumeracije lahko prirejamo spremenljivkam ter nad njimi izvajamo logični operaciji primerjanja vrednosti.

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

Da lahko realiziramo *runtime* operacije nad člani enumeracije, prevajalnik vsak član enumeracije nadomesti z njegovim indeksom v definiciji enumeracije. Za zgornji primer bo tako spremeljivka x vsebovala vrednost 0. V *if* stavku primerjamo vrednosti 0 in 1, iz česar sledi, da se bo izvedel *else* blok.

4.4 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 enumeraciji, ime elementa pa je kar njegov indeks v terki.

```
let x = (10, 5.5)
```

```
print(x.0)
print(x.1)
```

Program 4.7: Terka, sestavljena iz dveh vrednosti (Int, Double).

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).

```
let x = (lorem: "Lorem", "Ipsum")
print(x.lorem)
print(x.1)
# print(x.0) - napaka
```

Program 4.8: Terka s poimenovanim elementom.

Do elementov prav tako dostopamo z uporabo DOT operatorja [todo], 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. Ravno zaradi tega ne moremo odmika izračunati na enak način kot pri tabelah, kjer indeks pomnožimo z velikostjo podatkovnega tipa. Pri terkah zato odmik posameznega elementa od začetnega naslova terke izračunamo tako, da seštejemo velikosti podaktovnih tipov vseh elementov pred željenim elementom:

$$offset(element_i) = sum(size(element_0), size(element_1), ..., size(element_{n-1}))$$

Terke se pogosto uporabne v situacijah, kjer želimo, da funkcija vrne več vrednosti hkrati.

```
func evaluateBoard(_ ticTacToe: [Char]) (Int,
Int) {
    return (1, 10)
}
let eval = evaluateBoard(['X', ' ', 'X'])
print(eval.0) # '1'
print(eval.1) # '10'
```

Program 4.9: Funkcija, ki vrača dve vrednosti hkrati.

4.5 Razredi

Razred je sestavljena podatkovna struktura, ki, podobno kot terka, v sebi hrani spremenljivke različnih podatkovnih tipov. 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*. Do privatnih članov ne moremo dostopati izven razreda.

Metodam lahko dodamo 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*.

Poleg atributov in metod lahko razred vsebuje tudi definicije gnezdenih razredov, enumeracij ter vmesnikov.

V pomnilniku je instanca razreda (objekt) predstavljena podobno kot terka, t.j. kot tabela z vrednostmi različnih podatkovnih tipov. Zato tudi odmik elementov računamo na enak način.

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.

Zato se s tem ne ukvarjamo v razreševanju imen, ampak problem prepustimo razreševanju tipov. Razreševanje tipov zato delimo na dve pod-fazi (dva sprehoda po AST). V prvem sprehodu nas zanimajo samo podatkovni tipi definicij, izraze zaenkrat pustimo pri miru. Ko zberemo vse potrebne informacije o podatkovnih tipih članov razreda, lahko zgradimo razredni tip, ki predstavlja razred in njegove definicije.

Sedaj imamo dovolj informacij, da razrešimo imena znotraj DOT operatorja, kar storimo v drugem sprehodu.

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.

4.5.2 Instančne funkcije

Razred ima lahko v sebi definirane funkcije, ki se jim v OOP žargonu pogosto reče *metode*. Metode se od funkcij, ki niso definirane v razredu, razlikujejo v tem, da vsebujejo impliciten parameter, ki je referenca na objekt, ki metodo kliče. Ta parameter prevajalnik v metodo vstavi avtomatsko.

V C++ in Javi se ta parameter imenuje ***this***, v Pythonu, Swiftu in 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. Delno to stori tekom razreševanja imen, ko definiciji funkcije vstavi nov parameter, delno pa tekom razreševanja tipov, ko parametru nastavi tip.

4.5.3 Konstruktorji

Konstruktorji so posebne vrste metod, njihova naloga pa je kreacija objektov ter inicializacija atributov. Podobno kot ostalim metodam razreda tudi konstruktorju prevajalnik implicitno vstavi *self* parameter.

Naloga konstruktorja je, da na kopici, t.j. delu pomnilnika, kamor shranjujemo dinamično alicirane objekte (za razliko od sklada, kamor shranjujemo statične), alocira prostor, kjer 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 vstavi na začetek kode konstruktorja MALLOC gradnik, ki bo rezerviral prostor za objekt. Nato bo prevajalnik vrnjen naslov shranil 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 ga 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
        }
    }
```

Program 4.10: 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
```

Program 4.11: 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
        self.y = 100
    }
}

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

Program 4.12: 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()
```

Program 4.13: 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* tip, ki hrani podatke o statičnih definicijah razreda. 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 privzeto smatrajo 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'
```

Program 4.14: Enumeracija znotraj razreda.

Da lahko dostopamo do konstruktorjev znotraj gnezdenih razredov, jih prevajalnik avtomatično potisne v seznam statičnih metod razreda.

```
class Atheris {
    class NestedClass {
        var x = 20
    }
}
print(Atheris.NestedClass().x) # '20'
```

Program 4.15: Gnezden razred.

4.6 Dedovanje in polimorfizem

Dedovanje in polimorfizem sta, poleg enkapsulacije in abstrakcije, najpomembnejša koncepta OOP programiranja. 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.

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

[todo] Dynamic dispatch je proces izbire, katera implementacija polimorfne operacije naj se izvede v času **izvajanja** programa. Običajno lahko pri klicih funkcij že v času prevajanja izberemo, katera implementacija funkcije naj se kliče, saj obstaja samo ena. Pri razredih nastane problem, kadar dovolimo, da imajo lahko dedujoči se razredi svoje implementacije funkcij (oz. metod), kot je razvidno v spodnjem primeru:

```
class Fruit {
    func kind() {
        print("I am a Fruit!")
    }
}
class Apple: Fruit {
    override func kind() {
        print("I am an Apple!")
    }
}
class Orange: Fruit {
    override func kind() {
        print("I am an Orange!")
    }
}
```

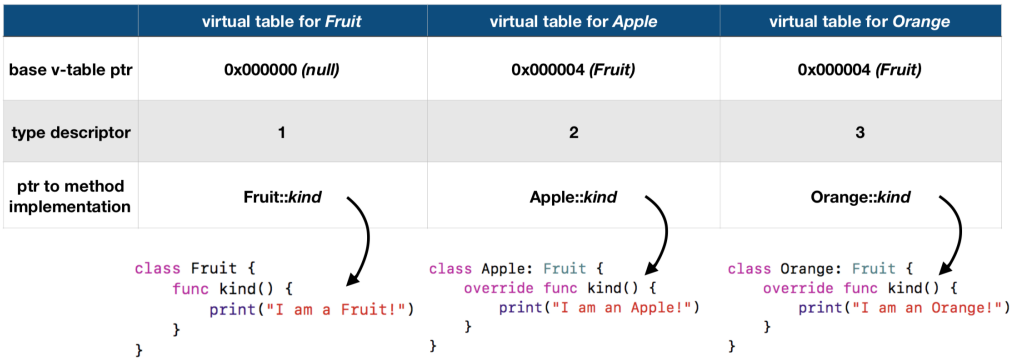
Program 4.16: Več implementacij za isto funkcionalnost.

Problem je možno rešiti na več načinov. Rešitev prevajalnika za Atheris temelji na rešitvi, ki jo ima tipično C++, t.j. z uporabo podatkovne strukture ***v-table***.

Virtualna tabela

Virtualna tabela je podatkovna stuktura, ki vsebuje kazalce na funkcije implementirane v danem razredu. Prevajalnik za vsak razred zgenerira in shrani v pomnilnik unikatno virtualno tabelo.

Vsak objekt v pomnilniku, poleg svojih atributov, hrani še kazalec na lastno virtualno tabelo.



Slika 4.2: Virtualne tabele za dane razrede iz 4.16.

Virtualna tabela vsebuje kazalce na naslove, kjer se nahajajo vse ne-statične 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 in je unikaten za vsak tip.

4.6.2 Realizacija

Problem rešimo tako, da kazalec na metodo shranimo v virtualno tabelo na tisti indeks, na katerem je motoda definirana v razredu. Prevajalnik za klic funkcije zgenerira vmesno kodo, ki izračuna naslov metode 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 in

$$address(method_i)$$

naslov metode na indeksu i.

4.7 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'
```

Program 4.17: Primer uporabe vmesnikov.

4.8 Abstraktni razredi

Programski jezik Atheris dopušča zmožnost deklariranja abstraktnih razredov z uporabo rezervirane besede *abstract*. Značilnost abstraktnih razredov je to, da ne moremo kreirati njihov instanc neposredno. Abstraktni razredi s tem dopuščajo možnost definiranja abstraktnih metod, ki jih mora vsak dedujoči-se ne-abstraktni razred implementirati. V tem pogledu so si abstraktni razredi in vmesniki zelo podobni. Abstraktne metode definiramo tako kot razrede z uporabo ključne besede *abstract*.

```
abstract class Shape {
    var len: Int
    abstract func area() Int
}

class Square: Shape {
    init(_ len: Int) {
        self.len = len
    }
    func area() Int {
        return self.len * self.len
    }
}

class Rect: Shape {
    var height: Int
    init(width: Int, height: Int) {
        self.len = width
        self.height = height
    }
    func area() Int {
        return self.len * self.height
    }
}

var shape: Shape
shape = Square(10)
print(shape.area()) # '100'
shape = Rect(width: 5, height: 10)
```

```
print(shape.area())  # '50'
```

Program 4.18: Primer uporabe abstraktnih razredov.

Naloga prevajalnika je, da zagotovi, da so v razredu implementirane vse abstraktne metode, ki so definirane v hiearhiji razredov. Če ne želimo implementirati vseh metod abstraktnega razreda, mora biti razred označen kot abstrakten.

Definirajmo abstraktni podatkovni strukturi seznam in sklad, ki definirata ustrezne abstraktne operacije:

```
abstract class AbstractList {
    private var storage: [Int]
    abstract func append(_ val: Int)
    abstract func remove(at index: Int)
}

abstract class AbstractStack: AbstractList {
    abstract func push()
    abstract func pop()
}
```

Razred, ki želi dedovati od *AbstractStack*, mora poleg metod *push* in *pop* implementirati tudi obe metodi v *AbstractList* razredu:

```
class Stack: AbstractStack {
    func append(_ val: Int) {}
    func remove(at index: Int) {}
    func push() {}
    func pop() {}
}
```

Program 4.19: Implementacija abstraktne podaktovne strukture.

4.9 Instance of operator

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

V programskem jeziku Atheris se operator imenuje *is*.

```
var object: A
```

```
object = B()
print(object is A) # 'true'
print(object is B) # 'true'
print(object is C) # 'false'
```

Program 4.20: Uporaba operatorja *is* za razrede iz sheme 4.2.

Ker je lahko razredna hiearhija obsežna, je potrebno preverjanje izvesti rekurzivno po celotni hiearhiji.

```
todo
```

Program 4.21: Algoritem za izračun ali je objekt instanca danega razeda.

4.10 Pretvarjanje tipov

Mehanizem pretvarjanja iz enega tipa v drug (angl. *type casting*) deluje po zelo podobnem principu kot *as* operator. Razlikujeta se predvsem v njunem rezultatu. Rezultat *as* operatorja je *true* ali *false*, medtem ko je pri neuspešnem pretvarjanju rezultat *null*, pri uspešnem pa objekt, pretvorjen v željen tip. 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
    print("cast failed")
}
else {
    print("cast succeeded") # izvede se 'else'
blok
}
```

Program 4.22: Pretvorba za razrede iz sheme 4.2.

4.11 Razširitve

Razširitev (angl. *extension*) je mehanizem, ki omogoča dodajanje funkcionalnosti obstoječim razredom. Razred lahko razširimo z metodami, enumeracijami in razredi.

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

Program 4.23: Razširitev razreda A.

Implementacija razširitev ni ravno zapletena; prevajalnik v fazi razreševanja tipov tekom prvega sprehoda po AST za vsako novo definicijo v razširitvi izračuna njen podatkovni tip, zagotovi, da razred ne vsebuje definicije z istim imenom, ter mu jo na koncu vstavi.

4.11.1 Razširitev z vmesnikom

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

```
class Collection {}
interface Iterable {
    func next() Int
}
extension Collection: Iterable {
    func next() Int {
        return 10
    }
}
let iterable = Collection()
print(iterable.next())
```

Program 4.24: Razširitev z vmesnikom.

4.12 Type inference

[todo] 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.

V primeru, da definicija spremenljivke nima eksplicitno navedenega podatkovnega tipa, prevajalnik najprej izračuna podatkovni tip izraza na desni strani prirejevalnega operatorja, ter ga nato dodeli 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
```

Program 4.25: Avtomatično prepoznavanje podatkovnih tipov.

Prepoznavanje tipov postane kompleksnejše pri tabelah, kadar njeni elementi niso istih tipov. Prevajalnik tekom razreševanja tipov vedno poišče najmanjši možen tip, ki je skupen vsem elementom 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 tudi tipa *Any* .

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

Program 4.26: Elementi tabele s skupnim nadrazredom.

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

Program 4.27: Elementi tabele nimajo skupnega tipa.

Poglavje 5

Meritve in testiranje

Delovanje programskega jezika Atheris je testirano na znanih algoritmih - *bubble sort*, *quick sort* in *fibonacci*. Testni primeri so izbrani tako, da preko njih preizkusimo delovanje čim večih funkcionalnosti jezika. Kot zanimivost je izmerjena tudi hitrost izvajanja programov in primerjana z osnovnim jezikom, za referenco pa je izmerjena še hitrost delovanja algoritmov v Pythonu (vsi rezultati v nadaljevanju so prikazani v sekundah).

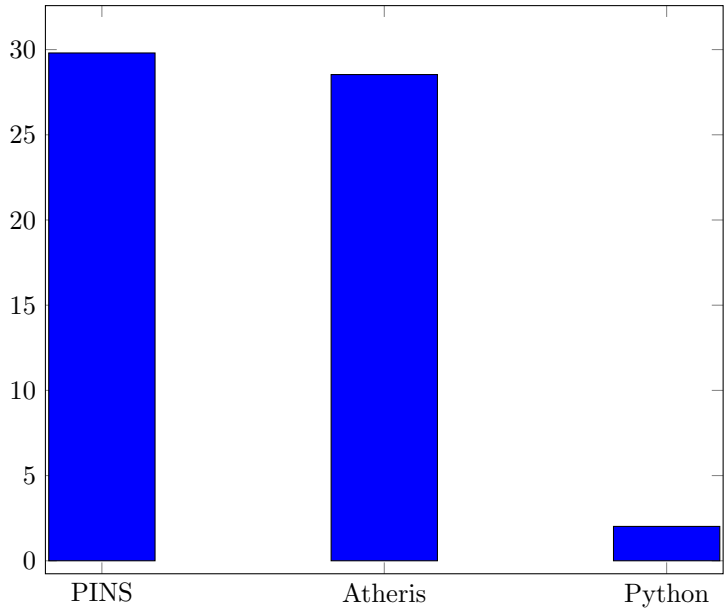
5.1 Meritve

5.1.1 Bubble sort

Z bubble sort algoritmom preverimo:

- dostopanje do pomnilnika (do elementov v tabeli na nekem indeksu)
- delovanje zank
- klicanje funkcij, prenos parametrov

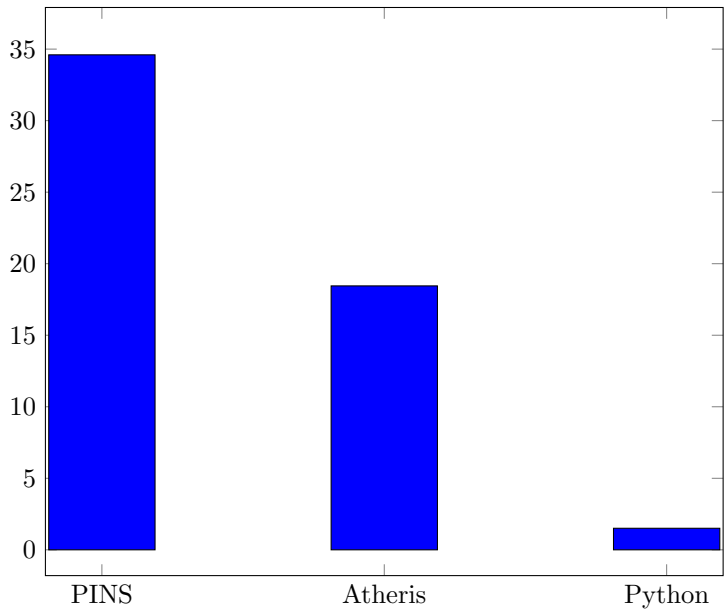
Algoritem izvedemo 100.000-krat na številih *8, 0, 3, 9, 2, 14, 10, 27, 1, 5, 8, -1, 26* in pri tem merimo čas izvajanja.



Rezultati niso presenetljivi, PINS in Atheris oba potrebujeta približno enako časa, medtem ko je Python (po pričakovanjih) bistveno hitrejši.

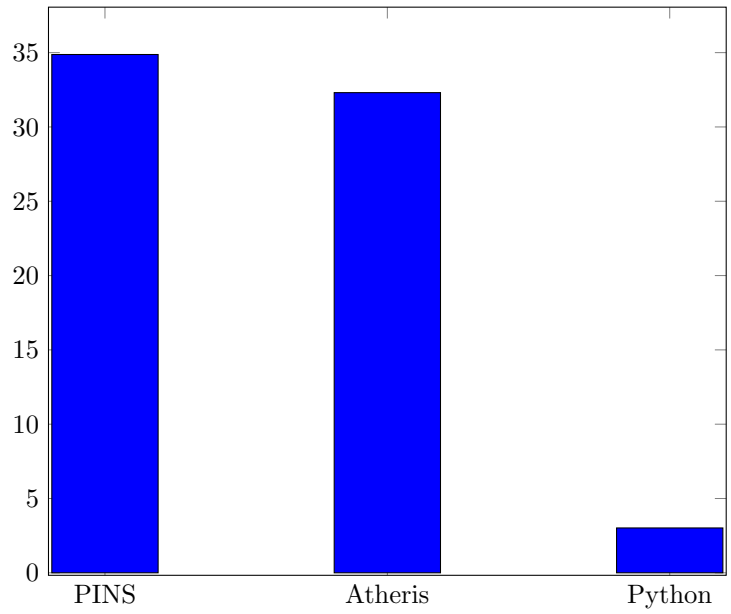
5.1.2 Quick sort

Z uporabo quick sort algoritma, poleg prej naštetih funkcionalnosti, stestiramo še delovanje rekurzije. Algoritem je izveden pod enakimi pogoji kot bubble sort.



5.1.3 Izračun n-tega fibonaccijevega števila

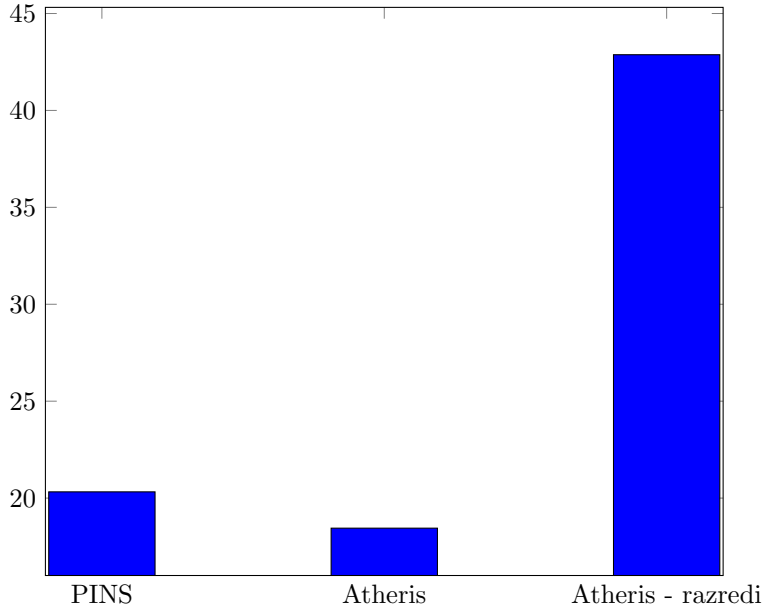
Zadnji testni algoritem je izračun n-tega fibonaccijevega števila z uporabo rekurzivne enačbe $fib(n) = fib(n-1) + fib(n-2); fib(0) = 1; fib(1) = 1$. Računamo $fib(35)$.



5.1.4 Quicksort s polimorfnimi operacijami

V razdelku 4.6.1 je podrobneje opisano delovanje polimorfnih operacij, kot so *dynamic dispatch*, *instance of* operator ter pretvarjanje tipov. Naštete operacije nam omogočajo kar nekaj priročnih funkcionalnosti, ki nam pri razvoju programske opreme pogosto pridejo prav. A pod določeno ceno, saj je izvedba teh operacij relativno počasna.

Zanimalo me je, kakšna je dejansko cena, ki jo moramo plačati za njihovo delovanje. Pripravljen je primer, ki testira prav to. Gre za implementacijo quick sort algoritma, ki se izvaja nad identičnimi vhodnimi podatki kot prejšnji primeri, s to razliko, da so števila ovita v objekte. Izvajanje primerjanja dveh objektov poteka z uporabo naštetih polimorfnih operacij.



Po pričakovanjih se izkaže, da je izvajanje res občutno počasnejše; in sicer skoraj 230%.

Poglavje 6

Sklepne ugotovitve

V okviru diplomske naloge so predstavljeni programski jeziki in prevajalniki, sodobne prakse načrtovanja in implementacije prevajalnikov ter njihova zgradba. Podrobneje sta opisana programska jezika PINS in Atheris, poudarek na slednjem, ki je nadgradnja prvega. Razširitve, ki jih prevajalnik za Atheris vsebuje napram prevajalniku za PINS, in njihova implementacija, so tekom naloge podrobneje opisane.

Programski jezik Atheris omogoča uporabo objektno usmerjenih gradnikov, ki jih vidimo v večini sodobnih objektnih jezikov; to so enumeracije, terke, razredi in vmesniki. Atheris podpira tudi razširitve razredov, podobno kot to lahko počnemo v Swiftu.

Za diplomsko nalogo je bilo načrtovano še kar nekaj dodatnih funkcionalnosti, ki zaenkrat še niso implementirane. Med njih spadajo generični podatkovni tipi in generične funkcije, opcijski podatkovni tipi in *closure* [1] funkcije. Njihova vgradnja bo prišla na vrsto v nadaljnem razvoju prevajalnika.

V razdelku 2.2.3 je omenjeno, da lahko semantična analiza stori več, kot le razreši imena in podatkovne tipe. Prevajalnik za Atheris vsebuje še tretjo semantično fazo, ki se imenuje *preverjanje inicializiranosti*. Naloga te faze je, da za vsako spremenljivko zagotovi, da ji je bila pred njeno uporabo nastavljena vrednost. Poleg tega prepreči, da lahko konstantam vrednost priredimo natanko enkrat. Faza je trenutno še v povojih in zaenkrat ne deluje tako, kot bi si želeli, zato ni opisana tekom diplomske naloge.

Tekom te diplomske naloge se osredotočamo predvsem na *front-end* del prevajalnika, kjer se izvede analiza izvorne kode in sinteza v vmesno. Preskočen je celoten zaledni del prevajalnika (*back-end*). Naloga zalednega dela je opti-

[1] todo razlaga

mizacija vmesne in generiranje dejanske strojne kode za ciljno arhitekturo. V prihodnje je cilj vgradnja v prevajalnik za Atheris izjemno močen zaledni sistem **LLVM** [2], na katerem temeljijo tudi zelo zmogljivi C++, Objective-C in Swift prevajalniki.

Poglavje 7

Priloge

7.1 Atheris

7.1.1 Bubble sort

```
func bubbleSort(_ list: [Int], size: Int) Int {
    var i = 0
    while i < size {
        var j = 0
        while j < size {
            if list[i] < list[j] {
                let tmp = list[i]
                list[i] = list[j]
                list[j] = tmp
            }
            j = j + 1
        }
        i = i + 1
    }
}

var list = [ 8, 0, 3, 9, 2, 14, 10, 27, 1, 5, 8, -1,
   26 ]
var i = 0
while i < 100000 {
    bubbleSort(list, size: list.count)
    list[0] = 8
```

```
    list[1] = 0
    list[2] = 3
    list[3] = 9
    list[4] = 2
    list[5] = 14
    list[6] = 10
    list[7] = 27
    list[8] = 1
    list[9] = 5
    list[10] = 8
    list[11] = -1
    list[12] = 26
    i = i + 1
}
```

7.1.2 Quick sort

```
func quickSort(_ list: [Int], low: Int, high: Int) {
    func partition(_ list: [Int], low: Int, high: Int)
    Int {
        let pivot = list[low]
        var i = low - 1
        var j = high + 1

        while true {
            while true {
                j = j - 1
                if list[j] <= pivot {
                    break
                }
            }

            while true {
                i = i + 1
                if list[i] >= pivot {
                    break
                }
            }
```

```
            if i < j {
                let tmp = list[i]
                list[i] = list[j]
                list[j] = tmp
            }
            else {
                return j
            }
        }
    }

    if low < high {
        let p = partition(list, low: low, high: high)
        quickSort(list, low: low, high: p)
        quickSort(list, low: p + 1, high: high)
    }
}

var list = [ 8, 0, 3, 9, 2, 14, 10, 27, 1, 5, 8, -1,
   26 ]
var i = 0
while i < 100000 {
    quickSort(list, low: 0, high: list.count - 1)
    list[0] = 8
    list[1] = 0
    list[2] = 3
    list[3] = 9
    list[4] = 2
    list[5] = 14
    list[6] = 10
    list[7] = 27
    list[8] = 1
    list[9] = 5
    list[10] = 8
    list[11] = -1
    list[12] = 26
    i = i + 1
}
```

7.1.3 Fibonacci

```
func fib(_ n: Int) Int {
    if n < 2 {
        return 1
    }
    else {
        return fib(n - 1) + fib(n - 2)
    }
}
fib(35)
```

7.1.4 Quick sort nad razredi

```
class Comparable {
    func compare(with object: Any) Int {
        if object is Comparable {
            let comparable = object as Comparable
            return self.value() - comparable.value()
        }
        else {
            return 0
        }
    }

    func value() Int {
        return 0
    }

    func toString() {
    }
}

class Integer: Comparable {
    var val: Int

    init(_ val: Int) {
        self.val = val
    }
```

```
    override func value() Int {
        return self.val
    }

    override func toString() {
        print(self.val)
    }
}

class Decimal: Comparable {
    var val: Int

    init(_ val: Int) {
        self.val = val
    }

    override func value() Int {
        return self.val
    }

    override func toString() {
        print(self.val)
    }
}

func partition(_ list: [Comparable], low: Int, high:
   Int) Int {
    let pivot = list[low]
    var i = low - 1
    var j = high + 1

    while true {
        while true {
            j = j - 1
            let el = list[j]
            if el.compare(with: pivot) <= 0 {
                break
```

```
            }
        }

        while true {
            i = i + 1
            let el = list[i]
            if el.compare(with: pivot) >= 0 {
                break
            }
        }

        if i < j {
            let tmp = list[i]
            list[i] = list[j]
            list[j] = tmp
        }
        else {
            return j
        }
    }
}

func quicksort(_ list: [Comparable], low: Int, high:
    Int) {
    if low < high {
        let p = partition(list, low: low, high: high)
        quicksort(list, low: low, high: p)
        quicksort(list, low: p + 1, high: high)
    }
}

var list = [ Integer(8), Decimal(0), Integer(3),
    Integer(9), Decimal(14), Decimal(10), Integer(27),
    Decimal(1), Decimal(5), Integer(8), Decimal(-1),
    Decimal(26) ]

var i = 0
while i < 100000 {
```

```
    quicksort(list, low: 0, high: list.count - 1)
    i = i + 1
}
```

7.2 PINS

7.2.1 Bubble sort

```
fun bubble(tab:arr[13] integer) : integer = (
    {for i = 0, 13, 1 :
        {for j = 0, 13, 1:
            {if tab[j] > tab[i] then
                (
                {tmp = tab[j]},
                {tab[j] = tab[i]},
                {tab[i] = tmp}, 0
                )
                { where var tmp : integer}
            }
        }
    },
    1
) { where var i : integer; var j : integer };

var tab: arr[13] integer;
var count: integer;

fun main(i:integer) : integer = (
    {while count < 100000:
        (
        {tab[0] = 8},
        {tab[1] = 0},
        {tab[2] = 3},
        {tab[3] = 9},
        {tab[4] = 2},
        {tab[5] = 14},
        {tab[6] = 10},
        {tab[7] = 27},
        {tab[8] = 1},
```

```
        {tab[9] = 5},
        {tab[10] = 8},
        {tab[11] = -1},
        {tab[12] = 26},
        bubble(tab),
        {count = count + 1}
        )
    },
    0
)
```

7.2.2 Quick sort

```
fun partition(tab: arr[13] integer, low: integer, high
    : integer) : integer = (
    {pivot = tab[low]},
    {i = low - 1},
    {j = high + 1},
    {loop = true},
    {while loop == true:
        (
        {p = true},
        {while p == true:
            (
            {j = j - 1},
            {if tab[j] <= pivot then
                {p = false}
            }
            )
        },
        {p = true},
        {while p == true:
            (
            {i = i + 1},
            {if tab[i] >= pivot then
               {p = false}
            }
            )
        },
```

```
        {if i < j then
            (
            {tmp = tab[j]},
            {tab[j] = tab[i]},
            {tab[i] = tmp},0
            )
        { where var tmp : integer}
        else ({loop = false})
        }
        )
    },
    j
) { where var pivot: integer; var i: integer; var j:
   integer; var p: logical; var loop: logical };

fun quickSort(tab: arr[13] integer, low: integer, high
   : integer) : integer = (
    {if low < high then
        (
        {p = partition(tab, low, high)},
        quickSort(tab, low, p),
        quickSort(tab, p + 1, high)
        )
    },
    0
) { where var p: integer }

var tab: arr[13] integer;
var count: integer;

fun main(i:integer) : integer = (
    {while count < 100000:
        (
        {tab[0] = 8},
        {tab[1] = 0},
        {tab[2] = 3},
        {tab[3] = 9},
        {tab[4] = 2},
```

```
        {tab[5] = 14},
        {tab[6] = 10},
        {tab[7] = 27},
        {tab[8] = 1},
        {tab[9] = 5},
        {tab[10] = 8},
        {tab[11] = -1},
        {tab[12] = 26},
        quickSort(tab, 0, tab.length - 1),
        {count = count + 1}
        )
    },
    0
)
```

7.2.3 Fibonacci

```
fun fib(x : integer) : integer = (
    { if x < 2 then
        ({rez = 1})
    else
        ({rez = fib(x - 1) + fib(x - 2)})
    },
    rez
) {where var rez : integer};

fun main(i:integer) : integer = (
    {i = fib(35)},
    0
)
```

7.3 Python

7.3.1 Bubble sort

```
def bubble(list):
    for i in range(len(list)):
        for j in range(len(list)):
            if list[i] < list[j]:
                tmp = list[i]
```

```
                list[i] = list[j]
                list[j] = tmp

startTime = time.time()

list = [ 8, 0, 3, 9, 2, 14, 10, 27, 1, 5, 8, -1, 26 ]
for i in range(100000):
    bubble(list)
    list = [8, 0, 3, 9, 2, 14, 10, 27, 1, 5, 8, -1,
   26]
print(time.time() - startTime)
```

7.3.2 Quick sort

```
import time

def partition(list, low, high):
    pivot = list[low]
    i = low - 1
    j = high + 1

    while True:
        while True:
            j = j - 1
            if list[j] <= pivot:
                break

        while True:
            i = i + 1
            if list[i] >= pivot:
                break

        if i < j:
            tmp = list[i]
            list[i] = list[j]
            list[j] = tmp
        else:
            return j
```

```
def quickSort(list, low, high):
    if low < high:
        p = partition(list, low, high)
        quickSort(list, low, p)
        quickSort(list, p + 1, high)

startTime = time.time()

list = [ 8, 0, 3, 9, 2, 14, 10, 27, 1, 5, 8, -1, 26 ]
for i in range(100000):
    quickSort(list, 0, len(list) - 1)
    list = [8, 0, 3, 9, 2, 14, 10, 27, 1, 5, 8, -1,
   26]
print(time.time() - startTime)
```

7.3.3 Fibonacci

```
import time

def fib(n):
    if n < 2:
        return 1
    return fib(n - 1) + fib(n - 2)

startTime = time.time()
fib(35)
print(time.time() - startTime)
```

Literatura

[1] Atheris chlorechis. Dosegljivo: https://qph.ec.quoracdn.net/main-qimg-6ee9bd139819545a1af652b204b5dc2e]. [Dostopano: 9.1.2018].

[2] Llvm. Dosegljivo: https://en.wikipedia.org/wiki/LLVM. [Dostopano: 14.1.2018].

[3] Modern compiler structure. Dosegljivo: http://anstoques.blogspot.si/2013/05/show-different-phases-of-compiler-with.html. [Dostopano: 9.1.2018].

[4] Programming language popularity index. Dosegljivo: https://www.tiobe.com/tiobe-index/. [Dostopano: 9.1.2018].

[5] Stack frames. Dosegljivo: https://loonytek.com/2015/04/28/call-stack-internals-part-1/. [Dostopano: 9.1.2018].

[6] 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: 9.1.2018].

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