

UNIVERZA NA PRIMORSKEM
Fakulteta za matematiko, naravoslovje in informacijske tehnologije

Iztok Savnik

SKRIPTA ZA PREDMET

**PROGRAMIRANJE II:
KONCEPTI PROGRAMSKIH
JEZIKOV**

ŠTUDIJSKI PROGRAM
RAČUNALNIŠTVO IN INFORMATIKA, 1. STOPNJA
UP FAMNIT

Koper, 2013.

Kazalo

To so zapiski za predmet Programiranje II na dodiplomskem študijskem programu Računalništvo in informatika, FAMNIT, Koper.

Del zapiskov vsebuje prevode sekcij iz knjige Dveloppement d'applications avec Objective Caml (angl. verzija: Developing Applications With Objective Caml) avtorjev Emmanuel Chailloux, Pascal Manoury, Bruno Pagano, O'REILLY & Associates, France.

I.Savnik, december 2012, Koper.

Poglavje 1

UVOD

Koncepti programskih jezikov so abstrakcije s katerimi predstavimo strukturo in obnašanje sistemov, ki jih s programom modeliramo. Koncepti programskih jezikov so realizirani v okviru konkretnih gradnikov programskih jezikov kot tudi v tehnikah za implementacijo programskih jezikov.

Osnovni koncepti uporabljeni v konkretnem programskem jeziku določajo *model jezika*. Poglejmo si pomembnejše modele programskih jezikov in osnovne koncepte, ki so značilni za posamezen model. Imperiativni programski jeziki temeljijo na podatkovnih in postopkovnih abstrakcijah kot so na primer spremenljivke, funkcije in procedure. Funkcijski jeziki so osnovani na matematični abstrakciji funkcije, ki je uporabljena v lambda računu. Objektni jeziki so zasnovani na osnovi konceptov kot so objekti in razredi ter dedovanje. Logični programski jeziki temeljijo na predikatnem računu oz. logiki.

Koncepte programskih jezikov lahko razdelimo tudi glede na *postopkovne* in *podatkovne* abstrakcije. Postopkovni gradniki programskih jezikov so iteracija, dekompozicija kode v funkcije in procedure, rekurzija ter drugi. Podatkovne abstrakcije so v programskih jezikih uporabljene za organizacijo in strukturiranje podatkov in programske kode. Osnovne podatkovne abstrakcije lahko vidimo kot primere uporabe matematičnih gradnikov kot so na primer n-terice, kartezijski produkt in unije.

Študij konceptov programskih jezikov pripomore k poznavanju izraznih zmožnosti gradnikov programskih jezikov in omogoča pravilno uporabo teh gradnikov. Dobro poznavanje konceptov programskih jezikov omogoča tudi korektno izbiro gradnikov pri modeliranju programskega sistema.

Poznavanje uveljavljenih tehnik za implementacije posameznih konceptov programskih jezikov, kot so na primer implementacija funkcij in rekurzije, omogoča pisanje bolj učinkovitih programov. Brez poznavanja implementacije gradnikov programskih jezikov ni mogoče učinkovito uporabljati implementacije gradnikov kot se je težje izogniti slabi uporabi gradnika.

1.1 Programski jeziki

Programski jezik je konceptualno orodje definirano v obliki jezika s katerim lahko zasnujemo, izrazimo in realiziramo računalniške programe. Idealen programski jezik nam pomaga pri implementaciji programskih sistemov tako, da nam omogoča oblikovanje in izražanje idej o programskem sistemu na najbolj naraven in učinkovit način. Velikokrat to pomeni, da je model jezika blizu človeškemu razmišljanju, ki ga uporablja pri reševanju problemov.

Po drugi strani nam sam programski jezik definira abstrakcije s katerimi lahko učinkovito in elegantno konstruiramo programske sisteme. Programski jezik nam torej definira konceptualno vesolje¹ v okviru katerega lahko predstavimo svoj sistem. Novinci, ki abstrakcij še ne poznajo se morajo preden bi lahko začeli programirati naučiti nov jezik konceptov s katerimi se lahko lepo in učinkovito izražajo pri reševanju problemov na določenem področju. Koncepti programskih jezikov uporabljeni v določenem programskem jeziku novinca torej učijo o konceptualnem vesolju, ki ga bo imel na razpolago pri pisanju programov.

Koncepti programskih jezikov predstavljajo koncepte za katere smo skozi raziskave in poskuse v zadnjih nekaj desetletjih ugotovili, da so najbolj primerni za programiranje informacijskih sistemov. Prenekateri jezikovni koncept programskih jezikov ima kompleksno definicijo, ki zahteva od programerjev dobro poznavanje koncepta, poznavanje formalnega ozadja koncepta, poznavanje možnih implementacij koncepta in izkušnje iz uporabe danega koncepta v konkretnih programskih okoljih.

...

1.2 Koncepti programskih jezikov

Koncepti programskih jezikov bodo predstavljeni na naslednji način. Najprej bomo identificirali posamezne koncepte. Pri tem bomo velikokrat uporabljali konkretne programske jezike, ki predstavljajo izvor predstavljenega koncepta. Programski jeziki, ki jih bomo uporabljali za predstavitev osnovnih idej so: Lisp, C, C++, Java, Ocaml, SML in drugi.

Vsak koncept bo predstavljen formalno z uporabo matematičnih orodij. Prikazali bomo formalne osnove konceptov, ki so pogosto zelo blizu matematičnim gradnikom. Na primer, koncept funkcije je blizu matematičnem pogledu na funkcije. Prav tako so blizu matematičnim pogledom podatkovne strukture, na primer, n-terice, vektorji, sezname ali polja.

Za vsakega od predstavljenih konceptov bomo podali množico primerov. Ocaml bo uporabljen kot referenčni jezik. Razlogi za izbiro Ocaml kot prvega jezika bodo predstavljeni v nadaljevanju. V primeru, da predstavljen koncept ni dobro pokrit z Ocaml

¹“A good programming language is a conceptual universe for thinking about programming”. Alan Perlis.

ga bomo predstavili v jeziku, ki je za to najbolj primeren. Pogosto bomo primerjali različne implementacije gradnikov v različnih programskih okoljih.

OCaml je izbran za jezik v katerem bo predstavljena večina primerov iz večih razlogov. Prvič, Meta-Language (ML) ki predstavlja osnovo OCaml je eden od bolje preučeni programskih jezikov. ML je v zadnjem času velikokrat uporabljen kot osnova za teoretične študije programskih jezikov.

Jedro programskega jezika OCaml je Lambda račun, ki ga bomo predstavili v naslednjem poglavju. Lambda račun je formalen jezik, ki je uporabljen za študij osnov matematike kot tudi za enega od osnovnih formalnih jezikov v teoriji programskih jezikov. Osnovni gradnik Lambda računa je λ -abstrakcija s katero lahko modeliramo matematične funkcije. V osnovi je OCaml funkcijski programski jezik, ki temelji na uporabi funkcij kot osnovnega gradnika jezika.

OCaml vsebuje poleg funkcijskega modela tudi imperativni model programskih jezikov. Imperativni programski jeziki so osnovani na uporabi spremenljivk s katerimi opišemo stanje sistema, na iteracijskih gradnikih ter na konceptu procedure. Imperativni gradniki OCaml so bili definirani zato, da je mogoče učinkovito implementirati sisteme za katere je imperativna rešitev najbolj primerna. Sisteme, ki uporabljajo matrike, na primer, je najbolj učinkovito implementirati z jezikom, ki vsebuje matrike kot osnoven gradnik jezika.

Programski jezik OCaml vsebuje tudi vse gradnike objektno-usmerjenih programskih jezikov. Objektni model programskih jezikov omogoča predstavitev modeliranega sistema z objekti, ki so grupirani v razrede, prototipne objekte od katerih instance (objekti) podedujejo strukturo in obnašanje. Koncepti objektnih jezikov so definirani zelo podobno drugim objektno-usmerjenim jezikom kot so na primer Java ali C++. Praviloma bomo primerjali implementacije posameznih gradnikov v različnih programskih okoljih.

Povrh naštetega vsebuje OCaml tudi zelo izrazen jezik za definicijo in uporabo modulov. Moduli omogočajo logično grupiranje programske kode v module, ki predstavljajo enoto prevajanja. Gradnike jezika za delo z moduli v OCaml lahko primerjamo z gradniki programskega jezika Modula 2, ki je eden od prvih jezikov definiran na osnovi abstrakcij modula.

Ločimo med jezikovnimi in implementacijskimi koncepti programskih jezikov. *Jezikovni koncepti* so pomembnejše abstrakcije uporabljene v programskih jezikih. Na primer, iteracija je abstrakcija s pomočjo katere lahko učinkovito predstavimo iterativno izvajanje delov kode programa. Primeri jezikovnih konceptov programskih jezikov so še funkcija, polimorfizem, razredi, moduli in drugi.

Implementacijski koncepti programskih jezikov predstavljajo pomembne pristope k implementaciji programskih jezikov. Na primer, organizacija spomina in delo s spominom programskega jezika lahko bistveno vpliva na način programiranja in na izvedbo programskega sistema. Prav tako vplivajo na način izvedbe programskega sistema drugi pomembni koncepti implementacije programskega jezika kot so mehanizmi za

prenos parametrov funkcij in procedur, izvedba imenskih prostorov programskega jezika, implementacija rekurzije, preverjanje tipov, implementacija razredov in instanc, in drugi.

Pri predstavitvi konceptov programskih jezikov bomo velikokrat primerjali definicijo in implementacijo posameznih gradnikov v različnih programskih jezikih. S primerjavo bomo ...

1.3 Meta-jezik

Meta-Language ML

- Meta Language (ML) je nastal nenamenoma pri izdelavi programa dokazovanje izjav s pomočjo LCF - Logic of computable functions.
- LCF je verzija lambda računa.
- Projekt se je izvajal v skupini za umetno ineteligenco na Univerzi Stanford. - Vodja projekta je bil Robin Milner.
- V okviru LCF je Meta-Language služil kot jezik za zapis dokazov; običajno bi spreminjali vsebino datoteke za izdelavo boljšega dokaza.
- Pomembno je tudi, da zdaj tvorijo nasledniki LCF drevo znanja o sklepanju s pomočjo računalnikov.

Lambda račun je osnoven formalizem za študij programskih jezikov.

- LCF oz. lambda račun tvori osnovni formalizem za vejo računalništva - teorijo programskih jezikov.
- Teorija programskih jezikov uporablja sklepanje za izpeljevanje tipov, evaluacijo izrazov, dokazovanje lastnosti programov, itd.
- Z razširjenim LCF lahko predstavimo statično in dinamično semantiko jezika. (?)
- Lahko preverimo ključne lastnosti jezikov kot so determinističnost evaluacije, enoličnost evaluacije, strogi tipi, itd.
- Lambda račun lahko vidimo kot izvorni programski jezik kot tudi osnovni jezik za opis semantike jezikov (?)
- Lambda račun si bomo ogledali v prvem delovnem poglavju.

Objektni Caml je referenčni jezik.

- Jedro Caml je čisti lambda račun.
- Eden izmed boljše teoretično obdelanih programskih jezikov.
- Sistem za delo s tipi. Izpeljava tipov. Caml ima stroge tipe.
- Vsebuje imperativne gradnike.

- Vsebuje parametrični polimorfizem.
- Funkcijski pogled na bolj kompleksne gradnike.
- Vsebuje module in funktorje.
- Lepa zasnova objektov in razredov.
- Vsebuje izjeme. Možne hierarhije izjem.
- Omogoča več programskih modelov.
- Imperativni + funkcijski + objektni programski model.

1.4 Uvrstitev področja v računalništvo

Osnovni koncepti programskih jezikov so zanimivi tudi za druga področja računalništva, ki tvorijo jedro (teoretičnega) računalništva.

1.4.1 Teorija jezikov in izračunljivost

Hierarhija jezikov
Regularni jeziki
Kontekstno neodvisni jeziki
Lambda račun in Turingov stroj
Izračunljivost
Rekurzivni jeziki
Turingovi jeziki

1.4.2 Teorija programskih jezikov

Formalni jezik *lambda račun*, ki ga bomo predstavili v naslednjih poglavjih bo služil kot osnova za predstavitev konceptov programski jezikov. Hkrati se uporablja tudi v *teoriji programskih jezikov* za študij matematičnih lastnosti algoritmov in jezikov. Primeri uporabe teorije programskih jezikov so dokazovanje pravilnosti programov, zaključitve izvajanja programov, determinističnost evaluacije izrazov jezikov, in podobno.

Logika in teorija tipov
Denotacijska semantika
Operacijska semantika
Lastnosti jezikov
Hierarhija v programskih jezikih
Enoličnost izračuna
Normalizacija, stroga normalizacija
Primitivna rekurzija

Rekurzivne funkcije
Gradniki višjega reda

Prevajalniki.

1.5 Pregled programskih jezikov

Poglejmo si razvoj programskih jezikov:

Lambda račun (1940)
Strojni jeziki, Fortran
Algol, Programski jezik C, Pascal, Modula
Funkcijski jeziki, Lisp, ML, Haskell
Objektni jeziki, C++, Objective C, Java, Eiffel
Visokonivojski jeziki, Elektronika, Informacijski sistemi, Integracijski jeziki
Skriptni jeziki, Sistemski jeziki Web jeziki, Uporabniški vmesniki, PHP, JSP, ASP
Prolog, Sicstus Prolog, SWI-Prolog, Datalog, Lambda Prolog

1.5.1 Strojni jeziki

Delo z ALU
Delo z registri
Ukazi in operandi
Različna naslavljanja
Zbirnik
Makroji
Procesorji

1.5.2 Fortran

John W. Backus, 1953, IBM
The IBM Mathematical Formula Translating System
IBM 360
Luknjane kartice
Gradniki Fortran
DO, GOTO, IF, SUBROUTINE, CALL

Zelo popularen jezik za numerično procesiranje
 Še vedno zelo živ jezik!
 Implicitni paralelizem
 Fortran program lahko lahko učinkovito paraleliziramo
 Edini takšen jezik!
 Fortran II, Fortran 77, Fortran III, Fortran 90, Fortran 95, Fortran 2003

1.5.3 Algolova družina

Družina programskih jezikov
 Funkcije so lahko del programa
 Funkcija ima lahko parametre
 Funkcija lahko vrne rezultat
 Funkcije si bomo natančno ogledali!
 Jezik ima bloke
 begin ... end
 Algol, Pascal, C, ... Modula

Programski jezik C

Dennis Ritchie, Bell Laboratories, 1972
 C je narejen za Unix – Unix je C okolje in obratno
 Razvit iz jezika B, ki je bil osnovan na BCPL
 B je bil jezik brez tipov, C dodaja nekatera preverjanja
 Gradniki jezika
 Strukturiran programski jezik zelo blizu OS in strojni opremi
 funkcije, operatorji, if, then, for, return
 Še danes najboljši jezik (C++ in C + iluzija objektov) za implementacijo sistemskih
 rutin ter resnih in hitrih sistemov
 Polja in kazalci so tesno povezani
 Polje je preprosto kazalec na prvi element polja
 Aritmetika za delo s kazalci
 Ritchie je napisal:
 C is quirky, flawed, and a tremendous success.

Pascal

Niklaus Wirth, ETH Zurich
 Eden najbolj razširjenih programskih jezikov v Evropi
 Strukturirani programski jezik

Procedure, parametri, funkcije
Blok za jezikovne konstrukte
Podatkovne strukture, kazalci, polja, variabini zapisi, ...
Veliko različnih implementacij
VAX Pascal, Turbo Pascal, ...
Delphi: zelo lep Objektni Pascal, učinkovito programiranje, uporabniški vmesniki, generiranje kode, ...
Modula-2, Modula-3, Oberon

1.5.4 Funkcijski jeziki

Angleška šola
Izhajajo iz formalnih jezikov
Lambda račun je razvil leta 1930 Alonzo Church
LISP
Razvil John McCarthy
Lisp je implementacija lambda-računa
Meta-Language
ML, Rob Milner
Funkcijski jeziki so boljše osnova za učenje?
Strogi tipi
Program je dokaz
Elementi deklarativnega programiranja
USA

1.5.5 Objektno usmerjeni jeziki

Prvi objektni jezik je bil Simula.
Simulacija dejanskega dogajanja.
Smalltalk je eden od najslavnejših jezikov
Adele Goldberg, Xerox, Palo Alto
Vse na tem svetu so samo objekti!
Vsak objekt je član razreda
Prototipni objekt
Razred je objekt!
Kompleksne hierarhije dedovanja
C++ + Java + vsi novi jeziki !
Java
J.Gosling, B.Joy, G.Steele, G.Bracha
The Java Language Specification

<http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf>

1.5.6 Skriptni jeziki

Web programski jeziki

V zadnjih letih se je pojavila množica jezikov, za delo na Web strežnikih
JSP, PHP, JavaScript, ASP, ...

Uporabljajo vire Web strežnika

Sistemske programski jeziki

Močna povezava z operacijskim sistemom

Awk, Perl, Python

Zelo učinkovito delo s sistemom

Novejši programski gradniki na sistemski način

Primerni za povezovanje komponent sistema

1.5.7 Visokonivojski jeziki

Informacijski sistemi

4GL, PL/SQL, SQL3

UML (specifikacije)

Integracijski jeziki

Python (multi-paradigm programming language)

Perl (OS, Web, Statistika,...)

Web jeziki (dostop do vsega)

Elektronika

VHDL

1.5.8 Logično programiranje

Programming in Logic, Robert Kowalski

Predikatni račun

Hornovi stavki

Unifikacija, resolucija

Zelo močen, enostaven in abstrakten jezik

Uporaba vračanja (back-tracking)

Deklarativen in proceduralen pomen Prologa

Operator CUT (!)

Jeziki za delo s podatkovnimi bazami
Datalog
Sicstus, SB Prolog, SWI Prolog, ...

Poglavje 2

LAMBDA RAČUN

Poglejmo najprej malce v zgodovino. Videli bomo na kakšen način je lambda račun povezan s programskimi jeziki ter zakaj predstavlja lambda račun enega izmed osnovnih formalnih jezikov računalništva.

Leibniz je imel naslednji ideal [?]: 1) kreiraj univerzalni jezik v katerem lahko izrazimo vse probleme in 2) poišči odločitven postopek s katerim rešimo vse probleme izražene v univerzalnem jeziku. Če se omejimo na matematične probleme, potem je prva točka Leibnizovega ideala rešena z uporabo neke oblike teorije množic in predikatnim računom oz. logiko. S prvo točko sta se ukvarjala Gottlob Frege in Bertrand Russel. Frege je definiral moderno obliko predikatnega računa, Russel pa je uvedel teorijo tipov, ki je postala del moderne logike.

Druga točka je postala pomemben filozofski problem: “lahko rešimo vse probleme izražene z univerzalnim jezikom”? Problem so popularno imenovali *Entscheidungsproblem*. Problem je bil rešen z negativnim izidom. Rešila sta ga neodvisno Church leta 1936 in Turing leta 1936/7. Za rešitev problema sta potrebovala jezik za opis problemov in formalno definicijo intuitivnega koncepta ‘rešljivosti’.

Church in Turing sta za reševanje problema uporabila različna jezika. Church je za jezik uporabil formalni sistem, ki ga imenujemo *lambda račun* medtem ko je Turing definiral abstrakten stroj, ki ga danes imenujemo *Turingov stroj*. Oba sta definirala pojem *izračunljive funkcije* vsak s svojim formalizmom. Turing je leta 1939 pokazal tudi, da sta jezika izrazno enako močna.

Lambda račun je osnoven formalen jezik programskih jezikov. Lambda abstrakcija predstavlja računalniško abstrakcijo funkcij in procedur. Z lambda računom lahko izrazimo katerokoli izračunljivo funkcijo enako kot s Turingovim strojem. V okviru *teorije programskih jezikov* [?] so lambda račun ter nadgradnje lambda računa uporabljene kot formalna osnova za definicijo hierarhije jezikov glede na izrazno moč jezikov.

2.1 Lambda račun in programski jeziki

Programski jezik Lisp je eden izmed prvih programskih jezikov. Razvil ga je John McCarthy leta 1958 na Massachusetts Institute of Technology (MIT). Lisp je genialno enostavna implementacija lambda računa. Moderne različice Lispa kot je na primer Scheme so še vedno v uporabi ter velikokrat služijo kot prvi programski jezik, ki se ga študenti učijo predvsem na ameriških univerzah.

V okviru Lispa je bila predlagana vrsta novih konceptov programskih jezikov, ki so se hitro uveljavili preko Lispa kot tudi v množici programskih jezikov, ki so bili razviti kasneje. Večino teh gradnikov, kot je na primer λ -abstrakcija, ter drugi predstavljeni spodaj, lahko vidimo kot koncepte predstavljene z λ -računom.

Abstrakcija funkcije kot je definirana v λ -računu služi kot osnovni koncept programskih jezikov uporabljen v vseh modelih programskih jezikih: funkcijskih, imperativnih kot tudi v novejših objektno-usmerjenih jezikih.

Poleg λ -abstrakcije so moderni programski jeziki prevzeli iz λ -računa tudi obravnavanje spremenljivk, ki so vezane na dan λ -izraz oz. spremenljivk, ki so vezane na dano definicijsko območje.

Interpreter Lispa je realiziran z istimi mehanizmi, ki so uporabljeni za evaluacijo λ -računa. Metoda za simbolično evaluacijo λ -izrazov, ki jo imenujemo β -redukcija je uporabna tudi širše, v okviru implementacije funkcij drugih funkcijskih kot tudi imperativnih programskih jezikov. Redukcijske strategije, ki so začetno služile pri dokazovanju enoličnosti evaluacije in determinističnosti λ -računa so kasneje uporabljene kot osnovne tehnike za implementacijo funkcijskih jezikov.

Očitno je zakaj je koristno študirati λ -račun zdaj, ko je minilo že skoraj celo stoletje od leta 1936, ko je Alonzo Church predlagal λ -račun kot formalizem za opis matematičnih problemov. λ -račun definira abstrakcijo funkcije, ki jo uporabljajo praktično vsi moderni programski jeziki. Podobno predstavljajo danes notacija in mehanizmi, ki so bili uporabljeni za študij lastnosti λ računa, izvirne ideje marsikaterega gradnika modernih programskih jezikov.

V nadaljevanju si bomo pogledali sintakso lambda računa ter semantiko lambda računa predstavljeno v obliki evaluacijskih pravil.

2.2 Osnove λ -računa

Pogejmo si najprej primer enostavnega izraza in funkcije, ki je definirana z danim izrazom.

Na primer, izraz $x + y$ izračuna vsoto spremenljivk x in y . Običajna matematična notacija za prejšnji izraz je $f(x) = x + y$. Izraz zapišemo kot funkcijo spremenljivke

x z uporabo *lambda abstrakcije*: $\lambda x.(x + y)$.

Opazimo lahko bistveno razliko med zapisoma: lambda notacija funkcije nima imena medtem, ko v matematični notaciji običajno zapišemo tudi ime funkcije f .

Naslednji izraz je podoben, le da vsebuje tri spremenljivke $x + 2 * y + z$. Iz izraza naredimo funkcijo dveh spremenljivk x in z z lambda izrazom $\lambda x.\lambda z.(x + 2 * y + z)$.

Aplikacija lambda izraza na vrednosti, ki je podana kot parameter se izvrši podobno kot pri matematičnih funkcijah. Spremenljivko povezano z lambda zamenjamo za vrednost ter ovrednotimo tako dobljen izraz. Poglejmo si aplikacijo na prejšnjih primerih.

$$\begin{aligned} (\lambda x.(x + y)) 3 &= 3 + y \\ (\lambda x.(\lambda z.(x + 2 * y + z))) 5 4 &= 4 + 2 * y + 5 \end{aligned} \quad (2.1)$$

Povezovanje: $\lambda x.f(fx) = \lambda x.(f(f(x)))$

Poglejmo si zdaj še nekaj primerov lambda izazov.

Lambda izraz z dvema lambda abstrakcijama in eno prosto spremenljivko:

$$\lambda x.\lambda y.x z y = \lambda x.(\lambda y.((x z) y)) \quad (2.2)$$

Še en primer povezovanja - porazdelitev z po dveh parametrih:

$$\lambda x.\lambda y.\lambda z.(x z) (y z) = \lambda x.(\lambda y.(\lambda z.((x z) (y z)))) \quad (2.3)$$

In še en primer ...

$$\lambda m.\lambda n.\lambda z.\lambda s.m (n z s) s = \lambda m.(\lambda n.(\lambda z.(\lambda s.((m ((n z) s)) s)))) \quad (2.4)$$

2.2.1 Definicija sintakse lambda računa

2.2.2 Proste in vezane spremenljivke

Vezana spremenljivka je "vrzel". Spremenljivka x je vezana v $\lambda x.(x + y)$. Funkcija $\lambda x.(x + y)$ je enaka funkciji $\lambda z.(z + y)$.

Primerjaj:

$$\begin{aligned} \int x + y dx &= \int z + y dz \\ \forall x P(x) &= \forall z P(z) \end{aligned} \quad (2.5)$$

Ime proste (=nevezane) spremenljivke je pomembno:

- Spremenljivka y je prosta v $\lambda x.(x + y)$.
- Funkcija $\lambda x.(x + y)$ ni enaka $\lambda x.(x + z)$.

Primer: y je prosta in povezana v $\lambda x.((\lambda y.y + 2)x) + y$.

2.3 Evaluacija

2.3.1 Substitucija

S substitucijo zamenjamo vse spremenljivke v danem izrazu z nekim izrazom, ki je lahko bodisi sestavljen izraz ali vrednost. Poglejmo si najprej opisna pravila za substitucijo N za x v M , kar zapišemo $[N/x]M$.

1. Preimenuj vezane spremenljivke v M in N tako, da so unikatne.
2. Izvedi tekstovno zamenjavo N za x v M .

Bolj formalno zapišemo pravila substitucije na naslednji način.

- $[N/x]x = N$
- $[N/x]z = z$, če $z \neq x$
- $[N/x](L M) = ([N/x]L)([N/x]M)$
- $[N/x](\lambda z.M) = \lambda z.([N/x]M)$, če $z \neq x \wedge z \notin FV(N)$

Poglejmo si nekaj primerov substitucije.

- $[y(\lambda x.x)/x]\lambda y.(\lambda x.x)y x$
- $[y(\lambda v.v)/x]\lambda z.(\lambda u.u)z x$
- $\lambda z.(\lambda u.u)z(y(\lambda v.v))$

2.3.2 Alfa konverzija

λ -izrazi, ki jih lahko pretvorimo med sabo s primenovanjem vezanih spremenljivk so identični.

Primer: $\lambda x.x$ je identičen $\lambda y.y$.

Konverzija: $\lambda x.M = \lambda y.([y/x]M)$, če $y \notin FV(M)$.

Preimenovanje vezanih spremenljivk - primer:

Aplikacija funkcije:

$(\lambda f.\lambda x.f(fx))(\lambda y.y + x)$

apliciraj 2X

dodaj x k argumentu

Slepa substitucija:

$$\lambda x.((\lambda y.y + x)((\lambda y.y + x)x)) = \lambda x.x + x + x$$

Preimenuj vezane spremenljivke:

$$\begin{aligned} & (\lambda f.\lambda z.f(fz))(\lambda y.y + x) \\ = & \lambda z.((\lambda y.y + x)((\lambda y.y + x)z))) = \lambda z.z + x + x \end{aligned} \quad (2.6)$$

2.3.3 Beta redukcija

V osnovnem lambda računu je edini način za ovrednotenje izrazov aplikacija funkcij na argumentih.

Beta redukcija:

- $(\lambda x.M)N \rightarrow [N/x]M$
- $(\lambda x.M)$ imenujemo redeks (iz angl. reducable expression)

Redukcija λ -izraza:

- β -redukcija M v N : $M \rightarrow_{\beta} N$ ali $M \rightarrow N$
- β -izpeljava M v N : $M \rightarrow_{\beta} N$

Končni rezultat je enolično določen

Primeri:

- $(\lambda x.xy)(uv) \rightarrow_{\beta} uv$
- $(\lambda x.\lambda y.x)zw \rightarrow_{\beta} (\lambda y.z)w \rightarrow z$
- $(\lambda x.\lambda y.x)zw \rightarrow_{\beta}^* z$

M je beta konvertibilen z N ali $M =_{\beta} N$, če $M = N$, ali $M \rightarrow^* B$ in $N \rightarrow^* B$.

Primer:

$$(\lambda x.x)z =_{\beta} (\lambda x.\lambda y.x)zw, \text{ zato ker } (\lambda x.\lambda y.x)zw \rightarrow^* z \text{ in } (\lambda x.x)z \rightarrow^* z. \quad (2.7)$$

Primeri evaluacije:

Funkcija identitete:

$$(\lambda x.x)E \rightarrow [E/x]x = E \quad (2.8)$$

Še en primer z identiteto:

$$\begin{aligned} & (\lambda f.f(\lambda x.x))(\lambda x.x) \rightarrow \\ & [(\lambda x.x)/f]f(\lambda x.x) = [(\lambda x.x)/f]f(\lambda y.y) \rightarrow \\ & (\lambda x.x)(\lambda y.y) \rightarrow \\ & [(\lambda y.y)/x]x = \lambda y.y \end{aligned} \quad (2.9)$$

Izpeljava, ki se ne zaključí:

$$(\lambda x.xx)(\lambda y.yy) \rightarrow [(\lambda y.yy)/x]xx = (\lambda x.xx)(\lambda y.yy) \rightarrow \dots \quad (2.10)$$

2.3.4 Funkcije višjega reda

Dana je funkcija f . Vrne funkcijo $f \circ f$.

$$\lambda f.\lambda x.f(fx) \quad (2.11)$$

Kako to dela?

$$\begin{aligned} & (\lambda f.\lambda x.f(fx))(\lambda y.y + 1) \\ & = \lambda x.(\lambda y.y + 1)((\lambda y.y + 1)x) \\ & = \lambda x.(\lambda y.y + 1)(x + 1) \\ & = \lambda x.(x + 1) + 1 \end{aligned} \quad (2.12)$$

Ista funkcija v Lispu. Dana je funkcija f . Vrne funkcijo $f \circ f$.

$$(\text{lambda}(f)(\text{lambda}(x)(f(fx)))) \quad (2.13)$$

Kako to dela?

$$\begin{aligned} & ((\text{lambda}(f)(\text{lambda}(x)(f(fx))))(\text{lambda}(y)(+y1))) \\ = & (\text{lambda}(x)((\text{lambda}(y)(+y1))((\text{lambda}(y)(+y1)x)))) \quad (2.14) \\ = & (\text{lambda}(x)((\text{lambda}(y)(+y1))(+x1))) \\ = & (\text{lambda}(x)(+(+x1)1)) \end{aligned}$$

2.4 Programiranje v lambda računu

2.4.1 Curry

Lambda račun nima funkcij z večimi argumenti vendar je enostavno doseči isti učinek s funkcijami višjega reda.

Naj bo M izraz s prostima spremenljivkama x in y . Čelimo napisati funkcijo F , ki za vsak par (N, L) zamenja x z N in y z L v izrazu M . V osnovnem lambda računu ne moremo napisati $F = \lambda(x, y).M$.

$$F = \lambda x. \lambda y. M$$

- F je funkcija, ki ob danem argumentu N vrne funkcijo, ki ob dani vrednosti za y vrne željeni rezultat.
- $FNL \rightarrow (\lambda y. [N/x]M)L \rightarrow [L/y][N/x]M$

Transformacijo, ki funkcijo z večimi argumenti prevede v funkcijo višjega reda imenujemo *Curry*.

2.4.2 Kombinatorji

Znane funkcije, ki se pogosto uporabljajo.

Funkcija identitete

$$I = \lambda x. x$$

Funkcija, ki zavrže dan argument y in izračuna funkcijo identitete

$$K = \lambda y. (\lambda x. x)$$

Porazdelitev zadnjega parametera na prva dva

$$S = \lambda x. \lambda y. \lambda z. (x z)(y z)$$

Funkcija, ki se ne konča

$$\Omega = (\lambda x. x x)(\lambda x. x x)$$

Definicija rekurzivnih funkcij

$$Y = \lambda f. (\lambda x. f(x x))(\lambda x. f(x x))$$

Pomembna lastnost Y je $Y F =_{\beta} F (Y F)$.

$$\begin{aligned} Y F &= \lambda f. (\lambda x. f(x x))(\lambda x. f(x x)) F \\ &\rightarrow (\lambda x. F(x x))(\lambda x. F(x x)) \\ &\rightarrow F((\lambda x. F(x x))(\lambda x. F(x x))) \\ &\leftarrow F((\lambda f. (\lambda x. f(x x))(\lambda x. f(x x))) F) \\ &= F(Y F) \end{aligned} \tag{2.15}$$

2.4.3 Logične vrednosti

Kako uvedemo logične vrednosti $\{True, False\}$ v Lambda račun.

Logične vrednosti so funkcije $True, False$

$$\begin{aligned} True &= \lambda t. \lambda f. t \\ False &= \lambda t. \lambda f. f \end{aligned}$$

if stavek: $if = \lambda l. \lambda m. \lambda n. l m n$

Primer izpeljave (redukcije) if stavka:

$$\begin{aligned} \text{if } True \ M \ N &= (\lambda l. \lambda m. \lambda n. l m n) \ True \ M \ N && \text{po definiciji} \\ &\rightarrow (\lambda m. \lambda n. \ True \ m \ n) \ M \ N && \beta\text{-redukcija} \\ &\rightarrow (\lambda n. \ True \ M \ n) \ N && \beta\text{-redukcija} \\ &\rightarrow \ True \ M \ N && \beta\text{-redukcija} \\ &= (\lambda t. \lambda f. t) \ M \ N && \text{po definiciji} \\ &\rightarrow (\lambda f. M) \ N && \beta\text{-redukcija} \\ &\rightarrow M && \beta\text{-redukcija} \end{aligned}$$

Logični IN

Logični IN lahko zapišemo v naslednji obliki:

$$And = \lambda b. \lambda c. b \ c \ False$$

V primeru, da je b resnično potem vrne c sicer $False$.
Izraz c vrne $True$ samo, če je izraz resničen...

Primer aplikacije funkcije And : $And\ True\ False$

2.4.4 Churchova števila

$$\begin{aligned} C_0 &= \lambda z. \lambda s. z \\ C_1 &= \lambda z. \lambda s. s\ z \\ C_2 &= \lambda z. \lambda s. s(s\ z) \\ &\vdots \\ C_n &= \lambda z. \lambda s. s(s(\dots(s\ z)\dots)) \end{aligned}$$

Število n je predstavljeno z funkcijo C_n , ki ima dva argumenta z (zero) in s (successor) ter aplicira n kopij funkcije s na z .

Število n je predstavljeno s funkcijo, ki nekaj naredi n -krat.

Običajne aritmetične operacije na Churchovih številih so:

$$\begin{aligned} Plus &= \lambda m. \lambda n. \lambda z. \lambda s. m(n\ z\ s)\ s \\ Times &= \lambda m. \lambda n. m\ C_0(Plus\ n) \end{aligned}$$

$$\begin{aligned} (Plus\ 1\ 2) &\rightarrow^* 3 \\ Plus\ (\lambda z. \lambda s. s\ z)\ (\lambda z. \lambda s. s(s\ z)) &\rightarrow \\ (\lambda m. \lambda n. \lambda z. \lambda s. m(n\ z\ s)\ s)\ (\lambda z. \lambda s. s\ z)\ (\lambda z. \lambda s. s(s\ z)) &\rightarrow \\ (\lambda n. \lambda z. \lambda s. (\lambda z. \lambda s. s\ z)(n\ z\ s)\ s)\ (\lambda z. \lambda s. s(s\ z)) &\rightarrow \\ \lambda z. \lambda s. (\lambda z. \lambda s. s\ z)((\lambda z. \lambda s. s(s\ z))\ z\ s)\ s &\rightarrow \\ \lambda z. \lambda s. (\lambda z. \lambda s. s\ z)((\lambda s. s(s\ z))\ s)\ s &\rightarrow \\ \lambda z. \lambda s. (\lambda z. \lambda s. s\ z)(s(s\ z))\ s &= \\ \lambda z. \lambda s. (((\lambda z. \lambda s. s\ z)\ (s(s\ z)))\ s) &\rightarrow \\ \lambda z. \lambda s. ((\lambda s. s(s\ z))\ s) &\rightarrow \\ \lambda z. \lambda s. s(s\ z) \end{aligned}$$

Seštevanje ni preveč zabavno :(

2.4.5 Fakulteta

Intuitivna definicija funkcije, ki izračuna fakulteto danega števila.

```
if n = 0 then 1
else n * (if n - 1 = 0 then 1
else (n - 1) * (if n - 2 = 0 then 1
else (n - 2) * ...
```

Rekurzijo lahko dosežemo s funkcijo $G = \lambda f. \langle telo\ f \rangle$ in $F = Y\ G$

$$\begin{aligned}
F &= Y G \\
&=_{\beta} G (Y G) \\
&=_{\beta} \langle telo (Y G) \rangle \\
&=_{\beta} \langle telo \langle telo (Y G) \rangle \rangle \\
&\vdots
\end{aligned}$$

Funkcijo *Factorial* lahko zdaj definiramo na sledeč način.

$$\begin{aligned}
Fact &= \lambda fact. \lambda n. \text{if } (IsZero n) C_1 (Times n (fact (Pred n))) \\
Factorial &= Y Fact
\end{aligned}$$

Poglejmo si izpeljavo fakultete za C_2 .

$$\begin{aligned}
Factorial C_2 &= Y Fact C_2 \\
&=_{\beta} Fact (Y Fact) C_2 \\
&=_{\beta} (\lambda fact. \lambda n. \text{if } (IsZero n) C_1 (Times n (fact (Pred n)))) (Y Fact) C_2 \\
&=_{\beta} (\lambda n. \text{if } (IsZero n) C_1 (Times n (Y Fact (Pred n)))) C_2 \\
&=_{\beta} \text{if } (IsZero C_2) C_1 (Times C_2 (Y Fact (Pred C_2))) \\
&=_{\beta} \text{if } False C_1 (Times C_2 (Y Fact C_1)) \\
&=_{\beta} Times C_2 (Y Fact C_1) \\
&= Times C_2 (Factorial C_1)
\end{aligned}$$

2.5 Uporaba lambda računa

Kje vse uporabljamo λ -račun.

- Preverjanje tipov, lambda račun s tipi.
- Formalen jezik za opis pomena stavkov v operacijski in denotacijski semantiki.
- Osnova za interpreter programskega jezika.
- Podroben študij gradnikov jezika.
- Obstaja veliko razširitev lambda računa.

2.5.1 Funkcijski jeziki

Pomembni aspekti λ -računa.

- λ ulovi "bistvo" povezovanja spremenljivk.
- Parametri funkcij.
- Deklaracije.
- Povezane spremenljivke se lahko primenujejo.
- Natančen opis funkcij.
- Enostavna simbolična evaluacija s substitucijo.

λ -račun lahko razširimo.

- Tipi.
- Strukture.
- Fiksne točke.
- Različnimi funkcijami.
- Spomin in stranski učinki.

Kaj je funkcijski jezik?

- Sinonim za: "Ni stranskih efektov".
- V okviru deklaracije x_1, x_2, \dots, x_n se vse morajo imeti te spremenljivke na njihovem definicijskem območju isto vrednost.
- Klicana funkcija ne sme vplivati na vrednost spremenljivk.
- Čisti funkcijski jezik: jezik s funkcijami, brez stranskih efektov.

Zakaj funkcijski jeziki?

- Sklepanje o programih.
- Enostavnejše je sklepati o funkcijskih programih.
- Bolj učinkovito zaradi možne vzporednosti.
- Algebrski zakoni.
- Optimizacija prevajalnika.

Sklepanje o programih.

- Dokaz pravilnosti programa.
- Moramo vedeti vse kar vpliva na program.
- Odvisnost od podatkovnih struktur v funkcijskih programih je eksplicitna.
- Enostavneje sklepamo o programih.
- Dokaz strogih tipov jezika.
- Determinističnost evaluacije programov.
- Orodje, ki omogoča boljši vpogled v kodo.

Poglavje 3

OSNOVNE STRUKTURE

Predstavljeni bodo naslednji osnovni gradniki programskih jezikov:

- vrednosti
- spremenljivke
- produkti
- sezname
- deklaracija vrednosti

ML:

Meta Language
eden od bolj popularnih funkcijskih jezikov
Edinburgh, 1974, skupina Robin Milner
obstaja množica dialektov

Ocaml kot referenčni jezik:

- čista definicija funkcij,
- formalne osnove struktur:
 - produkt,
 - seznam,
 - unije,
- vsebuje vrsto konceptov:
 - imperativni gradniki,
 - moduli,
 - objekti,
 - funktorji

bogato razvojno okolje ocaml:

knjižnice,
interpreter,
prevajalnik,
razhroščevalnik,
lex, yacc,
dobra dokumentacija.

Ocaml postaja široko uporabljen za vrsto namenov:

splošni programski jezik,
sistemski jezik,
delo z omrežjem.

3.1 Vrednosti

Enostavne vrednosti:

cela števila
realna števila
znaki
nizi
boolove vrednosti

Sestavljene vrednosti:

n-terice
seznam

3.1.1 Števila

Imamo dve vrsti števil: cela števila tipa `int` in realna števila tipa `float`. Standard IEEE 754 je uporabljene za predstavitev realnih števil. Če je rezultat operacije izven definiranega intervala ne pride do napake ampakse rezultat prilagodi intervalu.

Cela števila:

+ seštevanje
 - odštevanje in unarna negacija
 * množenje
 / celoštevilsko deljenje
 mod ostanek celoštevilskega deljenja

Primeri:

```

# 1 ;;
- : int = 1
# 1 + 2 ;;
- : int = 3
# 9 / 2 ;;
- : int = 4
# 11 mod 3 ;;
- : int = 2
(* limits of the representation *)
(* of integers *)
# 2147483650 ;;
- : int = 2

```

Realna števila

+. addition
 -. subtraction and unary negation
 *. multiplication
 /. division
 **. exponentiation

Primeri:

```

# 2.0 ;;
- : float = 2
# 1.1 +. 2.2 ;;
- : float = 3.3
# 9.1 /. 2.2 ;;
- : float = 4.13636363636
# 1. /. 0. ;;
- : float = inf
(* limits of the representation *)
(* of floating-point numbers *)
# 222222222222.11111 ;;
- : float = 222222222222

```

Vrednosti različnih tipov kot npr. int in float ne moremo primerjati direktno. Uporabiti moramo funkcije za pretvorbo med tipi.

Primeri:

```

# 2 = 2.0 ;;

```

Characters 5-8:

```
This expression has type float but is here used with type int
# 3.0 = float_of_int 3 ;;
- : bool = true
```

Tudi operacije nad celimi števili in realnimi števili so različne.

```
# 3 + 2 ;;
- : int = 5
# 3.0 +. 2.0 ;;
- : float = 5
# 3.0 + 2.0 ;;
Characters 0-3:
This expression has type float but is here used with type int
# sin 3.14159 ;;
- : float = 2.65358979335e-06
```

Nepravilen izračun kot je na primer deljenje z nič sproži izjemo, ki prekine izvajanje kode. Realna števila v Ocaml imajo definirano neskončno vrednost: Inf. Prav tako imamo predstavitev za nepravilen izračun: NaN3.

Funkcije na realnih številih:

```
ceil
floor
sqrt    square root
exp     exponential
log     natural log
log10   log base 10
```

Trigonometrične funkcije:

```
cos     cosine
sin     sine
tan     tangent
acos    arccosine
asin    arcsine
atan    arctangent
```

Primeri:

```
# ceil 3.4 ;;
- : float = 4
# floor 3.4 ;;
- : float = 3
# ceil (-.3.4) ;;
- : float = -3
# floor (-.3.4) ;;
- : float = -4
# sin 1.57078 ;;
```

```

- : float = 0.999999999867
# sin (asin 0.707) ;;
- : float = 0.707
# acos 0.0 ;;
- : float = 1.57079632679
# asin 3.14 ;;
- : float = nan

```

3.1.2 Znaki in nizi

Znaki ustrezajo celim številom med 0 in 255 po ASCII kodirni tabeli za prvih 128 znakov. Funkciji `char_of_int` in `int_of_char` podpirajo pretvorbo med znaki in celimi števili. Nizi so sekvence znakov določene dolžine. Operacijo konkatencije zapišemo `^`. Funkcije `int_of_string`, `string_of_int`, `string_of_float` in `float_of_string` pretvarjajo med števili in nizi.

```

# 'B' ;;
- : char = 'B'
# int_of_char 'B' ;;
- : int = 66
# "is a string" ;;
- : string = "is a string"
# (string_of_int 1987) ^ "is the year Caml was created" ;;
- : string = "1987 is the year Caml was created"

```

Tudi, če niz vsebuje cifre jih ne moremo uporabljati z operacijami nad znaki brez eksplícitne konverzije.

```

# "1999"+ 1 ;;
Characters 1-7:
This expression has type string but is here used with type int
# (int_of_string "1999") + 1 ;;
- : int = 2000

```

Množica funkcij nad znaki in nizi je zbranih v modulu `String`.

3.1.3 Boolove vrednosti

Boolovi vrednosti (tipa `bool`) sta: `true` in `false`. Zaradi zgodovinskih razlogov imata `and` in `or` dva zapisa.

```

not   negacija
&&   zaporedni logični in
&    sinonim za &&
or    sinonim za ||
||   zaporedni logični or

```

Primer:

```

# true ;;
- : bool = true
# not true ;;
- : bool = false
# true && false ;;
- : bool = false

```

Operaciji `&&` in `||` ter njuna sinonima ovrednotita najprej levi argument in v odvisnosti od vrednosti še desni argument. Primerjalne operacije, ki dajo vrednost `bool`, so naslednje.

```

=      strukturna enakost
==     fizična enakost
<>    negacija =
!=     negacija ==
<      manjše
>      večje
<=    manjše ali enako
>=    večje ali enako

```

Predstavljene operacije so polimorfične, ker znajo primerjati vrednosti različnih tipov. Edina omejitev je, da so vrednosti istega tipa.

```

# 1<=118 && (1=2 || not(1=2)) ;;
- : bool = true
# 1.0 <= 118.0 && (1.0 = 2.0 || not (1.0 = 2.0)) ;;
- : bool = true
# "one"< "two" ;;
- : bool = true
# 0 < '0' ;;
Characters 4-7:
This expression has type char but is here used with type int

```

Strukturna enakost preveri dve vrednosti tako, da pregleda celotno strukturo, metem ko fizična enakost prveri samo ali so vrednosti na istem pomnilniškem prostoru. Realna števila in nizi se obravnavajo kot strukturne vrednosti!

3.1.4 Unit

Tip unit opisuje množico, ki vsebuje en sam element, ki ga označimo ().

```
# () ;;  
- : unit = ()
```

Ta vrednost bo uporabljena za imperativne programe, ki imajo stranski učinek. Tip unit je podoben tipu void v programskem jeziku C.

3.2 Osnovni tipi

Kaj je tip?

Abstrakten koncept (razred) vrednosti.

Prototip vrednosti.

Interpretacija tipa je množica primerkov tipa.

Tip določa strukturo vrednosti.

Osnovni tipi.

Atomični tipi

Seznami

Produkti

Polja

3.2.1 Atomični tipi

Prej prikazani atomični tipi:

Celo število

Realno število

Boolove vrednosti

Znaki

Nizi

Unit

Primer uporabe atomičnih tipov

3.2.2 Sezname

Vrednosti istega tipa lahko združimo v seznam. Seznam je lahko bodisi prazen ali je sestavljen iz elementov istega tipa.

```
# [] ;;
- : 'a list = []
# [ 1 ; 2 ; 3 ] ;;
- : int list = [1; 2; 3]
# [ 1 ; "two"; 3 ] ;;
Characters 14-17:
This expression has type int list but is here used with type string list
```

Za dodajanje novega elementa v glavo seznama uporabljamo infiksni operator `::`, ki je ustreza Lispovem operatorju `cons`.

```
# 1 :: 2 :: 3 :: [] ;;
- : int list = [1;2;3]
```

Konkatenacijo dveh seznamov dosežemo z operatorjem `@`.

```
# [1]@[2;3];;
- : int list = [1;2;3]
# [1;2]@[3];;
- : int list = [1;2;3]
```

Druge funkcije za delo s seznamami so definirane v modulu `List`. Funkciji `hd` in `tl` vrneta glavo in rep seznama, če te vrednosti obstajata. Funkciji pokličemo kot `List.hd` in `List.tl`; nahajata se v modulu `List`.

```
# List.hd [ 1 ; 2 ; 3 ] ;;
- : int = 1
# List.hd [] ;;
Uncaught exception: Failure("hd")
```

V zadnjem primeru je res problem zahtevati prvi element, ker je seznam prazen. V tem primeru se sproži izjema.

3.2.3 N-terice

Vrednosti različnih tipov lahko združimo v pare ali bolj splošno, v n-terice. Vrednosti, ki sestavljajo n-terice so ločene z vejicami. Konstruktor tipa, ki definira n-terice je `*`. Na primer `int * string` je tip parov, ki imajo celo število za prvo komponento in niz za drugo.

```
# ( 12 , "October" ) ;;
- : int * string = 12, "October"
```

Ko ni dvoumnosti lahko zapišemo n-terico bolj enostavno:

```
# 12 , "October" ;;
- : int * string = 12, "October"
```

Funkciji `fst` in `snd` omogočata dostop do prve in druge komponente para.

```
# fst ( 12 , "October" ) ;;
- : int = 12
# snd ( 12 , "October" ) ;;
- : string = "October"
```

Predstavljeni funkciji `fst` in `snd` sta definirani nad pari. Funkciji sta polimorfični, ker imajo komponente parov lahko poljubni tip.

```
# fst ;;
- : 'a * 'b -> 'a = <fun>
# fst ( "October", 12 ) ;;
- : string = "October"
```

Tip `int * char * string` definira trojice katerih prvi element je tipa `int`, drugi tipa `char` in tretji tipa `string`. Primer vrednosti tega tipa je:

```
# ( 65 , 'B' , "ascii" ) ;;
- : int * char * string = 65, 'B', "ascii"
```

Opozorilo: Funkciji `fst` in `snd` v splošnem ne moremo aplicirati na n-terici, ki ni par.

```
# snd ( 65 , 'B' , "ascii" ) ;;
Characters 7-25:
This expression has type int * char * string but is here used with type
'a * 'b
```

Ločimo torej med tipom, ki določa par in tipom, ki določa trojico. Tip `int * int * int` je drugačen od tipov `(int * int) * int` in `int * (int * int)`. Nimamo funkcij za dostop do komponent: uporabljati je potrebno ujemanje vzorcev, ki so definirani kasneje.

3.3 Deklaracija vrednosti

Deklaracija poveže ime z vrednostjo. Imamo dve vrsti deklaracij: globalne in lokalne. V prvem primeru so deklarirane vrednosti vidne vsem izrazem. V drugem primeru so deklarirane vrednosti vidne v samo enem izrazu.

3.3.1 Globalne deklaracije

Sintaksa:

```
let name = expr ;;
```

Globalna deklaracija definira povezavo med imenom in vrednostjo izraza `expr`, ki je vidna vsem izrazom, ki sledijo deklaraciji.

```
# let yr = "1999";;
val yr : string = "1999"
# let x = int_of_string(yr) ;;
val x : int = 1999
# x ;;
- : int = 1999
# x + 1 ;;
- : int = 2000
# let new_yr = string_of_int (x + 1) ;;
val new_yr : string = "2000"
```

Lahko definiramo več povezav hkrati.

```
let name1 = expr1
and name2 = expr2
.
.
.
and namen = exprn ;;
```

Simultana deklaracija definira različne simbole na istem nivoju. Simboli niso poznani do konca deklaracije.

```
# let x = 1 and y = 2 ;;
val x : int = 1
val y : int = 2
# x + y ;;
- : int = 3
# let z = 3 and t = z + 2 ;;
Characters 18-19:
Unbound value z
```

Primer:

```
# let x = 2
  let y = x + 3;;
val x : int = 2
val y : int = 5
```

Globalna deklaracija se lahko zakrije z novo deklaracijo.

3.3.2 Lokalne deklaracije

Sintaksa:

```
let name = expr1 in expr2 ;;
```

Ime name je poznano samo med evaluacijo izraza expr2. Lokalna deklaracija poveže ime z vrednostjo expr1.

```
# let x1 = 3 in x1 * x1 ;;
- : int = 9
```

Lokalna deklaracija, ki poveže x1 z vrednostjo 3 je vidna samo med izvajanjem x1 * x1.

```
# x1 ;;
Characters 1-3:
Unbound value x1
```

Lokalna deklaracija zakrije vse predhodne deklaracije z istim imenom.

```
# let x = 2 ;;
val x : int = 2
# let x = 3 in x * x ;;
- : int = 9
# x * x ;;
- : int = 4
```

Lokalna deklaracija je izraz in je lahko uporabljena za konstrukcijo drugih izrazov.

```
# (let x = 3 in x * x) + 1 ;;
- : int = 10
```

Tudi lokalne deklaracije so lahko simultane.

Sintaksa:

```
let  name1 = expr1
and  name2 = expr2
.
.
.
and  namen = exprn
in   expr ;;
```

Primer:

```
# let a = 3.0 and b = 4.0 in sqrt (a*.a +. b*.b) ;;  
- : float = 5  
# b ;;  
Characters 0-1:  
Unbound value b
```

3.4 Imenski prostori

Poglavje 4

FUNKCIJSKI JEZIKI

Lambda abstrakcija:

direktna preslikava v Ocaml
osnovna oblika lambda računa je dostopna v Ocaml
lambda abstrakcija definira funkcijo,
aplikacija funkcije na argumentu,
parameter je poljuben tip.

Abstrakcija funkcije:

- lambda abstrakcija predstavlja funkcijo
- abstrakcija kode z imenom
- formalna osnova za množico gradnikov
- matematične osnove funkcije
- aplikacija in kompozicija funkcij tvori jezik
- funkcijski jeziki temeljijo na funkcijah
- sam jezik funkcij je Lisp

4.1 Funkcijski izrazi

Funkcijski izraz je sestavljen iz parametra in telesa funkcije. Formalni parameter je spremenljivka in telo funkcije je izraz. Parameter realizira lambda abstrakcijo.

Sintaksa:

```
function p -> expr
```

Funkcija, ki izračuna kvadrat parametra:

```
# function x -> x*x ;;
- : int -> int = <fun>
```

Ocaml sam izračuna tip. Tip funkcije `int -> int` pove, da funkcija pričakuje parameter tipa `int` in vrne rezultat tipa `int`. Aplikacija funkcije na argumentu se zapiše na naslednji način.

```
# (function x -> x * x) 5 ;;
- : int = 25
```

Aplikacija funkcije evaluirata telo funkcije tako, da se formalni parameter zamenja z dejanskim.

Telo funkcije je lahko katerikoli izraz—lahko je tudi funkcija.

```
# function x -> (function y -> 3*x + y) ;;
- : int -> int -> int = <fun>
```

Oklepaji niso potrebni. Lahko napišemo preprosto:

```
# function x -> function y -> 3*x + y ;;
- : int -> int -> int = <fun>
```

Tip tega izraza lahko beremo na običajen način kot tip funkcije, ki pričakuje dve celi števili in vrne celo število. V oviru funkcijskega jezika beremo takšne tipe kot funkcije, ki imajo argument tipa `int` in vrnejo funkcijo, katere argument je `int` in rezultat tipa `int`.

```
# (function x -> function y -> 3*x + y) 5 ;;
- : int -> int = <fun>
```

Funkcijo lahko uporabljamo na običajen način. Naslednji primer aplicira funkcijo na dveh parametrih.

```
# (function x -> function y -> 3*x + y) 4 5 ;;
- : int = 17
```

Ko napišemo `f a b`, predpostavljamo implicitno oklepaje, ki so definirani v skladu z levo asociativnim pravilom za povezovanje aplikacije funkcij: `(f a) b`.

Poglejmo zdaj aplikacijo

```
(function x -> function y -> 3*x + y) 4 5
```

bolj podrobno. Za izračun vrednosti tega izraza je potrebno najprej izračunati izraz


```
(function x -> function y -> 3*x + y) 4
```

kar predstavlja izraz, ki je ekvivalenten

```
function y -> 3*4 + y
```

Rezultat dobimo z zamenjavo x z 4 v $3*x + y$. Če zdaj apliciramo to funkcijo na argumentu 5 dobimo končno vrednost $3*4+5 = 17$:

```
# (function x -> function y -> 3*x + y) 4 5 ;;
- : int = 17
```

4.1.1 Števnost funkcije

Število argumentov funkcije imenujemo *števnost funkcije*. Uporaba funkcij, ki je pododovana od matematike, zahteva, da so argumenti funkcije obdani z oklepaji: $f(4,5)$. Oglejmo si še ta primer:

```
# function (x,y) -> 3*x + y ;;
- : int * int -> int = <fun>
```

Kot že sam tip funkcije pove pričakuje funkcija enega in ne dva parametra: par celih števil. Podajanje dveh argumentov funkciji, ki pričakuje par je napaka. Podobno je tudi podajanje para funkciji, ki pričakuje dva argumenta napaka.

```
# (function (x,y) -> 3*x + y) 4 5 ;;
Characters 29-30:
This expression has type int but is here used with type int * int
# (function x -> function y -> 3*x + y) (4, 5) ;;
Characters 39-43:
This expression has type int * int but is here used with type int
```

4.1.2 Alternativna sintaksa

Funkcijo lahko zapišemo tudi na bolj kompakten način: parametre preprosto naštejemo za imenom funkcije.

Sintaksa:

```
fun p1 ... pn -> expr
```

Krajši zapis omogoča, da se izognemo ponavljajočem zapisu ključne besede `function` in puščicam.

```
function p1 -> ... -> function pn -> expr
```

Primer:

```
# fun x y -> 3*x + y ;;
- : int -> int -> int = <fun>
# (fun x y -> 3*x + y) 4 5 ;;
- : int = 17
```

Predstavljena oblika zapisa funkcije se pogosto uporablja. Na primer, vse knjižnice jezika Ocaml uporabljajo to obliko.

4.2 Definicija funkcij

4.2.1 Deklaracija funkcijske vrednosti

Funkcijske vrednosti so deklarirane podobno kot v vseh ostalih jezikih, z uporabo konstrukta `let`.

```
# let succ = function x -> x + 1 ;;
val succ : int -> int = <fun>
# succ 420 ;;
- : int = 421
# let g = function x -> function y -> 2*x + 3*y ;;
val g : int -> int -> int = <fun>
# g 1 2;;
- : int = 8
```

Za poenostavitev zapisa je omogočena naslednja notacija.

Sintaksa:

```
let name p1 . . . pn = expr
```

Ta je enakovredna naslednjem polnem zapisu:

```
let name = function p1 -> ... -> function pn -> expr
```

Naslednji deklaraciji funkcij `succ` in `g` sta enakovredni prejšnjim deklaracijam.

```
# let succ x = x + 1 ;;
val succ : int -> int = <fun>
# let g x y = 2*x + 3*y ;;
val g : int -> int -> int = <fun>
```

Funkcijski značaj Ocaml je prikazan z naslednjim primerom. Funkcijo h1 dobimo z aplikacijo g na celem številu. Govorimo lahko o delni aplikaciji.

```
# let h1 = g 1 ;;
val h1 : int -> int = <fun>
# h1 2 ;;
- : int = 8
```

Lahko začnemo tudi s funkcijo g in definiramo funkcijo h2 tako, da fiksiramo vrednost drugega parametra y.

```
# let h2 = function x -> g x 2 ;;
val h2 : int -> int = <fun>
# h2 1 ;;
- : int = 8
```

Poglejmo si še primer funkcije, ki naredi kompozitum dveh funkcij. Funkcija compose sprejme dve funkciji in eno vrednost ter sestavi aplikacijo dveh funkcij na vrednosti.

```
# let compose f g x = f (g x) ;;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
# compose succ h2 1 ;;
- : int = 9
```

4.2.2 Infiksne funkcije

Nekatere funkcije z dvema argumenti so definirane z infiksno notacijo. Na primer, običajno pišemo $3 + 5$, kar pomeni aplikacijo $+$ na 3 in 7. Za uporabo simbola $+$ kot običajno funkcijo ga obdamo z oklepaji.

Sintaksa:

```
( op )
```

Naslednji primer definira funkcijo succ z uporabo $(+)$.

```
# ( + ) ;;
- : int -> int -> int = <fun>
# let succ = ( + ) 1 ;;
val succ : int -> int = <fun>
# succ 3 ;;
- : int = 4
```

Definiramo lahko tudi nove operatorje. Poglejmo si definicijo operacije $++$ za seštevanje parov.

```
# let ( ++ ) c1 c2 = (fst c1)+(fst c2), (snd c1)+(snd c2) ;;
val ++ : int * int -> int * int -> int * int = <fun>
# let c = (2,3) ;;
val c : int * int = 2, 3
# c ++ c ;;
- : int * int = 4, 6
```

Infiksne operatorje lahko definiramo samo nad simboli *, +, @, itd. in ne tudi z običajnimi črkami. Nekatere predefinirane funkcije so izjema temu pravilu. To so: or, mod, land, lor, lxor, lsl, lsr in asr.

4.2.3 Definijska območja

Izraz lahko ovrednotimo le, če so predhodno definirane vse spremenljivke, ki nastopajo v izrazu.

Na primer, v izrazu `e` stavka `let p=e` moraja biti definirane vse spremenljivke. Vrednost z imenom `p` se lahko pojavi v izrazu `e`, če je bila že predhodno definirana.

```
# let p = p ^ "-suffix" ;;
Characters 9-10:
Unbound value p
# let p = "prefix" ;;
val p : string = "prefix"
# let p = p ^ "-suffix" ;;
val p : string = "prefix-suffix"
```

V Ocaml so vrednosti statično povezane. Okolje, ki je uporabljeno pri evaluaciji izraza je definirano v času deklaracije in ne v času aplikacije.

```
# let p = 10 ;;
val p : int = 10
# let k x = (x, p, x+p) ;;
val k : int -> int * int * int = <fun>
# k p ;;
- : int * int * int = 10, 10, 20
# let p = 1000 ;;
val p : int = 1000
# k p ;;
- : int * int * int = 1000, 10, 1010
```

Funkcija `k` vsebuje prosto spremenljivko `p`. Ker je `p` definirana predhodno, je `k` legalna funkcija. Povezava med `p` v definiciji funkcije `k` je statična: določena je ob času deklaracije.

4.2.4 Omejitve tipov

Sintetizator tipov generira najbolj splošen tip izraza.

Sintaktična oblika omejitve s tipi je sledeča.

Sintaksa:

```
( expr : t )
```

Ko sintetizator tipov naleti na omejitev tipa jo bo vzel v poštev med konstrukcijo tipa izraza. Z omejitvijo tipa lahko:

- naredimo tip parametrov funkcije viden;
- prepovemo uporabo funkcije izven namenjenega konteksta;
- specificiramo tip izraza, ki bo uporabljen za spremenljive vrednosti .

Naslednji primeri bodo prikazali uporabnost omejitev tipov. Prvi primer prikaže omejitev funkcije `add`; parametri morajo biti tipa `int`.

```
# let add (x:int) (y:int) = x + y ;;
val add : int -> int -> int = <fun>
```

Naslednji dve funkciji predstavljata omejitev že obstoječe operacije za kreiranje para ter funkcije `compose`, ki naredi kompozitum dveh funkcij, ki sta tokrat omejeni s tipom `int -> int`.

```
# let make_pair_int (x:int) (y:int) = x,y;;
val make_pair_int : int -> int -> int * int = <fun>
# let compose_fn_int (f : int -> int) (g : int -> int) (x:int) =
  compose f g x;;
val compose_fn_int : (int -> int) -> (int -> int) -> int -> int = <fun>
```

V naslednjem primeru definiramo simbol `nil` kot vrednost tipa `string list`. Pri poskusu kreacije seznama z glavo `'H'` in repom `nil` dobimo napako, ker glava ni istega tipa kot element repa.

```
# let nil = ( [] : string list);;
val nil : string list = []
# 'H' :: nil;;
Characters 5-8:
This expression has type string list but is here used with type char list
```

Omejevanje polimorfizma na ta način omogoča boljšo kontrolo tipov izrazov z omejevanjem izpeljevanja tipov podizrazov. Za omejitve tipa lahko uporabimo katerikoli definiran tip.

```
# let llnil = ( [] : 'a list list) ;;
val llnil : 'a list list = []
# [1;2;3]:: llnil ;;
- : int list list = [[1; 2; 3]]
```

Simbol `llnil` je seznam seznamov kateregakoli tipa.

Ukvarjamo se z omejitvami in ne z zamenjevanjem sinteze tipov z eksplicitnimi tipi. Omejevanje tipov ne more posplošiti tip. Na primer, v naslednjem primeru tip funkcije ne bo `'a -> 'a -> 'a` ampak `int -> int -> int`.

```
# let add_general (x:'a) (y:'a) = add x y ;;
val add_general : int -> int -> int = <fun>
```

Omejitve tipov bodo uporabljene tudi pri vmesnikih modulov kot tudi pri deklaraciji razredov.

4.3 Rekurzivne funkcije

Definicija simbola je rekurzivna, če uporablja simbol pri lastni definiciji.

Pravkar smo videli, da gradnik `let` tega ne podpira. Za deklaracijo rekurzivne funkcije uporabljamo poseben sintaktični gradnik.

Sintaksa:

```
let rec name = expr ;;
```

Ključno besedo `rec` lahko uporabljamo tudi v kontekstu, kjer ima funkcija parametre.

Sintaksa:

```
let rec name p1 . . . pn = expr ;;
```

Poglejmo si funkcijo `sigma`, ki izračuna vsoto (pozitivnih) celih števil od nič do vrednosti argumenta (vključno z argumentom).

```
# let rec sigma x = if x = 0 then 0 else x + sigma (x-1) ;;
val sigma : int -> int = <fun>
# sigma 10 ;;
- : int = 55
```

Opazimo lahko, da se funkcija ne zaključi v primeru, da je argument negativen.

Rekurzivna vrednost je v splošnem funkcija ! Prevajalnik zavrne nekatere rekurzivne definicije, katerih vrednost ni funkcija.

```
# let rec x = x + 1 ;;
Characters 13-18:
```

This kind of expression is not allowed as right-hand side of 'let rec'

Deklaracija z let rec lahko kombiniramo med sabo z uporabo simultane deklaracije. V tem primeru so vse funkcije definirane na istem nivoju let znane vsem ostalim definicijam. To med ostalim omogoča definicijo navzkrižno rekurzivnih funkcij.

```
# let rec even n = (n<>1) && ((n=0) or (odd (n-1)))
  and odd n = (n<>0) && ((n=1) or (even (n-1)));;
val even : int -> bool = <fun>
val odd : int -> bool = <fun>
# even 4 ;;
- : bool = true
# odd 5 ;;
- : bool = true
```

Na isti način so lahko tudi lokalne deklaracije rekurzivne. Naslednji primer vsebuje prejšnjo definicijo funkcije sigma, ki je zdaj zavarovana: preveri se legalnost argumenta.

```
# let sigma x =
  let rec sigma_rec x = if x = 0 then 0 else x + sigma_rec (x-1) in
  if (x<0) then "error: negative argument"
  else "sigma = " ^ (string_of_int (sigma_rec x)) ;;
val sigma : int -> string = <fun>
```

4.3.1 Primeri rekurzivnih funkcij

Poglejmo si zdaj implementacijo nekaterih znanih funkcij nad števili. Začetni primer sigma prikaže definicijo vsote naravnih števil 0..n. Poglejmo si še primer izračuna vsote prvih n členov vrste $\sum_{1..n} 1/n^2$.

```
# let rec vrsta n =
  if n = 0 then 0.0 else 1.0/.(float_of_int n)**2.0 +. vrsta (n-1);;
val fib : int -> int = <fun>
# vrsta 10000;;
- : float = 1.64483407184806496
```

Naslednji primer prikaže definicijo funkcije, ki izračuna n -to Fibonaccijevo število.

```
# let rec fib n =
  if n < 2 then 1 else fib(n-1) + fib(n-2);;
val fib : int -> int = <fun>
# fib 33;;
- : int = 5702887
```

4.4 Polimorfizem

Nekatere funkcije lahko definiramo tako, da sprejemajo parametre različnih tipov. Na primer kreacija para iz dveh vrednosti ne zahteva različne funkcije za vsak tip argumenta. Podobno je tudi funkcija za dostop do prve komponente para lahko zadosti splošna, da deluje z različnimi tipi parametra.

```
# let make_pair a b = (a,b) ;;
val make_pair : 'a -> 'b -> 'a * 'b = <fun>
# let p = make_pair "paper" 451 ;;
val p : string * int = "paper", 451
# let a = make_pair 'B' 65 ;;
val a : char * int = 'B', 65
# fst p ;;
- : string = "paper"
# fst a ;;
- : char = 'B'
```

Funkcije imenujemo *polimorfične*, če lahko vračajo vrednosti ali imajo za enega izmed parametrov tip, ki ni specificiran.

Objektni Caml poišče vedno najbolj splošen tip za vsak izraz. Spremenljivke tipa, ki jih uporabljamo za označevanje nespecificiranih tipov imenujemo 'a in 'b.

Polimorfične funkcije omogočajo pisanje generične kode brez da bi izgubili varnost statičnega preverjanja tipov. Čeprav je funkcija `make_pair` polimorfična ima kreiran par dobro definiran tip.

4.4.1 Primeri polimorfičnih funkcij in vrednosti

Naslednji primeri polimorfičnih funkcij imajo funkcijske parametre katerih tipi so parametrizirani.

Funkcija `app` aplicira funkcijo na argument.

```
# let app = function f -> function x -> f x ;;
val app : ('a -> 'b) -> 'a -> 'b = <fun>
```

Tako lahko `app` apliciramo na funkciji `odd` kot je bila definirana prej.

```
# app odd 2 ;;
- : bool = false
```

Funkcija identitete vzame parameter in ga vrne takšnega kot je.


```
# let id x = x ;;
val id : 'a -> 'a = <fun>
# app id 1 ;;
- : int = 1
```

Naslednji primer prikaže definicijo polimorfične funkcije `compose`, ki poveže funkciji `add1` in `mul5` z uporabo stavka `let`.

```
# let compose f g x = f (g x) ;;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
# let add1 x = x+1 and mul5 x = x*5 in compose mul5 add1 9 ;;
- : int = 50
```

Običajne vrednosti (razen funkcij) so tudi lahko polimorfične. Na primer, prazen seznam je takšna vrednost.

```
# let l = [] ;;
val l : 'a list = []
```

Naslednji primer demonstrira, da *sinteza tipa* izhaja iz omejitev izraza.

```
# let t = List.tl [2] ;;
val t : int list = []
```

Tip funkcije `List.tl` je `'a list -> 'a list` zato funkcija aplicirana na seznam celih števil vrne seznam celih števil. Če je rezultat prazen seznam to ne spremeni tipa rezultata.

Objektni Caml generira parametrični tip za vse funkcije, ki na nobeden način ne omejujejo danega tipa. Takšen polimorfizem imenujemo *parametrični polimorfizem*.

4.4.2 Curry oblika funkcij

Curry operacija je pomemben koncept funkcijskega programiranja. Poimenovana je po logiku Haskell Curry. Funkcijo z več argumenti spremenimo v Curry obliko, kjer ima vsaka funkcija en sam argument.

Naslednja operacija `my_add` je definirana v Curry obliki.

```
# let my_add x y = x + y ;;
val my_add : int -> int -> int = <fun>
# my_add 3 4 ;;
- : int = 7
```

Še ena enakovredna definicija iste funkcije.

```
# let my_add x =
  function y -> x + y;;
val my_add : int -> int -> int = <fun>
# my_add 3 4;; (* parenthesized as (myadd 3) 4 *)
- : int = 7
# let inc3 = my_add 3;;
val inc3 : int -> int = <fun>
# inc3 4;;
- : int = 7
```

Nova funkcija `my_add` je funkcija, ki vrača funkcijo. Na primer, pri klicu `(my_add 3) 4` podamo parametra na sledeč način. Najprej pokličemo funkcijo in dobimo nazaj funkcijo, ki jo pokličemo in ji podamo še drugi parameter.

Pri uporabi Curry oblike funkcij imajo vse funkcije vedno en sam argument. Oporaba Curry oblike se pojavi naravno pri pisanju funkcij, ki vračajo funkcije, kot na primer funkcija `map`. Funkcije večih argumentov v knjižnicah so običajno v Curry obliki.

```
# List.map;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Funkcije imajo seveda lahko za parametre tudi pare ali v splošnem n -terice, ki nadomestijo uporabo večih parametrov oz. večih funkcij.

```
# let my_pair_add (x,y) = x+y;;
val my_pair_add : int * int -> int = <fun>
# my_pair_add (2,3);;
- : int = 5
```

Naslednji funkciji `curry` in `uncurry` spremenita obliko funkcije iz funkcije s parametrom, ki ima strukturo n -terice, v Curry obliko funkcije, ter obratno.

```
# let curry f = function x -> function y -> f (x,y);;
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
# let uncurry f = function (x,y) -> f x y;;
val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
# uncurry my_add;;
- : int * int -> int = <fun>
# curry my_pair_add;;
- : int -> int -> int = <fun>
# uncurry map;;
- : ('_a -> '_b) -> '_a list -> '_b list = <fun>
# curry(uncurry my_add);;
- : int -> int -> int = <fun>
```

4.4.3 Funkcije nad seznamami

Začnemo s funkcijo `null`, ki preveri ali je seznam prazen.

```
# let null l = (l = []) ;;
val null : 'a list -> bool = <fun>
```

Naprej definiramo funkcijo `size`, ki izračuna dolžino seznama.

```
# let rec size l =
  if null l then 0
  else 1 + (size (List.tl l)) ;;
val size : 'a list -> int = <fun>
# size [] ;;
- : int = 0
# size [1;2;18;22] ;;
- : int = 4
```

Funkcija preveri, če je argument prazen seznam in v tem primeru vrne 0. Sicer prišteje ena k dolžini repa.

Naslednja funkcija obrne seznam, ki je podan kot parameter funkcije.

```
# let rec reverse l =
  if l=[] then []
  else (reverse (List.tl l)) @ [(List.hd l)];;
val reverse : 'a list -> 'a list = <fun>
# reverse [1; 2; 3; 4];;
- : int list = [4; 3; 2; 1]
```

Naslednja funkcija obrne vsebino parov, ki so elementi seznama. Funkcija je polimorfična, saj so lahko tipi komponent parov poljubni.

```
# let rec flip l =
  if l=[] then []
  else let head = List.hd l in
        let rest = List.tl l in
        (snd head,fst head)::(flip rest);;
val flip : ('a * 'b) list -> ('b * 'a) list = <fun>
# flip [(1,2);(3,4)];;
- : (int * int) list = [(2, 1); (4, 3)]
```

Naslednje funkcije preoblikujejo seznam z uporabo rekurzije. Prvi dve funkciji vzamejo ali odrežejo prvih n elementov seznama. Drugi dve funkciji sestavijo oz. razcepijo seznam parov.

```
# let rec take (k, l) =
  if null l then []
  else if k>0 then List.hd(l)::take(k-1,List.tl(l)) else [];;
val take : int * 'a list -> 'a list = <fun>

# let rec drop (k, l) =
  if null l then []
  else if k>0 then drop(k-1,List.tl l) else List.hd(l)::(List.tl l);;
val drop : int * 'a list -> 'a list = <fun>

# let rec combine (l1, l2) =
  if null(l1) && null(l2) then []
  else (List.hd(l1),List.hd(l2)) :: combine (List.tl(l1), List.tl(l2));;
val combine : 'a list * 'b list -> ('a * 'b) list = <fun>

# let rec split l =
  if null(l) then ([],[])
  else
    let l1 = split(List.tl(l))
    in (fst(List.hd l) :: fst(l1), snd(List.hd(l)) :: snd(l1));;
val split : ('a * 'b) list -> 'a list * 'b list = <fun>
```

Sezname lahko obravnavamo kot množice. Naslednja funkcija preveri članstvo v seznamu oz. množici. Sledita funkciji s katerima sta realizirani operaciji razlika in unija nad seznamami.

```
# let rec member x l =
  if l=[] then false
  else if x = List.hd(l) then true
  else member x (List.tl l);;
val member : 'a -> 'a list -> bool = <fun>
# member 3 [2;3;1];;
- : bool = true
# let rec inter(xs, ys) =
  if xs=[] then []
  else let x = List.hd xs
    in if (member x ys) then x :: inter(List.tl xs, ys)
    else inter(List.tl xs, ys);;
val inter : 'a list * 'a list -> 'a list = <fun>
# inter ([1;2;3],[4;2]);;
- : int list = [2]
# let rec union(xs,ys) =
  if xs=[] then ys
```

```

    else let x = List.hd xs in
          let xr = union (List.tl xs,ys) in
            if (member x ys) then xr else x::xr;;
val union : 'a list * 'a list -> 'a list = <fun>
# union ([1;2;3], [1;4;5]);;
- : int list = [2; 3; 1; 4; 5]

```

4.5 Funkcije višjega reda

4.5.1 Osnove

Funkcija je lahko rezultat funkcije. Prav tako je funkcija lahko argument funkcije.

Funkcijo, ki vzame kot argument drugo funkcijo ali vrne funkcijo kot rezultat imenujemo *funkcija višjega reda*.

```

# let h = function f -> function y -> (f y) + y ;;
val h : (int -> int) -> int -> int = <fun>

```

Pozor: aplikacija je levo asociativna medtem ko so funkcijski tipi desno asociativni. Tip funkcije h lahko zapišemo na sledeč način.

```

(int -> int) -> int -> int   ali   (int -> int) -> (int -> int)

```

Funkcije višjega reda omogočajo elegantno delo s seznama. Na primer, funkcija map aplicira funkcijo na vseh elementih seznama in vrne rezultate aplikacije kot seznam.

```

# let rec map f l =
    if null l then []
    else f(List.hd l)::(map f (List.tl l));;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# let square x = string_of_int (x*x) ;;
val square : int -> string = <fun>
# map square [1; 2; 3; 4] ;;
- : string list = ["1"; "4"; "9"; "16"]

```

Funkcija map je definirana že v standardni knjižnici Ocaml oz. v sistemskem modulu List.

```

# List.map ;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>

```

Poglejmo si še en primer. Funkcija `for_all` preveri ali vsi elementi seznama zadostujejo danemu pogoju. Tudi funkcija `for_all` je definirana v modulu `List`. Poglejmo si implementacijo.

```
# let rec for_all f l =
  if null l then true
  else (f (List.hd l)) && for_all f (List.tl l);;
val for_all : ('a -> bool) -> 'a list -> bool = <fun>
# for_all (function n -> n<>0) [-3; -2; -1; 1; 2; 3] ;;
- : bool = true
# for_all (function n -> n<>0) [-3; -2; 0; 1; 2; 3] ;;
- : bool = false
```

4.5.2 Primeri funkcij višjega reda

Izraz `iterate n f` izračuna vrednost funkcije `f` `n` krat.

```
# let rec iterate n f =
  if n = 0 then (function x -> x)
  else compose f (iterate (n-1) f) ;;
val iterate : int -> ('a -> 'a) -> 'a -> 'a = <fun>
```

Funkcija `iterate` preveri ali je `n` enak 0. Če je potem vrne funkcjo identitete, in če ni naredi kompozicijo `f` z `n-1` kratno iteracijo `f`.

Z uporabo `iterate` lahko definiramo eksponent kot iteracijo z množenjem.

```
# let rec power i n =
  let i_times = ( * ) i in
  iterate n i_times 1 ;;
val power : int -> int -> int = <fun>
# power 2 8 ;;
- : int = 256
```

Funkcija `power` iterira `n`-krat funkcijski izraz `i.times`, potem aplicira rezultat na 1, kar izračuna `n`-to potenco celega števila.

Multiplikacijska tabela

Želimo napisati funkcijo `multab`, ki izračuna multiplikacijsko tabelo celega števila, ki je podano kot argument.

Najprej definiramo funkcijo `apply_fun_list` tako da, če je `f_list` seznam funkcij potem `apply_fun_list x f_list` vrne seznam aplikacij vsakega elementa `f_list` na `x`.

```
# let rec apply_fun_list x f_list =
  if null f_list then []
  else ((List.hd f_list) x) :: (apply_fun_list x (List.tl f_list)) ;;
val apply_fun_list : 'a -> ('a -> 'b) list -> 'b list = <fun>
# apply_fun_list 1 [( + ) 1;( + ) 2;( + ) 3] ;;
- : int list = [2; 3; 4]
```

Funkcija `mk_mult_fun_list` vrne seznam funkcij, ki pomnožijo argument z `i`, kjer gre `i` od 0 do `n`.

```
# let mk_mult_fun_list n =
  let rec mmfl_aux p =
    if p = n then [ ( * ) n ]
    else ( ( * ) p ) :: (mmfl_aux (p+1))
  in (mmfl_aux 1) ;;
val mk_mult_fun_list : int -> (int -> int) list = <fun>
```

Poglejmo kako dobimo multiplikacijsko tabelo 7:

```
# let multab n = apply_fun_list n (mk_mult_fun_list 10) ;;
val multab : int -> int list = <fun>
# multab 7 ;;
- : int list = [7; 14; 21; 28; 35; 42; 49; 56; 63; 70]
```

Iteracija nad seznamom

Funkcija `fold_left` `f a [e1; e2; ... ; en]` vrne `f (... (f (f a e1) e2) ... en)`. Imamo torej `n` aplikacij funkcije `f`.

```
# let rec fold_left f a l =
  if null l then a
  else fold_left f ( f a (List.hd l) ) (List.tl l) ;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

Funkcija `fold_left` omogoča kompaktno definicijo funkcije, ki izračuna vsoto elementov celoštevilskega seznama.

```
# let sum_list = fold_left (+) 0 ;;
val sum_list : int list -> int = <fun>
# sum_list [2;4;7] ;;
- : int = 13
```

Lahko pa isto funkcijo uporabimo tudi za konkatencijo seznama nizov.

```
# let concat_list = fold_left (^) "";;
val concat_list : string list -> string = <fun>
# concat_list ["Hello "; "world"; "!"] ;;
- : string = "Hello world!"
```

Funkcija višjega reda, ki je podobna `fold_left` vendar aplicira binarno funkcijo `f` od zadnjega elementa seznama proti prvemu se imenuje `fold_right`. Aplikacije funkcije `fold_right a [e1;e2;...;en]` se prevede v `f e1 (f e2 (... (f en a)...))`.

```
# let rec fold_right f a l =
  if null l then a
  else f (List.hd l) (fold_right f a (List.tl l)) ;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```


Poglavje 5

IMPERATIVNI JEZIKI

Za razliko od funkcijskega programiranja kjer računamo z aplikacijami funkcij na argumentih je imperativno programiranje bližje strojni predstavitvi: uporabljamo spomin, ki ga program med delovanjem spreminja z akcijami.

Imperativni programski jeziki izhajajo iz strojnih jezikov, ki se izvajajo direktno na procesorju. Akcije se v začetnih imperativnih programskih jezikih imenujejo inštrukcije. Imperativni program je predstavljen s sekvenco inštrukcij.

Izvajanje vsake inštrukcije spreminja stanje spomina, kar predstavlja enega izmed osnovnih principov imperativnega programiranja. Spominske lokacije do katerih v zgodnjih imperativnih programih lahko dostopamo preko *spremenljivk* predstavljajo lahko zapis celega števila, realnega števila ali bolj kompleksne strukture kot je npr. zapis.

Model imperativnega računanja iz druge strani temelji na matematičnih formulah. Programi predstavljajo izračun sekvence formul, ki se lahko ponovi z uporabo *razvejivne* *stavka* ali *zank* ter spremenljivk, kot vmesnih izračunov. Programski jezik Fortran (Formula Translator) se šteje kot prvi imperativni oz. tudi prvi programski jezik.

Z Algolsko vejo programskih jezikov, ki so bili razviti okoli leta 1970 so programi dobili strukturo z uporabo *blokov* programske kode ter *podprogramov*, ki so definirani podobno kot v funkcijskih programskih jezikih funkcije.

Najbolj pomembni predstavniki programskih jezikov, ki vsebujejo gradnike za definicijo in uporabo spremenljivk, blokov in procedur so programski jeziki C, Pascal in Cobol. *Strukturirano programiranje* uvedeno s temi programskimi jeziki omogoča pisanje programov, ki niso urejeni v vrstice ampak lahko kodo poljubno strukturiramo v bloke in procedure.

Imperativni programski jeziki so običajno bolj primerni za numerične izračune. Pre nekateri algoritmi se dajo enostavneje izraziti z uporabo imperativnega stila programiranja. Na primer, računanje z matrikami je veliko enostavneje in tudi bolj učinkovito realizirati z uporabo imperativnega programskega jezika.

Motivacija za integracijo imperativnih gradnikov v funkcijski jezik je torej zmožnost izražanja nekaterih algoritmov s primernimi gradniki programskih jezikov. Izbira med funkcijskimi in imperativnimi gradniki programskih jezikov mora biti vodena s prednostmi, ki jih omogoča uporaba določenih konceptov programskih jezikov.

Predstavitev matrik s pomočjo dvo-dimenzionalnih polj, katerih elemente lahko enostavno naslavljamo z uporabo običajnih matematičnih abstrakcij npr. $A[1, 1]$, $A[1, 2]$, ... ter uporaba iteracijskih gradnikov omogoča učinkovito implementacijo operacij nad matrikami kot so npr. seštevanje, množenje, itd.

Caml vsebuje nekatere vrste podatkovnih struktur katerih vrednosti so spremenljive, običajne iteracijske gradnike za kontrolo izvajanja programov (npr. for stavek) in knjižnice za vhodno/izhodne operacije. V programskem jeziku Caml naslednji gradniki realizirajo imperativni model programiranja.

- reference na konstante, ki realizirajo spremenljivke;
- spremenljive podatkovne strukture polja in zapisi s spremenljivimi komponentami;
- vhodno-izhodne operacije; ter
- kontrolne strukture kot so zanke in izjeme.

5.1 Spremenljivke

5.1.1 Vrednosti in spremenljivke

Spremenljivke lahko vidimo kot simbole, ki predstavljajo vrednost nekega tipa. V naslednjem primeru definiramo spremenljivko *i* v programskem jeziku Java.

```
int i;
```

Spremenljivka *i* predstavlja pomnilniško lokacijo kjer je shranjena vrednost spremenljivke *i*. Ime spremenljivke in pomnilniški naslov spremenljivke sta shranjena v *tabeli simbolov*, ki jo vsebuje vsak program.

V primeru statične spremenljivke lahko prevajalnik določi naslov spremenljivke in ga uporablja pri prevajanju izrazov, ki vsebujejo dano spremenljivo.

Program dostopa do vrednosti spremenljivke definirane v funkciji ali znotraj bloka tako, da dobi v tabeli simbolov pomnilniški naslov spremenljivke. Glede na tip spremenljivke se interpretira vsebina danega pomnilniškega naslova.

Poglejmo si še primer uporabe spremenljivk v algoritmu za izračun največjega skupnega delitelja, ki ga je definiral Euclid.

```
public static int gcd(int p, int q) {
    while (q != 0) {
        int temp = q;
        q = p % q;
        p = temp;
    }
    return p;
}
```

5.1.2 Spremenljivke in reference

Caml vsebuje gradnike za uporabo *kazalcev* (referenc) na atomične in strukturirane vrednosti. Sintaksa gradnikov za delo s kazalci je blizu Algolskim programskim jezikom kot so npr. C ali C++. Poglejmo si naprej kako so definirani kazalci v programskem jeziku C.

V programskem jeziku C imamo dva pomembna operatorja za delo s kazalci. Operator '&' ima lahko za argument poljubno spremenljivko. Operator vrne naslov spremenljivke kot rezultat.

Drugi operator, ki deluje obratno operatorju '&' je operator za ovrednotenje naslova '*'. Argument operatorja '*' je naslov vrednosti spremenljivke in rezultat operatorja je vrednost spremenljivke na danem naslovu.

```
int x = 1, y = 2, z = 3;
int *ip;
ip = &x;
```

Zgornji primer definira dve spremenljivki x, y in z, ki imata začetno vrednost 1, 2 in 3. Spremenljivka ip je definirana kot kazalec na celo število. Kazalcu ip priredimo naslov spremenljivke x. Vrednost kazalca ip dobimo z uporabo operatorja '*'. Izraz *ip ima torej vrednost enako spremenljivki x (1).

```
y = *ip;
*ip = 0;
ip = &z;
```

Prvi stavek v zgornjem primeru priredi vrednost *ip spremenljivki y. Spremenljivka y ima torej zdaj vrednost enako x (1), ker ip predstavlja naslov x. Po izvršitvi drugega stavka dobi spremenljivka x vrednost 0, medtem ko spremenljivka y ostane nespremenjena (ima vrednost 1). Zadnji stavek spremeni vrednost ip, ki zdaj kaže na naslov spremenljivke z.

Gradnika programskega jezika Caml, ki ustrežata '&' in '*' označimo 'ref' in '!'. Operator 'ref' služi kot konstruktor tipa kot tudi kot operator, ki vrne naslov vrednosti ali spremenljivka. Operator '!' ovrednoti kazalec oz. prikaže vrednost, ki se nahaja na naslovu predstavljenim s kazalcem.

Poglejmo si primer definicije kazalca v Caml. V naslednjem primeru najprej definiramo kazalec na vrednost 3. Vrednost kazalca izpišemo z drugim stavkom (!x;).

```
# let x = ref 3 ;;
val x : int ref = {contents=3}
# !x;;
- : int = 3
# x ;;
- : int ref = {contents=3}
```

Objektni Caml ima definiran polimorfični referenčni tip ref, ki ga lahko vidimo kot tip kazalcev na poljubno vrednost. Tip ref je definiran kot zapis z eno samo spremenljivo komponento, kar si bomo bolj natančno ogledali v naslednjem poglavju.

```
type 'a ref = {mutable contents:'a}
```

V naslednjem primeru prikažemo uporabo operatorja '!' za ovrednotenje kazalca ter infiksno funkcijo ':=' s katero lahko spreminjamo vsebino na katero kaže kazalec.

```
# !x ;;
- : int = 3
# x := 4 ;;
- : unit = ()
# !x ;;
- : int = 4
# x := !x+1 ;;
- : unit = ()
# !x ;;
- : int = 5
```

V zgornjem primeru najprej ovrednotimo kazalec x in dobimo vrednost 3. Naslednji stavek priredi novo vrednost vrednosti kazalca x, ki jo nato še izpišemo. Naslednji stavek priredi novo vrednost vsebini na katero kaže kazalec x; nova vrednost je stara vrednost plus ena.

5.2 Sekvenčna kontrola

Najbolj primitivna kontrola v programskem jeziku je *sekvenca*. Programski ukazi so urejeni v zaporedje tako, da se izvajajo po vrsti.

Vejitveni stavki omogočajo implementacijo pogojnega izvajanja blokov kode in realizacijo zank—blokov kode, ki se ponavljajo v skladu s kontrolnimi spremenljivkami zank. *Zanke* so osnovni mehanizmi za programiranje v začetnih imperativnih programskih jezikih kot npr. C, Pascal, Basic in drugi.

Dekompozicija problema v funkcije omogoča kontrolo izvajanja s pomočjo funkcijskih struktur. Problem je razdeljen na podprobleme, kar je odraženo v hierarhični strukturi funkcij.

Rekurzija omogoča uporabo drugačne kontrole izvajanja: rekurzivna funkcija se razvije v sekvenco ali drevo rekurzivnih klicev, ki jih vodimo s pogojnimi stavki v telesu funkcije.

Vzorci so zelo izrazni gradniki programskih jezikov s pomočjo katerih lahko realiziramo vrsto sekvenčne kontrole—izvajanje programa se ujame na enem izmed vzorcev, ki so nanizani v sekvenco, na osnovi dane vrednosti, ki je lahko strukturirana.

Pri uporabi razredov in objektov so na voljo druge vrste kontrolnih mehanizmov, ki omogočajo vodenje izvajanja sistema. Na osnovi dedovanja je definirana dejanska koda, ki se sproži ob klicu dane metode. Povezovanje med imenom metode z danimi parametri in dejansko kodo metode imenujemo *dinamično povezovanje*. Kontrolo višjenivojskih konceptov si bomo podrobneje ogledali v poglavjih, ki predstavijo module in razrede.

Višjenivojski programski jeziki kot so na primer Java, C++, Modula, Perl, Python, itd. vsebujejo večino izmed prej naštetih kontrolnih gradnikov.

Kontrolni stavki za implementacijo sekvenčne kontrole so:

- Sekvence
- Pogojni stavki
- Iteracije
- Vzorci

5.2.1 Sekvence

Sekvenca je zaporedje izrazov programskega jezika, ki se izvajajo eden za drugim. Sintaksa sekvenc je definirana na sledeč način.

Sintaksa:

```
expr1 ; ... ; exprn
```

Sekvenca izrazov je izraz, katerega vrednost je vrednost zadnjega izraza v sekvenci. Naslednji primer najprej izpiše niz, nakar se ovrednoti aritmetični izraz, ki vrne rezultat sekvence.

```
# print string "2 = "; 1+1 ;;
2 = - : int = 2
```

Sekvence skupaj s stranskimi učinki dajo običajno konstrukcijo imperativnega programskega jezika.

```
# let x = ref 1 ;;
val x : int ref = {contents=1}
# x:=!x+1 ; x:=!x*4 ; !x ;;
- : int = 8
```

V primeru, da je vrednost izraza pred podpičjem različna od unit, Ocaml vrednost zavrže in o tem opozori.

```
# print_int 1; 2 ; 3 ;;
Characters 14-15:
Warning: this expression should have type unit.
1- : int = 3
```

Če se hočemo izogniti sporočilu, lahko uporabljamo funkcijo ignore.

```
# print int 1; ignore 2; 3 ;;
1- : int = 3
```

V primeru, da je vrnjena vrednost tipa funkcija, potem Ocaml sumi, da je bil pozabljen parameter funkcije.

```
# let g x y = x := y ;;
val g : 'a ref -> 'a -> unit = <fun>
# let a = ref 10;;
val a : int ref = {contents=10}
# let u = 1 in g a ; g a u ;;
Characters 13-16:
Warning: this function application is partial,
maybe some arguments are missing.
- : unit = ()
# let u = !a in ignore (g a) ; g a u ;;
- : unit = ()
```

V splošnem je priporočljivo uporabljati oklepaje, da je explicitno vidno definicijsko območje. Sintaksa uporabe oklepajev je naslednja.

Sintaksa:

```
( expr )
begin expr end
```

Poglejmo si program “Višje/Nižje”, ki poskuša uganiti število tako da usmerja igralca z “višje” in “nižje”.

```
# let rec hilo n =
  print_string "type a number: ";
  let i = read_int () in
  if i = n then print_string "BRAVO\n\n"
  else
    begin
      if i < n then print_string "Higher\n"
      else print_string "Lower\n";
      hilo n
    end ;;
val hilo : int -> unit = <fun>
```

Isti program v Javi ...

5.2.2 Pogojni stavki

Kontrola izvajanja na osnovi pogojnega stavka je eden izmed osnovnih mehanizmov začetnih strojnih jezikov. Razvejitev je bila implementirana na osnovi inštrukcije za primerjavo vrednosti spominske lokacije z danim registrom.

V primeru, da je primerjava uspela se je izvedla naslednja instrukcija sicer pa druga naslednja. Če uporabimo na teh dveh mestih stavke GOTO potem lahko dobimo eno izmed oblike IF stavka.

Pogojni stavek IF je ena od nepogrešljivih kontrolnih struktur vsakega programskega jezika. Pogojni stavek vodi kontrolo izvajanja na osnovi boolovega pogoja.

Sintaksa:

```
if expr1 then expr2 else expr3
```

Izraz expr1 je tipa bool. Izraza expr2 in expr3 morata biti istega tipa.

```
# if 3=4 then 0 else 4 ;;
- : int = 4
# if 3=4 then "0" else "4";;
- : string = "4"
# if 3=4 then 0 else "4";;
```

Characters 20-23:

This expression has type string but is here used with type int

Pogojni izraz je tudi funkcija, ki vrne neko vrednost.

```
# (if 3=5 then 8 else 10) + 5 ;;
- : int = 15
```

5.2.3 Zanke

Iteracijske kontrolne strukture so osnovni gradniki imperativnih jezikov.

Pogojni stavek uporabljen za vejitve programov ter realizacijo zanke ima smisla lahko šele, če imamo možnost spreminjanja vsebine spremenljivk.

V Ocaml imamo dva iteracijska gradnika: `for` stavek in `while` stavek. Oba stavka sta del programskega jezika Ocaml in vračata vrednost `()` tipa `unit`. Poglejmo si najprej sintakso `for` stavka.

Sintaksa:

```
for name = expr1 to expr2 do expr3 done
for name = expr1 downto expr2 do expr3 done
```

Izraza `expr1` in `expr2` sta tipa `int`. če `expr3` ni tipa `unit`, potem prevajalnik izpiše opozorilo.

```
# for i=1 to 10 do
    print_int i;
    print_string " "
done;
print_newline ();;
1 2 3 4 5 6 7 8 9 10
- : unit = ()
# for i=10 downto 1 do
    print_int i;
    print_string " "
done;
print_newline ();;
10 9 8 7 6 5 4 3 2 1
- : unit = ()
```

Sintaksa stavka `while` je definirana na naslednji način.

Sintaksa:

```
while expr1 do expr2 done
```

Izraz `expr1` mora biti tipa `bool` in kot pri stavku `for`, če `expr2` ni tipa `unit` prevajalnik izpiše opozorilo.


```
# let r = ref 1
  in while !r < 11 do
      print_int !r ;
      print_string " ";
      r := !r+1
    done ;;
1 2 3 4 5 6 7 8 9 10 - : unit = ()
```

Pomembno je razumeti, da so iteracijski gradniki tipa unit.

```
# let f () = print_string "-- end\n";;
val f : unit -> unit = <fun>
# f (for i=1 to 10 do print_int i; print_string " " done) ;;
1 2 3 4 5 6 7 8 9 10 -- end
- : unit = ()
```

Niz “– end”je izpisan po tem, ko se izpišejo števila 1 do 10. Primer demonstrira, da se argumenti funkcij ovrednotijo preden se predajo funkciji kot dejanski argument.

V imperativnem programskem jeziku se telo iteracijskega stavka realizira s pomočjo stranskih učinkov in ne preko vrednosti, ki jih vrne.

```
# let s = [5; 4; 3; 2; 1; 0] ;;
val s : int list = [5; 4; 3; 2; 1; 0]
# for i=0 to 5 do List.tl s done ;;
Characters 17-26:
Warning: this expression should have type unit.
- : unit = ()
```

5.2.4 Vzorci

Osnovne ideje vzorcev:

delno izražene ideje,
 uporaba kompleksnih vrednosti,
 opis vrednosti ali konceptualne strukture,
 sprotno prirejanje vrednosti spremenljivkam.

Vrste vzorcev:

vzorci nad števili in nizi,
 vzorci na osnovi strukture vrednosti,
 vzorci na strukturah, seznamih in unijah,

Ujemanje vzorcev:

implementacija funkcij,
prirejanje vrednosti vzorcu.

Ujemanje vzorcev

Vzorec je struktura zgrajena z eno izmed danih podatkovnih struktur programskega jezika, konstant primitivnih tipov, spremenljivk ter predefiniranih gradnikov za definicijo vzorcev, ki jih imenujemo jokerji.

Ujemanje vzorcev se izvaja na vrednostih. Uporablja se za ujemanje strukture vrednosti in vodenje izvajanja programa v skladu z ujemanjem.

Sintaksa:

```
match expr with
  | p1 -> expr1
  .
  .
  | pn -> exprn
```

Vrednost izraza `expr` se prepoznava sekvenčno na vzorcih `p1, ..., pn`. Če se vzorec ujema z vrednostjo, se izvede pripadajoča koda. Vzorci v sekvenci so istega tipa. Vertikalna črta pred prvim vzorcem je opsijska.

Poglejmo si dva načina definicije ujemanja vzorcev za implementacijo funkcije `imply` tipa `(bool * bool) -> bool`, ki implementira logično implikacijo.

Prva verzija `imply` realizira pravilnostno tabelo tako, da našteje vse možne primere.

```
# let imply v = match v with
  (true,true)      -> true
  | (true,false) -> false
  | (false,true)  -> true
  | (false,false) -> true;;
val imply : bool * bool -> bool = <fun>
```

Druga verzija uporabi spremenljivke za združevanje večih primerov.

```
# let imply v = match v with
  (true,x) -> x
  | (false,x) -> true;;
val imply : bool * bool -> bool = <fun>
```

Vzorci morajo nujno biti *linearni*—nobena spremenljivka se ne pojavi več kot enkrat v enem vzorcu. Sicer bi lahko napisali naslednjo funkcijo.

```
# let equal c = match c with
  (x,x) -> true
  | (x,y) -> false;;
Characters 35-36:
This variable is bound several times in this matching
```

To bi pomenilo, da prevajalnik zna izvajati test enakosti. Če bi uporabljali fizično enakost potem bi dobili prešibek sistem, ki ne more izvajati enakost nad seznamami, na primer. Po drugi strani, če bi uporabljali strukturno enakost, lahko pride do cikličnih referenc. Na primer, graf je rekurzivna podatkovne struktura.

Simbol, ki se lahko ujema z vsemi možnimi vrednostmi, imenujemo joker. Uporabimo ga lahko na primer za dodatno poenostavitev funkcije `imply`.

```
# let imply v = match v with
  (true,false) -> false
  | _ -> true;;
val imply : bool * bool -> bool = <fun>
```

Definicija z ujemanjem vzorcev mora pokriti popolno množico možnih primerov za vrednost, ki jo primerjamo. Če to ni res prevajalnik izpiše sporočilo:

```
# let is_zero n = match n with 0 -> true ;;
Characters 17-40:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1
val is_zero : int -> bool = <fun>
```

Če je dejanski parameter različen od nič potem funkcija ne ve katero vrednost naj vrne. Primer lahko kompletiramo z uporabo jokerja.

```
# let is_zero n = match n with
  0 -> true
  | _ -> false ;;
val is_zero : int -> bool = <fun>
```

Če se v času izvajanje ne izbere nobeden primer ujemanja se sproži izjema.

```
# let f x = match x with 1 -> 3 ;;
Characters 11-30:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
0
```

```

val f : int -> int = <fun>
# f 1 ;;
- : int = 3
# f 4 ;;
Uncaught exception: Match_failure("", 11, 30)

```

Izjema Match Failure se sproži pri klicu f 4. Predstavljena koda ne obravnava izjeme.

Kombiniranje večih vzorcev da nov vzorec, ki se sproži na vrednosti v skladu z originalnimi vzorci.

Sintaksa:

```
p1 | ... | pn
```

Nov vzorec dobimo s kombinacijo vzorcev p1, ... in pn. Edina omejitev pri definiciji novega vzorca je v tem, da ne smemo definirati novih imen. Vsak od vzorcev lahko vsebuje samo konstante in jokerje. Naslednji primer definira funkcijo, ki preveri, če je črka samoglasnik.

```

# let is_a_vowel c = match c with
    'a' | 'e' | 'i' | 'o' | 'u' | 'y' -> true
  | _ -> false ;;
val is_a_vowel : char -> bool = <fun>
# is_a_vowel 'i' ;;
- : bool = true
# is_a_vowel 'j' ;;
- : bool = false

```

V kontekstu ujemanja vzorcev na znakih je precej nadležno konstruiranje kombinacij vzorcev na znakih, ki ustrezajo intervalom znakov. Če bi hoteli pisati vzorce na znakih je potrebno vsaj 26 vzorcev in jih kombinirati. Za poenostavitev takšnih vzorcev omogoča Ocaml pisanje intervalov.

Sintaksa:

```
'c1' .. 'cn'
```

je ekvivalentno: 'c1' | 'c2' | ... | 'cn'.

Na primer vzorec '0' .. '9' ustreza vzorcu '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'. Prva oblika je enostavnejša za branje in pisanje.

Z uporabo kombiniranih vzorcev in intervalov definiramo funkcijo za kategorizacijo znakov po večih kriterijih.

```
# let char_discriminate c = match c with
  | 'a' | 'e' | 'i' | 'o' | 'u' | 'y'
  | 'A' | 'E' | 'I' | 'O' | 'U' | 'Y' -> "Vowel"
  | 'a'..'z' | 'A'..'Z' -> "Consonant"
  | '0'..'9' -> "Digit"
  | _ -> "Other";;
val char_discriminate : char -> string = <fun>
```

Vrstni red skupin vzorcev ima pomen. V zgornjem primeru je prvi interval vsebovan v drugem vendar ni preverjen dokler ni preverjen prvi.

5.2.5 Ujemanje parametrov in vzorci

Ujemanje vzorcev se uporablja za definicijo funkcij na osnovi primerov. Poglejmo si sintaktični konstrukt function, ki omogoča ujemanje vzorcev na parametrih.

Sintaksa:

```
function | p1 -> expr1
        | p2 -> expr2
        .
        .
        .
        | pn -> exprn
```

Pokončna črta pred prvim vzorcem je opsijska. Dejansko vedno, ko definiramo funkcijo uporabljamo ujemanje vzorcev: konstrukcija funkcije "function x -> expression" uporablja en sam vzorec, ki je reduciran na spremenljivko. Poglejmo si še primer.

```
# let f = function (x,y) -> 2*x + 3*y + 4 ;;
val f : int * int -> int = <fun>
```

Dejansko je oblika:

```
function p1 -> expr1 | . . . | pn -> exprn
```

ekvivalentna:

```
function expr -> match expr with p1 -> expr1 | . . . | pn -> exprn
```

Če uporabimo še ekvivalenco deklaracij, ki smo jo predstavili pri funkcijah dobimo:

```
# let f (x,y) = 2*x + 3*y + 4 ;;
val f : int * int -> int = <fun>
```

Poglejmo si še enkrat funkcijo sigma, ki izračuna vsoto (pozitivnih) celih števil od nič do vrednosti argumenta (vključno z argumentom).

```
# let rec sigma = function
  0 -> 0
  | x -> x + sigma (x-1) ;;
val sigma : int -> int = <fun>
# sigma 10 ;;
- : int = 55
```

Enako kot pri uporabi stavka match morajo tudi vzorci funkcije pokriti vse možne vrednosti parametra. V primeru, da to ni res je ujemanje neizčrpano.

```
# let is_zero 0 = true ;;
Characters 13-21:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1
val is_zero : int -> bool = <fun>
```

Med preverjanjem vzorcev je včasih koristno imenovati dele ali celotne vzorce. Naslednja sintaktična oblika predstavlja povezovanje imena z vzorcem.

Sintaksa:

```
( p as name )
```

To je koristno, če želimo razstaviti vrednost in obdržati predstavo o njeni celoti. V naslednjem primeru bo definirana funkcija min_rat, ki vrne manjše racionalno število od komponent para racionalnih števil.

```
# let min_rat pr = match pr with
  ((_,0),p2) -> p2
  | (p1,(_,0)) -> p1
  | ((n1,d1) as r1), ((n2,d2) as r2)) ->
    if (n1 * d2 ) < (n2 * d1) then r1 else r2;;
val min_rat : (int * int) * (int * int) -> int * int = <fun>
```

Za primerjavo racionalnih števil jih moramo razstaviti zato, da lahko imenujemo števec in imenovalce (n1, n2, d1 in d2) hkrati moramo tudi referencirati par (r1 in r2), da bi lahko vrnili enega izmed njiju. Na ta način nam ni potrebno ponovno konstruirati parov iz komponent.

Ujemanje vzorcev s stražarji

Primerjanje vzorcev z uporabo stražarjev ustreza evalvaciji pogoja takoj po tem, ko je vzorcu zadoščeno. Če je rezultat evalvacije pravilen pogoj potem se dejansko tudi izvrši akcija sicer se nadaljuje primerjanje vzorcev.

Sintaksa:

```
match expr with
  .
  .
  .
  | pi when condi -> expri
  .
  .
  .
```

Naslednji primer uporablja dva stražarja za preverjanje enakosti dveh racionalnih števil.

```
# let eq_rat cr = match cr with
  ((_,0),(_,0)) -> true
  | ((_,0),_) -> false
  | (_,(_,0)) -> false
  | ((n1,1), (n2,1)) when n1 = n2 -> true
  | ((n1,d1), (n2,d2)) when ((n1 * d2) = (n2 * d1)) -> true
  | _ -> false;;
val eq_rat : (int * int) * (int * int) -> bool = <fun>
```

Če preverjanje pogoja stražarja ne uspe se preverjanje vzorcev nadaljuje na petem vzorcu.

Preverjanje ali definirani vzorci pokrivajo vse možnosti je ob prisotnosti stražarjev težje. Ocaml predpostavi, da pogoji lahko niso pravilni, zato ni mogoče vedeti pred izvajanjem ali bodo pogoji stražarjev izpolnjeni ali ne.

Naslednji primer pokaže, da ni mogoče predvideti ali vzorci pokrivajo vse možnosti pred časom izvajanja.

```
# let f = function x when x = x -> true;;
Characters 10-40:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
—
val f : 'a -> bool = <fun>
```

5.2.6 Ujemanje vzorcev na seznamih

Kot smo prej videli je seznam lahko:

- bodisi prazen (seznam oblike []),
- ali sestavljen iz prvega elementa (glava) in podseznama (rep). Seznam je potem oblike h::t.

Ta dva načina pisanja seznamov se lahko uporabi za vzorce.

```
# let rec size x = match x with
  [] -> 0
  | _::tail -> 1 + (size tail) ;;
val size : 'a list -> int = <fun>
# size [] ;;
- : int = 0
# size [7;9;2;6];;
- : int = 4
```

Poglejmo si kako izgleda prejšnji primer funkcije `fold_left` napisan z uporabo vzorcev na seznamih.

```
# let rec fold_left f a = function
  [] -> a
  | head :: tail -> fold_left f (f a head) tail ;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
# fold_left (+) 0 [8;4;10];;
- : int = 22
```

5.2.7 Deklaracija vrednosti preko ujemanja vzorcev

Deklaracija vrednosti lahko uporablja ujemanje vzorcev. V najenostavnejši obliki deklaracija npr. `let x = 18` postavi vzorcu `x` vrednost 18. Na levi strani deklaracije je dovoljen vsak vzorec; spremenljivke vzorca se povežejo z vrednostjo s katero se ujema.

```
# let (a,b,c) = (1, true, 'A');;
val a : int = 1
val b : bool = true
val c : char = 'A'
# let (d,c) = 8, 3 in d + c;;
- : int = 11
```

Definicijsko območje spremenljivk je običajno statično definicijsko območje lokalnih deklaracij. Recimo, da je spremenljivka `c` ostala povezana z vrednostjo `'A'`.


```
# a + (int_of_char c);;
- : int = 66
```

Enako kot v primeru kakršnekoli druge uporabe ujemanja vzorcev je lahko deklaracija vrednosti neizčrpana.

```
# let [x;y;z] = [1;2;3];;
Characters 5-12:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val x : int = 1
val y : int = 2
val z : int = 3
# let [x;y;z] = [1;2;3;4];;
Characters 4-11:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
Uncaught exception: Match_failure("", 4, 11)
```

Dovoljena je uporaba poljubnega vzorca, ki vsebuje konstruktorje, jokerje in sestavljene vzorce.

```
# let head :: 2 :: _ = [1; 2; 3] ;;
Characters 5-19:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val head : int = 1
# let _ = 3. +. 0.14 in "PI";;
- : string = "PI"
```

Zadnji primer ni uporaben v funkcijskem svetu, ker sestavljena vrednost ni imenovana in je zato izgubljena.

5.2.8 Primeri funkcij z vzorci

V nadaljevanju bo predstavljena uporaba ujemanja na primerih funkcij, ki so bile definirane v Sekciji ??.

Funkcijo size lahko zdaj definiramo na sledeč način.

```
# let rec size = function
  [] -> 0
```

```

    | _ -> 1 + (size (List.tl l)) ;;
val size : 'a list -> int = <fun>
# size [] ;;
- : int = 0
# size [1;2;18;22] ;;
- : int = 4

```

Funkcija `append` je ena izmed standardnih funkcij funkcijskega programskega jezika. Večina jezikov jo ima že implementirano znotraj samega jezika kot npr. Ocaml s funkcijo `(@)`.

```

# let rec append = function
  [] , l -> l
  | h::t, l -> h :: append (t,l);;
val append : 'a list * 'a list -> 'a list = <fun>
# append ([1;2],[3;4;5]);;
- : int list = [1; 2; 3; 4; 5]

```

Curry oblika funkcije `append` je sledeča .

```

# let rec curried_append l1 l2 =
  match l1 with
  [] -> l2
  | h::t -> h :: curried_append t l2;;
val curried_append : 'a list -> 'a list -> 'a list = <fun>
# curried_append [1;2] [3;4;5];;
- : int list = [1; 2; 3; 4; 5]

```

Naslednja funkcija obrne seznam. Spet je uporabljena linearna rekurzija. Ustavitveni pogoj rekurzije je `l = []`. Funkcija se rekurzivno pokliče na repu vhodnega seznama. Zapis funkcije z uporabo vzorcev je precej enostavnejši kot zapis z uporabo stavkov `if`.

```

# let rec reverse l = match l with
  [] -> []
  | head :: tail -> (reverse tail) @ [head];;
val reverse : 'a list -> 'a list = <fun>
# reverse [1;2;3;4];;
- : int list = [4; 3; 2; 1]

```

Naslednja funkcija obrne vsebino parov v seznamu. Tudi ta funkcija je veliko bolj razumljiva v zapisu, ki uporablja vzorce.

```

# let rec flip l = match l with
  [] -> []
  | (a,b) :: tail -> (b,a) :: (flip tail);;

```

```
val flip : ('a * 'b) list -> ('b * 'a) list = <fun>
# flip [(1,2);(3,4)];;
- : (int * int) list = [(2, 1); (4, 3)]
```

Poglejmo si še implementacijo operacij nad množicami z uporabo vzorcev.

```
# let rec member x l = match l with
  [] -> false
  | a :: _ when a=x -> true
  | _ -> member x (List.tl l);;
val member : 'a -> 'a list -> bool = <fun>
# member 3 [2;3;1];;
- : bool = true
# let rec inter (xs, ys) = match xs with
  [] -> []
  | x :: xr when (member x ys) -> x :: (inter (xr, ys))
  | _ :: xr -> inter(xr, ys);;
val inter : 'a list * 'a list -> 'a list = <fun>
# inter ([1;2;3],[4;2]);;
- : int list = [2]
# let rec union (xs, ys) = match xs with
  [] -> ys
  | x :: xr when (member x ys) -> (union (xr, ys))
  | x :: xr -> x :: union (xr, ys);;
val union : 'a list * 'a list -> 'a list = <fun>
# union ([1;2;3],[1;4;5]);;
- : int list = [2; 3; 1; 4; 5]
```

Za vajo lahko naredite sledeče. Primerjajte implementacije funkcij za delo z množicami v Sekciji ?? in tukaj.

5.3 Implementacija funkcij

- Kako je funkcija realizirana?
- Kaj se zgodi ob funkcijskem klicu
- Prenos parametrov
- Življenska doba spremenljivk
- Aktivacijski zapisi
- Statični aktivacijski zapisi
- Sklad aktivacijskih zapisov
- Vgnezdene funkcije

Povezovanje spremenljivk.

- Spremenljivke so (dinamično) povezane z vrednostmi.
- Vrednosti se morajo nekje shraniti.
- Spremenljivke morajo biti nekako povezane s spominskimi lokacijami.
- Kako?

Funkcijski jeziki srečajo imperativne...

- Imperativni jeziki slonijo na konceptu spominske lokacije: `a := 0`
- Shrani nič na spominsko lokacijo `a`
- Funkcijski jeziki to skrijejo: `val a = 0`
- Poveži `a` z vrednostjo nič
- Oboji morajo povezati spremenljivke z vrednostmi, ki so predstavljene v spominu.
- V obeh primerih pridemo do istega vprašanja o povezovanju.

5.3.1 Prenos parametrov

Poglejmo si definicijo funkcije:

```
# let inc a = a+1;;
val inc : int -> int = <fun>
# inc 3;;
- : int = 4;;
```

Kako se prenašajo parametri?

Formalen parameter `a` je dostopen znotraj funkcije.
Dejanski parameter `3` je prenešen iz okolja klica funkcije `inc`.

Pogledali si bomo kako je lahko implementiran prenos parametra.
 Predstavljena bosta dva načina prenosa parametrov: prenos po vrednosti in prenos po referenci. Preden si ogledamo prenos parametrov bo najprej bolj podrobno predstavljeno kako se ujamejo imena formalnih in dejanskih parametrov.

Ujemanje parametrov

Kako se ujemajo formalni in dejanski parametri?
 Kateri formalni parametri se ujemajo s katerimi dejanskimi parametri?
 Najbolj pogost primer: pozicijski parametri.
 Korespondenca je določena s pozicijo.

N-ti formalni parameter se ujema z n-tim dejanskim parametrom.

Imenovani parametri.

Ujemanje je določeno s pomočjo imen parametrov.

Ada: DIVIDE(DIVIDEND => X, DIVISOR => Y);

Ujemanje aktualnega parametra X s formalnim parametrom DIVIDEND in Y z DIVISOR

Vrstni red je tukaj nepomemben.

Mešano: imena in pozicije.

Večina jezikov, ki podpira imenovane parametre dovoljuje oboje:

Ada, Fortran, Dylan, Python.

Prvi parameter v seznamu je lahko pozicijski, ostali pa so lahko določeni z imeni.

Opcijske in privzete vrednosti.

Opcijsko privzete vrednosti:

Seznam formalnih parametrov vsebuje tudi seznam privzetih vrednosti, ki se uporabijo, če manjka pripadajoči dejanski parameter.

Omogoča kratek zapis nekaterih tipov prekritih (angl. overloaded) funkcij.

Neomejen seznam parametrov.

Nekateri jeziki dovoljujejo, da je seznam dejanskih parametrov poljubno velik:

C, C++, in skriptni jeziki kot npr. JavaScript, Python, in Perl.

Uporabiti je potrebno rutine iz knjižnice za dostop do dejanskih parametrov.

Luknja v statičnem sistemu tipov, ker se tipov parametrov ne more preveriti v času prevajanja.

Tipični primer funkcije: printf v C.

```
int printf(char *format, ...) { body }
```

Prenos po vrednosti

Pri prenosu parametra po vrednosti je formalni parameter podoben lokalni spremenljivki, ki se nahaja v aktivacijskem zapisu klicane metode. Edina razlika je v tem, da se formalni parameter inicializira z vrednostjo dejanskega parametra ob klicu metode.

- Najenostavnejša metoda.
- Široko uporabna.
- Edini tip prenosa v Javi.

Poglejmo si najprej primer v Javi. Funkcija plus prišteje vrednost spremenljivke b k vrednosti spremenljivke a, ki se vrne kot rezultat funkcije plus. Po klicu funkcije plus znotraj funkcije f se prepiseta vrednosti spremenljivk x in y v spremenljivke a in b.

```
int plus(int a, int b) {
    a += b;
    return a;
}
void f() {
    int x = 3;
    int y = 4;
    int z = plus(x, y);
}
```

Poglejmo si isti primer napisan v Ocaml. Funkcija plus ima dva parametra: a in b. Oba sta tipa int, ker je uporabljeno celoštevilsko seštevanje. Ob klicu funkcije plus se v telesu funkcije f prepiseta vrednosti spremenljivk x in y v parametra a in b.

```
# let plus (a,b) = let a = a + b in a;;
val plus : int * int -> int = <fun>
# let f =
    let x = 3
    and y = 4
    in plus (x,y);;
val f : int = 7
```

Parameter a je redefiniran v telesu funkcije plus: vrednost spremenljivke a dobi vrednost a + b. Nova vrednost spremenljivke a se ne prenese izven okolja kot bi se v primeru prenosa parametrov po referenci.

Prenos po referenci

Pri prenosu parametrov po referenci se naslov (lvalue) dejanskega parametra izračuna preden se metoda pokliče. Znotraj metode se naslov uporabi za naslov (lvalue) pripadajočega formalnega parametra. Formalni parameter je torej nadomestno ime za dejanski parameter — drugo ime za isto spominsko lokacijo.

- Ena od starejših metod: Fortran, Pascal, itd.
- Najbolj učinkovita za velike objekte.
- Še vedno se pogosto rabi.

Poglejmo si primer prenosa parametrov po referenci, kot bi bil definiran v Javi.

```
void plus(int a, by-reference int b) {
    b += a;
}
void f() {
    int x = 3;
    plus(4, x);
}
```

Parameter b postane po klicu funkcije plus spremenljivka, ki ima isti naslov vrednosti kot x. Ob spremembi b se hkrati spremeni tudi vrednost x. Poglejmo si še implementacijo istih funkcij v C.

```
void plus(int a, int *b) {
    *b += a;
}
void f() {
    int x = 3;
    plus(4, &x);
}
```

Prenos po referenci je zelo pomemben pri prenosu večjih struktur kot so na primer polja, objekti, itd. Java praviloma uporablja prenos po referenci za vse naštetje strukture.

Drugi načini prenosa parametrov

Prenos po rezultatu.

- Pri prenašanju vrednosti parametra po rezultatu je formalni parameter enak kot (neinicijalizirana) lokalna spremenljivka v aktivacijskem zapisu.
- Po klicu metode se končna vrednost formalnega parametra priredi pripadajočemu dejanskemu parametru.

Prenos po vrednosti in rezultatu.

Pri prenosu parametrov po vrednosti in rezultatu je formalni parameter enak kot lokalna spremenljivka v aktivacijskem zapisu klicane metode. Vrednost se inicializira z vrednostjo pripadajočega dejanskega parametra pred izvajanjem metode. Po tem, ko se metoda konča se prepíše formalni parameter v dejanskega.

Ekspanzija makroja.

Pri prenosu parametrov z makro ekspanzijo se telo makroja evaluiira v kontekstu metode, ki kliče makro. Vsak dejanski parameter se evaluiira pri vsaki uporabi pripadajočega formalnega parametra v kontekstu pojavitve formalnega parametra oz. v kontekstu metode, ki kliče makro. Poglejmo si primere uporabe makrojev v programskem jeziku C.

```
#include <stdio.h>

#define IZRAZ          1 + 2 + 3 + 4
#define ABS(x)        ((x) < 0) ? -(x) : (x)
#define MAX(a,b)      ((a < b) ? (b) : (a))

int main ()
{
    printf ("%d\n", IZRAZ);
    printf ("%d\n", ABS(-5));
    printf ("Večja vrednost od 2 in 9 je %d\n", MAX(2,9));
}
```

5.3.2 Aktivacijski zapisi

- Življenska doba funkcije od klica do vrnitve imenujemo *aktivacija funkcije*.
- Vsaka aktivacija ima svoje povezave spremenljivk s spominskimi lokacijami.
- Spremenljivka se kreira v okviru aktivacije funkcije.

Bloki in spremenljivke.

- Bloki so v vseh jezikih konstrukt, ki vsebuje definicije in določa področje programa kjer so definicije uporabne.
- Blok določa *definijsko območje* spremenljivk.
- Blok je definiran z gradnikom let kot tudi s funkcijo.
- Poglejmo si primer definicije bloka v Caml.

```
# let x = 1
  and y = 2
  in x+y;;
- : int = 3
```

V naslednjem primeru definiramo funkcijo cube, ki definira blok s spremenljivko x.

```
# let cube x = x*x*x;;
val cube : int -> int = <fun>
```


Večkratne alternative vsebujejo večkratne bloke. Poglejmo si primer funkcije definirane z vzorci, ki ima tri bloke.

```
# let f = function
  a::b::_ -> a+b
  | [a] -> a
  | [] -> 0;;
val f : int list -> int = <fun>
```

Poglejmo si še primer v Javi. Naslednji blok je definiran v stavku while. Blok vsebuje lokalno definicijo spremenljivke c, medtem ko so ostale spremenljivke definirane v nadrejenih blokih.

```
while (i < 0) {
  int c = i*i*i;
  p += c;
  q += c;
  i -= step;
}
```

V večini jezikov lahko spremenljivke definirane v vgnezenem bloku ponovno definirajo spremenljivko z istim imenom. V naslednjem primeru notranji blok redefinira spremenljivko. Ocaml sporoči tudi, da ena izmed spremenljivk ni bila uporabljena.

```
# let n = 1 in let n = 2 in n;;
Warning Y: unused variable n.
- : int = 2
```

Aktivacije blokov.

Aktivacija bloka je življenska doba enega izvajanja bloka. V naslednjem primeru imamo dva bloka: zunanji definiran s funkcijo fact in notranji definiran z vgnezenim let stavkom. Parameter n je definiran samo znotraj funkcije medtem ko je spremenljivka b definirana samo v vgnezenem bloku.

```
# let rec fact n =
  if (n=0) then 1
  else let b = fact (n-1) in n*b;;
```

Življenska doba spremenljivk.

- Življenska doba spremenljivke definirane znotraj danega bloka je čas v katerem je blok aktiviran.

- Življenska doba je torej vezana na definicijsko območje.
- Temu pravimo *dinamična alokacija* spremenljivk.
- Večina imperativnih jezikov vsebuje načine za deklaracijo spremenljivk, ki so vezane na eno samo spominsko lokacijo za celoten čas izvajanja programa.
- Uporabljamo *statično alokacijo*.
- Območje definicije torej ni vedno povezano z življensko dobo.
- C in C++ dovolita definicijo statičnih spremenljivk v funkcijah.

```
int nextcount() {
    static int count = 0;
    count = count + 1;
    return count;
}
```

Druge življenske dobe spremenljivk.

- Objektno usmerjeni jeziki uporabljajo življensko dobo, ki je vezana na življensko dobo objektov.
- Nekateri jeziki imajo spremenljivke katerih vrednosti so persistentne.
- Življenska doba spremenljivk se razteza na več izvajanj programa.

Aktivacijski zapis in imenski prostor.

- Vse spremenljivke definirane pri aktivaciji funkcije ali bloka so shranjene v *aktivacijskem zapisu*.
- Tabela simbolov znotraj aktivacijskega zapisa določa *imenski prostor*.

Komponente aktivacijskega zapisa.

- 1) *Kontrolni kazalec*. - Naslov aktivacijskega zapisa okolja (običajno funkcija) iz katerega se sprožila aktivacija funkcije.
- 2) *Kazalec dostopa* (angl. access pointer).
 - Naslov nadrejenega imenskega prostora (aktivacijskega zapisa).
 - Kot bomo videli lahko imamo različne načine povezovanja imenskih prostorov: statično in dinamično.
- 3) Naslov za vrnitev funkcije.
- 4) Parametri.
- 5) Lokalne spremenljivke.
- 6) Začasni rezultat funkcije.
- 7) Naslov kamor je potrebno zapisati rezultat po vrnitvi.

Aktivacijski zapis bloka.

- Ko se začne izvajati blok je potrebno definirati podatkovne strukture, ki hranijo spremenljivke bloka
- Več možnosti:
 - 1) Vnaprejšnja definicija pomnilniškega prostora za spremenljivke bloka znotraj aktivacijskega zapisa funkcije.
 - 2) Razširitev aktivacijskega zapisa funkcije ob vstopu v blok (čiščenje ob vnitvi).
 - 3) Zaseži ločene aktivacijske bloke.

Implementacija aktivacijskih zapisov.

- 1) Statičen zapis:
 - Lokacija aktivacijskega zapisa se je določila v času prevajanja.
- 2) Dinamičen zapis:
 - Lokacija trenutnega aktivacijskega zapisa je znana v času izvajanja.
 - Funkcija mora vedeti kako poiskati naslov trenutnega aktivacijskega zapisa.
 - Pogosto se ta naslov hrani v registru.

5.3.3 Statični aktivacijski zapisi

Statična alokacija.

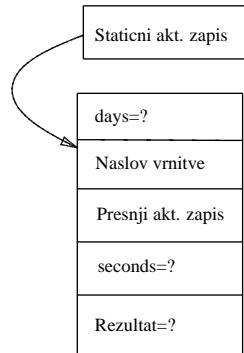
- Najenostavnejši pristop:
- Alociraj en aktivacijski zapis za eno funkcijo.
- Statična alokacija.
- Starejši dialekti Fortrana uporabljajo ta sistem.
- Enostavno in hitro.

Slabe lastnosti.

- Vsaka funkcija ima en aktivacijski zapis
- Ob danem času je lahko živa le ena aktivacija funkcije.
- Moderni programski jeziki ne izpolnjujejo teh pogojev.
- Rekurzija
- Multinitenje

Poglejmo si primer statičnega aktivacijskega zapisa za funkcijo days2ms. Statični aktivacijski zapis je predstavljen na Sliki ??.

```
# let days2ms days =
let seconds = days*24*60*60
in seconds * 1000;;
```



Slika 5.1: Statični aktivacijski zapis za days2ms

5.3.4 Skladi aktivacijskih zapisov

- Za podporo rekurziji moramo zaseči novi aktivacijski zapis za vsako aktivacijo.
- = Dinamična alokacija:
 - Aktivacijski zapis se alocira, ko se funkcija pokliče.
 - V večini jezikov (npr. C) se aktivacijski zapis dealocira, ko se funkcija izvrši.
- = Sklad aktivacijskih zapisov:
 - Okvirji na skladu se naložijo ob klicu in sprostijo ob vrnitvi.

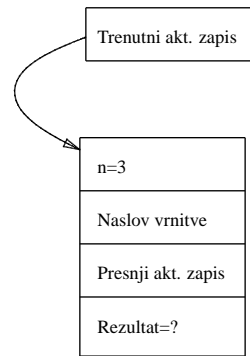
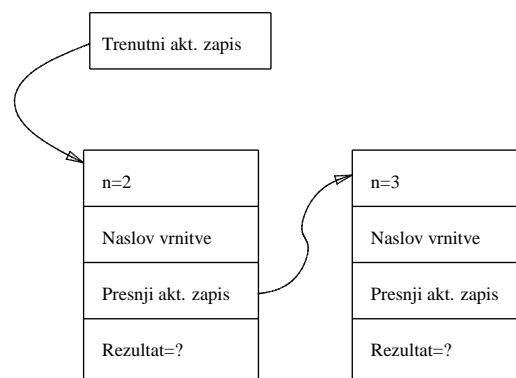
Povezovanje imenskih prostorov.

- = Statično povezovanje imenskih prostorov.
 - Naslov nadrejenega imenskega prostora je naslov aktivacijskega zapisa, ki je strukturno nadrejen klicani funkciji.
- = Dinamično povezovanje imenskih prostorov.
 - Naslov nadrejenega imenskega prostora je naslov aktivacijskega zapisa iz katerega se je klicala funkcija.

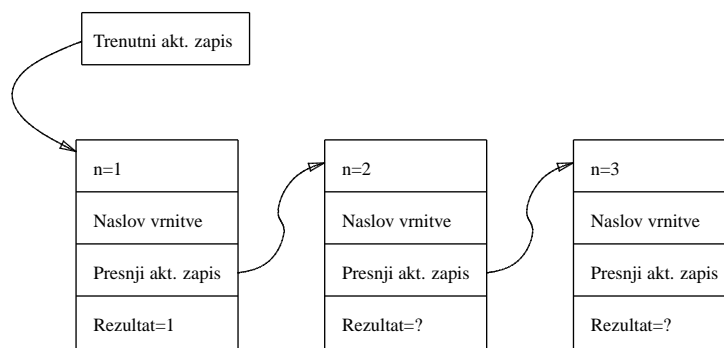
Poglejmo si primer evaluacije funkcije `fact`. Prikazali bomo zaporedje kreacij aktivacijskih zapisov ob rekurziji ter sproščanje aktivacijskih zapisov ob vračanju iz rekurzije.

```
# let rec fact n =
  if (n=0) then 1
  else let b = fact (n-1) in n*b;;
```

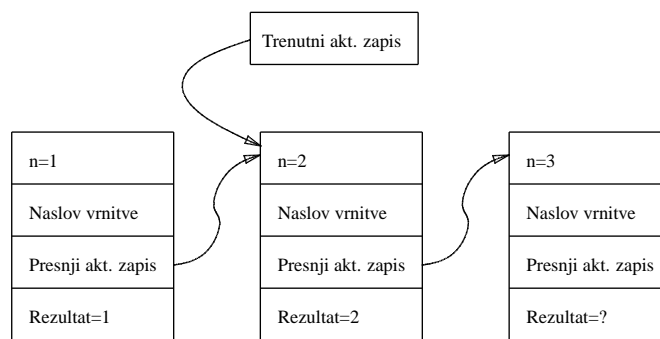
Stanje verige aktivacijskih zapisov je prikazano ob kreaciji in ob sprostitvi vsakega aktivacijskega zapisa v verigi.

Slika 5.2: Evaluiramo `fact(3)`

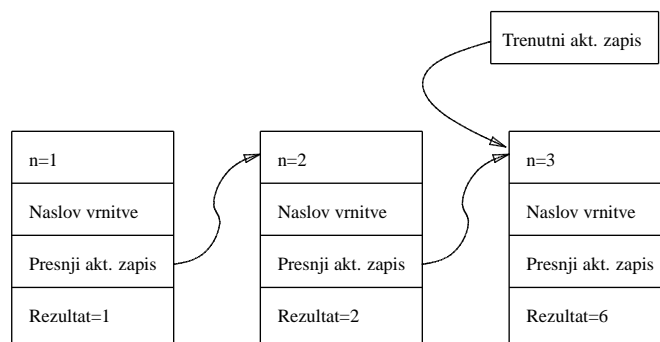
Slika 5.3: Stanje v spominu tik pred tretjo aktivacijo funkcije



Slika 5.4: Stanje v spominu tik preden se tretja aktivacija funkcije zaključi



Slika 5.5: Druga aktivacija se zaključí



Slika 5.6: Tik preden se prva aktivacija funkcije zaključí

- Večina programskih jezikov dovoljuje funkcije vgnezdene v drugih funkcijah.
- Vgnezdena funkcija lahko dela s spremenljivkami, ki so definirane v zunanjih funkcijah.
- Uporabimo običajno pravilo območja bloka.
- ML, Ada, Pascal, itd.

Poglejmo si malce bolj obsežen primer klica funkcije, ki ima vgnezdeno funkcijo. Vgnezdena funkcija mora dostopati do spremenljivk, ki so definirane v strukturno nadrejeni funkciji z uporabo kazalca dostopa, ki kaže na nadrejeni imenski prostor.

Funkcija `quicksort` sortira seznam tako, da seznam razdeli na tiste elemente, ki so manjši od izbranega pivota in tiste, ki so večji od pivota. Izbrane dele vhodnega seznama rekurzivno sortiramo s klicem funkcije `quicksort` nad vsakim od delov. Sortirane dele spnemo v nov sortiran seznam skupaj s pivotom.

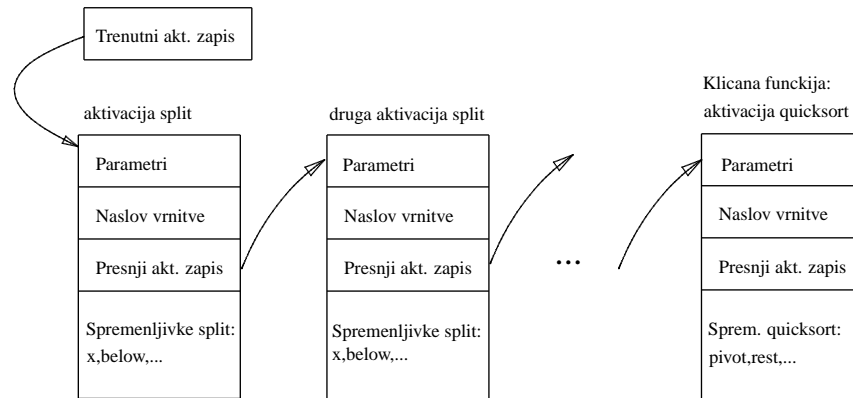
```
# let rec quicksort = function
  [] -> []
  | pivot::rest ->
    let rec split = function
      [] -> ([],[])
      | x::tail ->
        let (below, above) = split tail
        in
          if x<pivot then (x::below, above)
          else (below, x::above)
    in let (below, above) = split rest
      in quicksort below @ [pivot] @ quicksort above;;
val quicksort : 'a list -> 'a list = <fun>
```

Dostop do nadrejenih imenskih prostorov.

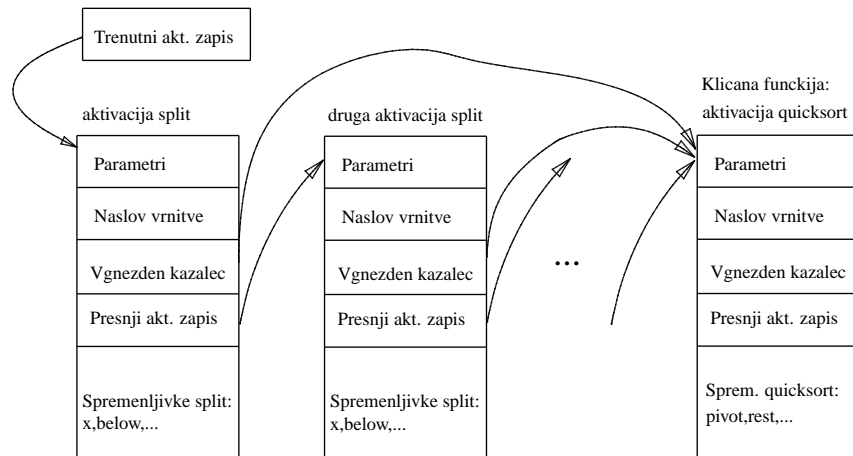
- Kako naj aktivacija notranje funkcije (`split`) najde aktivacijski zapis zunanje funkcije (`quicksort`)?
- To ni nujno prejšnji aktivacijski zapis, ker lahko notranja funkcija pokliče drugo notranjo funkcijo.
- Notranja funkcija se lahko izvaja rekurzivno tako kot `split`...
- Notranja funkcija potrebuje dostop do naslova zadnjega aktivacijskega bloka zunanje funkcije.
- Uporaba kazalca dostopa.

Uporaba in postavljanje kazalca dostopa.

- Enostavno, če obstaja samo en nivo gnezdenja:
- Klic zunanje funkcije \rightarrow null



Slika 5.7: Veriga akt.zapisov pri klicu quicksort



Slika 5.8: Veriga akt.zapisov s kazalcem dostopa pri klicu quicksort

- Zunanja funkcija kliče notranjo \rightarrow postavi kazalec dostopa na aktivacijski zapis zunanje funkcije.
- Notranja funkcija kliče notranjo \rightarrow postavi kazalec dostopa na isto vrednost kot funkcija, ki kliče.
- Bolj kompleksen scenarij dobimo, če imamo več nivojev gnezdenja.

Več nivojev gnezdenja.

- Reference na istem nivoju uporabljajo $((f_1 \text{ do } v_1) f_2 \text{ do } v_2) f_3 \text{ do } v_3$ trenutni aktivacijski zapis.
- Reference n nivojev gnezdenja navzgor se verižijo n vgnezdenih nivojev navzgor.

5.4 Polja

Polja so enodimenzionalna podatkovne strukture, ki vsebujejo poznano število elementov istega tipa. Elementi polja so urejeni glede na indekse elementov polja, ki imajo celoštevilске vrednosti od 0 do $n-1$.

5.4.1 Definicija polja in osnovne operacije

Polje definiramo tako, da zapišemo vse elemente polja med oklepaji `[]` in `[]`. Elementi so ločeni s podpičji.

```
# let v = [| 3.14; 6.28; 9.42 |] ;;
val v : float array = [|3.14; 6.28; 9.42|]
```

Funkcija za kreiranje polja `Array.create` vzame število elementov polja in začetno vrednost elementov ter vrne novo polje.

```
# let v = Array.create 3 3.14;;
val v : float array = [|3.14; 3.14; 3.14|]
```

Za dostop in modifikacijo konkretnega elementa uporabimo indeks elementa:

Sintaksa:

```
expr1.( expr2 )
expr1.( expr2 ) <- expr3
```

`expr1` mora biti polje katerega vrednosti imajo tip `expr3`. Izraz `expr2` mora seveda biti tipa `int`. Tip modifikacijske funkcije je `unit`.

Prvi element ima indeks 0 in indeks zadnjega elementa je dolžina polja minus 1. Oklepaji okoli indeksa so nujni.

```
# v.(1) ;;
- : float = 3.14
# v.(0) <- 100.0 ;;
- : unit = ()
# v ;;
- : float array = [|100; 3.14; 3.14|]
```

Če je indeks uporabljen pri dostopu do elementa izven meja polja se sproži izjema v času dostopa.

```
# v.(-1) +. 4.0;;
Uncaught exception: Invalid_argument("Array.get")
```

Preverjanje mej se izvaja med izvajanjem programa kar ga upočasnja. Preverjanje je nujno, da bi se izognili dostop do spomina izven meja definirane polja.

...dolžina polja...

5.4.2 Primeri funkcij nad polji

Poglejmo si nekaj primerov operacij za delo s polji in implementacijo v Ocaml. Naslednji primer prikaže definicijo celoštevilskega polja v, ki ima 10 elementov.

```
# let n = 10;;
val n : int = 10
# let v = Array.create n 0;;
val v : int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
```

Polje napolnimo z uporabo for stavka tako, da vsebuje vsak element indeks danega elementa.

```
# for i=0 to (n-1) do v.(i)<-i done;;
- : unit = ()
# v;;
- : int array = [|0; 1; 2; 3; 4; 5; 6; 7; 8; 9|]
```

Naslednji primer prikaže implementacijo operacije, ki sešteje vse elemente polja v in vrne vsoto kot rezultat.

```
# let sum = ref 0
  in for i=0 to (n-1) do sum := (!sum) + v.(i) done; !sum;;
- : int = 45
```

Vsebino elementov polja lahko obrnemo z uporabo enega for stavka. Indeks for stavka teče do polovice polja oz. do indeksa $(n/2-1)$ ¹. Vsebino polja zamenujemo po parih elementov: prvi in zadnji, drugi in predzadnji in tako naprej do $(n/2-1)$, kar je

```
# let tmp=ref 0
  in for i=0 to (n/2-1) do
    tmp := v.(i);
    v.(i) <- v.(n-i-1);
    v.(n-i-1) <- (!tmp);
  done;;
- : unit = ()
# v;;
- : int array = [|9; 8; 7; 6; 5; 4; 3; 2; 1; 0|]
```

Poglejmo si zdaj malce težji problem, kjer preverimo ali je dan niz celih števil u podniz niza celih števil v. V primeru, da napišemo celoten program z eno samo funkcijo je rešitev kar kompleksna.

Osnovna ideja rešitve je sledeča. Funkcija subarray u v naj preveri ali je u pod-polje polja v. Z indeksom i se pomikamo po polju od prvega elementa proti zadnjemu do elementa z indeksom n-m, ki je zadnji od koder lahko še primerjamo polje. Na vsakem mestu določenim z i pogledamo ali se polje u ujema z poljem v na danem mestu i.

```
# let u = [|2;3|];;
val u : int array = [|2; 3|]
# let m = 2;;
val m : int = 2
# let subarray u v =
  let found = ref false
  and i = ref 0
  in while ((!i<=(n-m)) && not !found) do
    found := true;
    for j=0 to (m-1) do
      if v.(!i+j) != u.(j) then
        found := false
    done;
    i := !i+1
  done;
  !found;;
val subarray : 'a array -> 'a array -> bool = <fun>
# subarray u v;;
- : bool = true
```

¹Delejenje s celoštevilskimi argumenti vrne navzdol zaokrožen celoštevilski rezultat.

Veliko enostavneje je problem razbiti na dva dela. Najprej definiramo funkcijo `prefix`, ki preveri ali je dan niz `u` podniz niza `v` na `i`-tem mestu.

```
# let prefix u v i =
  let found = ref true
  in for j=0 to (m-1) do
    if v.(i+j) != u.(j) then
      found := false
  done;
  !found;;
val prefix : 'a array -> 'a array -> int -> bool = <fun>
# prefix u v 0;;
- : bool = false
# prefix u v 3;;
- : bool = true
```

Zdaj je potrebno samo še pogledati ali se `u` ujema z na enim izmed možnih mest znotraj polja `v`.

```
let subarray u v =
  let found = ref false
  and i = ref 0
  in while ((!i<=(n-m)) && not !found) do
    found := prefix u v !i;
    i := !i+1
  done;
  !found;;
```

5.4.3 Znakovni nizi

Znakovni nizi se obravnavajo kot posebni primeri polj, ki vsebujejo znake. Zaradi učinkovite predstavitve je definiran poseben tip znakovnih nizov ter metode za dostop do elementov.

Sintaksa:

```
expr1.[expr2]
```

Elemente znakovnih nizov lahko spreminjamo.

Sintaksa:

```
expr1.[expr2] <- expr3
```

Poglejmo si par primerov.

```
# let s = "hello";
val s : string = "hello"
# s.[2];;
- : char = 'l'
# s.[2]<-'Z';;
- : unit = ()
# s;;
- : string = "heZlo"
```

5.4.4 Implementacija polja

Polje je v spominu predstavljeno na zaporednih lokacijah. Vsak element polja je predstavljen s sekvenco besed, katere velikost je odvisna od velikosti binarnega zapisa tipa elementa. Elementi polja si od začetnega naslova polja sledijo zaporedno.

V primeru, da so elementi polja cela števila potem so predstavljena s sekvenco dveh, štirih ali osmih zlogov odvisno od velikosti celega števila. V Ocaml so cela števila predstavljena bodisi z 31 ali z 63 biti v odvisnosti od uporabljenega konkretnega tipa celih števil npr. `int32`.

Poglejmo si nekaj primerov definicije polj v Ocaml preko katerih lahko bolj spoznamo fizično predstavitev polj. Naslednja definicija polja kreira polje celih števil dolžine 3. Začetno vsi elementi vsebujejo vrednost 0.

```
# let v = Array.create 3 0;;
val v : int array = [|0; 0; 0|]
# let m = Array.create 3 v;;
val m : int array array = [| [|0; 0; 0|]; [|0; 0; 0|]; [|0; 0; 0|] |]
```

Polje `m` je definirano na osnovi polja `v`. Elementi polja `m` so začetno inicializirani na začetni naslov polja `v`. Če zdaj spremenimo en element polja `v`, potem se hkrati spremenijo vse vrstice matrike.

```
# v.(0) <- 1;;
- : unit = ()
# m;;
- : int array array = [| [|1; 0; 0|]; [|1; 0; 0|]; [|1; 0; 0|] |]
```

V primeru, da je inicializacijska vrednost vektorja atomična vrednost se le-ta pomnoži s kopiranjem vsebine, medtem ko se v primeru podane strukturirane vrednosti le-ta deli tolikokrat kolikokrat je pomnožena.

Strukturirane vrednosti so namreč predstavljene s kazalcem na vsebino strukture–pri pomnoževanju se prepisuje samo naslov.

Polja realnih števil so izjema: čeprav so realna števila strukturirane vrednosti se prepiše vedno vsebina in ne kazalec.

Pogljemo si še en primer, kjer je potrebno poznavanje interne predstavitve polj. Ko prekopiramo polje ali spnemo dva polja dobimo rezultat, ki je novo polje. Sprememba na originalnem polju ne povzroči spremembe na novo nastalih poljih, ker so nove vrednosti samostojni objekti in si ne delijo vrednosti z originali.

```
# let v2 = Array.copy v ;;
val v2 : int array = [|1; 0; 0|]
# let m2 = Array.copy m ;;
val m2 : int array array = [| [|1; 0; 0|]; [|1; 0; 0|]; [|1; 0; 0|] |]
# v.(1) <- 352 ;;
- : unit = ()
# v2 ;;
- : int array = [|1; 0; 0|]
# m2 ;;
- : int array array = [| [|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|] |]
```

Polje m2 kreiramo tako, da kreiramo novo polje kamor se prepiše polje m. Če spremenimo vrednost vektorja v katerega naslov je bil uporabljen za inicializacijo elementov polja m, potem se spremeni tudi polje m2. Elementi polja m in m2 so naslovi polja v.

5.4.5 Matrike

Abstrakcija matrika:

n vrstic in m stolpcev
matematična predstava matrike
tabelarična abstrakcija matrike

Implementacija matrike:

fiksna struktura matrike v spominu
polja enostavnih vrednosti in referenc,
polja polj

Primeri matrike:

definicija matrike
inicializacija matrike
operacije za delo z matrikami

Poglejmo si najprej običajne, pravokotne matrike, ki se pogosto uporabljajo za implementacijo numeričnih metod. Naslednja matrika ima velikost 4x4 – indeksa tečeta od 0 do 3.

```
# let m = Array.create_matrix 4 4 0;;
val m : int array array =
  [| [|0; 0; 0; 0|]; [|0; 0; 0; 0|]; [|0; 0; 0; 0|]; [|0; 0; 0; 0|] |]
# for i=0 to 3 do m.(i).(i) <- 1; done;;
- : unit = ()
# m;;
- : int array array =
  [| [|1; 0; 0; 0|]; [|0; 1; 0; 0|]; [|0; 0; 1; 0|]; [|0; 0; 0; 1|] |]
# m.(1);;
- : int array = [|0; 1; 0; 0|]
```

Matrika je podobno kot je nakazano na primeru polj, konstruirana iz polja kazalcev na polja celih števil, v zgornjem primeru.

Oblika matrike ni nujno, da je pravilna oz. pravokotna. Vsebino elementov polja lahko predstavljajo polja različne dolžine. Takšna matrika lahko na primer shrani vrstice teksta; vsaka vrstica je predstavljena s poljem vgnezenim v polje.

Poglejmo si primer polja polj celih števil. Naslednja matrika vsebuje koeficiente Pascalovega trikotnika.

```
# let t = [|
          [|1|];
          [|1; 1|];
          [|1; 2; 1|];
          [|1; 3; 3; 1|];
          [|1; 4; 6; 4; 1|];
          [|1; 5; 10; 10; 5; 1|]
        |] ;;
val t : int array array =
  [| [|1|]; [|1; 1|]; [|1; 2; 1|]; [|1; 3; 3; 1|]; ... |]
# t.(3) ;;
- : int array = [|1; 3; 3; 1|]
```

Elementi polja t z indeksom i imajo dolžino je $i + 1$. Pri delu s takšnimi matrikami moramo izračunati dolžino vsake vrstice.

5.4.6 Računanje z matrikami

V naslednjih primerih si bomo ogledali matrike realnih števil ter nekaj osnovnih operacij nad matrikami.

Najprej bomo napisali definirali dve matriki m in u , ki imata dimenzije 3x3.

```

# let n = 3 and m=3;;
val n : int = 3
val m : int = 3
# let a = Array.create_matrix n m 0.0;;
val a : float array array =
  [| [|0.; 0.; 0. |]; [|0.; 0.; 0. |]; [|0.; 0.; 0. |] |]
# let b = Array.create_matrix n m 0.0;;
val b : float array array =
  [| [|0.; 0.; 0. |]; [|0.; 0.; 0. |]; [|0.; 0.; 0. |] |]

```

Vsota matrik a in b je definirana z naslednjo funkcijo, ki sešteje istoležne elemente matrik.

```

# let add_mat a b =
  let r = Array.create_matrix n m 0.0 in
  for i = 0 to (n-1) do
    for j = 0 to (m-1) do
      r.(i).(j) <- a.(i).(j) +. b.(i).(j)
    done
  done ; r;;
val add_mat : float array array -> float array array -> float array array = <fun>
# a.(0).(0) <- 1.0; a.(1).(1) <- 2.0; a.(2).(2) <- 3.0;;
- : unit = ()
# b.(0).(2) <- 1.0; b.(1).(1) <- 2.0; b.(2).(0) <- 3.0;;
- : unit = ()
# add_mat a b;;
- : float array array = [| [|1.; 0.; 1. |]; [|0.; 4.; 0. |]; [|3.; 0.; 3. |] |]

```

Produkt matrik a in b je definiran z naslednjo funkcijo, ki izračuna produkt na običajen način.

```

# let mul_mat a b =
  let r = Array.create_matrix n m 0.0 in
  for i = 0 to (n-1) do
    for j = 0 to (m-1) do
      let c = ref 0.0 in
      for k = 0 to (m-1) do
        c := !c +. a.(i).(k) *. b.(k).(j)
      done;
      r.(i).(j) <- !c
    done
  done; r;;
# mul_mat a b;;
- : float array array = [| [|0.; 0.; 1. |]; [|0.; 4.; 0. |]; [|9.; 0.; 0. |] |]

```


Poglavje 6

TIPI

Teorija tipov

1901: Russellov paradoks

Teorija množic, ki jo je definiral Cantor vsebuje paradox

Teorija tipov je odgovor na odkritje paradoksa

Vsak matematični objekt je označen s "tipom"

Russel prvi predlaga teorijo tipov leta 1908

Predikatni račun s teorijo tipov ne vsebuje paradoksa

Tipi in λ -račun

Originalen λ -račun, ki ga je razvil Church, tudi vsebuje paradoks

Kleene-Rosser paradoks λ -računa

Church po odkritju paradoksa predlaga λ -račun s tipi

- Churcheva teorija tipov (Church; 1940)

- λ -izrazi so označeni s tipi

- tipi omejujejo možne interpretacije λ -izrazov

Tipi

Tip je semantično gledano en nivo višji objekt od konkretnih vrednosti

S tipi definiramo klasifikacijo objektov in izrazov

Interpretacija tipa predstavlja množico objektov

- povezava s teorijo množic

- primer: int predstavlja tip množice celih števil

Enostavni tipi označujejo enostavne vrednosti

Strukturirani tipi označujejo strukturirane vrednosti

Vrste tipov

Elementarni tipi
 Produkti in n-terice
 Zapisi
 Unije
 Parametrizirani tipi
 Rekurzivni tipi

Strukturiranje podatkov in programa

S tipi definiramo strukture, ki so uporabljene v programu
 Definicija konceptualnega prostora programa
 Nižji nivoji strukture programa
 - produkti, zapisi, unije, rekurzivni tipi
 Strukturirani tipi so uporabljene za strukturiranje podatkov in programa
 - programi so tipi?
 - izvajanje programa je definirano z istimi strukturami kot podatki
 - sezname, unije, produkti
 Višji nivoji strukture modulov in objektov
 - Poglavlji moduli in razredi

Sklepanje s tipi

Iz tipov elementov izraza (programa) izračunamo tip izraza
 Poznamo tipe vseh elementarnih vrednosti in spremenljivk
 - na primer `i:int, 2.0:float, "Niz":string`
 Poznamo tipe vseh operatorjev in funkcij v izrazu
 Iz tipov vrednosti in tipov funkcij izračunamo tip rezultata
 Poglejmo si primer izraza: `List.hd l + 10`

```
10:int, l:int list
(+) : int->int->int, List.hd : 'a list -> 'a
List.hd l + 10 : int
```

Preverjanje tipov

Preverjanje ujemanja tipov izrazov
 Preverjamo v času prevajanja oz. v času izvajanja
 - Velikokrat ne moremo vedeti tip izraza v času prevajanja
 - preverjanje tipov v času izvajanja upočasnjuje izvajanje programa
 - statično preverjanje tipov: C++, Ocaml, C#, itd.
 - dinamično preverjanje tipov: Lisp, Javascript, Ruby, itd.

- oboje: C++, Java, C#, Perl, itd.
- Programski jezik s strogimi tipi striktno preverja tipe vseh izrazov programa
- nekateri prog.jeziki zahtevajo striktno označevanje tipov
- to ni nujno saj je velikokrat jasno kakšnega tipa je izraz

Uporaba tipov

Organizacija programov

- podatkovne strukture in strukture programa (zgoraj)

Dokumentacija programov

- s primernim imenovanjem tipov dokumentiramo program

Odkrivanje napak v programih

- `1+true-"banana"=?`
- s striktno uporabo tipov se izognemo napakam

Podpora optimizaciji programov

- z uporabo tipov vemo več o programu in objektih
- tipi v Fortranu so bili začetno uporabljeni za razlikovanje numeričnih izračunov
- $(100 * 17) + (121.21 * 3.14)$: katera operacija bo uporabljena?

6.1 Deklaracije tipov

Z deklaracijami tipov definiramo nove tipe na osnovi obstoječih tipov in konstruktorjev tipov.

Uporabljamo dva glavna konstruktorja tipov: produkt za n-terice ali zapise in vsota za unije¹

Deklaracije tipov uporabljajo ključno besedo `type`.

Sintaksa:

```
type name = typedef ;;
```

Za razliko od deklaracije spremenljivk je deklaracija tipov privzeto rekurzivna. Deklaracije tipov lahko kombiniramo tako da dobimo medsebojno rekurzivne tipe.

Sintaksa:

```
type name1 = typedef1
and name2 = typedef2
.
```

¹Uvesti formalen pogled na tipe s sintakso Ocaml. Sekcija vsebuje bolj natančno predstavitev specifičnega tipa (*).

```
and namen = typedefn ;;
```

Naslednji primer tipa definira tip `int_pair`. Tip je definiran kot produkt `int*int`. Primerki tega tipa so pari celih števil.

```
# type int_pair = int*int;;
type int_pair = int * int
# let v:int_pair = (1,1);;
val v : int_pair = (1, 1)
```

Vrednost `v` smo označili s tipom `int_pair`. Kot si bomo podrobneje ogledali kasneje, v primeru, da vrednosti nebi označili s tipom Ocaml izračuna najbolj splošen tip.

Deklaracije tipov so lahko tudi prametrizirane s spremenljivkami tipov. Spremenljivka tipa se vedno začne z ' :

Sintaksa:

```
type 'a name = typedef ;;
```

Ko imamo več parametrov tipa jih deklariramo kot n-terico pred imenom tipa. Samo parametri definirani na levi strani deklaracije se lahko pojavijo na desni strani.

Sintaksa:

```
type ('a1 . . . 'an ) name = typedef ;;
```

Poglejmo si še primer parmetriziranega tipa `pair`, katerega komponente so lahko poljubnega tipa.

```
# type ('a,'b) pair = 'a*'b;;
type ('a, 'b) pair = 'a * 'b
# let v:(char,int) pair = ('a',1);;
val v : (char, int) pair = ('a', 1)
```

V naslednjih sekcijah bo predstavljena vrsta primerov definicij tipov. Najprej bodo predstavljeni produkti tipov in nato še zapisi ter unije.

6.2 Enostavni tipi

6.3 Strukturirani tipi

6.3.1 Produkti

Naslednji primer predstavlja definicijo tipa `triple` kot trojico celih števil. Vidimo lahko, da Ocaml vedno izračuna najbolj splošen tip, kot v primeru spremenljivke `l`. Tip vrednosti lahko omejimo z eksplicitnim označevanjem tipa vrednosti, kot v primeru spremenljivke `t`.

```
# type triple = int*int*int;;
type triple = int * int * int
# let l = 1,2,3;;
val l : int * int * int = (1, 2, 3)
# let t:triple = 1,2,3;;
val t : triple = (1, 2, 3)
```

Naslednji primer prikaže definicijo polimorfičnega tipa `triple`, ki ima lahko komponente poljubnega tipa. Kot v prejšnjem primeru je potrebno definirati tip vrednosti eksplicitno.

```
# type ('a,'b,'c) triple = 'a*'b*'c;;
type ('a, 'b, 'c) triple = 'a * 'b * 'c
# let t = 1,'1',"1";;
val t : int * char * string = (1, '1', "1")
# let t:(int,char,string) triple = 1,'1',"1";;
val t : (int, char, string) triple = (1, '1', "1")
```

Poglejmo si še definicijo parametričnega tipa `paired_with_integer`—`par`, kjer je prva komponenta fiksnega tipa `int` in druga komponenta parameter. Ocaml interno pri tiskanju tipov preimenuje parametre v interno obliko: `'a`, `'b`, itd.

```
# type 'param paired_with_integer = int * 'param ;;
type 'a paired_with_integer = int * 'a
# type specific_pair = float paired_with_integer ;;
type specific_pair = float paired_with_integer
# let x:specific_pair = (3, 3.14) ;;
val x : specific_pair = 3, 3.14
```

6.3.2 Zapisi

Zapisi so *n*-terice, ki imajo imenovane komponente. Zapis vedno ustreza neki deklaraciji novega tipa. Tip zapisa je definiran z deklaracijo imena in tipa vsake posamezne komponente.

Sintaksa:

```
type name = { name1 : t1 ; . . . ; namen : tn } ;;
```

Kompleksna števila definiramo na sledeč način.

```
# type complex = { re:float; im:float } ;;
type complex = { re: float; im: float }
```

Primerek zapisa danega tipa kreiramo tako, da priredimo vrednost vsaki posamezni komponenti.

Sintaksa:

```
{ namei1 = expr1 ; . . . ; namein = exprin } ;;
```

Primerek tipa kompleksnega števila kreiramo na naslednji način.

```
# let c = {re=2.;im=3.} ;;
val c : complex = {re=2; im=3}
# c = {im=3.;re=2.} ;;
- : bool = true
```

V primeru, da nekatere komponente manjkajo se generira naslednja napaka.

```
# let d = { im=4. } ;;
Characters 9-18:
Some labels are undefined
```

Do komponente polja lahko dostopamo na dva načina: z uporabo pike ali z ujemanjem vzorcev.

Notacija s piko je naslednja:

Sintaksa:

```
expr.name
```

Izraz `expr` mora biti zapis, ki vsebuje komponento z danim imenom. Ujemanje vzorcev omogoča branje večih komponent hkrati.

Sintaksa:

```
{ namei = pi ; . . . ; namej = pj }
```

Ni potrebno, da uporabimo vse komponente zapisa v vzorcu.

Funkcija `add_complex` dostopa do komponent zapisa z uporabo pike, medtem ko funkcija `mult_complex` dostopa do komponent preko ujemanja vzorcev.

```
# let add_complex c1 c2 = {re=c1.re+c2.re; im=c1.im+c2.im};;
val add_complex : complex -> complex -> complex = <fun>
# add_complex c c ;;
- : complex = {re=4; im=6}
# let mult_complex c1 c2 = match (c1,c2) with
  ({re=x1;im=y1},{re=x2;im=y2}) -> {re=x1*x2-.y1*.y2;
                                   im=x1*.y2+.x2*.y1} ;;
val mult_complex : complex -> complex -> complex = <fun>
# mult_complex c c ;;
- : complex = {re=-5; im=12}
```

Prednosti zapisov v primerjavi z n-tericami so naslednje: opisni način dostopa do komponent zaradi imen polj, imena polj omogočajo enostaven način definicije ujemanja vzorcev, dostop je omogočen na uniformen način preko imen in vrstni red komponent ni več pomemben.

Naslednji primer prikaže razlike dostopa do komponent zapisa v primerjavi z n-tericami.

```
# let a = (1,2,3) ;;
val a : int * int * int = 1, 2, 3
# let first tr = match tr with x,_,_ -> x ;;
val first : 'a * 'b * 'c -> 'a = <fun>
# first a ;;
- : int = 1
# type triplet = {x1:int; x2:int; x3:int} ;;
type triplet = { x1: int; x2: int; x3: int }
# let b = {x1=1; x2=2; x3=3} ;;
val b : triplet = {x1=1; x2=2; x3=3}
# let first tr = tr.x1 ;;
val first : triplet -> int = <fun>
# first b ;;
- : int = 1
```

Poglejmo si še primer dostopa do komponent zapisa z ujemanjem vzorcev. Pri ujemanju vzorcev ni potrebno specificirati vseh polj zapisa.

```
# let first tr = match tr with {x1=x} -> x;;
val first : triplet -> int = <fun>
# first b;;
- : int = 1
```

Spremenljivi zapisi

Komponente zapisov lahko deklariramo kot spremenljive. Možnost spreminjanja polj zapisov dosežemo s ključno besedo `mutable`, ki jo lahko uporabimo pri definiciji polja zapisa.

Sintaksa:

```
type name = { ...; mutable name1: t ; ... }
```

Poglejmo si primer definicije tipa zapisa za predstavitev točk ravnine.

```
# type point = { mutable xc : float; mutable yc : float } ;;
type point = { mutable xc: float; mutable yc: float }
# let p = { xc = 1.0; yc = 0.0 } ;;
val p : point = {xc=1; yc=0}
```

Vrednost polj, ki so definirana s ključno besedo `mutable` lahko spreminjamo z naslednji stavkom.

Sintaksa:

```
expr1.name <- expr2
```

Izraz `expr1` mora biti tipa zapis, ki ima polje `name`. Operator za popravljanje polja vrne tip `unit`.

```
# p.xc <- 3.0 ;;
- : unit = ()
# p ;;
- : point = {xc=3; yc=0}
```

Funkcijo za premikanje točk napišemo s spreminjanjem vrednosti komponent. Za zapis sekvence prirejanj uporabimo lokalno deklaracijo z lovljenjem vzorcev.

```
# let moveto p dx dy =
    let () = p.xc <- p.xc +. dx
    in p.yc <- p.yc +. dy ;;
val moveto : point -> float -> float -> unit = <fun>
# moveto p 1.1 2.2 ;;
- : unit = ()
# p ;;
- : point = {xc=4.1; yc=2.2}
```

Pri definiciji zapisa lahko mešamo spremenljive in nespremenljive komponente. Spreminjamo lahko samo tiste komponente, ki so definirane kot `mutable`.


```

# type t = { c1 : int; mutable c2 : int } ;;
type t = { c1: int; mutable c2: int }
# let r = { c1 = 0; c2 = 0 } ;;
val r : t = {c1=0; c2=0}
# r.c1 <- 1 ;;
Characters 0-9:
The label c1 is not mutable
# r.c2 <- 1 ;;
- : unit = ()
# r ;;
- : t = {c1=0; c2=1}

```

Kasneje bomo predstavili primer uporabe zapisov s spremenljivimi komponentami za implementacijo sklada.

6.3.3 Vsote

Za razliko od n-teric in zapisov, ki ustrezajo kartezijskemu produktu, ustreza deklaracija vsot unijam množic. Različni tipi so zbrani v en sam tip.

Različne člane unije identificiramo s pomočjo *konstruktorjev*, ki podpirajo na eni strani konstrukcijo vrednosti danega tipa in po drugi strani ujemanje vzorcev za dostop do komponent vrednosti.

Aplikacija konstruktorja nekega tipa na argumentu zagotavlja, da je vrnjena vrednost pripada danemu tipu.

Vsota je deklarirana tako, da so podana imena konstruktorjev in imena tipov morebitnih argumentov.

Sintaksa:

```

type    name = ...
        | Namei ...
        | Namej of tj ...
        | Namek of tk * ...* tl ... ;;

```

Ime konstruktorja je izbran identifikator. Imena konstruktorjev se vedno začnejo z veliko črko.

Konstrukcija konstant

Konstruktor, ki ne pričakuje argumenta imenujemo konstruktor konstant. Konstruktor konstant lahko uporabljamo v izrazih kot konstanto.

```
# type coin = Heads | Tails;;
type coin = | Heads | Tails
# Tails;;
- : coin = Tails
```

Na podoben način lahko definiramo tip `bool`.

Konstruktorji z argumenti

Konstruktor ima lahko argumente. Ključna beseda `of` napove tipe konstruktorjevega argumenta. Naštevaje konstruktorjev omogoča združevanje različnih tipov v en sam tip.

Poglejmo si klasičen primer definicije podatkovnega tipa, ki predstavlja karte neke igre, v našem primeru Tarota. Tipi `suit` in `card` so definirani na naslednji način.

```
# type suit = Spades | Hearts | Diamonds | Clubs ;;
# type card =
  King of suit
  | Queen of suit
  | Knight of suit
  | Knave of suit
  | Minor_card of suit * int
  | Trump of int
  | Joker ;;
```

Kreacija vrednosti tipa `card` se izvrši preko aplikacije konstruktorja na vrednosti primernega tipa.

```
# King Spades ;;
- : card = King Spades
# Minor_card(Hearts, 10) ;;
- : card = Minor_card (Hearts, 10)
# Trump 21 ;;
- : card = Trump 21
```

Poglejmo si zdaj funkcijo, ki konstruira seznam vseh kart za dano barvo.

```
# let rec interval a b = if a = b then [b] else a :: (interval (a+1) b) ;;
val interval : int -> int -> int list = <fun>
# let all_cards s =
  let face_cards = [ Knave s; Knight s; Queen s; King s ]
  and other_cards = List.map (function n -> Minor_card(s,n)) (interval 1 10)
  in face_cards @ other_cards ;;
val all_cards : suit -> card list = <fun>
# all_cards Hearts ;;
```

```
- : card list =
[Knave Hearts; Knight Hearts; Queen Hearts; King Hearts;
 Minor_card (Hearts, 1); Minor_card (Hearts, 2); Minor_card (Hearts, 3);
 Minor_card (Hearts, ...); ...]
```

Pravila za Tarot lahko najdemo na naslovu <http://www.pagat.com/tarot/frtarot.html>.

Unije in ujemanje vzorcev

Za delo z vrednostmi vsote uporabljamo ujemanje vzorcev. Naslednji primer konstruira pretvorbo iz vrednosti tipa `suit` in `card` v tekstovne nize.

```
# let string_of_suit = function
  Spades   -> "spades"
  | Diamonds -> "diamonds"
  | Hearts   -> "hearts"
  | Clubs    -> "clubs";;
val string_of_suit : suit -> string = <fun>
# let string_of_card = function
  King c           -> "king of " ^ (string_of_suit c)
  | Queen c        -> "queen of " ^ (string_of_suit c)
  | Knave c         -> "knave of " ^ (string_of_suit c)
  | Knight c        -> "knight of " ^ (string_of_suit c)
  | Minor_card (c, n) -> (string_of_int n) ^ "of " ^ (string_of_suit c)
  | Trump n         -> (string_of_int n) ^ "of trumps"
  | Joker           -> "joker";;
val string_of_card : card -> string = <fun>
```

Postavljanje vzorcev v vrsto omogoča enostavno branje funkcije.

6.4 Relacija podtip

= Zakaj potrebujemo relacijo podtip.

- Urejenost matematičnih objektov.
- Pretvorbe tipov (angl. coercion).

= Definicija relacije podtip na:

- Osnovni tipi.
- N-terice, zapisi.
- Unije.

= Primeri relacije podtip v Ocaml sintaksi.

6.5 Rekurzivni tipi

Rekurziven tip je tip, katerega definicija vsebuje eno ali več referenc na definiran tip.

Rekurzivni tipi so nepogrešljivi v kateremkoli algoritmičnem programskem jeziku za definicijo rekurzivnih podatkovnih struktur: sezname, kopice, drevesa, grafi, itd.

6.5.1 Sezname

Ena izmed osnovnih rekurzivnih podatkovnih struktur uporabljena v večini programskih jezikov je *seznam*. Seznam vsebuje glavo in rep, kjer je rep definiran rekurzivno kot seznam. Seznane lahko implementiramo na več različnih načinov. V nadaljevanju bo predstavljena implementacija z uporabo parov in implementacija z uporabo zapisov.

Poglejmo si najprej implementacija seznamov z uporabo vsot in parov. Seznam plist je definiran kot vsota, ki je bodisi konstanta Nil ali vozlišče s konstruktorjem Node in tipom 'a * plist. Druga komponenta para je torej spet seznam tipa plist.

```
# type 'a plist = Nil | Elm of 'a * 'a plist;;
type 'a plist = Nil | Elm of 'a * 'a plist
# let a = Elm(1,Elm(2,Elm(3,Elm(4,Nil))));;
val a : int plist = Elm (1, Elm (2, Elm (3, Elm (4, Nil)))
```

Poglejmo si zdaj primer implementacije funkcije nad seznamom. Naslednja funkcija sešteje vrednosti seznama tipa int plist, ki je podan kot parameter.

```
# let rec sum l = match l with Nil -> 0 | Elm(v,t) -> v+sum t;;
val sum : int plist -> int = <fun>
# sum a;;
- : int = 10
```

Ko enkrat kreiramo seznam tipa 'a plist ne moremo več spreminjati vsebine seznama. Vrednosti parov po kreaciji namreč ne moremo spreminjati.

Naslednja implementacija seznamov temelji na uporabi zapisov.

```
# type 'a rnode = { mutable cont:'a; mutable next:'a rlist }
and 'a rlist =
  Nil
  | Elm of 'a rnode;;
type 'a rnode = { mutable cont : 'a; mutable next : 'a rlist; }
and 'a rlist = Nil | Elm of 'a rlist
# let l1 = Elm {cont=1;next=Elm {cont=2;next=Nil}};;
val l1 : int rlist =
  Elm {cont = 1; next = Elm {cont = 2; next = Nil}}
```

Definirali bomo vse standardne operacije nad seznamami predstavljenimi z rlist.

Prva funkcija, ki jo bomo implementirali je funkcija `cons` v `l`, ki doda nov element `v` v glavo seznama `l`. Funkcijo definiramo tudi z infiksno operacijo `**`.

```
# let cons v l = Elm {cont=v; next=l};;
val cons : 'a -> 'a rlist -> 'a rlist = <fun>
# let l2 = cons 3 (cons 4 Nil);;
val l2 : int rlist =
  Elm {cont = 3; next = Elm {cont = 4; next = Nil}}
# let ( ** ) v l = cons v l;;
val ( ** ) : 'a -> 'a rlist -> 'a rlist = <fun>
# let l3 = 5**6**Nil;;
val l3 : int rlist =
  Elm {cont = 5; next = Elm {cont = 6; next = Nil}}
```

Najprej bomo definirali funkciji `head` in `tail`, ki vrneti glav oz. rep seznama. Za obravnavanje napak potrebujemo še definicijo izjeme `EmptyList`.

```
# exception EmptyList;;
exception EmptyList
# let head l = match l with Nil -> raise EmptyList | Elm r -> r.cont;;
val head : 'a rlist -> 'a = <fun>
# let tail l = match l with Nil -> raise EmptyList | Elm r -> r.next;;
val tail : 'a rlist -> 'a rlist = <fun>
# head l1;;
- : int = 1
# tail l1;;
- : int rlist = Elm {cont = 2; next = Nil}
```

Naslednji primer prikaže implementacijo funkcije `length`, ki izračuna dolžino seznama. Tudi ta funkcija je ena izmed standardnih funkcij na seznamih.

```
# let rec length l = match l with Nil -> 0 | Elm {next=t} -> 1+length t;;
val length : 'a rlist -> int = <fun>
# length l1;;
- : int = 2
```

Vrednosti elementov seznama, ki so shranjene kot vrednosti komponente `cont` lahko spreminjamo, ker smo obe komponenti definirali kot `mutable`. Poglejmo si dve funkciji `get` in `set`, s katerima lahko preberem vrednost `i`-tega elementa seznama oz. lahko postavimo novo vrednost `i`-tega elementa seznama.

Naslednja funkcija `get` `l` `n` ima dva parametra: seznam `l` in celo število `n`, večje ali enako 0. Funkcija vrne `n`-ti element seznama `l`, če ta obstaja. Prvi element ima indeks 0.

```
# let rec get l n = match l with
  Nil -> raise EmptyList
```

```

    | Elm r when n=0 -> r.cont
    | Elm r -> get r.next (n-1);;
val get : 'a rlist -> int -> 'a = <fun>
# get l1 1;;
- : int = 2

```

Elemente seznama tipa `rlist` lahko spremenimo. Naslednja funkcija `set` spremeni vrednost poljubnega elementa seznama na dano vrednost. Funkcija `set i l v` v spremeni *i*-ti element seznama *l* na vrednost *v*. Prvi element seznama ima indeks 0.

```

# let rec set i l v = match l with
  Nil -> ()
  | Elm r when i=0 -> r.cont <- v
  | Elm r -> set (i-1) r.next v;;
val set : int -> 'a rlist -> 'a -> unit = <fun>
# set 0 l1 0;;
- : unit = ()
# l1;;
- : int rlist = Elm {cont = 0; next = Elm {cont = 2; next = Nil}}

```

V nadaljevanju bomo definirali dve verziji funkcije `append l1 l2`. Prva verzija bo vrnila na nov seznam, ki vsebuje kopije elementov `l1` in `l2`. Druga verzija `append` bo preprosto spela konec seznama `l1` s seznamom `l2` ter vrnila kot rezultat seznam `l1`.

Poglejmo si torej najprej verzijo `append`, ki kreira nov seznam. Funkcija deluje tako, da se rekurzivno sprehodimo najprej po `l1` in potem še po seznamu `l2`. Ob obisku vsakega elementa kreiramo nov element, ki bo del novega seznama `l1 l2`.

```

# let rec append l1 l2 = match l1,l2 with
  Elm r1,_ -> Elm { cont=r1.cont; next=append r1.next l2 }
  | Nil,Elm r2 -> Elm { cont=r2.cont; next=append Nil r2.next }
  | Nil,Nil -> Nil;;
val append : 'a rlist -> 'a rlist -> 'a rlist = <fun>
# append l1 l2;;
- : int rlist =
Elm
  {cont = 1;
  next =
    Elm {cont = 2; next = Elm {cont = 3; next = Elm {cont = 4; next = Nil}}}}

```

Definirajmo zdaj še drugo verzijo, ki spne dva seznama preko spremenljivke `next` zadnjega elementa `l1`. Funkcija deluje tako, da se najprej sprehodimo do konca prvega seznama. Ko pridemo do zadnjega elementa povežem naslednika s prvim elementom drugega seznama. Funkcija vrne referenco na `l1` v primeru, da `l1` ni bil `Nil` oz. `l2` sicer.

```

# let rec append1 l1 l2 = match l1 with
  Nil -> l2

```

```

| Elm r when r.next=Nil -> r.next <- l2; l1
| Elm r -> ignore (append1 r.next l2); l1;;
val append1 : 'a rlist -> 'a rlist -> 'a rlist = <fun>
# append1 l1 l2;;
- : int rlist =
Elm {cont = 1; next = ... Elm {cont = 4; next = Nil}}}}

```

Poučno se je igrati z uporabo funkcije `append1`. Poglejmo si kaj se je zares zgodilo s seznamoma `l1` in `l2`. Seznamu `l1` je pripet na konec seznam `l2`. `l1` je torej sestavljen iz elementov seznamov (pred `append1`) `l1` in `l2`. `l2` še vedn kaže na isti element kot pred uporabo operacije `append1`.

```

# l1;;
- : int rlist =
Elm {cont = 1; next = ... Elm {cont = 4; next = Nil}}}}
# l2;;
- : int rlist = Elm {cont = 3; next = Elm {cont = 4; next = Nil}}

```

Definirajmo zdaj še seznam `l3`, ki ga bomo pripeli na konec novega seznama `l1`. Ker smo sezname samo povezovali so, razen povezav preko komponente `next` na koncu seznamov, elementi ostali takšni kot so bili. Poglejmo si na koncu še stanje vseh treh seznamov.

```

# let l3 = 5**6**Nil;;
val l3 : int rlist = Elm {cont = 5; next = Elm {cont = 6; next = Nil}}
# append1 l1 l3;;
- : int rlist =
Elm {cont = 1; next = ... Elm {cont = 6; next = Nil}}}}}}
# l1;;
- : int rlist =
Elm {cont = 1; next = ... Elm {cont = 6; next = Nil}}}}}}
# l2;;
- : int rlist =
Elm {cont = 3; next = ... Elm {cont = 6; next = Nil}}}}
# l3;;
- : int rlist = Elm {cont = 5; next = Elm {cont = 6; next = Nil}}

```

Poglejmo si zdaj še dve verziji funkcije `reverse l`, ki vrne seznam z obrnjenimi elementi seznama `l`. Prva verzija funkcije `reverse`, podobno kot v primeru funkcije `append`, kreira nov seznam. Druga verzija `reverse` vrne seznam, ki je sestavljen iz elementov seznama `l`, ki imajo popravljene vrednosti komponente `next`.

```

# let rec reverse l = match l with
  Nil -> Nil
| Elm {cont=v; next=t} -> append (reverse t) (Elm {cont=v;next=Nil});;
val reverse : 'a rlist -> 'a rlist = <fun>

```

Definirajmo zdaj nov seznam `l` in ga obrnimo z uporabo `reverse`. Kot vidimo, je seznam `l` ostal takšen kot je bil.

```
# let l = 1**2**3**Nil;;
val l : int rlist =
  Elm {cont = 1; next = Elm {cont = 2; next = Elm {cont = 3; next = Nil}}}
# reverse l;;
- : int rlist =
Elm {cont = 3; next = Elm {cont = 2; next = Elm {cont = 1; next = Nil}}}
# l;;
- : int rlist =
Elm {cont = 1; next = Elm {cont = 2; next = Elm {cont = 3; next = Nil}}}
```

Poglejmo si še definicijo funkcije `reverse1`, ki elemente seznama obrne tako, da ustrezno preveže povezave med zapisi.

```
# let rec reverse1 l = match l with
  Nil -> Nil
  | Elm r -> let t = reverse1 r.next
             in r.next <- Nil;
             append1 t (Elm r);;
val reverse : 'a rlist -> 'a rlist = <fun>
```

Uporabimo spet seznam `l`, ki smo ga definirali prej. Preden obrnemo seznam si zapomnimo referenco na zadnji element seznama `l`. Referenco `l1` dobimo z uporabo funkcije `tail`. Po klicu funkcije `reverse1` predstavlja spremenljivka `l` še vedno isti seznam, vendar je komponenta `next` prvega elementa dobila vrednost `Nil`. Ker je zadnji element `l` zdaj prvi, predstavlja zdaj `l1` obrnjen seznam.

```
# let l1 = tail (tail l);;
val l1 : int rlist = Elm {cont = 3; next = Nil}
# let l2 = reverse1 l;;
val l2 : int rlist =
  Elm {cont = 3; next = Elm {cont = 2; next = Elm {cont = 1; next = Nil}}}
# l;;
- : int rlist = Elm {cont = 1; next = Nil}
# l1;;
- : int rlist =
Elm {cont = 3; next = Elm {cont = 2; next = Elm {cont = 1; next = Nil}}}
```

6.5.2 Binarna drevesa

Poglejmo si najprej definicijo binarnega iskalnega drevesa, katerega vozlišča so označena z vrednostmi.

Slika 6.1: Binarno iskalno drevo.

```
# type 'a bin_tree =
  Empty
  | Node of 'a bin_tree * 'a * 'a bin_tree ;;
```

Podatkovno strukturo bomo uporabili za program, ki sortira zapise. Binarno iskalno drevo ima naslednjo lastnost: v vsakem vozlišču velja, da so v levem poddrevesu vrednosti, ki so manjše in v desnem poddrevesu vrednosti, ki so večja od oznake vozlišča.

Naslednja slika predstavlja primer binarnega iskalnega drevesa celih števil. Prazna drevesa so označena s kvadrati medtem, ko so notranja vozlišča predstavljena s krogi, ki vsebujejo vrednost.

Naslednja funkcija izlušči sortiran seznam iz drevesa s pregledovanjem drevesa po principu “v globino”.

```
# let rec list_of_tree = function
  Empty -> []
  | Node(lb, r, rb) -> (list_of_tree lb) @ (r :: (list_of_tree rb)) ;;
val list_of_tree : 'a bin_tree -> 'a list = <fun>
```

Binarno iskalno drevo lahko konstruiramo iz seznama z uporabo naslednje funkcije.

```
# let rec insert x = function
  Empty -> Node(Empty, x, Empty)
  | Node(lb, r, rb) -> if x < r then Node(insert x lb, r, rb)
                       else Node(lb, r, insert x rb) ;;
val insert : 'a -> 'a bin_tree -> 'a bin_tree = <fun>
```

Funkcijo, ki pretvori seznam v drevo dobimo z iteracijo funkcije insert po elementih seznama.

```
# let rec tree_of_list = function
  [] -> Empty
  | h :: t -> insert h (tree_of_list t) ;;
val tree_of_list : 'a list -> 'a bin_tree = <fun>
```

Sortirna funkcija je potemtakem kompozicija funkcij tree_of_list in list_of_tree.

```
# let sort x = list_of_tree (tree_of_list x) ;;
val sort : 'a list -> 'a list = <fun>
# sort [5; 8; 2; 7; 1; 0; 3; 6; 9; 4] ;;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
```

Poglejmo si zdaj še nekaj klasičnih operacij na urejenih binarnih drevesih.

Najprej bo predstavljena funkcija `height`, ki izračuna višino drevesa. Funkcija `height` je izračunana na naslednji način. V primeru praznega drevesa je višina nič. Če ima drevo vozlišče, potem je višina celega drevesa za eno večja od višine večjega poddrevesa.

```
# let rec height = function
  Empty -> 0
  | Node( l, _, r ) -> 1 + max (height l) (height r);;
val height : 'a bin_tree -> int = <fun>
# height a;;
- : int = 9
```

Funkcija `member` pogleda ali je celo število `x` element binarnega iskalnega drevesa, ki je podano kot drugi parameter. Poglejmo najprej najenostavnejšo metodo, ki ne upošteva urejenosti.

```
# let rec member x = function
  Empty -> false
  | Node(_,e,_) when e=x -> true
  | Node(l,_,r) -> member x l || member x r;;
val member : 'a -> 'a bin_tree -> bool = <fun>
```

Pri iskanju lahko uporabimo dejstvo, da so za dano vozlišče v elementi levega poddrevesa v manjši od `v` in elementi desnega poddrevesa v večji od `v`.

```
# let rec member x = function
  Empty -> false
  | Node(_,e,_) when e=x -> true
  | Node(l,v,r) when x<v -> member x l
  | _ -> member x r;;
val member : 'a -> 'a bin_tree -> bool = <fun>
```

Poglejmo si zdaj prvi način pregleda drevesa: *pregled drevesa v globino*. V vsakem vozlišču drevesa izpišemo najprej levo poddrevo, nato izpišemo ključ vozlišča in nato še desno poddrevo.

```
# let rec print_depth_first = function
  Empty -> print_string "."
  | Node(l,e,r) -> print_depth_first l;
                    print_int e;
                    print_depth_first r;;
val print_depth_first : int bin_tree -> unit = <fun>
```

Prejšnji izpis binarnega drevesa bomo opremili še z izpisi, ki povejo kam se premika izvajanje funkcije. Znak `<` pomeni, da se izvajanje pomika v levo poddrevo, znak `>` pomeni premik v desno poddrevo in znak `^` bo pomenil, da se izvajanje funkcije pomika navzgor.

```
# let rec print_depth_first = function
  Empty -> print_string "."
  | Node(l,e,r) -> print_string "<"; print_depth_first l;
                  print_string "^"; print_int e;
                  print_string ">"; print_depth_first r;;
val print_depth_first : int bin_tree -> unit = <fun>
```

Naslednji izpis predstavlja drugi klasičen primer izpisa binarnega drevesa: *izpis po nivojih*. Najprej bomo definirali funkcijo, ki izpiše en nivo binarnega drevesa. Parameter funkcije `print_level` bo števec nivojev: vsakič, ko se pomaknemo v levo ali v desno poddrevo odštejemo enico. Izpišemo samo tista vozlišča, kjer je nivo enak 0.

```
# let rec print_level i = function
  Empty -> ()
  | Node(_,e,_) when i=0 -> print_int e; print_string " "
  | Node(l,_,r) -> print_level (i-1) l;
                  print_level (i-1) r;;
val print_level : int -> int bin_tree -> unit = <fun>
```

Poglejmo si funkcijo `print_level` bolj natančno. Prvi vzorec pravi, da v primeru praznega poddrevesa ne izpišemo nič. Drugi vzorec ulovi primer, ko je števec nivojev enak nič in je potrebno izpisati vozlišče. Tretji vzorec pokrije ujemanje z vozlišči, kjer je števec $i > 0$, torej je potrebno izpisati nivo $i-1$ za vsako poddrevo danega vozlišča.

Zdaj lahko uporabimo implementacijo funkcije `print_level` za izpis celotnega drevesa po nivojih.

```
# let print_levelwise t =
  for i=0 to (height t)-1 do
    print_level i t;
    print_string "\n"
  done;;
val print_levelwise : int bin_tree -> unit = <fun>
```

6.5.3 Splošna drevesa

Splošno drevo, kjer vozlišča nimajo fiksnega števila naslednikov je lahko predstavljeno z naslednjo strukturo.

```
# type 'a tree = Empty
  | Node of 'a * 'a tree list ;;
```

Prazno drevo je predstavljeno z `Empty`. Listi so vozlišča brez vej oblike `Node(x,[])`, ali v degenerirani obliki `Node(x, [Empty;Empty; ...])`.

Ker so veje drevesa predstavljene s *seznamom poddreves* za vsako vozlišče drevesa lahko pri implementaciji funkcij za delo s splošnimi drevesi uporabljamo poznane višjenivojske funkcije implementirane v modulu List. Na ta način dobimo zelo abstraktno kodo, ki je hkrati tudi razumljiva.

Poglejmo si najprej primer funkcije, ki pogleda ali element pripada drevesu. Za preverjanje članstva elementa v drevesu uporabimo naslednji algoritem: če je drevo prazno potem e ne pripada drevesu, sicer e pripada drevesu, če je bodisi enak oznaki vozlišča ali pripada eni izmed vej.

```
# let rec belongs e = function
  Empty -> false
  | Node(v, bs) -> (e=v) or (List.exists (belongs e) bs) ;;
val belongs : 'a -> 'a tree -> bool = <fun>
```

Za izračun višine drevesa bomo uporabili naslednjo definicijo: prazno drevo ima višino 0, sicer je višina drevesa enaka višini največjega poddrevesa.

```
# let rec height =
  let max_list l = List.fold_left max 0 l in
  function
    Empty -> 0
    | Node(_, bs) -> 1 + (max_list (List.map height bs)) ;;
val height : 'a tree -> int = <fun>
```

Poglejmo spet pregled drevesa v globino. Z uporabo funkcije List.iter apliciramo funkcijo print_depth_first nad vsakim poddrevesom danega vozlišča in nato izpišemo ključ vozlišča.

```
# let rec print_depth_first = function
  Empty -> print_string "."
  | Node(e,l) -> List.iter print_depth_first l; print_int e;;
val print_depth_first : int tree -> unit = <fun>
```

6.5.4 Funkcijski tipi

Tip argumenta konstruktorja je lahko poljuben—lahko je tudi funkcijski tip. Naslednji tip konstruira sezname katerih so vse vrednosti razen zadnje funkcije.

```
# type 'a listf =
  Val of 'a
  | Fun of ('a -> 'a) * 'a listf ;;
type 'a listf = | Val of 'a | Fun of ('a -> 'a) * 'a listf
```

Ker so funkcijske vrednosti tudi vrednosti lahko konstruiramo naslednji tip listf.

```
# let eight_div = (/) 8 ;;
val eight_div : int -> int = <fun>
# let gl = Fun (succ, (Fun (eight_div, Val 4))) ;;
val gl : int listf = Fun (<fun>, Fun (<fun>, Val 4))
```

Še funkcija, ki uporablja ujemanje vrednosti tipa `int listf` za evaluacijo izraza tega tipa.

```
# let rec compute = function
  Val v -> v
  | Fun(f, x) -> f (compute x) ;;
val compute : 'a listf -> 'a = <fun>
# compute gl;;
- : int = 3
```

6.5.5 Rekurzivne vrednosti, ki niso funkcije

Rekurzivna deklaracija ne-funkcijskih vrednosti omogoča konstrukcijo krožnih podatkovnih struktur. Naslednji primer prikaže definicijo krožne podatkovne strukture.

```
# let rec l = 1 :: l ;;
val l : int list =
  [1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; ...]
```

Rekurzivna funkcija se lahko izvaja na takšni strukturi v neskončnost.

```
# size l ;;
Stack overflow during evaluation (looping recursion?).
```

Strukturna enakost za takšne vrednosti ni uporabna. Fizično enakost, ki preveri enakost na osnovi enakosti naslovov vrednosti, lahko uporabimo.

```
# l==l ;;
- : bool = true
```

Če torej definiramo enaka seznama ne smemo uporabljati strukturne enakosti, ker se izvajanje primerjanja nebi ustavilo. Ni priporočljivo pognati naslednji stavek.

```
let rec l2 = 1::l2 in l=l2 ;;
```

...

6.6 Primeri uporabe tipov

V nadaljevanju bodo prikazani primeri implementacije funkcij in podatkovnih struktur z uporabo predstavljenih tipov.

6.6.1 Implementacija sklada

Podatkovno strukturo 'a stack bomo implementirali z uporabo zapisov, ki vsebuje polje elementov in spremenljivko s prvo prosto pozicijo v polju. Poglejmo si najprej pripadajoč tip.

```
# type 'a stack = { mutable ind:int; size:int; mutable elts : 'a array } ;;
```

Polje ind vsebuje kazalec na prvi prost element sklada, polje size vsebuje maksimalno velikost sklada in elts vsebuje elemente sklada.

Operacije na skladu so naslednje.

- init: inicializacija sklada,
- push: postavljanje elementov na sklad, in
- pop: pobiranje elementov iz sklada.

```
# let init_stack n = {ind=0; size=n; elts = [| |]} ;;
val init_stack : int -> 'a stack = <fun>
```

Uporabili bomo dve izjemi za primera ko uporabnik želi pobirati iz praznega sklada in ko želi postaviti element na poln sklad.

```
# exception Stack_empty ;;
# exception Stack_full ;;
# let pop p =
  if p.ind = 0 then raise Stack_empty
  else (p.ind <- p.ind - 1; p.elts.(p.ind)) ;;
val pop : 'a stack -> 'a = <fun>
# let push e p =
  if p.elts = [| |] then
    (p.elts <- Array.create p.size e;
     p.ind <- 1)
  else if p.ind >= p.size then raise Stack_full
  else (p.elts.(p.ind) <- e; p.ind <- p.ind + 1) ;;
val push : 'a -> 'a stack -> unit = <fun>
```

Poglejmo si manjši primer uporabe sklada.

```

# let p = init_stack 4 ;;
val p : '_a stack = {ind=0; size=4; elts=[|]}
# push 1 p ;;
- : unit = ()
# for i = 2 to 5 do push i p done ;;
Uncaught exception: Stack_full
# p ;;
- : int stack = {ind=4; size=4; elts=[|1; 2; 3; 4|]}
# pop p ;;
- : int = 4
# pop p ;;
- : int = 3

```

Če hočemo preprečiti proženje izjeme `Stack_full` pri dodajanju elementa na sklad lahko povečamo polje. Da bi to lahko naredili mora biti polje `size` spremljivo.

```

# type 'a stack =
  {mutable ind:int ; mutable size:int ; mutable elts : 'a array} ;;
# let init_stack n = {ind=0; size=max n 1; elts = [|]} ;;
# let push e p =
  if p.elts = [|]
  then
    begin
      p.elts <- Array.create p.size e;
      p.ind <- 1
    end
  else if p.ind >= p.size then
    begin
      let nt = 2 * p.size in
      let nv = Array.create nt e in
      for j=0 to p.size-1 do nv.(j) <- p.elts.(j) done ;
      p.elts <- nv;
      p.size <- nt;
      p.ind <- p.ind + 1
    end
  else
    begin
      p.elts.(p.ind) <- e ;
      p.ind <- p.ind + 1
    end ;;
val push : 'a -> 'a stack -> unit = <fun>

```

Potrebno je biti previden s podatkovnimi strukturami, ki se lahko širijo brez meja. Poglejmo si še en primer uporabe novega sklada.

```

# let p = init_stack 4 ;;
val p : '_a stack = {ind=0; size=4; elts=[|]}
# for i = 1 to 5 do push i p done ;;
- : unit = ()

```

```
# p ; ;  
- : int stack = {ind=5; size=8; elts=[1; 2; 3; 4; 5; 5; 5; 5]}
```

Zelo koristno je, če napišemo program tako, da sklad sprosti spomin, ko ni več potreben.

6.6.2 Implementacija vrste

Vrsta s poljem:

Uporaba dveh kazalcev: začetek in konec vrste

Operacije: dodaj na začetek in konec, odvzami iz začetka in konca

Poglavje 7

OBJEKTI IN RAZREDI

Abstrakcije za modeliranje *postopkov*, ki smo jih predstavili v prejšnjih poglavjih so sekvence, vejitveni stavki, iteracijski stavki in vzorci. S temi abstrakcijami lahko učinkovito predstavimo aritmetične izračune, algoritme, ki temeljijo na iteraciji, in druge postopke, ki uporabljajo enostavne podatkovne strukture kot so na primer, polja, matrike, sezname in podobno. Abstrakcije za modeliranje postopkov so običajno dostopne že v zbirniku.

Funkcije in funkcijska dekompozicija problema v podprobleme omogočata enostavnejšo modeliranje bolj kompleksnih postopkov—kompleksne funkcije lahko razdelimo na več pod-funkcij, ki jih lahko enostavno povezujemo z abstrakcijami za modeliranje postopkov. Z uporabo funkcij in postopkovnih gradnikov programskih jezikov lahko učinkovito predstavimo kompleksne algoritme. Do sedaj našete abstrakcije so značilne za Algolsko vejo programskih jezikov, kot so na primer, Pascal ali C.

Tipi dodajo programskemu jeziku zmožnost strukturiranja podatkov z uporabo konstruktorjev tipov kot so na primer produkt, zapis in unija. Postopkovne abstrakcije lahko skupaj s tipi uporabimo za definicijo postopkov nad kompleksnimi podatkovnimi strukturami. Funkcijske abstrakcije uporabljamo za definicijo obnašanja določenih podatkovnih struktur, kot tudi za povezovanje podatkovnih in funkcijskih objektov v višje-nivojske funkcijske strukture.

Objektni model programskih jezikov izhaja iz jezika Simula, ki je bil namenjen za modeliranje in simulacijo sistemov, ki so opisani z diskretnimi dogodki. V okviru simulacijskega jezika Simule so bili prvič definirani koncepti objektno usmerjenih programskih jezikov kot npr. razred in objekt. Jezik Simula je imel izjemen vpliv na razvoj jezika Smalltalk [?], ki se smetra za enega od prvih objektno usmerjenih splošnih programskih jezikov. Stroustrup, avtor C++, prav tako piše, da je Simula imela zelo velik vpliv na zasnovu jezika C++.

Objektni model definira popolnoma drugačne abstrakcije kot so koncepti za definicijo sekvenčne kontrole, funkcije in tudi tipi. Razredi modelirajo koncepte za katere lahko

predstavimo strukturo, klasifikacijo ter druga razmerja med razredi-koncepti. Razred definira tudi obnašanje primerkov razreda; definira sporočila in kodo, ki se bo izvajala ob določenem sporočilu.

Objektni model je blizu simulacijskem pogledu na reševanje problemov: izdelava modela sistema. Uporabljamo abstrakcije veliko bližje človekovemu načinu predstavitve problema. Razred si lahko predstavljamo kot *koncept* za katerega definiramo predstavitev statične strukture in obnašanja. Analogija z modelom dejanskih objektov in idej gre še naprej z modeliranjem dejanskega obnašanja simuliranega modela preko interne logike objektov ter sporočil, ki si jih objekti pošiljajo za reševanje določenega problema.

7.1 Objektni model programskega jezika

Za razliko od imperativnih programov, kjer vrstimo operacije v sekvence, ali funkcijskega programiranja, kjer je izvajanje vodeno z dekompozicijo funkcij, je objektno-usmerjeno programiranje vodeno podatkovno!

Objekti in razredi nudijo drugačno organizacijo programov v razrede in pripadajoče objekte. Razred predstavlja združitev podatkov–objektov istega tipa–in operacij oz. metod, ki jih lahko izvajamo nad danimi objekti. Metode definirajo obnašanje objektov.

Metodo aktiviramo s pošiljanjem sporočila objektu. Ko objekt sprejme sporočilo izvrši akcijo, ki ustreza metodi definirani s sporočilom. Opisana akcija je drugačna od aplikacije funkcije na argumentu, ker se objektu pošlje sporočilo.

Od objekta je odvisno katera metoda se bo dejansko izvedla. Zakasnjeno povezovanje med imenom metode in kodo naredi obnašanje sistemov bolj fleksibilno: isto sporočilo lahko pošljem zelo različnim objektom. Koda, ki uporablja sporočila je iz tega razloga tudi bolj prilagodljiva spremembam in omogoča enostavnejšo možnost ponovne uporabe.

V objektno-usmerjenem jeziku definiramo razmerja med razredi in objekti. Razredi definirajo kako objekti komunicirajo preko sporočil. Najbolj pomembni razmerji med razredi sta specializacija/generalizacija ter kompozicija/dekompozicija.

V primeru razmerja specializacija/generalizacija tipično uporabljamo dedovanje lastnosti bolj splošnih razredov na bolj specifične razrede. Lastnosti lahko redefiniramo v okviru bolj specifičnega razreda ter njegovih primerkov.

Kompozicija oz. dekompozicija se lahko izraža v objektno-usmerjenih jezikih preko referenc med razredi.

Pri definiciji hierarhije razredov definirane na osnovi specializacije/generalizacije razredov se pojavita dve vrsti polimorfizma.

Parametrični polimorfizem smo si že ogledali. Parametrizirani tipi omogočajo definicijo zelo splošnih funkcij, ki jih lahko instanciramo za konkretno uporabo na konkretnih tipih. Parametrični polimorfizem ima več oblik.

Polimorfizem, ki se pojavi zaradi podtipov imenujemo polimorfizem vsebovanosti. Zaradi hierarhije razredov ne moremo vedno vedeti katera metoda se bo izvršila za dano sporočilo. Podobno kot parametrični polimorfizem ima tudi polimorfizem vsebovanosti več različnih oblik, ki si jih bomo ogledali v posebnem poglavju.

Poudariti je potrebno, da je Ocaml eden od redkih jezikov, ki vsebujejo parametrični polimorfizem kot tudi polimorfizem vsebovanosti. Navkljub kompleksnemu sistemu tipov imamo v Ocaml še vedno učinkovito statično sklepanje s tipi.

7.2 Razredi

Objektna razširitev omogoča definicijo razredov, instanc, hierarhijo dedovanja, parametrizirane razrede in abstraktne razrede. Vmesniki razredov so definirani preko definicije razredov, lahko pa jih definiramo bolj natančno z definicijo signature, podobno kot v primeru modulov.

7.2.1 Definicija razreda

Najenostavnejša sintaksa za definicijo razredov je naslednja.

Sintaksa:

```
class name p1 ... pn =
  object
  .
  .
  .
  instance variables
  .
  .
  .
  methods
  .
  .
  .
end
```

p_1, \dots, p_n so parametri konstruktorja razreda; izpustimo jih v primeru, da razred nima parametrov.

Spremenljivke primerka so naslednje:

Sintaksa:

```
val name = expr
```

ali

```
val mutable name = expr
```

Če je podatkovno polje definirano kot spremenljivo (mutable), lahko spreminjamo njegovo vrednost. Sicer je vrednost vedno tista, ki jo izračunamo med kreiranjem objekta.

Metode deklariramo na sledeč način.

Sintaksa:

```
method name p1 . . . pn = expr
```

Razred lahko vsebuje tudi druge stavke, ki opisujejo omejitve lastnosti razreda. Te stavke bomo spoznavali skupaj s temami objektno-usmerjenega programiranja.

Gradnike za definicijo razredov si bomo ogledali na primeru definicije razreda point.

Podatkovna polja x in y vsebujeta koordinate točke. Dve metodi omogočata dostop do teh podatkovnih polj (get_x in get_y), dve metodi za premikanje točk: absoluten premik z moveto in relativen premik z rmoveto, ena metoda pretvori podatke točke v niz (to_string) in ena metoda izračuna razdaljo točke do izhodišča.

Poglejmo si najprej zelo enostavno varianto razreda point, ki vsebuje samo del opisanih lastnosti.

```
#class point =
  object
    val mutable x = 0
    method get_x = x
    method move d = x <- x + d
  end;;
class point :
  object val mutable x : int method get_x : int method move : int -> unit end
```

Zdaj lahko kreiramo novo točko oz. primerek razreda point.

```
#let p = new point;;
val p : point = <obj>
```

Spremenljivka `p` je tipa `point`. Kot bomo kasneje videli je pomen `point` tip `<get_x : int; move : int -> unit>`.

Zdaj lahko pokličemo nekaj metod na objektu `p`.

```
#p#get_x;;
- : int = 0
#p#move 3;;
- : unit = ()
#p#get_x;;
- : int = 3
```

Razred `point` ima lahko parametre, ki predstavljajo začetne vrednosti `x` koordinate.

```
# class point = fun x_init ->
  object
    val mutable x = x_init
    method get_x = x
    method move d = x <- x + d
  end;;
class point :
  int ->
  object val mutable x : int method get_x : int method move : int -> unit end
```

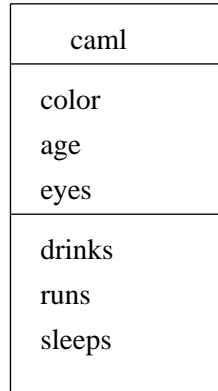
Kot v primeru definicije funkcij lahko zgornji zapis okrajšamo.

```
#class point x_init =
  object
    val mutable x = x_init
    method get_x = x
    method move d = x <- x + d
  end;;
class point :
  int ->
  object val mutable x : int method get_x : int method move : int -> unit end
```

Poglejmo si zdaj primer kreacije točke.

```
#new point;;
- : int -> point = <fun>
#let p = new point 7;;
val p : point = <obj>
```

Parameter `x_init` je seveda viden znotraj celotnega razreda. Tako lahko dodamo metodo `get_offset`.



Slika 7.1: Grafična predstavitev razreda

```
#class point x_init =
  object
    val mutable x = x_init
    method get_x = x
    method get_offset = x - x_init
    method move d = x <- x + d
  end;;
class point :
  int ->
  object
    val mutable x : int
    method get_offset : int
    method get_x : int
    method move : int -> unit
  end
```

7.2.2 Grafična notacija razredov

Uporabljali bomo UML notacijo za opis tipov Ocaml. Razredi so opisani s pravokotniki, ki imajo tri dele. Področje na vrhu vsebuje ime razreda, srednji del vsebuje seznam spremenljivk primerka (podatkovna polja), in spodnji del vsebuje metode razreda.

Slika ?? podaja primer grafične predstavitev razreda caml, ki ima spremenljivke color, age in eyes, ter metode drinks, runs in sleeps. Običajno dodamo še podatke o tipih spremenljivk in metod razreda npr. color:string, age:int in eyes:string, ter drinks : unit->unt, runs : unit->unit in sleeps : unit->unit.

7.2.3 Kreiranje in inicializacija objekta

Objekt je primerek razreda in predstavlja vrednost. Primerke generiramo z uporabo generičnega konstrukcijskega primitiva `new`, ki vzame razred in inicializacijske vrednosti kot argumente.

Sintaksa:

```
new name expr1 . . . exprn
```

Naslednji primer kreira primerke razreda `point` iz različnih inicializacijskih vrednosti.

```
# let p1 = new point 0;;
val p1 : point = <obj>
# let p2 = new point 3;;
val p2 : point = <obj>
# let xcoord = 8;;
val xcoord : int = 8
# let p3 = new point xcoord;;
val p3 : point = <obj>
```

Definicija razreda lahko vsebuje izraze, ki se ovrednotijo pred kreacijo objekta. To je koristno za izpolnjevanje invariant. Na primer, točko lahko prilagodimo na mrežo pred kreacijo.

```
# class adjusted_point x_init =
  let origin = (x_init / 10) * 10 in
  object
    val mutable x = origin
    method get_x = x
    method get_offset = x - origin
    method move d = x <- x + d
  end;;
class adjusted_point :
  int ->
  object
    val mutable x : int
    method get_offset : int
    method get_x : int
    method move : int -> unit
  end
```

Metoda lahko pošlje sporočilo samemu sebi. Za ta namen mora biti objekt znan znotraj samega sebe. V ta namen uporabljamo posebno spremenljivko, ki referencira sam objekt. Običajno poimenujemo takšno spremenljivko `self`, čeprav to ni nujno v Ocaml.

```
# class printable_point x_init =
  object (s)
    val mutable x = x_init
    method get_x = x
    method move d = x <- x + d
    method print = print_int s#get_x
  end;;
class printable_point :
  int ->
  object
    val mutable x : int
    method get_x : int
    method move : int -> unit
    method print : unit
  end
# let p = new printable_point 7;;
val p : printable_point = <obj>
# p#print;;
7- : unit = ()
```

Spremenljivka `s` je dinamično povezana ob klicu metode. Več o spremenljivki `self` pri uporabi skupaj z dedovanjem bomo videli kasneje.

Let stavki se lahko torej izvršijo pred kreacijo objekta. Mogoče je izvajati kodo tudi takoj po kreaciji objekta. Začetno kodo lahko napovemo s ključno besed `initializer`.

```
# class printable_point x_init =
  let origin = (x_init / 10) * 10 in
  object (self)
    val mutable x = origin
    method get_x = x
    method move d = x <- x + d
    method print = print_int self#get_x
    initializer print_string "new point at "; self#print; print_newline()
  end;;
class printable_point :
  int ->
  object
    val mutable x : int
    method get_x : int
    method move : int -> unit
    method print : unit
  end

# let p = new printable_point 17;;
new point at 10
val p : printable_point = <obj>
```

Z uporabo tega mehanizma lahko realiziramo enostaven način inicializacije objekta iz

aspekta danega razreda. Kasneje bomo videli, da lahko isti objekt inicializira več razredov iz hierarhije dedovanja. Inicializatorje ne moremo prekriti. Nasprotno, izvajajo se eden za drugim sekvenčno.

7.2.4 Pošiljanje sporočila

Poglejmo si zdaj kompleten primer definicije razreda point in še nekatere podrobnosti.

```
# class point (x_init,y_init) =
  object
    val mutable x = x_init
    val mutable y = y_init
    method get_x = x
    method get_y = y
    method moveto (a,b) = x <- a ; y <- b
    method rmoveto (dx,dy) = x <- x + dx ; y <- y + dy
    method to_string () =
      "(" ^ (string_of_int x) ^ ", " ^ (string_of_int y) ^ ")"
    method distance () = sqrt (float(x*x + y*y))
  end ;;
```

V večini objektnih jezikov je za dostop do metod objekta uporabljena notacija s piko. Ker je v Ocaml ta rezervirana za dostop do komponent zapisov in modulov je uporabljen simbol #.

Sintaksa:

```
obj#name p1 ... pn
```

Sporočilo z imenom name se pošlje objektu obj. Argumenti p1,...,pn se pričakujejo skupaj z imenom metode. Metoda mora biti definirana v razredu objekta in mora biti vidna v tipu. Tipi argumentov morajo ustrezati tipom formalnih parametrov. Naslednji primeri prikažejo več klicev metod.

```
# p1#get_x;;
- : int = 0
# p2#get_y;;
- : int = 4
# p1#to_string () ;;
- : string = "( 0, 0)"
# p2#to_string () ;;
- : string = "( 3, 4)"
# if (p1#distance () ) = (p2#distance () )
  then print_string ("That's just chance\n")
  else print_string ("We could bet on it\n");;
We could bet on it
- : unit = ()
```

Objekti tipa `point` se lahko uporabljajo v polimorfičnih funkcijah enako kot vse ostale vrednosti uporabljene v Ocaml.

```
# p1 = p1 ;;
- : bool = true
# p1 = p2 ;;
- : bool = false
# let l = p1::[];;
val l : point list = [<obj>]
# List.hd l;;
- : point = <obj>
Warning Object equality is defined as physical equality.
```

Problem bomo razčistili pri opisu relacije podtip.

Nekatere metode ne potrebujejo parametrov: taki metodi sta `get_x` in `get_y`. Podatkovna polja običajno beremo z metodami, ki nimajo parametrov—te metode imenujemo tudi *bralci* oz. *pisalci*.

Po deklaraciji točke sistem izriše tip razreda.

```
class point :
  int * int ->
  object
    val mutable x : int
    val mutable y : int
    method distance : unit -> float
    method get_x : int
    method get_y : int
    method moveto : int * int -> unit
    method rmoveto : int * int -> unit
    method to_string : unit -> string
  end
```

Zapis vsebuje tip objektov definirane razreda; tip bomo okrajšano označili s `point`. *Tip objekta* je seznam imen in tipov metod. V našem primeru je tip `point` predstavljen z naslednjim zapisom.

```
< distance : unit -> unit; get_x : int; get_y : int;
  moveto : int * int -> unit; rmoveto : int * int -> unit;
  to_string : unit -> unit >
```

Konstruktor primerkov tipa `point` ima tip `int*int -> point`. Konstruktor omogoča konstrukcijo točke na osnovi parametrov, ki podajo začetne vrednosti točke. Točko konstruiramo iz para celih števil, ki predstavljajo koordinate. Konstruktor `point` se uporablja s ključno besedo `new`.

Definiramo lahko tip razreda:

```
# type simple_point = <get_x : int; get_y : int; to_string : unit -> unit> ;;
type simple_point = <get_x : int; get_y : int; to_string : unit -> unit>
```

Tip `point` ne vsebuje vse podatke: spremenljivke objekta niso del tipa. Tip vsebuje samo metode vključno z bralci in pisalci.

Deklaracija razreda je torej deklaracija tipa. Kot posledica: tip ne sme vsebovati prostih spremenljivk tipa. Kasneje bomo pogledali to temo bolj podrobno preko parametriziranih razredov.

7.2.5 Privatne metode

Metodo lahko definiramo kot privatno s ključno besedo `private`. Metoda se bo pojavila v vmesniku razreda vendar ne tudi v instanci razreda. Privatno metodo lahko pokličemo le iz ostalih metod danega razreda; ne moremo pa je sprožiti izven definicije razreda.

Sintaksa:

```
method private name = expr
```

Za prikaz uporabe privatnih metod razširimo spet razred `point` tako, da si zapomni koordinate točke pred izvršenim premikom točke z metodama `moveto` in `rmoveto`. Dodamo še metodo `undo`, ki izniči zadnji premik.

```
# class point (x0,y0) =
  object(self)
    val mutable x = x0
    val mutable y = y0
    val mutable old_x = x0
    val mutable old_y = y0
    method get_x = x
    method get_y = y
    method private mem_pos () = old_x <- x ; old_y <- y
    method undo () = x <- old_x; y <- old_y
    method moveto (x1, y1) = self#mem_pos (); x <- x1; y <- y1
    method rmoveto (dx, dy) = self#mem_pos (); x <- x+dx; y <- y+dy
    method to_string () =
      "(" ^ (string_of_int x) ^ ", " ^ (string_of_int y) ^ ")"
    method distance () = sqrt (float(x*x + y*y))
  end ;;
class point :
  int * int ->
  object
    val mutable old_x : int
    val mutable old_y : int
    val mutable x : int
```

```

val mutable y : int
method distance : unit -> float
method get_x : int
method get_y : int
method private mem_pos : unit -> unit
method moveto : int * int -> unit
method rmoveto : int * int -> unit
method to_string : unit -> string
method undo : unit -> unit
end
# let p = new point (0, 0);;
val p : point = <obj>

```

Razred `point` razširimo z novima komponentama `old_x` in `old_y`, ki hranita stare vrednosti koordinat `x` in `y`. Dodamo metodo `mem_pos`, ki shrani trenutne vrednosti `x` in `y`. Ker nebi želeli, da ima uporabnik dostop do te metode jo definiramo kot *privatno*.

Redefinirati je potrebno še metodi `moveto` in `rmoveto` tako, da si zapomnita trenutno pozicijo pred klicem metod za premikanje točke z uporabo `mem_pos`.

7.3 Agregacija

Edini razmerji med razredi, ki sta zadosti dobro definirani v primerjavi z splošnimi asociacijami med razredi, sta specializacija oz. generalizacija ter kompozicija (agregacija) oz. dekompozicija.

Agregacijska relacija, ki jo imenujemo tudi Has-a: razred `C2` je v relaciji Has-a z razredom `C1` če ima `C2` podatkovno polje katerega tip je `C1`. Relacijo lahko posplošimo: `C2` ima vsaj eno polje katerega razred je `C1`.

Relacija dedovanja, ki jo imenujemo Is-a: razred `C2` je podrazred razreda `C1` če `C2` razširi obnašanje `C1`. Ena izmed prednosti objektnega programiranja je možnost razširitve obnašanja obstoječega razreda ter hkrati ponovno uporabo obstoječe kode. Nov razred podeduje vse lastnosti nadrejenega razreda—podatkovna polja in metode.

7.3.1 Primer agregacije

Definirali bomo sliko, ki je sestavljena iz točk. Razred `picture` vsebuje podatkovno polje, ki je vektor objektov razreda `point`. Razred `picture` torej povezuje (relacija agregacije) razred `point` z uporabo relacije Has-a.

```

# class picture n =
  object
    val mutable ind = 0

```

```

    val tab = Array.create n (new point(0,0))
    method add p = tab.(ind)<-p ; ind <- ind + 1
    method remove () = if (ind > 0) then ind <-ind-1
    method to_string () =
      let s = ref "["
      in for i=0 to ind-1 do s:= !s ^ " " ^ tab.(i)#to_string () done ;
      (!s) ^ "]"
  end ;;
class picture :
  int ->
  object
    val mutable ind : int
    val tab : point array
    method add : point -> unit
    method remove : unit -> unit
    method to_string : unit -> string
  end
end

```

Za izgradnjo slike najprej kreiramo objekt razreda picture in vstavimo točke v vektor.

```

# let pic = new picture 8;;
val pic : picture = <obj>
# pic#add p1; pic#add p2; pic#add p3;;
- : unit = ()
# pic#to_string () ;;
- : string = "[ ( 0, 0) ( 3, 4) ( 3, 0) ]"

```

7.3.2 Grafična notacija za agregacijo

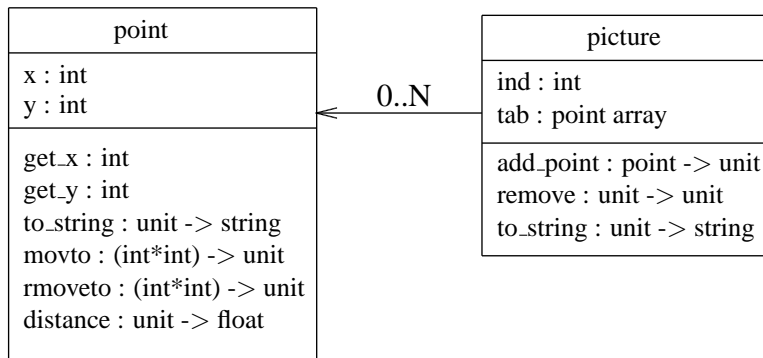
Relacijo med razredom, ki predstavlja nadrejen razred (celoto), in razredom, ki predstavlja komponente (dele) predstavimo s pomočjo puščice, ki je usmerjena proti komponenti. Nad puščico se napiše števnost relacije min..max.

Števnost predstavimo z minimalno števnostjo min, ki je bodisi 0 ali 1, ter z maksimalno števnostjo max, ki je lahko konkretna vrednost n ($n > 1$) ali N, ki predstavlja poljubno število večje od 1.

Relacija med razredom picture in razredom point predstavimo grafično s Sliko ??
Puščica označuje agregacijo: slika ima 0 ali več točk.

7.4 Specializacija

Specializacija je ena izmed najbolj pomembnih abstrakcij objektnega programiranja. Če je razred c2 specializacija razreda c1 potem c2 *podeduje* vse lastnosti c1. V okviru novega razreda c2 lahko tudi ponovno definiramo metode in jih tako specializiramo.



Slika 7.2: Agregacijska relacija

Sintaksa dedovanja je sledeča.

Sintaksa:

```
inherit name1 p1 . . . pn [ as name2 ]
```

Parametri p_1, \dots, p_n so pričakovani parametri razreda $name_1$. Opcijsko ime razreda se uporablja za dostop do metod razreda. Ta možnost se uporablja v primeru da otrok redefinira metodo starša.

7.4.1 Primer enostavnega dedovanja

Razred `point` bomo razširili z `barvo`, ki bo definirana kot podatkovno polje tipa `string`. Dodamo tudi metodo `get_color`, ki vrne vrednost podatkovnega polja `c`. Na koncu ostane še metoda `to_string`, ki jo je potrebno popraviti.

Spremenljivki `x` in `y` v funkciji `to_string` sta polja razreda in ne inicializacijska parametra.

```

# class colored_point (x,y) c =
  object
    inherit point (x,y)
    val mutable c = c
    method get_color = c
    method set_color nc = c <- nc
    method to_string () = "(" ^ (string_of_int x) ^
      "," ^ (string_of_int y) ^ ")" ^
      "[" ^ c ^ "]"
  end ;;
class colored_point :

```

```

int * int ->
string ->
object
  val mutable c : string
  val mutable x : int
  val mutable y : int
  method distance : unit -> float
  method get_color : string
  method get_x : int
  method get_y : int
  method moveto : int * int -> unit
  method rmoveto : int * int -> unit
  method set_color : string -> unit
  method to_string : unit -> string
end

```

Argumenti konstruktorja za obarvano točko so par koordinat točke ter barva. Obnašanje primerka razreda določajo podedovane metode, novo definirane metode ter redefinirane metode.

```

# let pc = new colored_point (2,3) "white";;
val pc : colored_point = <obj>
# pc#get_color;;
- : string = "white"
# pc#get_x;;
- : int = 2
# pc#to_string () ;;
- : string = "(2,3)[white] "
# pc#distance;;
- : unit -> float = <fun>

```

Pravimo, da je razred `point` starš razreda `colored_point` in obratno `colored_point` je otrok razreda `point`.

7.4.2 Reference: `self in super`

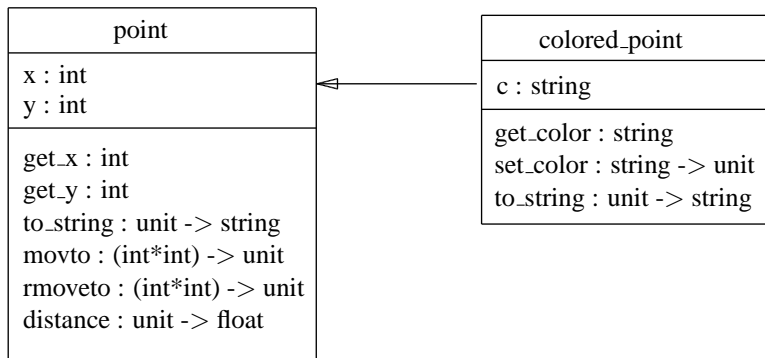
Ko definiramo metodo razreda je koristno imeti dostop do metod nadrejenega razreda. Zaradi tega razloga ima Ocaml možnost imenovanja danega razreda in nadrejene razrede. V zadnjem primeru damo ime z uporabo opsijskega parametra gradnika `inherit`. V prvem primeru lahko imenujemo objekt takoj za ključno besedo `object`.

Pri definiciji metode `to_string` za obarvane točke je enostavneje uporabiti obstoječo metodo iz razreda `point` in jo prilagoditi, kot napisati popolnoma novo metodo. Na ta način uporabimo že obstoječo kodo programa.

```

# class colored_point (x,y) c =

```



Slika 7.3: Relacija dedovanja

```

object (self)
  inherit point (x,y) as super
  val mutable c = c
  method get_color = c
  method set_color nc = c <- nc
  method to_string () = super#to_string () ^
    "[" ^ c ^ "]"
end ;;
  
```

Za imenovanje starša in otroka lahko uporabimo poljubno ime, čeprav je običajno, da uporabljamo imeni `self` in `super`. Izbira drugih imen je nujna pri večkratnem dedovanju za razlikovanje staršev.

Pozor. Ni mogoče doseči spremenljivke nadrejenega razreda, če je prekrita z istoimensko spremenljivko v otroku.

7.4.3 Grafična notacija dedovanja

Relacija dedovanja je označena s puščico, ki je usmerjena proč od otroka. V grafični notaciji pokažemo samo metode, ki so nove ali redefinirane v kontekstu otroka. Slika ?? prikaže razmerje med razredoma `colored_point` in `point`.

7.4.4 Inicializacija objektov

Ključna beseda `initializer` se uporablja za specifikacijo kode, ki se izvaja med konstrukcijo objekta. Inicializator lahko vsebuje poljubno kodo kot ostale metode.

Sintaksa:


```
initializer expr
```

Spet bomo razširili razred `point` tako, da bomo naredili izpis med kreacijo objekta.

```
# class point (x_init,y_init) =
  object (self)
    val mutable x = x_init
    val mutable y = y_init
    method get_x = x
    method get_y = y
    method moveto (a,b) = x <- a ; y <- b
    method rmoveto (dx,dy) = x <- x + dx ; y <- y + dy
    method to_string () =
      "(" ^ (string_of_int x) ^ "," ^ (string_of_int y) ^ ")"
    method distance () = sqrt (float(x*x + y*y))
    initializer
      Printf.printf ">> Creation of point: %s\n" (self#to_string ());
  end ;;
```

Dodajmo zdaj še en razred z imenom `verbose_point`, ki podrobno opiše točko ob kreaciji. Razred `verbose_point` je specializacija `point`.

```
# class verbose_point p =
  object (self)
    inherit point p as super
    method to_string () = "point=" ^ (super#to_string ()) ^
      ",distance=" ^ string_of_float (self#distance ())
    initializer
      Printf.printf ">> Creation of verbose point: %s\n"
        (self#to_string ())
  end ;;
```

```
# new verbose_point (1,1);;
>> Creation of point: (1,1)
>> Creation of verbose point: point=(1,1), distance=1.41421356237
- : verbose_point = <obj>
```

Inicializatorji se prožijo torej od najvišjega razreda proti bolj specifičnim razredom. Najprej se je kreirala točka kot primerek `point` in šele potem se je kreirala točka kot primerek `verbose_point`.

7.4.5 Večkratno dedovanje

- = Primer večkratnega dedovanja.
- = Problemi večkratnega dedovanja.
- Napačna uporaba večkratnega dedovanja.
- Nixon diamant.

- = Verzije večkratnega dedovanja.
- C++
- Ocaml
- ... = Implementacija v sekciji o impl.

7.4.6 Pretvorba tipov

Večina objektnih programskih jezikov omogoča obravnavanje objekta razreda R_1 kot primerka nadrazreda R_2 . Običajno imamo lahko torej spremenljivko tipa R_2 , ki ji lahko priredimo objekt tipa R_1 ali npr. kolekcijo tipa R_2 , ki vsebuje tudi primerke tipa R_1 .

Pri obravnavi primerka R_2 kot primerka R_1 imenujemo pretvorbo objekta tipa R_2 v objekt tipa R_1 *pretvorbo tipov* (angl. coercion). Primerek R_2 pri pretvorbi ohrani vse metode, ki so definirane v razredu R_2 vendar jih po pretvorbi tipa ni več mogoče sprožiti. Pravzaprav s pretvorbo tipov povemo prevajalniku ali tolmaču naj od zdaj naprej obravnava dan primerek tipa R_2 kot primerek tipa R_1 in to je vse¹.

Operacijo pretvorbe tipov označimo z “:”>”. Pretvorbo tipov objektov v Ocaml lahko naredimo na dva načina, ki sta predstavljena z naslednjo sintakso.

Sintaksa:

```
(name : sub_type :> super_type)
(name :> super_type)
```

Poglejmo si zdaj primer pretvorbe tipa primerka razreda `colored_point` v primerek razreda `point`. Najprej kreiramo primerek razreda `colored_point` in ga shranimo v spremenljivki `cp`. Z operacije pretvorbe tipov spremenimo tip `cp` v tip `point` in objekt shranimo v spremenljivko `p`.

```
# let cp = new colored_point (1,1) "red";;
val cp : colored_point = <obj>
# let p = (cp :> point);;
val p : point = <obj>
# p#get_color ();;
Error: This expression has type point
      It has no method get_color
# p#to_string ();;
- : string = "(1,1)[red]"
```

Objekt določen s spremenljivko `p` nima več dostopne metode `get_color`, očitno pa ta metoda je dostopna iz funkcije `to_string`.

¹Večji poudarek na relaciji podtip! (*)

7.4.7 Dinamično povezovanje

Dinamično povezovanje je povezovanju kode metode z imenom metode v času izvajanja. *Statično povezovanje* je povezovanje kode metode z imenom v času prevajanja. Ocaml uporablja dinamično povezovanje–koda ki se izvaja ob proženju metode se določi v času izvajanja.

Klasičen primer potrebe po dinamičnem povezovanju je primer, ko imamo kolekcijo objektov in želimo vsem objektom v kolekciji poslati isto sporočilo. Ker so objekti v kolekciji lahko primerki različnih razredov (iz ene hierarhije razredov), se lahko za specifičen objekt izvrši specifična metoda.

Poglejmo si primer. Slika je bila definirana z razredom `picture`, ki vsebuje seznam točk. V prejšnjih primerih smo definirali tri vrste točk: `point`, `colored_point` in `verbose_point`.

```
# class picture n =
  object
    val mutable ind = 0
    val tab = Array.create n (new point(0,0))
    method add p = tab.(ind)<-p ; ind <- ind + 1
    method remove () = if (ind > 0) then ind <-ind-1
    method to_string () =
      let s = ref "["
        in for i=0 to ind-1 do s:= !s ^ " " ^ tab.(i)#to_string () done ;
          (!s) ^ "]"
    end ;;
```

Slika bom zdaj sestavili iz več različnih točk, ki so primerki različnih razredov. Naslednja koda definira sliko, ki vsebuje tri točke: primerek razreda `point`, primerek razreda `colored_point` in primerek razreda `verbose_point`.

```
# let pic = new picture 3;
>> Creation of point: (0,0)
val pic : picture = <obj>
# pic#add (new point (1,1));
  pic#add ((new colored_point (2,2) "red") :> point);
  pic#add ((new verbose_point (3,3)) :> point);;
- : unit = ()
# pic#to_string () ;;
- : string = "[ (1,1) (2,2)[red] point=(3,3),distance=4.24264068712]"
```

Razreda `colored_pont` in `verbose_point` redefinirata funkcijo `to_string`, zato je izpis drugačen. Razred `colored_point` izpiše še barvo medtem ko `verbose_point` izpiše še razdaljo do izhodišča.

V zgornjem primeru smo kolekciji objektov poslali sporočilo `to_string`, ki se izvede tako, da se vsakemu elementu kolekcije pošlje sporočilo `to_string`. Ker ima

vsak razred definirano svojo metodo `to_string` dobimo različne izpise. Naslednji primer prikaže izvajanje iste metode `to_string`, ki se obnaša drugače v odvisnosti od konkretnega razreda kateremu objekt pripada.

Definirajmo še en razred `colored_point_1`, ki podeduje vse lastnosti od razreda `colored_point`; ta novi razred redefinira metodo `get_color` vendar ne redefinira `to_string`.

```
# class colored_point_1 coord c =
  object
    inherit colored_point coord c
    val true_colors = ["white"; "black"; "red"; "green"; "blue"; "yellow"]
    method get_color = if List.mem c true_colors then c else "UNKNOWN"
  end ;;
```

Metoda `to_string` je ista za oba razreda obarvanih točk.

```
# let p1 = new colored_point (1,1) "Blue";;
val p1 : colored_point = <obj>
# p1#to_string () ;;
- : string = "( 1, 1) [Blue] "
# let p2 = new colored_point_1 (1,1) "Blue";;
val p2 : colored_point_1 = <obj>
# p2#to_string () ;;
- : string = "( 1, 1) [UNKNOWN] "
```

Povezava metode `get_color` znotraj `to_string` ni fiksna v času prevajanja. Koda, ki se izvaja nad danim objektom se določi v času izvajanja na osnovi razreda objekta.

Za instanco `colored_point` aktiviranje sporočila objektu `to_string` povzroči izvajanje metode `get_color`, ki je definirana v razredu `colored_point`. Po drugi strani pošiljanje sporočila `to_string` primerku `colored_point_1` sproži aktiviranje metode `to_string` v staršu, ki pa pokliče metodo `get_color` v `colored_point_1`.

7.4.8 Polimorfizem vsebovanosti

Kombinacija zakasnjene povezovanja in podtipov omogoča novo vrsto polimorfizma: *polimorfizem vsebovanosti*. Ta omogoča obravnavanje vrednosti kateregakoli podtipa kot vrednosti pričakovanega supertipa.

Čeprav statični tipi garantirajo, da se bo pri pošiljanju sporočila vedno našla ustrezna metoda, je obnašanje metode lahko različno v odvisnosti od konkretnega objekta oz. od tipa tega objekta.

Relacija podtip je definirana na tipih: uporabniško definiranih tipih kot tudi na tipih modulov in razredov². Dedovanje je po drugi strani definirano s strukturo razredov oz. z razmerji `IS_A` med razredi, ki so uporabljena kot kanal za dedovanje.

²Predstaviti relacijo *podtip* v poglavju Tipi (*).

Različno od glavnega toka programskih jezikov kot npr. C++, Java, in SmallTalk, so podtipi v Ocaml različni pojem od dedovanja.

Glavni razlog za to je v tem, da imajo lahko instance nekega razreda c2 tip, ki je podtip tipa razreda c1, četudi razred c2 ne deduje od razreda c1. Razred `colored_point` na primer je lahko definiran neodvisno od razreda `point`. Tip razreda `colored_point` je še vedno podtip tipa razreda `point`.

Obnašanje družine razredov

Pod imenom *polimorfizem* (lahko tudi: večobličnost) mislimo:

1. zmožnost aplikacije funkcije na argumentih različnih “oblik” oz. tipov in
2. pošiljanje sporočil objektom različnih tipov oz. “oblik”.

Prvi primer smo predstavili v okviru funkcijskega/imperativnega jedra Ocaml–imenovali smo ga *parametrični polimorfizem*. Polimorfični parametri funkcije imajo parametrični tip oz. spremenljivke tipov. Tako definirana funkcija bo izvajala isto kodo za različne tipe parametrov. Koda ni odvisna od strukture parametrov.

Relacija podtip v povezavi z zakasnjanim povezovanjem predstavlja novo vrsto polimorfizma metod: *polimorfizem vsebovanosti*. Ta dovoljuje, da lahko pošljemo isto sporočilo objektom različnih tipov, ki pa morajo imeti skupni super-razred.

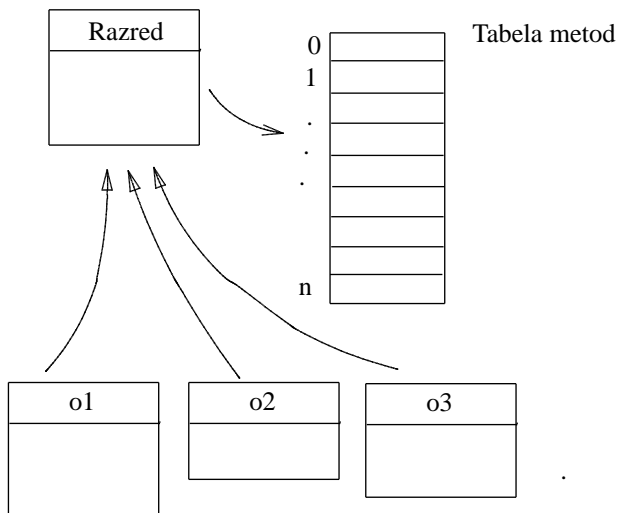
V enem izmed prejšnjih primerov smo konstruirali seznam točk, kjer so nekatere izmed točk primerki razreda `colored_point`, ki so se obravnavale kot točke. Pošiljanje sporočil različnim objektom seznama povzroči proženje različnih metod v odvisnosti od razreda objekta.

Enako sporočilo m lahko torej pošljemo primerkom razredov iz drevesa podtipov razreda c, ki vsebuje definicijo m. Za razliko od parametričnega polimorfizma je koda, ki se izvaja za isto sporočilo *različna* za primerke različnih razredov.

Z uporabo parametriziranih razredov lahko uporabljamo obe vrsti polimorfizma skupaj: parametrični polimorfizem in polimorfizem vsebovanosti.

7.5 Tip objektov

7.5.1 Razredi in tipi



Slika 7.4: Predstavitev razredov in objektov

7.5.2 Podrazredi in podtipi

7.6 Implementacija razredov in objektov

- = Predstavitev objektov in razredov.
- = Implementacija dedovanja.
- = Implementacija večkratnega dedovanja (*).
- = Preverjanje tipov razredov (*).

7.6.1 Predstavitev objektov

Objekt je razdeljen na dva dela: *variabilen del* in *fiksni del*. Variabilni del vsebuje spremenljivke objekta, kot v primeru zapisov. Fiksni del ustreza tabeli metod, ki si jo delijo vse instance razreda. Predstavitev razredov in objektov je podana na Sliki ??.

Tabela metod je redka tabela (angl. sparse table) funkcij. Vsaki metodi ustreza enoličen identifikator, ki je uporabljen kot indeks v tabelo metod.

Predpostavljamo, da imamo "strojno inštrukcijo" `GETMETHOD(o,n)`, ki vzame dva parametra: objekt o in indeks n , in vrne funkcijo, ki je povezana s tem indeksom. Rezultat klica metode `GETMETHOD(o,n)` bomo imenovali f_n .

7.6.2 Razpečevanje sporočila

Pri prevajanju sporočila `send o#m` se izračuna indeks `n` metode z imenom `m` in generira koda za apliciranje metode `GETMETHOD(o,n)`, ter rezultata funkcije f_n na objektu `o`. Dinamično (zakasnjeno) povezovanje je realizirano preko klica metode `GETMETHOD` v času izvajanja.

Pošiljanje sporočila samemu sebi preko `self` se tudi prevede v iskanje indeksa za metodo sporočila, čemur sledi klic funkcije iz tabele metod.

Poglejmo si še kaj se dogaja pri uporabi dedovanja. Ker ima ime metode vedno isti indeks navkljub redefiniciji v pod-razredu se nova metoda doda v tabelo pod-razreda pod isti indeks kot redefinirana metoda v nadrejenem razredu.

Pošiljanje sporočila `to_string` instanci razreda `point` bo sprožilo prevajanje imena metode v tabeli razreda `point`, metem ko bo pošiljanje iste metode razredu `colored_point` sprožilo prevajanje imena metode v tabeli razreda `colored_point`. Indeks metode bo v obeh primerih enak.

Zaradi te invariance indeksa so podtipi koherentni z izvajanjem. Če je primerek `colored_point` eksplicitno omejen s tipom `point`, potem se pri pošiljanju sporočila `to_string` izračuna indeks metode iz razreda `point`. Ta indeks je enak indeksu `to_string` v razredu `colored_point`. Iskanje se bo odvijalo v tabeli, ki je povezana s primerkom `colored_point`.

7.7 Generativnost

Poleg zmožnosti modeliranja problema z uporabo *agregacije* in *klasifikacije* nudi objektno usmerjeno programiranje zmožnost ponovne uporabe in spreminjanja obstoječih razredov. Razširjanje razreda mora ohranjati statično preverjanje tipov programskega jezika.

Z uporabo *abstraktnih razredov* lahko izpostavimo kodo in naredimo družino podrazredov, ki funkcionira kot komunikacijski protokol. Abstraktni razred fiksira imena in tipe sporočil, ki jih lahko sprejmejo instance otrok abstraktnega razreda. Ta tehnika je zelo uporabna skupaj z večkratnim dedovanjem.

Večkratno dedovanje in abstraktni razredi ... (*)

Parametrizirani razredi omogočajo definicijo razredov s parametričnimi tipi. Razred lahko tako instanciramo z naborom konkretnih tipov za parametre. Na ta način dobimo parametrični polimorfizem, ki omogoča zelo generativen opis razreda.

7.7.1 Abstraktni razredi

V abstraktnih razredih so nekatere metode definirane brez telesa. Takšne metode imenujemo abstraktne. Abstrakten razred ne moremo instancirati—ne moremo uporabiti metode `new`. Ključno besedo `virtual` uporabljamo za abstraktne metode in razrede.

Sintaksa:

```
class virtual name = object . . . end
```

Razred mora biti definiran kot abstrakten, če vsebuje vsaj eno abstraktno metodo.

Sintaksa:

```
method virtual name : type
```

Če podrazred redefinira abstraktno metodo potem ta lahko postane *konkretna*, sicer mora biti razred označen kot abstrakten.

Na primer, želimo definirati množico predstavljenih objektov, ki vsebujejo metodo `print`. Le-ta pretvori vsebino objekta v niz znakov.

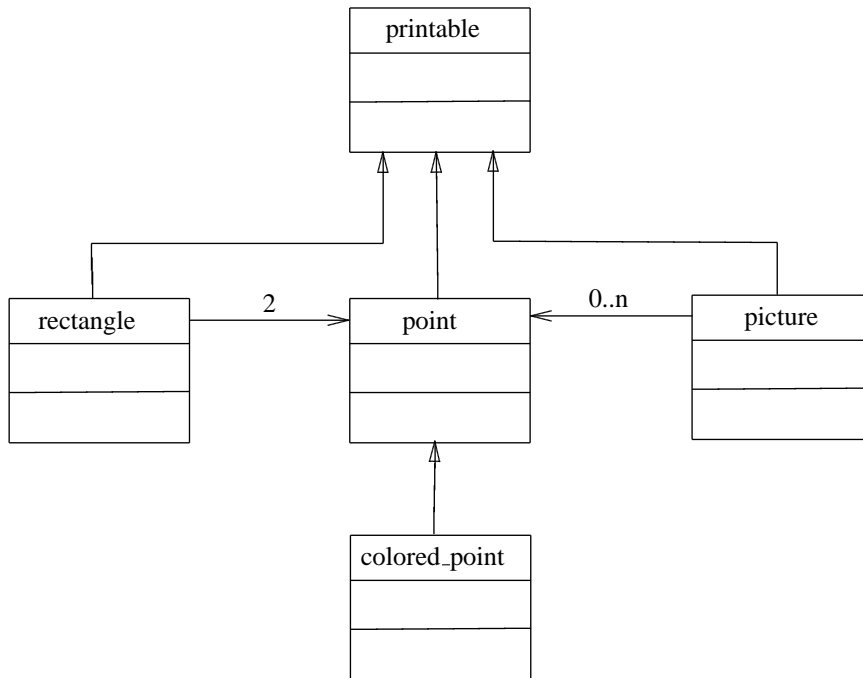
Vsi takšni objekti potrebujejo metodo `to_string` zato definiramo razred `printable`. Niz znakov se spreminja v odvisnosti od razreda objekta. Metodo `to_string` definiramo abstraktno zaradi česar so razred `printable` kot tudi vsi nasledniki abstraktni.

```
# class virtual printable () =
  object(self)
    method virtual to_string : unit -> string
    method print () = print_string (self#to_string ())
  end ;;
class virtual printable :
  unit ->
  object
    method print : unit -> unit
    method virtual to_string : unit -> string
  end
```

Abstratnost razreda in njegovih metod je vidna iz tipa, ki ga izpiše interpreter.

Iz tega razreda definiramo hierarhijo, ki je predstavljena na Sliki ???. Razrede `point`, `colored_point` in `picture` definiramo tako, da dodamo deklaracijam spremenljivk in metod vrstico `inherit printable ()` s čimer dobijo razredi metodo `print` po vseh vejah dedovanja.

Poglejmo si zdaj primere kreacije objektov razredov `point`, `colored_point` in `picture`, ter uporabo metode `print` nad kreiranimi objekti.



Slika 7.5: Razmerja med razredi predstavljivih objektov

```
# let p = new point (1,1) in p#print () ;;
(1,1)- : unit = ()
# let pc = new colored_point (2,2) "blue" in pc#print () ;;
(2,2)[blue]- : unit = ()
# let t = new picture 3 in t#add (new point (1,1)) ;
                                t#add (new point (3,2)) ;
                                t#add (new point (1,4)) ;
                                t#print () ;;
[(1,1) (3,2) (1,4)]- : unit = ()
```

Podrazred `rectangle` predstavljen spodaj deduje od razreda `printable` in definira metodo `to_string`. Spremenljivke objekta `llc` in `urc` pomenijo točka spodnjega levega kota ter točka zgornjega desnega kota pravokotnika.

```
# class rectangle (p1,p2) =
  object
    inherit printable ()
    val llc = (p1 : point)
    val urc = (p2 : point)
    method to_string () = "["^llc#to_string ()^","^urc#to_string ()^"]"
  end ;;
class rectangle :
  point * point ->
  object
    val llc : point
    val urc : point
    method print : unit -> unit
    method to_string : unit -> string
  end
```

Razred `rectangle` deduje od abstraktnega razreda `printable` in podeduje metodo `print`. Vsebuje dve spremenljivki tipa `point`: `llc` in `urc`. Metoda `to_string` pošlje sporočili `to_string` točkama, ki jih določata spremenljivki `llc` in `urc`.

```
# let r = new rectangle (new point (2,3), new point (4,5));;
val r : rectangle = <obj>
# r#print () ;;
[(2,3),(4,5)]- : unit = ()
```

Abstraktni razredi in večkratno dedovanje

Z večkratnim dedovanjem lahko dan razred podeduje podatkovna polja in metode iz večih razredov. V primeru identičnih imen polj ali metod se upošteva zadnja deklaracija po vrstnem redu deklaracij dedovanja.

Metodo iz natančno določenega razreda lahko referenciramo s povezovanjem imen s podedovanimi razredi z uporabo stavka `inherit`.

To ne velja za spremenljivke primerka: če podedovan razred prekrije spremenljivko instance prejšnjega podedovanega razreda, zadnja ni več dostopna³.

Večkratno dedovanje poveča ponovno uporabnost razredov⁴.

Poglejmo si definicijo abstraktnega razreda `geometric_object`, ki deklarira dve virtualne metodi `compute_area` in `compute_peri` za izračun področja in obsega.

```
# class virtual geometric_object () =
  object
    method virtual compute_area : unit -> float
    method virtual compute_peri : unit -> float
  end;;
```

Potem redefiniramo razred `rectangle` na naslednji način.

```
# class rectangle_1 ((p1,p2) : point * point) =
  object
    inherit printable ()
    inherit geometric_object ()
    val llc = p1
    val urc = p2
    method to_string () =
      "["^(llc#to_string () )^", "^(urc#to_string () )^"]"
    method compute_area () =
      float ( abs(urc#get_x - llc#get_x) * abs(urc#get_y - llc#get_y))
    method compute_peri () =
      float ( (abs(urc#get_x - llc#get_x) + abs(urc#get_y - llc#get_y)) * 2)
  end;;

class rectangle_1 :
  point * point ->
  object
    val llc : point
    val urc : point
    method compute_area : unit -> float
    method compute_peri : unit -> float
    method print : unit -> unit
    method to_string : unit -> string
  end
```

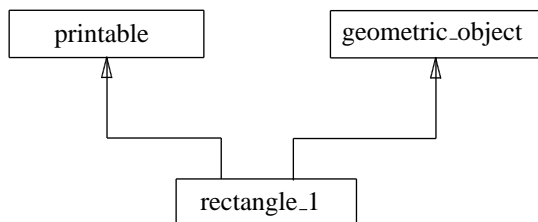
Klasifikacijska hierarhija razredov je prikazana na Sliki ??.

Da bi se izognili pisanju metod definiranih v razredu `rectangle` lahko dedujemo direktno od razreda `rectangle` kot je prikazano na Sliki ??.

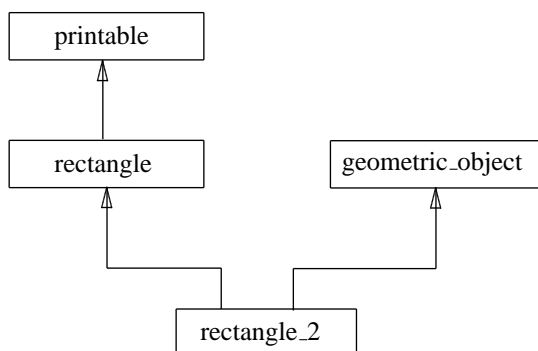
V razredu `rectangle_2` je tako potrebno definirati še samo abstraktne metode abstraktnega razreda `geometric_object`.

³Bolj natančen opis predstavitve spremenljivk objekta (*).

⁴Primer referenciranja nadrejenih razredov (*).



Slika 7.6: Večkratno dedovanje



Slika 7.7: Večkratno dedovanje (2)

```
# class rectangle_2 (p2 : point * point) =
  object
  inherit rectangle p2
  inherit geometric_object ()
  method compute_area () =
    float ( abs(urc#get_x - llc#get_x) * abs(urc#get_y - llc#get_y))
  method compute_peri () =
    float ( (abs(urc#get_x - llc#get_x) + abs(urc#get_y - llc#get_y)) * 2)
end;;
```

7.7.2 Parametrizirani razredi

Parametrizirani razredi omogočajo uporabo *parametričnega polimorfizma*. Kot pri deklaracijah tipov so lahko tudi razredi parametrizirani s spremenljivimi tipov. To omogoča še dodatne možnosti za generativnost in ponovno uporabo kode.

Sintaksa se zelo malo razlikuje od sintakse parametriziranih tipov; parametri tipov so definirani med oglatimi oklepaji.

Sintaksa:

```
class ['a,'b,...] name = object . . . end
```

Tip Ocaml se napiše kot običajno: ('a,'b,...) name.

Primer slike picture, ki ima parametriziran tip s katerim določimo elemente slike oz. vrh hierarhije razredov, katerih primerki bodo elementi slike (*).

Poskusimo zdaj definirati parametrični razred pair, ki bo shranjeval pare vrednosti (objektov) poljubnih tipov oz. razredov. Naivna definicija razreda pair bi bila:

```
# class pair x0 y0 =
  object
  val x = x0
  val y = y0
  method fst = x
  method snd = y
end ;;
```

Characters 6-106:

Some type variables are unbound in this type:

```
class pair :
  'a ->
  'b -> object val x : 'a val y : 'b method fst : 'a method snd : 'b end
The method fst has type 'a where 'a is unbound
```

Dobimo napako tipa, ki je bila že omenjena pri definiciji razreda `point`. Sporočilo napake pravi, da je bila spremenljivka tipa `'a` prirejena parametru `x0` (kot tudi `x` in `fst`) in `'a` ni povezan.

Kot v primeru *parametriziranih tipov* je potrebno *parametrizirati razred* `pair` z dvema spremenljivkama ter omejiti tipa konstrukcijskih parametrov `x0` in `y0` za korektno izvajanje sklepanja s tipi.

```
# class ['a,'b] pair (x0:'a) (y0:'b) =
  object
    val x = x0
    val y = y0
    method fst = x
    method snd = y
  end ;;
class ['a, 'b] pair :
  'a ->
  'b -> object val x : 'a val y : 'b method fst : 'a method snd : 'b end
```

Mehanizem za sklepanje s tipi izpiše vmesnik razreda parametriziran s spremenljivkama `'a` in `'b`. Ko se konstruira parametriziran razred se tipi parametrov instancirajo s tipi konstrukcijskih parametrov.

```
# let p = new pair 2 'X';;
val p : (int, char) pair = <obj>
# p#fst;;
- : int = 2
# let q = new pair 3.12 true;;
val q : (float, bool) pair = <obj>
# q#snd;;
- : bool = true
```

V deklaracijah razredov so tipi predstavljeni v oglatih oklepajih, medtem ko so v predstavitvi tipov predstavljeni v navadnih oklepajih.

Dedovanje parametriziranih razredov

Pri dedovanju od parametriziranih razredov je potrebno deklarirati parametre razreda. Poglejmo si primer. Definirajmo razred `acc_pair`, ki deduje od `('a,'b) pair`; dodamo dve metodi za dostop do polj: `get1` in `get2`,

```
# class ['a,'b] acc_pair (x0 : 'a) (y0 : 'b) =
  object
    inherit ['a,'b] pair x0 y0
    method get1 z = if x = z then y else raise Not_found
    method get2 z = if y = z then x else raise Not_found
```

```

    end;;
class ['a, 'b] acc_pair :
  'a ->
  'b ->
  object
    val x : 'a
    val y : 'b
    method fst : 'a
    method get1 : 'a -> 'b
    method get2 : 'b -> 'a
    method snd : 'b
  end
# let p = new acc_pair 3 true;;
val p : (int, bool) acc_pair = <obj>
# p#get1 3;;
- : bool = true

```

Parametre tipov lahko definiramo bolj natančno tako, da uporabimo *konkretne tipe*. Poglejmo si primer para točk.

```

# class point_pair (p1,p2) =
  object
    inherit [point,point] pair p1 p2
  end;;
class point_pair :
  point * point ->
  object
    val x : point
    val y : point
    method fst : point
    method snd : point
  end

```

Razred `point_pair` ne potrebuje več parametrov tipa, ker sta `'a` in `'b` že dokončno definirana.

Meta-programski pogled na parametrične razrede. (*)⁵

⁵Parametrizirani tipi vsebujejo meta-programске koncepte. Konstrukcija konkretnega tipa na osnovi parametriziranega tipa je oblika meta-programiranja. Razred se konstruira iz definicije parametriziranega tipa in konkretnih tipov, ki so uporabljeni kot parametri izraza višjega reda.

Poglavje 8

MODULI

Modularna zasnova programov in *modularno programiranje* omogoča dekompozicijo programov v več programskih enot, ki jih imenujemo moduli. Module lahko razvijamo samostojno, neodvisno od preostalega dela programskega sistema.

Modul lahko prevedemo ločeno od ostalega programa. Programer ne potrebuje dostopa do izvorne kode modula ampak lahko dela preprosto s prevedeno kodo.

Programer mora poznati *vmesnik* do modulov, ki vsebuje vrednosti, funkcije, tipe in pod-module, ki jih modul nudi uporabnikom navzven.

Vmesnik modula skriva podrobnosti implementacije modula. Vse kar programi, ki uporabljajo modul vedo o modulu je dostopno preko vmesnika; natančna implementacija modula je skrita.

Vzdrževalec modula ima precejšnjo fleksibilnost pri razvoju implementacije modula. Dokler se ne spremeni vmesnik se pomen funkcij modula ne spreminja. Uporabnik modula ne bo opazil spremembe v implementaciji. Na ta način lahko bistveno poenostavimo razvoj kompleksnih sistemov.

Vmesnik modula torej zakrije dele implementacije, ki jih avtor noče dati v javno uporabo.

Ocaml omogoča definicijo *parametriziranih modulov*. Te moduli lahko vzamejo druge module za parametre in s tem omogočijo ponovno uporabo kode.

8.1 Moduli kot enota prevajanja

Objektni caml vsebuje množico predefiniranih modulov, ki jih lahko enostavno uporabimo v programu. V tej sekciji si bomo ogledali kako lahko sami definiramo nov

modul.

8.1.1 Moduli v programskem jeziku C

Eden izmed prvih načinov na katerega je bila razdeljena koda v programskih jezikih je bila delitev kode v *datoteke*, ki vsebujejo del celotnega programa, ter v *vmesnike*, ki vsebujejo deklaracije funkcij in podatkovnih struktur v datotekah.

Ena datoteka programa v programskem jeziku C vsebuje kodo, ki je vezana na nek koncept. Na primer, datoteka vsebuje funkcije in podatkovne strukture za delo z določeno vhodno/izhodno napravo. Ali, datoteka lahko vsebuje funkcije za delo z določeno podatkovno strukturo kot je na primer asociativno polje.

Običajno zraven datoteke s podaljškom “.c”, ki vsebuje kodo modula, definiramo še datoteko vmesnika s podaljškom “.h”, ki vsebuje deklaracije funkcij in podatkovnih struktur modula.

Datoteko s kodo v programskem jeziku C lahko prevedemo samostojno v objektno kodo, brez ostalega dela programa. Če modul uporablja še druge module (datoteke) potem pri prevajanju tega modula uporabljamo tudi vmesnike preostalih modulov.

Primer C modula ... (?).

Datoteke s kodo v programskem jeziku C niso zares pravi moduli kot jih bomo definirali kasneje. Iz stališča celotnega programa napisanega v programskem jeziku C predstavlja ena datoteka samo del programa, ki tvori celoto. Deklaracije podatkovnih struktur in funkcij se sicer lahko pojavljajo kjerkoli drugje v programu, ker služijo za napovedovanje: prevajalniku povedo, da se bo funkcija, ki jo deklariramo pojavila nekje kasneje v kodi.

Konceptualna zasnova modulov v obliki enostavnih datotek v programskem jeziku C je služila kot vzgled pri definiciji modulov v množici programskih jezikov kot so Modula in tudi Ocaml.

8.1.2 Moduli kot enota prevajanja v Ocaml

Podobno kot v programskem jeziku C lahko tudi v Ocaml pišemo dele programa v datoteke, ki služijo kot moduli. Enako kot v programskem jeziku C imamo dve datoteki: vmesnik in implementacijo modula.

Datoteka s kodo modula ima končnico “.ml” in vmesnik modula je shranjen v datoteki s podaljškom “.mli”. Podobno kot v programskem jeziku C je tudi datoteka z implementacijo v Ocaml lahko uporabljena samostojno brez vmesnika.

Poglejmo si naprej primer modula, ki vsebuje implementacijo sklada. Modul je shranjen v datoteki `Stack.ml`. Implementacija sklada je osnovana na seznamih in je sicer del standardne knjižnice `Ocaml`.

```

type 'a t = { mutable c : 'a list }
exception Empty
let create () = { c = [] }
let clear s = s.c <- []
let push x s = s.c <- x :: s.c
let pop s = match s.c with hd :: tl -> s.c <- tl; hd
           | [] -> raise Empty
let length s = List.length s.c
let iter f s = List.iter f s.c

```

Modul definira tip `t`, ki je zapis `s` spremenljivo komponento `c` tipa `'a list`. Tip `t` predstavlja *abstrakten podatkovni tip* in je osnovna podatkovna struktura modula. Za modulom delamo tako, da najprej kreiramo podatkovno strukturo tipa `Stack.t`, ki predstavlja sklad. Podatkovno strukturo je potem potrebno podati vsem funkcijam modula kot parameter.

Napišimo še eno datoteko `primer.ml`, ki uporablja modul `Stack.ml`. Koda v datoteki najprej kreira sklad, doda par elementov na sklad in prebere iz sklada ter izpiše.

```

let s = Stack.create ();;
Stack.push 1 s; Stack.push 2 s; Stack.push 3 s;;
Printf.printf "elementi: %i, %i in %i\n"
              (Stack.pop s) (Stack.pop s) (Stack.pop s);;

```

Vrednosti, tipe in izjeme definirane v modulu lahko referenciramo z uporabo notacije s piko (`Module.identifier`), ali z uporabo gradnika `open`. Zgornja koda je lahko zapisana z uporabo operacije `open` na naslednji način.

```

open Stack;;
let s = create ();;
push 1 s; push 2 s; push 3 s;;
Printf.printf "elementi: %i, %i in %i\n"
              (pop s) (pop s) (pop s);;

```

Program prevedemo z uporabo prevajalnika `ocamlc`. Unix ukazi za prevajanje in zagon programa so sledeči.

```

$ ocamlc -o primer stack.ml primer.ml
$ ./primer
elementi: 1, 2 in 3

```

V primeru, da modul nima vmesnika velja, da so dostopne vse strukture definirane v modulu. Z vmesnikom lahko omejimo množico funkcij in podatkovnih struktur dostopnih od zunaj modula. Bolj podrobna predstavitev načinov uporabe vmesnika modula bo podana v naslednji sekciji.

V vmesniku modula so zapisani tipi in vrednosti, ki so dostopni izven modula ter funkcije, ki so dostopne izven modula. Vrednosti in funkcije vmesnika deklariramo na sledeč način.

```
val nom : type
```

Poglejmo si zdaj še datoteko `stack.mli`, ki vsebuje vmesnik modula. V vmesniku so našete vse funkcije definirane v `stack.ml` razen funkcije `iter`.

```
type 'a t
exception Empty

val create: unit -> 'a t
val push: 'a -> 'a t -> unit
val pop: 'a t -> 'a
val clear : 'a t -> unit
val length: 'a t -> int
```

Za tip `t`, ki je definiran v modulu `stack.ml`, smo v vmesniku zapisali samo ime tipa. Pravimo da je tip `t` *abstrakten*—uporabniki modula `Stack` nimajo dostopa do strukture tipa `t` in do strukture primerkov tipa. Dostop do primerkov tipa je omogočen samo preko funkcij.

8.1.3 Povezovanje vmesnikov in implementacij

Sklad je sestavljen iz dveh delov: a) implementacije, ki poda *definicije* tipov, spremenljivk in funkcij, in b) vmesnik, ki vsebuje *deklaracije* izvoženih definicij.

Vse deklaracije morajo imeti ustrezne definicije v implementacijskem delu. Podobno morajo biti tudi vsi definirani tipi, spremenljivke in funkcije, ki naj bodo dostopni izven modula, deklarirani v vmesniku.

Relacija med vmesnikom in implementacijo ni simetrična. Implementacija lahko vsebuje več definicij kot je deklaracij v vmesniku. Na primer, pri implementaciji pogosto uporabljamo pomožne funkcije in vrednosti. Te dodatne funkcije niso dostopne od zunaj.

Podobno lahko tudi vmesnik omeji tip definicije. Poglejmo na primer modul, ki definira funkcijo identitete `id: 'a -> 'a`. V vmesniku lahko deklariramo `id` z drugim tipom npr. `id: int -> int`.

Ker je vmesnik modula ločen od implementacije je mogoče imeti več implementacij in isti vmesnik. Na ta način lahko testiramo implementacije neke strukture z različnimi algoritmi za iste operacije. Poglejmo si na primer implementacijo sklada s polji namesto s seznamami.

Datoteko z implementacijo moramo spet imenovati `stack.ml`. Abstraktni tip `t` predstavlja sklad kot zapis z dvema komponentama: kazalec na prvi prosti element in polje, ki vsebuje sklad. Polje po potrebi povečamo vedno ko postane sklad premajhen.

```

type 'a t = { mutable sp : int; mutable c : 'a array }
exception Empty
let create () = { sp=0 ; c = [|]| }
let clear s = s.sp <- 0; s.c <- [|]|
let size = 5
let increase s = s.c <- Array.append s.c (Array.create size s.c.(0))
let push x s =
  if s.sp >= Array.length s.c then increase s ;
  s.c.(s.sp) <- x ;
  s.sp <- succ s.sp
let pop s =
  if s.sp = 0 then raise Empty
  else let x = s.c.(s.sp) in s.sp <- pred s.sp ; x
let length s = s.sp
let iter f s = for i = pred s.sp downto 0 do f s.c.(i) done

```

Ta implementacija zadostuje pogojem vmesnika `stack.mli`. Uporabljamo jo lahko torej namesto prejšnje implementacije sklada s seznamom. Uporabniki modula se lahko niti ne zavedajo, da je implementacija modula, ki ga uporabljajo spremenjena.

8.2 Jezik modulov

Vmesnik jezika modulov imenujemo signatura in njegovo implementacijo strukturo. Večkrat bomo uporabili ime “modul” namesto strukture.

Sintaksa za deklaracijo signature in strukture je naslednja.

Sintaksa:

```

module type NAME = sig declarations end
module Name = struct definitions end

```

Ime modula se mora začeti z veliko črko. Ta omejitev ne velja za signature, čeprav bomo tudi za signature uporabljali veliko začetnico.

Signature in strukture ni nujno, da so povezane z imenom. Uporabljamo lahko tudi anonimne signature in strukture. Sintaksa za definicijo anonimnih signatur in struktur je naslednja.

Sintaksa:

```
sig declarations end
struct definitions end
```

Imeni signatura in struktura se lahko nanašajo na signature in strukture definirane z ali brez imena.

Vsaka struktura ima privzeto signaturo, ki jo izračuna sistem za delo s tipi. Ta signatura vsebuje vse definicije, ki se pojavijo v strukturi, označene z najbolj splošnim tipom.

Sintaksa:

```
module Name : signature = structure
module Name = (structure : signature)
```

Ko je signatura definirana eksplicitno, sistem preveri da je vsaka komponenta definirana v signaturi definirana tudi v strukturi. Definicije v strukturi so lahko kvečjemu bolj *splošne* kot tiste definirane v signaturi oz. obratno: s signaturami lahko *omejimo* tip definicije v strukturi.

Dostop do komponent modula je omogočen z uporabo notacije s piko.

Sintaksa:

```
Name1.name2
```

Ime name2 je določeno z imenom Name1, ki označuje modul. Ime modula lahko izpustimo po uporabi ukaza open.

Sintaksa:

```
open Name
```

V definicijskem območju tega ukaza lahko uporabljamo name2 za delo s komponentami modula Name. V primeru konfliktov vedno na novo odprti modul prekrije definicije prejšnjega modula.

8.2.1 Primer modula

Poglejmo si zdaj primer modula, ki realizira več podatkovnih struktur: vrsto in sklad.

Uporabili bomo domiselno podatkovno stukturo, ki vsebuje par seznamov. Prvi seznam služi za dodajanje in odvzemanje na začetku vrste, drugi seznam pa za dodajanje in jemanje na koncu vrste. V obeh primerih dodajamo in odvezemamo elemente iz zacetka seznama. Vedno ko zmanjka elementov v enem izmed parov seznamov obrnemo drugi seznam, ga postavimo na mesto praznega in nadaljujemo z delom.

Tip abstraktne podatkovne strukture PairOfLists je referenca na par seznamov.

```

# module PairOfLists = struct
  type 'a t = ('a list * 'a list) ref
  exception Empty
  let create () = ref ([], [])

  let enqueue x queue =
    let front, back = !queue in
      queue := (x::front, back)

  let rec dequeue queue =
    match !queue with
    | (front, x :: back) -> queue := (front, back);
      x
    | ([], []) -> raise Empty
    | (front, []) -> queue := ([], List.rev front);
      dequeue queue

  let push x queue = enqueue x queue

  let rec pop queue =
    match !queue with
    | (x::front,back) -> queue := (front,back); x
    | ([],[]) -> raise Empty
    | ([],back) -> queue := (List.rev back,[]);
      pop queue
end;;

```

Poglejmo si primer uporabe podatkovne strukture `PairOfLists`. Kreiramo podatkovno strukturo, dodamo dva elementa na začetek vrste in odvzamemo oba elementa, prvega iz konca vrste in drugega iz začetka vrste.

```

# let q = PairOfLists.create ();;
val q : ('_a list * '_b list) ref = {contents = ([], [])}
# PairOfLists.enqueue 1 q; PairOfLists.push 2 q;;
- : unit = ()
# q;;
- : (int list * '_a list) ref = {contents = ([2; 1], [])}
# PairOfLists.dequeue q;;
- : int = 1
# q;;
- : (int list * int list) ref = {contents = ([], [2])}
# PairOfLists.pop q;;
- : int = 2

```

8.2.2 Moduli in skrivanje informacij

V nadaljevanju bo predstavljen primer skrivanja abstraktnega tipa modula. Definirali bomo tip s katerim predstavimo običajen sklad Stack oz. vmesnik modula s katerim bomo lahko omejili implementacijo konkretnega sklada.

```
# module type Stack =
  sig
    type 'a t
    exception Empty
    val create: unit -> 'a t
    val push: 'a -> 'a t -> unit
    val pop: 'a t -> 'a
  end ;;
```

Sklad lahko zdaj definiramo na osnovi modula PairOfLists in vmesnika Stack.

```
# module Stack1 = (PairOfLists:Stack);;
module Stack1 : Stack
```

Kreirajmo zdaj konkreten sklad in preizkusimo delovanje sklada.

```
# let s = Stack1.create ();;
val s : '_a Stack1.t = <abstr>
# Stack1.push 1 s;;
- : unit = ()
# Stack1.push 2 s;;
- : unit = ()
# Stack1.pop s;;
- : int = 2
# Stack1.pop s;;
- : int = 1
```

Ob kreaciji sklada s Ocaml izpiše da je kreirana podatkovna struktura abstraktna. Uporabnik sklada Sklad1 ne more videti definicijo abstraktne podatkovne strukture, ker je zakrita z vmesnikom Stack kjer je definirano samo ime tipa.

V primeru, da je abstraktni tip modula v vmesniku samo deklariran nimamo vpogleda tudi do konkretne podatkovne strukture s katerim je implementiran sklad.

Abstrakcija nad tipi zagotavlja, da je edini način za konstrukcijo in manipulacijo vrednosti danega tipa preko funkcij, ki so izvožene iz modula.

V naslednjem primeru najprej izpišemo vrednost q, ki je primerek abstraktnega tipa PairOfLists, potem pa še vrednost s, ki je primerek tipa definiranega z Stack1.

```
# PairOfLists.push 1 q; PairOfLists.push 2 q;;
```



```

- : unit = ()
# Stack1.push 1 s; Stack1.push 2 s;;
- : unit = ()
# q;;
- : (int list * int list) ref = {contents = ([2; 1], [])}
# s;;
- : int Stack1.t = <abstr>

```

V prvem primeru vidimo primerek tipa `t` definirane z modulom `PairOfLists`, v drugem primeru pa smo izpisali vrednost abstraktnega podatkovnega tipa, ki ni vidna navzven. V obeh primerih je uporabljen isti tip vrednosti le da je pri drugi vrednosti zakrit tip.

8.2.3 Več pogledov na modul

Jezik modulov in omejevanje modulov s signaturami omogoča kreiranje različnih pogledov na isto strukturo.

V prejšnji sekciji smo implementirali modul `PairOfLists`, ki vsebuje podatkovne strukture in operacije s katerimi lahko reliziramo sklad in vrsto. Sklad smo že definirali tako, da smo omejili implementacijo modula `PairOfLists` z vmesnikom `Stack`.

Definirajmo zdaj še tip modulov za delo z vrsto `Queue`. Vmesnik bo vseboval samo funkcije za delo z vrsto—ostale funkcije bomo zakrili tako, da jih ne bomo deklarirali z vmesnikom.

```

# module type Queue =
  sig
    type 'a t
    exception Empty
    val create: unit -> 'a t
    val enqueue: 'a -> 'a t -> unit
    val dequeue: 'a t -> 'a
  end ;;

```

Zdaj lahko definiramo modul `Queue1` iz implementacije modula `PairOfLists` in vmesnika `Queue`. Kreiramo še primerek abstraktnega tipa `Queue1.t` in vstavimo na začetek vrste števila 1 in 2 ter jih nato izločimo iz konca vrste.

```

# module Queue1 = (PairOfLists:Queue);;
module Queue1 : Queue
# let v = Queue1.create ();;
val v : '_a Queue1.t = <abstr>
# Queue1.enqueue 1 v; Queue1.enqueue 2 v;;
- : unit = ()
# Queue1.dequeue v;;

```

```
- : int = 1
# Queue1.dequeue v;;
- : int = 2
```

Isti modul `PairOfLists` nam je z dvema vmesnikoma služil za definicijo dveh različnih pogledov. Najprej smo definirali sklad `Stack1` in nato še vrsto `Queue1`.

8.3 Primeri implementacij modulov

V tem razdelku bomo predstavili primera implementacije dveh modulov: modul za delo s prioriteto vrsto in modul za delo z binarnim drevesom.

8.3.1 Modul za prioriteto vrsto

Primarna motivacija modulov je a) združevanje medseboj povezanih definicij kot so na primer podatkovni tipi in pripadajoče operacije in b) konsistentno poimenovanje definiranih tipov, vrednosti in funkcij.

Poglejmo si še en primer modula, ki implementira prioriteto vrsto.

```
# module PrioQueue =
  struct
    type priority = int
    type 'a queue = Empty | Node of priority * 'a * 'a queue * 'a queue
    let empty = Empty
    let rec insert queue prio elt =
      match queue with
      | Empty -> Node(prio, elt, Empty, Empty)
      | Node(p, e, left, right) ->
          if prio <= p
          then Node(prio, elt, insert right p e, left)
          else Node(p, e, insert right prio elt, left)
    exception Queue_is_empty
    let rec remove_top = function
      | Empty -> raise Queue_is_empty
      | Node(prio, elt, left, Empty) -> left
      | Node(prio, elt, Empty, right) -> right
      | Node(prio, elt, (Node(lprio, left, _, _) as left),
                    (Node(rprio, right, _, _) as right)) ->
          if lprio <= rprio
          then Node(lprio, left, remove_top left, right)
          else Node(rprio, right, left, remove_top right)
    let extract = function
      | Empty -> raise Queue_is_empty
```

```

        | Node(prio, elt, _, _) as queue -> (prio, elt, remove_top queue)
    end;;
module PrioQueue :
sig
    type priority = int
    type 'a queue = Empty | Node of priority * 'a * 'a queue * 'a queue
    val empty : 'a queue
    val insert : 'a queue -> priority -> 'a -> 'a queue
    exception Queue_is_empty
    val remove_top : 'a queue -> 'a queue
    val extract : 'a queue -> priority * 'a * 'a queue
end

```

Izven strukture dostopamo do komponent modula z uporabo notacije s piko. Na primer, `PrioQueue.insert` je v kontekstu vrednosti funkcija `insert`, kot je definirana znotraj modula `PrioQueue`. Podobno je `PrioQueue.queue` v kontekstu tipov tip `queue` definiran v `PrioQueue`.

```

# PrioQueue.insert PrioQueue.empty 1 "hello";;
- : string PrioQueue.queue =
PrioQueue.Node (1, "hello", PrioQueue.Empty, PrioQueue.Empty)

```

8.3.2 Modul za binarno drevo

```

# module type BINARYTREE =
sig
    type 'a btree
    val empty: 'a btree
    val member: 'a ->'a btree -> bool
    val insert: 'a ->'a btree -> 'a btree
    val extract: 'a btree -> 'a * 'a btree
    exception Tree_is_Empty
end
module type BINARYTREE =
sig
    type 'a btree
    val empty : 'a btree
    val member : 'a -> 'a btree -> bool
    val insert : 'a -> 'a btree -> 'a btree
    val extract : 'a btree -> 'a * 'a btree
    exception Tree_is_Empty
end

```

```

# module BinaryTree: BINARYTREE =
  struct
    type 'a btree = Empty | Node of 'a * 'a btree * 'a btree
    let empty = Empty
    let rec member x bt =
      match bt with
      | Empty -> false
      | Node(y, left, right) ->
        if x = y then true
        else if x < y then member x left else member x right
    let rec insert x bt =
      match bt with
      | Empty -> Node(x, Empty, Empty)
      | Node(y, left, right) ->
        if x <= y then Node(y, insert x left, right)
        else Node(y, left, insert x right)
    exception Tree_is_Empty
    let rec remove_top bt =
      match bt with
      | Empty -> raise Tree_is_Empty
      | Node(_, left, Empty) -> left
      | Node(_, Empty, right) -> right
      | Node(_, (Node(z, _, _) as left), right) ->
        Node(z, remove_top left, right)
    let extract bt =
      match bt with
      | Empty -> raise Tree_is_Empty
      | Node(y, _, _) as bt -> (y, remove_top bt)
  end;;
module Make : BINARYTREE

# module BT = BinaryTree;;
module BT :
  sig
    type 'a btree = 'a Make.btree
    val empty : 'a btree
    val member : 'a -> 'a btree -> bool
    val insert : 'a -> 'a btree -> 'a btree
    val extract : 'a btree -> 'a * 'a btree
    exception Tree_is_Empty
  end
# let x = BT.insert 10 BT.empty;;
val x : int BT.btree = <abstr>
# let x = BT.insert 15 x;;
val x : int BT.btree = <abstr>
# let x = BT.insert 5 x;;
val x : int BT.btree = <abstr>

```

8.4 Funktorji

Moduli so zaprte entitete, ki so definirane enkrat samkrat—ko je enkrat definiran abstraktni tip z uporabo jezika modulov, potem ni več mogoče dodajati novih metod brez modifikacije samega modula.

Dodajanje novih operacij, ki so odvisne od predstavitve tipa zahteva popravljanje izvorne kode modulov kot tudi vmesnika modula oz. signaturo modula. Na ta način seveda dobimo drugačen modul in uporabniki danega modula morajo biti ponovno prevedeni.

Če spremembe modula ne vsebujejo sprememb obstoječih signatur potem ostane prostanek programa pravilen in ga ni potrebno popravljati—potrebno ga je le ponovno prevesti.

Parametrizirani moduli ali *funktorji* dodajo modulom dodatno fleksibilnost pri definiciji generične kode—moduli so definirani na osnovi modulov, ki so podani kot argumenti funktorja. Koda funktorja je tako veliko bolj prilagodljiva konkretnim potrebam in se da veliko lažje ponovno uporabiti po potrebi.

8.4.1 Parametrizirani moduli

Parametrizirani moduli so v enakem razmerju z običajnimi moduli kot so funkcije v razmerju z običajnimi vrednostmi.

Podobno kot klic funkcije zgradi novo vrednost iz danih parametrov, parametriziran modul konstruira nov modul iz danih vrednosti, tipov in modulov, ki so podani kot parametri.

Parametrizirane module imenujemo tudi *funktorji*.

Funktorji razširijo jezik modulov z gradniki, ki bistveno izboljšajo ponovno uporabo kode in strukture.

Funktorji so definirani s sintakso, ki je blizu sintaksi funkcij.

Sintaksa:

```
functor ( Name : signature ) -> structure
```

Naslednji primer prikaže funktor, katerega parameter je modul Q znotraj katerega je definiran en sam tip $Q.t$. Funktor `Couple` konstruira en sam tip, ki definira tip kot par $Q.t * Q.t$.

```
# module Couple = functor ( Q : sig type t end ) ->
  struct type couple = Q.t * Q.t end ;;
```

```
module Couple :
  functor(Q : sig type t end) -> sig type couple = Q.t * Q.t end
```

Podobno kot v primeru funkcij lahko uporabljamo okrajšave za definicijo in imenovanje funktorja.

Sintaksa:

```
module Name1 ( Name2 : signature ) = structure
```

Funktor ima lahko več parametrov.

Sintaksa:

```
functor ( Name1 : signature1 ) ->
  .
  .
  .
  functor ( Namen : signaturen ) ->
    structure
```

Sintaktična okrajšava za definicijo in imenovanje funktorja se lahko razširi na funktorje z večimi argumenti.

Sintaksa:

```
module Name (Name1:signature1) . . . (Namen:signaturen) = structure
```

Aplikacija funktorja na argumente se napiše na sledeč način.

Sintaksa:

```
module Name = functor ( structure1 ) . . . ( structuren )
```

Vsak parameter je potrebno zapisati v svojih oklepajih. Rezultat aplikacije funkcije je lahko bodisi enostaven modul ali delno apliciran funktor, odvisno od števila parametrov funktorja.

Zaprta funktor ne referencira drugih modulov kot tistih, ki so definirani kot parametri. Takšen funktor komunicira z drugimi moduli samo eksplicitno!

Z uporabo zaprtih funktorjev maksimiziramo ponovno uporabnost funktorja–moduli, ki jih funktor uporablja so določeni samo ob kreiranju. Formalno imamo tesno zvezo med zaprtimi funkcijami in zaprtimi funktorji.

8.4.2 Primer funktorja

Funktorji so “funkcije” iz struktur v strukture.

Funktorji se uporabljajo za predstavitev parametrizirane strukture: struktura A je parametrizirana s strukturo B ali preprosto funktor F s formalnim parametrom B (skupaj s signaturo B) ki vrne novo strukturo A.

Funktor F lahko apliciramo na eno ali več implementacij B1 modula B, ki potem vrne ustrezno strukturo A1,...,An.

Naslednji primer prikaže strukturo, ki implementira množice kot sortirane sezname. Struktura je parametrizirana s strukturo ki poda tip elementov množice kot tudi funkcijo urejenosti nad tem tipom:

```
# type comparison = Less | Equal | Greater;;
type comparison = Less | Equal | Greater

# module type ORDERED_TYPE =
  sig
    type t
    val compare: t -> t -> comparison
  end;;
module type ORDERED_TYPE = sig type t val compare : t -> t -> comparison end

# module Set =
  functor (Elt: ORDERED_TYPE) ->
  struct
    type element = Elt.t
    type set = element list
    let empty = []
    let rec add x s =
      match s with
      [] -> [x]
      | hd::tl ->
        match Elt.compare x hd with
        Equal -> s (* x is already in s *)
        | Less -> x :: s (* x is smaller than all elements of s *)
        | Greater -> hd :: add x tl
    let rec member x s =
      match s with
      [] -> false
      | hd::tl ->
        match Elt.compare x hd with
        Equal -> true (* x belongs to s *)
        | Less -> false (* x is smaller than all elements of s *)
        | Greater -> member x tl
  end;;
module Set :
```

```

functor (Elt : ORDERED_TYPE) ->
sig
  type element = Elt.t
  type set = element list
  val empty : 'a list
  val add : Elt.t -> Elt.t list -> Elt.t list
  val member : Elt.t -> Elt.t list -> bool
end

```

Z aplikacijo funktorja Set na strukturo, ki implementira urejen tip, dobimo operacije nad množicami tega tipa:

```

# module OrderedString =
  struct
    type t = string
    let compare x y = if x=y then Equal else if x < y then Less else Greater
  end;;
module OrderedString :
  sig type t = string val compare : 'a -> 'a -> comparison end

# module StringSet = Set(OrderedString);;
module StringSet :
  sig
    type element = OrderedString.t
    type set = element list
    val empty : 'a list
    val add : OrderedString.t -> OrderedString.t list -> OrderedString.t list
    val member : OrderedString.t -> OrderedString.t list -> bool
  end

# StringSet.member "bar" (StringSet.add "foo" StringSet.empty);;
- : bool = false

```

8.4.3 Funktorji in ponovna uporaba kode

Standardna knjižnica Ocaml vsebuje tri module, ki definirajo funktorje. Dva od teh uporabljata kot argument totalno urejen podatkovni tip–modul z naslednjo signaturo.

```

# module type OrderedType =
  sig
    type t
    val compare: t -> t -> int
  end ;;
module type OrderedType = sig type t val compare : t -> t -> int end

```


Funkcija `compare` vzame dva argumenta tipa `t` in vrne negativno število, če je prvi argument manjši od drugega, nič, če sta enaka in pozitivno število, če je prvi argument večji od drugega.

Poglejmo si primer totalno urejenih parov celih števil, ki so opremljeni z leksikografsko urejenostjo.

```
# module OrderedIntPair =
  struct
    type t = int * int
    let compare (x1,x2) (y1,y2) =
      if x1 < y1 then -1
      else if x1 > y1 then 1
      else if x2 < y2 then -1
      else if x2 > y2 then 1
      else 0
  end ;;
module OrderedIntPair :
  sig type t = int * int val compare : 'a * 'b -> 'a * 'b -> int end
```

Funktor `Make` iz modula `Map` vrne modul, ki implementira asociativne tabele katerih ključi so vrednosti urejenega tipa, ki je podan kot parameter. Ta modul omogoča operacije podobne operacijam na asociativnih seznamih, vendar uporablja bolj učinkovito in bolj kompleksno podatkovno strukturo (uravnotežena binarna drevesa).

```
# module AssocIntPair = Map.Make (OrderedIntPair) ;;
module AssocIntPair :
  sig
    type key = OrderedIntPair.t
    and 'a t = 'a Map.Make(OrderedIntPair).t
    val empty : 'a t
    val add : key -> 'a -> 'a t -> 'a t
    val find : key -> 'a t -> 'a
    val remove : key -> 'a t -> 'a t
    val mem : key -> 'a t -> bool
    val iter : (key -> 'a -> unit) -> 'a t -> unit
    val map : ('a -> 'b) -> 'a t -> 'b t
    val fold : (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
  end
```

Funktor `Map.Make` omogoča konstruiranje asociativnih tabel nad ključi danega tipa za katere smo napisali funkcijo za primerjanje. Poglejmo si primer kreiranja asociativnega polja, vstavljanje elementov in iskanje elementa.

```
# open AssocIntPair;;
# let a = add (1,1) "ena,ena" empty;;
val a : string AssocIntPair.t = <abstr>
# let a = add (2,2) "dva,dva" a;;
```

```
val a : string AssocIntPair.t = <abstr>  
# let a = add (3,3) "tri,tri" a;;  
val a : string AssocIntPair.t = <abstr>  
# find (2,2) a;;  
- : string = "dva,dva"
```

Literatura

- [1] Emmanuel Chailloux, Pascal Manoury, Bruno Pagano, Developing Applications With Objective Caml, O'REILLY & Associates, 2000, France.
- [2] Henk Barendregt, Erik Barendsen, Introduction to Lambda Calculus, Revised edition, March 2000.
- [3] Robert Harper. Programming in Standard ML. Draft, 2005.
- [4] Robert Harper, Practical Foundations for Programming Languages, Draft, CMU, 2010.
- [5] Jeff Kramer, Is Abstraction The Key To Computing?, Communications of the ACM, Vol.50, No.4, April 2007.
- [6] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy and Jérôme Vouillon, The Objective Caml system release 3.10, Documentation and user's manual, INRIA, 2007.
- [7] John C. Mitchell, Concepts in programming languages, Cambridge University Press, 2003.
- [8] Benjamin Pierce, Foundation Calculi for Programming Languages, Handbook of Computer Science, 1995.
- [9] Benjamin Pierce, Types and Programming languages, MIT Press, 2002.
- [10] Tennet, The Denotational Semantics of Programming Languages, CACM, vol. 19, no. 8, 1976.
- [11] Adam B. Webber, Modern Programming Languages: A Practical Introduction, Franklin, Beedle & Associates, Inc., 2003.

Dodatek A

LISP

Lisp (1960)

Pogled v zgodovino

J. McCarthy

Nekatere stare ideje se zdijo stare

Nekatere stare ideje se zdijo nove

Primer elegantnega, minimalističnega jezika

Ni C, C++, Java!

Priložnost za drugačno razmišljanje

Lisp JE! razširjen λ -račun Genialno enostavna realizacija λ -računa

Osnova večini modernih programskih jezikov

Prikazane bodo splošne teme in zasnova jezika

Gradimo od korenin navzgor!

Lisp

Obstaja zelo veliko dialektov

Lisp 1.5, CommonLisp, ..., Scheme, ...

CommonLisp ima zelo veliko dodatnih gradnikov

Predstavljeni bodo gradniki Scheme in Common Lisp

A.1 Izrazi, atomi in sezname

Enostavni izrazi:

> (+ 4 9)

13

> (- 5 7)

```
-2
> (* 3 9)
27
> (/ 15.0 2)
7.5
```

Atomi in pari

Atomi jezika so ključne besede in števila

Osnovni podatkovni tipi

```
<atom> ::= <smb1> | <number>
<smb1> ::= <char> | <smb1><char> | <smb1><digit>
<num> ::= <digit> | <num><digit>
```

Pari (cons) s piko:

- (A.B) je par ali cons.
- Simbolični izraz = s-izrazi (sexp):

```
<sexp> ::= <atom> | (<sexp> . <sexp>)
```

Primeri s-izrazov

- Notacija s pari: (b . (c . (d . nil)))
- Okrajšava: (a b c d)

Atomi in sezname so edini tipi začetnega Lispa

Atomi in sezname

Atomi so besede in števila.

```
1
25
342
mouse
factorial
x
```

Sezname so sekvence atomov in seznamov ločenih s presledki in oklepaji.

```
()
(())
((( )))
((a b c))
((1 2) 3 4)
(mouse (monitor 512 342) (keyboard US))
(defun factorial (x) (if (eql x 0) 1 (* x (factorial (- x 1)))))
```

Funkcije na atomih in parih

- Imamo par (A . B)
- Operacija car
- Vrne podatkovni del A (cons)

Primer:

```
> (cons 1 nil)
(1)
> (cons A B)
(A B)
```

Operacija cdr: Vrne preostali del seznama B

Primer:

```
> (cons 3 (cons 2 (cons 1 ())))
(3 2 1)
> (car (3 2 1))
3
> (cdr (3 2 1))
(2 1)
```

Funkcija quote

- Radi bi predstavili literale.
- Sintaksa: (quote datum)
- Isti pomen: '

Primer:

```
> (quote a)
a
> 'a
a
> (quote (2 3 5 7 11 13 17 19))
(2 3 5 7 11 13 17 19)
```

A.2 Predefinirane funkcije

A.2.1 Funkcija define

- Povezovanje imena z računskim objektom.
- Sintaksa: (define var expr) (define a 2) (define b (+ 25 (* 3 25)))
- Definicija funkcije -č kasneje podrobno.
- Sintaksa: (define (name formal_parameters) body) (define (square x) (* x x))

Enakost: Funkcija: =

```
> (= 2 (* 2 2))
#f
> (= 2 (* 1 2))
#t
```

Funkcija: eq?

```
> (eq? 'foo 'foo)
#t
> (eq? 'foo 'fooo)
#f
> (define foo 'foo)
> (eq? foo 'foo)
#t
```

Funkcija: eqv?

```
> (eqv? (cons 'a 'b) (cons 'a 'b))
#t
> (eqv? (* 2 2) (sqrt 16))
#f
> (eqv? 'foo 'foo)
#t
> (eqv? "(substring \"foobar\"0 0))
#f
```

Funkcija: equal?

```
> (equal? (cons 'a 'b) (cons 'a 'b))
#t
> (equal? 'foo 'foo)
#t
> (equal? "(substring \"foobar\"0 0))
#t
```


A.2.2 Funkcija let

- Definira množico spremenljivk v seznamu
- Izračuna izraz na osnovi spremenljivk

Sintaksa:

```
(let ((var1 exp1)
      (var2 exp2)
      ...
      (varn expn))
  body)
```

Primeri:

```
> let ((a 3)
      (b 4)
      (c 5))
      (* (+ a b) c))
35
> a
Error: Unbound variable

>(define x 10)
>(+ (let ((x 5))
      (* x (+ x 2))) x)
45
```

Lokalnost!

A.3 Pogojne funkcije

Pogojne funkcije

- if
- cond
- case

A.3.1 Funkcija if

Sintaksa:

```
(if condition consequent alternative)
(if condition consequent)
```

Primer:

```
(define minsquare
  (lambda (a b)
    (if (< a b)
        (square a)
        (square b))))
```

A.3.2 Funkcija cond

Sintaksa:

```
(cond (condition1 consequent1)
      (condition2 consequent2)
      . . .
      (else alternative))

((or (and (<= x z) (<= z y))
     (and (<= z x) (<= x y)))
 (+ x z))
(else (+ y z))))
```

A.3.3 Funkcija case

Sintaksa:

```
(case key clause1 clause2 ...)
```

Primer:

```
> case (+ 3 4)
  ((7) 'seven)
  ((2) 'two)
  (else 'nothing))
```

```
seven
> (case 'a
    ((a b c d) 'first)
    ((e f g h) 'second)
    (else 'rest))
first
```

A.3.4 Funkcije and, or in not

```
(and expr1 expr2 ... exprn)
(or expr1 expr2 ... exprn)
(not expr)
```

Primer:

```
>(define a 3)
>(or (< 2 5) (number? a))
#t
```

A.4 Funkcije

Oblika

```
(lambda ( parameters ) ( function_body ) )
```

Sintaksa izhaja iz lambda računa

$$\lambda x.\lambda y.x + y \tag{A.1}$$

Definicija funkcije brez imena:

```
(lambda (x y) (+ x y))
```

Funkcija z imenom:

```
(define sum (lambda (x y) (+ x y)))
```

A.4.1 Append

```
(define append
  (lambda (ls1 ls2)
    (if (null? ls1)
        ls2
        (cons (car ls1) (append (cdr ls1) ls2)))))
> (trace append) (append)
> (append '(a b c) '(d e f))
| (append (a b c) (d e f))
| (append (b c) (d e f))
| | (append (c) (d e f))
| | (append () (d e f))
| | (d e f)
| | (c d e f)
| (b c d e f)
| (a b c d e f)
(a b c d e f)
```

Primer (Common Lisp):

```
> (defun secret-number (the-number)
  (let ((the-secret 37))
    (cond ((= the-number the-secret) 'that-is-the-secret-num)
          ((< the-number the-secret) 'too-low)
          ((> the-number the-secret) 'too-high))))
SECRET-NUMBER
> (secret-number 11)
TOO-LOW
> (secret-number 99)
TOO-HIGH
> (secret-number 37)
THAT-IS-THE-SECRET-NUMBER
```

A.4.2 Rekurzivne funkcije

Radi bi izrazili funkciju f

$$(f\ x) = (\text{cond } ((\text{eq } x\ 0)\ 0) (\text{true } (+\ x\ (f\ (-\ x\ 1))))))$$

Poskus:

$$(\text{lambda } (x) (\text{cond } ((\text{eq } x\ 0)\ 0) (\text{true } (+\ x\ (f\ (-\ x\ 1))))))$$

Toda f v telesu funkcije ni definirana

McCarthy 1960: uporabi operator "label"

```
(label f
  (lambda (x) (cond ((eq x 0) 0) (true (+ x (f (- x 1 ))))))))
```

Primer: Povežemo lambda funkcijo z imenom

```
(define f
  (lambda (x)
    (cond ((eq x 0) 0)
          (true (+ x (f (- x 1 ))))))))
```

Fibonacci:

```
(define fib
  (lambda (n)
    (cond ((= n 0) 0)
          ((= n 1) 1)
          (else (+ (fib (- n 1))
                   (fib (- n 2)))))))
```

Fakulteta (Common Lisp):

```
(defun factorial(x)
  (if (eql x 0) 1
      (* x (factorial(- x 1)))))
```

Primer:

```
>(trace power)
POWER
```

Common Lisp

```
(defun power (x y)
  (cond ((= y 0) 1)
        (else (* x (power x (1- y))))))
```

```
>(power 3 4)
1> (POWER 3 4)
2> (POWER 3 3)
3> (POWER 3 2)
4> (POWER 3 1)
5> (POWER 3 0)
<5 (POWER 1)
```

```
<4 (POWER 3)
<3 (POWER 9)
<2 (POWER 27)
<1 (POWER 81)
```

Imenovani let

- Omogoča bolj splošen iteracijski konstrukt.
- Zanimiv gradnik, ki daje iluzijo iteracije.

```
(define factorial (lambda (n)
  (let iter ((product 1)
            (counter 1))
    (if (> counter n)
        product
        (iter (* counter product) (+ counter 1))))))
```

A.5 Funkcije višjega reda

Funkcije, ki bodisi

- Vzamejo funkcije kot argument
- Vrnejo funkcije kot rezultat

Primer: kompozicija funkcij

```
(define compose
  (lambda (f g)
    (lambda (x) (f (g x)))))
```

Primer: maplist

```
(define maplist (lambda (f x)
  (cond ((null x) ())
        (true (cons (f(car x)) (maplist f (cdr x) )
                    )))))
```

Primer:

```
(define twice
  (lambda (f) (lambda (x) (f (f x)))))
```

```
(define inc (lambda (x) (+ 1 x)))
> ((twice inc) 2)
4
```

Primer:

```
(define inc (lambda (x) (+ x 1)))
> (maplist inc '(1 2 3))
(2 3 4)
```

A.6 Implementacije

A.6.1 Stranski učinki

čisti Lisp: ni stranskih učinkov

Dodatne operacije so dodane za učinkovitost

```
(rplaca x y)  zamenjaj car x z y
(rplacd x y)  zamenjaj cdr x z y
```

Kaj pomeni učinkovitost?

- Je (rplaca x y) hitrejši od (cons y (cdr x)) ?
- Hitro je boljše?

Primer:

```
(define p '(A B))
(rplaca p 'C)
(rplacd p 'D)
p
-> (C . D)
```

Scheme:

```
(define rplaca set-car!)
(define rplacd set-cdr!)
```

A.6.2 Evaluacija izraza

Zanka: beri-evaluiraj-izpiši

Klic funkcije (funkcija arg1 ... argn)

- Evaluiraj vse argumente
- Podaj vrednosti argumentov funkciji

Posebne oblike stavkov ne ovrednotijo vseh argumentov

- Primer (cond (p1 e1) ... (pn en))
- Od leve proti desni
- Poišči prvi pi z vrednostjo true, evaluiraj ei
- Primer (quote A) ne ovrednoti A

A.6.3 Programi kot podatki

Programi in podatki imajo isto predstavitev

Funkcija eval se uporablja za evaluacijo seznama

Primer: zamenjaj x za y v z in evaluiraj

```
(define substitute (lambda (x y z)
  (cond ((null z) z)
        ((atom z) (cond ((eq z y) x) (else z)))
        (else (cons (substitute x y (car z))
                      (substitute x y (cdr z))))))

(define substitute-and-eval
  (lambda (x y z) (eval (substitute x y z))))

(substitute-and-eval '3 'x '(+ x 1))
```