KATEDRA INFORMATIKY, PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO, OLOMOUC

PARADIGMATA PROGRAMOVÁNÍ 2A
MAKRA I

Slajdy vytvořili Vilém Vychodil a Jan Konečný

Opakování (kvazikvotování)

```
`(1 2 3 4 5)              ⟹  (1 2 3 4 5)
`(1 (+ 2 3) 4 5)          ⟹  (1 (+ 2 3) 4 5)
`(1 ,(+ 2 3) 4 5)         ⟹  (1 5 4 5)
`(1 ((,(+ 2 3))) 4 5)     ⟹  (1 ((5)) 4 5)

(define s '(a b c))
`(1 ,s 2)                 ⟹  (1 (a b c) 2)
`(1 ,@s 2)                ⟹  (1 a b c 2)

`(1 '2 3)   ⟹  (1 (quote 2) 3)
`'(1 2 3)   ⟹  (quote (1 2 3))
'`(1 2 3)   ⟹  (quasiquote (1 2 3))

`(1 `(2 3))               ⟹  (1 (quasiquote (2 3)))
`(1 `(,(+ 1 2) 3))        ⟹  (1 (quasiquote ((unquote (+ 1 2)) 3)))
`(1 ,`(,(+ 1 2) 3))       ⟹  (1 (3 3))
```

Problém: chceme upravit `if` tak,
aby při absenci alternativního výrazu vracel `#f`

nyní máme:

```
(if (= 1 2) 'blah)
```
$\Longmapsto$ nedefinovaná hodnota

chceme:

```
(new-if (= 1 2) 'blah)
```
$\Longmapsto$ `#f`

nabízí se vyřešit pomocí nové procedury:

```
(define new-if
  (lambda (elem1 elem2)
    (if elem1 elem2 #f)))
```

při volání `new-if` je vždy vyhodnocen i druhý argument:

```
(new-if #f blah-blah)
```
$\Longmapsto$ `Error`

Potřebujeme: během vyhodnocování každý výraz tvaru
`(new-if expr1 expr2)`

nahradit výrazem:
`(if expr1 expr2 #f)`
bez toho aniž by se vyhodnocovaly `expr1` a `expr2`

Jinými slovy:

- potřebujeme zavést předpis, který bude provádět „transformaci kódu“
- **transformace** . . . konkrétní část kódu je nahrazena jinou
- po transformaci proběhne **vyhodnocení transformovaného kódu**

```
(define x 11)
(new-if (even? x) (+ x 1))
```
⇓ transformace
```
(if (even? x) (+ x 1) #f)
```
⇓ vyhodnocení
```
#f
```

Jak se dívat na transformaci?

- můžeme si ji představit jako (klasickou) proceduru,
  které jsou předány argumenty v nevyhodnocené podobě
- transformace se v některých jazycích nazývá **makroexpanze**

```
;; transformační procedura pro new-if
(define new-if-trans
  (lambda (test expr . alt)
    (list 'if test expr
          (if (null? alt)
              #f
              (car alt)))))
```

```
(new-if-trans 'e1 'e2 'e3)  ⟹  (if e1 e2 e3)
(new-if-trans 'e1 'e2)      ⟹  (if e1 e2 #f)
(new-if-trans '(even? x) '(+ x 1))
  ⟹  (if (even? x) (+ x 1) #f)
```

kratší řešení pomocí kvazikvotování

```
(define new-if-trans
  (lambda (test expr . alt)
    `(if ,test
         ,expr
         (begin #f ,@alt))))
```

příklady transformace:

```
(new-if-trans 'expr1 'expr2 'expr3)
```
$\longmapsto$ *(if expr1 expr2 (begin #f expr3))*

```
(new-if-trans 'expr1 'expr2)
```
$\longmapsto$ *(if expr1 expr2 (begin #f))*

```
(new-if-trans '(even? x) '(+ x 1))
```
$\longmapsto$ *(if (even? x) (+ x 1) (begin #f))*

nouzové řešené `new-if`, které se již chová jak má
manuální spuštění transformační procedury
následné vyhodnocení transformovaného výrazu

```
(eval (new-if-trans '(even? x) '(+ x 1)))
```

Výhody řešení:

- pokud je první výraz nepravdivý, alternativní výraz není vyhodnocen
- touto konstrukcí (new-if) lze zastavit rekurzi

Nevýhody řešení:

- všechny předávané argumenty musíme explicitně kvotovat
- transformovaný výraz musíme ručně vyhodnotit pomocí `eval`
- `eval` ve většině interpretů pracuje jen v globálním prostředí
- volání je nepřehledné

problém s lexikálními vazbami

```
(let ((x 10))
  (eval (new-if-transformer '(even? x) '(+ x 1))))
```

$\Longmapsto$ `error: x not bound`

částečné řešení: použití (`the-environment`)
ve většině interpretů nebude fungovat

```
(let ((y 10))
  (eval (new-if-transformer '(even? y) '(+ y 1))
        (the-environment)))
```

navíc jsme se nezbavili nepřehledného kódu

Řešení problému: zavedení **maker**

MAKRA – dva základní pohledy na makra

1. **POHLED:** Makra jsou „rozšířením syntaxe jazyka“
   - **makro** = dáno definicí svého transformačního předpisu
   - po načtení výrazu (READ) je v něm provedena makroexpanze
     tuto fázi provádí tzv. *preprocesor*
   - až po dokončení expanze všech maker nastává vyhodnocování výrazu
   - nemá smysl uvažovat pojem „aplikace makra“
   - takto na makra pohlíží většina PJ: C, DrScheme, Common LISP,...

Výhody přístupu:

- preprocesor a vlastní `eval` jsou zcela nezávislé
- preprocesor může být aktivován okamžitě po načtení výrazu
- umožňuje snadnou kompilaci kódu
  (v kompilovaném kódu již pochopitelně „žádná makra nejsou“)

Nevýhody přístupu:

- makra jsou „mimo jazyk“ (často se zapisují odlišně, třeba v C)
- makra nejsou elementy prvního řádu

MAKRA – dva základní pohledy na makra

2. POHLED: Makra jsou „speciální elementy jazyka“

- **makro** = element jazyka obsahující ukazatel na transf. proceduru
- **transformační procedura** . . . klasická procedura
- je potřeba rozšířit `eval`:
  případ, kdy se první prvek seznamu vyhodnotí na makro
- makra jsou „uživatelsky definované speciální formy“
- takto na makra budeme pohlížet my (dále třeba PJ: M4, TEX)

Výhody přístupu:

- makra jsou elementy prvního řádu
- s makry lze pracovat „jako s daty“,
  mohou dynamicky vznikat/zanikat za běhu programu
- můžeme uvažovat koncept „anonymního makra“

Nevýhody přístupu:

- k makroexpanzi dochází až při činnosti `eval`
- prakticky znemožňuje účinnou kompilaci kódu
- při neuváženém používání maker komplikuje ladění programu

Motivační příklady definice maker

```
(define-macro new-if
  (lambda (<test> <expr> . <alt>)
    (list 'if <test> <expr>
          (if (null? <alt>)
              #f
              (car <alt>)))))

;; new-if: pomocí kvazikvotování
(define-macro new-if
  (lambda (<test> <expr> . <alt>)
    `(if ,<test>
         ,<expr>
         (begin #f ,@<alt>))))
```

Příklad použití makra

```
;; new-if: pomocí kvazikvotování
(define-macro new-if
  (lambda (<test> <expr> . <alt>)
    `(if ,<test>
         ,<expr>
         (begin #f ,@<alt>))))

(let ((x 10))
  (new-if (even? x) (+ x 1)))
```

⇓ aktivace transformační procedury makra

```
(if (even? x) (+ x 1) (begin #f))
```

⇓ vyhodnocení výrazu v prostředí, kde x má vazbu 10

```
11
```

Rozšíření EVAL

$\mathrm{Eval}[E, \mathcal{P}]$:

(A) Pokud je E `cislo`, ... jako obvykle

(B) Pokud je E `symbol`, ... jako obvykle

(C) Pokud je E `seznam` tvaru `(`E_1␣E_2␣`·`␣E_n`)`, pak nejprve provedeme vyhodnocení prvního prvku E_1 v prostředí $\mathcal{P}$ a výslednou hodnotu označíme F_1, to jest $F_1 := \mathrm{Eval}[E_1, \mathcal{P}]$. Mohou nastat čtyři situace:

(C.1) Pokud F_1 je **procedura**, ... jako obvykle

(C.2) Pokud F_1 je **speciální forma**, ... jako obvykle

(C.3) Pokud F_1 je **makro** jehož transformační procedura je T, pak

1. $F' := \mathrm{Apply}[T, E_2, \ldots, E_n]$
   (F' je výsledkem aplikace transf. procedury na nevyhodnocené arg.)
2. Výsledek vyhodnocení F elementu E v prostředí $\mathcal{P}$ je definován $F := \mathrm{Eval}[F', \mathcal{P}]$
   (F je výsledek vyhodnocení elementu F' v prostředí $\mathcal{P}$).

(C.e) Pokud F_1 není **procedura, speciální forma,** ani **makro**, pak vyhodnocení končí chybou „CHYBA: První prvek seznamu ...“.

Ladění maker: základní princip

- potlačíme vyhodnocení transformovaného kódu
- využívá dodatečné KVOTOVÁNÍ

```
(define-macro new-if
  (lambda (test expr . alt)
    (list 'quote
          (list 'if test expr
                (if (null? alt) #f (car alt))))))

(new-if #f blah-blah)  ⟹  (if #f blah-blah #f)

(define-macro new-if
  (lambda (test expr . alt)
    `'(if ,test ,expr (begin #f ,@alt))))

(new-if #f blah-blah)  ⟹  (if #f blah-blah (begin #f))
```

chceme vytvořit `and2` dvou argumentů vracející `#t` nebo `#f`
chceme vytvořit pouze s pomocí `if`

```
;; nedostačující řešení pomocí procedury:
(define and2
  (lambda (elem1 elem2)
    (if elem1
        (if elem2
            #t
            #f)
        #f)))
```

předchozí má vážný nedostatek:
`(and2 #f blah-blah)` $\Longmapsto$ `error` (chceme `#f`)

`and2` se dvěma argumenty vracející konjunkci

potřebujeme: během vyhodnocování každý výraz
`(and2 expr1 expr2)`
nahradit výrazem
`(if expr1 (if expr2 #t #f) #f)`
bez toho aniž by se vyhodnocovaly `expr1` a `expr2`

Řešení pomocí makra:

```
(define-macro and2
  (lambda (expr1 expr2)
    `(if ,expr1
         (if ,expr2
             #t
             #f)
         #f)))
```

Ukázky použití and2

```
(and2 1 (+ 1 2))
```

⇓

```
(if 1 (if (+ 1 2) #t #f) #f)
```

⇓

```
#t
```

```
(and2 #f blah-blah)
```

⇓

```
(if #f (if blah-blah #t #f) #f)
```

⇓

```
#f
```

```
(and2 #t #f)
```

⇓

```
(if #t (if #f #t #f) #f)
```

⟼ #f

Anaforický `if`: `if*`

- `if*`, který pracuje stejně jako `if`, ale umožňuje v druhém a třetím výrazu používat symbol `$result`, který bude vždy navázaný na výsledek vyhodnocení prvního výrazu

praktické rozšíření, místo:

```
(if (member 'b '(a b c d))
    (list 'nalezen (member 'b '(a b c d)))
    'blah)
```

stačí napsat:

```
(if* (member 'b '(a b c d))
     (list 'nalezen $result)
     'blah)
```

$\Longmapsto$ *(nalezen (b c d))*

Řešení: během vyhodnocování každý výraz

```
(if* expr1 expr2 expr3)
```

potřebujeme nahradit výrazem:

```
(let (($result expr1))
  (if $result expr2 expr3))
```

a to opět bez vyhodnocování `expr1` až `expr3`

`if*` jako makro

```
(define-macro if*
  (lambda (test expr . alt)
    `(let (($result ,test))
       (if $result
           ,expr
           ,@alt))))
```

Ukázky použití `if*`

bez `$result` se chová jako normální `if`

```
(if* 1 2 3)
```

⇓

```
(let (($result 1)) (if $result 2 3))
```

⇓

```
2
```

příklad použití `$result`

```
(if* 1 $result 3)
```

⇓

```
(let (($result 1)) (if $result $result 3))
```

⇓

```
1
```

Složitější ukázka použití `if*`

```
(if* (member 'b '(a b c d))
     (list 'nalezen $result)
     'blah)
```

⇓

```
(let (($result (member (quote b)
                       (quote (a b c d)))))
  (if $result
      (list (quote nalezen) $result)
      (quote blah)))
```

⇓

```
(nalezen (b c d))
```

- Všimněte si: v expandovaném výrazu nejsou žádné ‘ ’,

if pomocí cond

```
(define-macro if
  (lambda (test expr alt)
    `(cond (,test ,expr)
          (else ,alt))))
```

if pomocí cond (bez nutnosti mít alternativní větev)

```
(define-macro if
  (lambda (test expr . alt)
    `(cond (,test ,expr)
          (else (begin #f ,@alt)))))
```

podobné jako předchozí, ale vracíme nedefinovanou hodnotu

```
(define-macro if
  (lambda (test expr . alt)
    `(cond (,test ,expr)
          (else (begin (cond) ,@alt)))))
```

`cond` pomocí `if`
musíme přepsat jeden `cond`-výraz pomocí několika `if`ů

```
;; základní cond pomoci if (pomoci rekurzivního vnoření)
(define-macro cond
  (lambda clist
    (let dive-ifs ((clist clist))
      (if (null? clist)
          `(if #f #f)
          (if (equal? (caar clist) 'else)
              (cadar clist)
              `(if ,(caar clist)
                   ,(cadar clist)
                   ,(dive-ifs (cdr clist))))))))
```

Příklad použití

```
(cond ((= x 3) 'blah)
      ((> x 10) (+ 1 x))
      ((prop? x y) (list x y))
      (else (f 20)))
```

⇓

```
(if (= x 3)
    (quote blah)
    (if (> x 10)
        (+ 1 x)
        (if (prop? x y)
            (list x y)
            (f 20))))
```

⇓

...

`cond` pomocí `if`
musíme přepsat jeden `cond`-výraz pomocí několika `if`ů

```
;; základní cond pomoci if (řešeno jako rekurzivní makro)
(define-macro cond
  (lambda clist
    (if (null? clist)
        '(if #f #f)
        (if (equal? (caar clist) 'else)
            (cadar clist)
            `(if ,(caar clist)
                 ,(cadar clist)
                 (cond ,@(cdr clist)))))))
```

Příklad použití

```
(cond ((= x 3) 'blah)
      ((> x 10) (+ 1 x))
      ((prop? x y) (list x y))
      (else (f 20)))
```

⇓

```
(if (= x 3)
    (quote blah)
    (cond ((> x 10) (+ 1 x))
          ((prop? x y) (list x y))
          (else (f 20))))
```

⇓

...

Rozšířená verze `define`

`define` jsme zatím používali pouze ve tvaru

```
(define symbol ◁vyraz▷)
```

v R6RS Scheme je `define` zaveden taky ve tvaru

```
(define (symbol ◁argumenty▷...) ◁vyrazy▷...)
```

Příklad: faktoriál

```
(define (f n)
  (if (= n 1)
      1
      (* n (f (- n 1)))))
```

Příklad: nepovinné argumenty

```
(define (f x y . args)
  (list x y args))
```

Rozšířená verze `define`

Pokud by náš interpret nedisponoval rozšířeným `define`, pak bychom jej mohli vyrobit jako makro:

```
(define-macro def
  (lambda (first . args)
    (if (symbol? first)
        `(define ,first ,@args)
        `(define ,(car first)
           (lambda ,(cdr first)
             ,@args)))))
```

Příklad použití:

```
(def (f n)
  (if (= n 1)
      1
      (* n (f (- n 1)))))
(f 6)  ⟼  720
```

Konjunkce libovolně mnoha argumentů pomocí if

```
;; základní verze
(define-macro and
  (lambda args
    (if (null? args)
        #t
        `(if ,(car args)
             (and ,@(cdr args))
             #f))))
```

```
(and 1 2 3)  ⟼  #t
```

protože:

```
(and 1 2 3)  ⟼  (if 1 (and 2 3) #f)  ⟼  ···
(and 2 3)    ⟼  (if 2 (and 3) #f)  ⟼  ···
(and 3)      ⟼  (if 3 (and) #f)  ⟼  ···
(and)        ⟼  #t  ⟼  #t
```

Konjunkce libovolně mnoha argumentů pomocí if

```
;; zlepšená verze (zobecněné pravdivostní hodnoty)
(define-macro and
  (lambda args
    (if (null? args)
        #t
        (if (null? (cdr args))
            (car args)
            `(if ,(car args)
                 (and ,@(cdr args))
                 #f)))))
```

nyní se již chová jako klasický and:

```
(and)          ⟼  #t
(and 1 2 3)    ⟼  3
(and 1 #f 3)   ⟼  #f
```

Disjunkce libovolně mnoha argumentů pomocí `if`

```
;; základní verze
(define-macro or
  (lambda args
    (if (null? args)
        #f
        `(if ,(car args)
             #t
             (or ,@(cdr args))))))
```

Příklad použití:

```
(or)          ⟹  #f
(or 1 2 3)    ⟹  #t chtěli bychom 1
(or #f 2 3)   ⟹  #t chtěli bychom 2
```

Disjunkce libovolně mnoha argumentů pomocí `if`

```
;; rozšířená verze
(define-macro or
  (lambda args
    (if (null? args)
        #f
        (if (null? (cdr args))
            (car args)
            `(if ,(car args)
                 ,(car args)
                 (or ,@(cdr args)))))))
```

Chová se (zdánlivě) v pořádku:

```
(or)          ⟹  #f
(or 1 2 3)    ⟹  1
(or #f 2 3)   ⟹  2
```

Problémy s implementací disjunkce pomocí `if`

- naše implementace `or`: dvakrát vyhodnocuje pravdivé argumenty (kromě posledního)
- důsledek: nechová se jako klasický `or` pokud použijeme vedlejší efekt

Chování klasického `or`

```
(let ((x 0))
  (or (begin (set! x (+ x 1))
             x)
      blah))  ⟼  1
```

Chování našeho `or`:

```
(let ((x 0))
  (or (begin (set! x (+ x 1))
             x)
      blah))  ⟼  2
```

Problémy s implementací disjunkce pomocí `if`

```
;; pokus o řešení předchozího problému
(define-macro or
  (lambda args
    (if (null? args)
        #f
        (if (null? (cdr args))
            (car args)
            `(let ((result ,(car args)))
               (if result
                   result
                   (or ,@(cdr args))))))))
```

Nyní už jsme předchozí problém odstranili, . . .

```
(let ((x 0))
  (or (begin (set! x (+ x 1)) x)
      blah))  ⟼  1
```

... ale nový problém jsme vyrobili

Klasický `or`:

```
(let ((result 10))
  (or #f result))  ⟹  10
```

Náš `or`:

```
(let ((result 10))
  (or #f result))
  ⇓
(let ((result #f))
  (if result result (or result)))  ⟹  #f
```

- došlo k překrytí symbolu `result` symbolem stejného jména, který je používán uvnitř našeho makra
- tomuto efektu se říká **symbol capture** (**variable capture**)
- v další lekci ukážeme čisté řešení tohoto problému