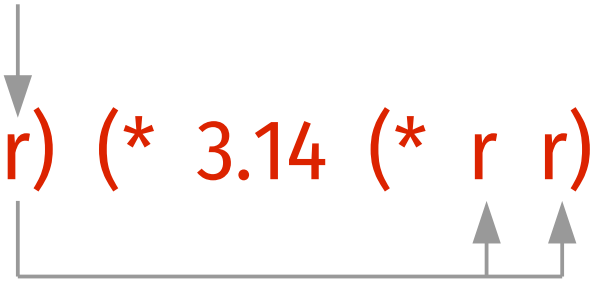
# Definice funkce

- **(define (*jméno-funkce argumenty*) *výraz*)**
- funkce má dané *jméno*
- identifikátory tvořící *argumenty* jsou při volání nahrazeny hodnotami získanými vyhodnocením skutečných argumentů funkce
- o přiřazení hodnot argumentům rozhoduje pořadí
- následně se vyhodnotí *výraz* a výsledná hodnota je výsledkem volání funkce

# Příklad funkce

- definujme funkci plocha-kruhu; argumentem je poloměr r, počítá Pi * $r^2$
- (define (**plocha-kruhu** r) (* 3.14 (* r r)))
- nyní lze použít např.

(plocha-kruhu 5)



(define (plocha-kruhu r) (* 3.14 (* r r)))

vypočte se (* 3.14 (* 5 5))

# Pokračování – prstenec

- definovanou funkci lze použít v definici dalších funkcí
- například plocha prstence:

```
(define (plocha-prstence vnejsi vnitrni)
   (– (plocha-kruhu vnejsi) (plocha-kruhu vnitrni)))
```

- 
```
(plocha-prstence 8 5)
(– (plocha-kruhu 8) (plocha-kruhu 5))
(– (* 3.14 (* 8 8)) (* 3.14 (* 5 5)))
```

# Komentáře

- řádek začínající středníkem je komentářem
- doporučuje se popisovat v komentářích syntax a význam definovaných funkcí
  - první řádek – **kontrakt**: jméno, typy argumentů a výsledku
  - další řádky: slovní popis funkce
- ;;plocha-prstence: číslo číslo -> číslo
  ;;vypočte plochu prstence s poloměrem 'vnejsi'
  ;;a poloměrem otvoru 'vnitrni'
  (define (plocha-prstence vnejsi vnitrni) ... )

# Program ve Scheme

- sada spolupracujících funkcí
- nedělejte velké kroky (dlouhé a složité funkce)
- vhodná je dekompozice – každá funkce vyřeší logicky uzavřenou část problému
- k vyřešení dílčích podproblémů volá další funkce (rozděl a panuj)
- snazší, srozumitelnější a lépe laditelné

# Kvadratická rovnice (1)

```
;kvadrovnice: číslo číslo číslo -> seznam čísel
;vypočte kořeny kvadratické rovnice
;ax^2 + bx + c = 0
; vrací seznam kořenů
(define (kvadrovnice a b c)
   (list (koren1 a b c)
         (koren2 a b c)))
```

# Kvadratická rovnice (2)

```
;koren1,2: číslo číslo číslo -> číslo
;vypočte kořen kvadratické rovnice podle vzorce
;(–b + sqrt(D)) / 2a  a  (–b – sqrt(D)) / 2a
(define (koren1 a b c)
   (/ (+ (– b) (sqrt (diskrim a b c))) (* 2 a)))
(define (koren2 a b c)
   (/ (– (– b) (sqrt (diskrim a b c))) (* 2 a)))

;diskrim: číslo číslo číslo -> číslo
;vypočte diskriminant kvadratické rovnice: b^2 – 4ac
(define (diskrim a b c)
   (– (* b b) (* 4 a c)))
```

# Srozumitelnost programů

- komentáře a volné místo činí program snadněji srozumitelným
- je záhodno používat mnemotechnické identifikátory
- strukturu výrazu lze zvýraznit odsazením, které vyznačí úrovně vnoření jednotlivých konstrukcí

```
(define (koren1 a b c)
   (/ (+ (– b)
         (sqrt (diskrim a b c)))
      (* 2 a)))
```

# Seznamy

- seznamy mají velmi důležitou úlohu – základní datová struktura
- dvě základní procedury:
  **(car *seznam*)** vydá první prvek seznamu
  **(cdr *seznam*)** vydá zbytek seznamu (bez 1. prvku)
- též **first** a **rest**
- (car '(1 2 3)) ... 1
  (cdr '(1 2 3)) ... (2 3)
  (car (cdr '(1 2 3))) ... 2

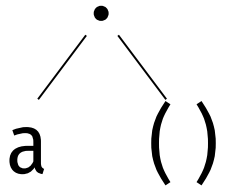
# Vytvoření seznamu (1)

- **cons** spojí své dva argumenty do seznamu; druhým argumentem by měl být seznam
- (cons 1 '(2 3)) ... (1 2 3)
  (cons 1 '()) ... (1)
- interně je seznam tvořen páry; pár má dvě položky zvané car a cdr; zapisován jako (car . cdr)
- **seznam** je buď prázdný seznam nebo pár obsahující první prvek (car) a seznam-zbytek (cdr)

# Seznamy a páry

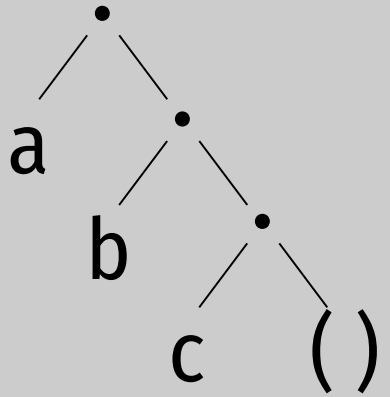
(a)

(a . ())



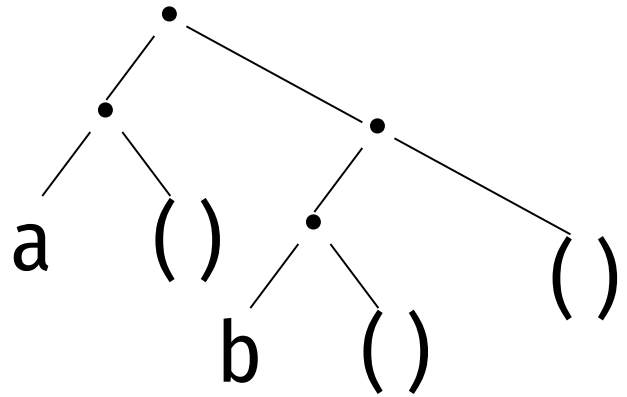

(a b c)

(a . (b . (c . ())))




((a) (b))

((a . ()) . ((b . ()) . ()))



# Pravidla pro seznamy

- všechny seznamy mají konečnou délku a jsou ukončeny prázdným seznamem
- cons ve skutečnosti vytvoří pár
- nesprávný seznam (improper list) je pár, který nekončí prázdným seznamem, nejedná se o seznam – např. (a . b) nebo (a  b  c . d)

# Vytvoření seznamu (2)

- **(list *arg1 arg2 ... argN*)**
  vyhodnotí své argumenty a vytvoří seznam obsahující výsledky
- (list (* 5 4) "ok" (+ (/ 4 2) (* 3 5)))
  vytvoří seznam (20 "ok" 17)
- list je vhodný, pokud známe všechny prvky seznamu
- cons se často používá při rekurzi: „spoj tuto hodnotu a seznam vytvořený rekurzivním voláním“

# Další funkce pro seznamy

- (list? *x*) — je x seznam?
- (pair? *x*) — je x pár?
- (null? *x*) — je x prázdný seznam?
  (empty? x)
- (list *operandy*) — vyhodnotí operandy a vytvoří z nich seznam
- (length *seznam*) — počet prvků seznamu
- (append *seznamy*) — spojí seznamy do jednoho
- (reverse *seznam*) — obrátí pořadí prvků v seznamu

# Konstanty

- definují se pomocí define, ovšem v jednodušším tvaru:
  **(define *jméno výraz*)**
- příklad:

```
(define Pi 3.1415926)
(define (plocha-kruhu r)
    (* Pi r r))
```

- pojmenování hodnot přináší lepší srozumitelnost a snadnější modifikovatelnost

# Pravdivostní výrazy

- atomické výrazy: predikáty, konstanty #t a #f
- Booleovské funkce:
  - **and** (and (< 0 x) (< x 10))
  - **or** (or (integer? x) (rational? x))
  - **not** (not (list? x))
- příklad: y leží mezi min a max

```
(define (mezi? min y max)
   (and (<= min y)
        (<= y max)))
```

# Porovnání různých typů

- **čísla**: =, <, > a spol.
- **symboly** ('kolo): symbol=?
  - jen rovnost, větší/menší nemá smysl
  - (symbol=? 'kolo 'kolo) ... #t
- **řetězce**: string=?
  - lze i lexikografické porovnání string<?, string>=?,...

# Obecné porovnání

- **equal?** – jsou argumenty stejné?
  - rekurzivně prochází strukturu – lze i seznamy apod.
  - (define test (list 1 2 (+ 2 1)))
    (equal? '(1 2 3) test) ... #t
- **eq?** – odkazují oba argumenty na totéž?
  - (eq? '(1 2 3) test) ... #f
  - výsledek závisí na implementaci – raději nepoužívat
- **eqv?** – jako eq?, u čísel a znaků porovná hodnoty

# Podmíněný výraz (1)

- syntaxe:

```
(cond
    (podmínka1  výraz1)
    ...
    (podmínkaN  výrazN)
    (else  výraz)
)
```

- podmínkami jsou pravdivostní výrazy

# Podmíněný výraz (2)

- *podmínky* se vyhodnocují postupně podle pořadí ve zdrojovém kódu
- **první úspěšná se použije** – Scheme vyhodnotí příslušný *výraz* a výsledek vyhodnocení se stane výsledkem celého podmíněného výrazu
- else je nepovinné, totéž co #t
- k ohraničení jednotlivých větví je zvykem místo () používat []

# Příklad – sign

```
;sign: číslo -> číslo
;pro kladné hodnoty vrací +1, pro záporné –1,
;pro 0 vrací 0
(define (sign n)
   (cond
      [(< 0 n) 1]
      [(> 0 n) –1]
      [else 0]
   )
)
```

# Nebezpečí podmínek

- pokud se podmínky překrývají, závisí na pořadí
- např. ve výrazu

```
(cond
    [(< x 100) 100]
    [(< x 50) 50]
)
```

nebude druhá větev nikdy použita – jakákoli hodnota menší než 50 je také menší než 100 a bude použita první větev

# Možná řešení

- **učinit podmínky disjunktními**

```
(cond
   [(and (<= 50 x) (< x 100)) 100]
   [(< x 50) 50]
)
```

- **uspořádat podmínky (konkrétnější dopředu)**

```
(cond
   [(< x 50) 50]
   [(< x 100) 100]
)
```

# if

- zjednodušená forma podmíněného výrazu
- **(if *podmínka výraz1 výraz2*)**
- obvyklý význam – platí-li *podmínka*, vyhodnotí se *výraz1*, jinak *výraz2*
- příklad: absolutní hodnota

```
(define (abs x)
   (if (< x 0)(– x)
               x))
```

# Kvadratická rovnice lépe

```
;kvadrovnice: číslo číslo číslo -> seznam čísel
;vypočte kořeny kvadratické rovnice
;ax^2 + bx + c = 0
; vrací seznam kořenů
(define (kvadrovnice a b c)
   (if (= 0 (diskrim a b c))
      (list (koren1 a b c))
      (list (koren1 a b c) (koren2 a b c))
))
```

# Podmínky a chyby

- podmínkami lze kontrolovat přípustnost dat
- nevyhovující data způsobí chybové hlášení
- **(error '*jméno-funce* "*chybové hlášení*")**
  ukončí vyhodnocování a zobrazí chybu
- (define (**prumer** L)
  (if (empty? L)
  (error 'prumer "seznam nesmí být prázdný")
  ( ... )
  ))

# Zpracování seznamů

- máme k dispozici přístup jen k prvnímu prvku a (celému) zbytku seznamu
- funkce musí zpracovat první prvek a rekurzivně zavolat sama sebe ke zpracování zbytku
- např. hledání hodnoty v seznamu
  - **není** obsažena v prázdném seznamu
  - **je** obsažena, pokud je totožná s prvním prvkem seznamu **nebo** je obsažena v jeho zbytku (rekurzivní krok)

# Hledání hodnoty

```
(define (hledej? hodnota L)
   (cond
      [(empty? L) #f]
      [(= hodnota (car L)) #t]
      [else (hledej? hodnota (cdr L))]
   ))
```

- rekurzivní funkce
- klasické funkcionální programování nepoužívá cykly

# Návrh rekurzivní funkce

- **nejprve testovat cílové situace** (prázdný seznam, hodnota nalezena, úkol splněn,...), pro ně je řešení triviální
- po nich sestavit výraz určující výsledek
- jeho součástí jsou podvýrazy obsahující volání sama sebe pro **zjednodušenou** úlohu (typicky zbytek seznamu)

# Faktoriál

- pro n<=1 je faktoriál 1
- jinak rekurzivně zavoláme pro n–1 a výsledek znásobíme n
- 
```
(define (! n)
  (cond
    [(<= n 1) 1]
    [else (* n (! (– n 1)))]
  ))
```

# Délka seznamu

- prázdný seznam má délku 0
- jinak je délka seznamu = 1 + délka zbytku
- 
```
(define (length L)
  (cond
    [(empty? L) 0]
    [else (+ 1 (length (cdr L)))]
  ))
```

vytvořeno s podporou
projektu ESF