Wyznaczanie wyrażeń Scheme'a

W jaki sposób Scheme wyznacza wyrażenia?

$$(procedure\ arg_1\ \ldots\ arg_n)$$

- Znajdź wartość $procedure$
- Znajdź wartość arg_1
- Znajdź wartość arg_n
- Zastosuj wartość $procedure$ do wartości $arg_1\ \ldots\ arg_n$

Wyznaczanie wyrażeń Scheme'a

Przykład

Wyrażenie `(+ 3 4)`

Wartością procedury `+` *jest dodawanie*

Wartością `3` *jest liczba* `3`

Wartością `4` *jest liczba* `4`

Stosując dodawanie `+` *dla liczb* `3` *i* `4` *daje to* `7`

Wartością jest obiekt `7`

Wyznaczanie wyrażeń Scheme'a

Przykład

Znajdź wartość zagnieżdżonego wyrażenia

```
(* (+ 3 4) 2)
```

Wartością `*` *jest procedura mnożenia*

Wartością `(+ 3 4)` *jest liczba* `7`

Wartością `2` *jest liczba* `2`

Mnożąc `7` *przez* `2` uzyskujemy `14`

Odpowiedź brzmi `14`

Wyznaczanie wyrażeń Scheme'a

- Reguła ta jest słuszna dla oceniania procedur ale nie dla cytowanego (`quote`) wyrażenia
- Podwyrażenia procedury są oceniane, zaś podwyrażenia cytowanego nie
- Wyznaczanie wartości wyrażeniom `quote` jest podobne do oceny stałych obiektów
- Wartością wyrażenia `quote` przykładowo `(quote object)` jest po prostu *object*

Wyznaczanie wyrażeń Scheme'a

- Wyznaczanie wartości wyrażeń w Scheme-ie ma charakter dowolnego kierunku
  - od lewej do prawej
  - od prawej do lewej
  - jakikolwiek inny dowolny kierunek
- Wartości podwyrażeń mogą być wyznaczane w dowolnych kierunkach dla tego samego wyrażenia

Wyznaczanie wyrażeń Scheme'a

Przykład. Wyznacz wartość wyrażenia

```
((car (list + - * /)) 2 3)
```

Procedura ma postać

```
(car (list + - * /))
```

Wartością

```
(car (list + - * /))
```

jest operacja dodawania

Tak jakby procedura była po prostu zmienną +

Zmienne i wyrażenie `let`

- Oznaczenia:
  - *`expr`* jest wyrażeniem Scheme-a zawierające zmienną *`var`*
  - *`var`* posiada wartość *`val`* uzyskaną z oceny wyrażenia *`expr`*
- `let` – forma składniowa
- `let` zawiera listę par zmienna-wyrażenie oraz sekwencję wyrażeń stanowiące ciało formy składniowej `let`

Zmienne i wyrażenie `let`

- Ogólna postać wyrażenia `let`

  `(let ((`*var val*`)...)`exp_1 exp_2 `...)`

- `let` związuje (przypisuje) zmiennym wartości
- Mówimy o zmiennych związanych poprzez `let`-przypisanie
- Wyrażenie `let` jest często stosowane w celu uproszczenia wyrażeń zawierających dwa identyczne wyrażenia

Zmienne i wyrażenie `let`

- Wartość wspólnego podwyrażenia jest wyznaczana tylko raz

```
(+ (* 4 4) (* 4 4))      ⇒  32
```

```
(let ((a (* 4 4)))
  (+ a a))               ⇒  32
```

Zmienne i wyrażenie `let`

Przykład użycia formy składniowej `let`

```
(let ((x 2))
  (+ x 3))                    ⇒  5

(let ((y 3))
  (+ 2 y))                    ⇒  5

(let ((x 2) (y 3))
  (+ x y))                    ⇒  5
```

Zmienne i wyrażenie `let`

Przykład. Jaka będzie wartość wyrażenia?

```
(let ((list1 '(a b c))
      (list2 '(d e f)))
   (cons (cons (car list1)
               (car list2))
         (cons (car (cdr list1))
               (car (cdr list2)))))
```

⇒ ((a . d) b . e)

Zmienne i wyrażenie `let`

Przykład użycia formy składniowej *`let`*

```
(let ((f +))
  (f 2 3))
```

⇒ 5

```
(let ((f +) (x 2))
     (f x 3))
```

⇒ 5

```
(let ((f +) (x 2) (y 3))
     (f x y))
```

⇒ 5

Zmienne i wyrażenie `let`

- Zmienne związane przez wyrażenie `let` są widoczne tylko wewnątrz procedury `let`
- Przykładowo:

```
(let ((+ *))
   (+ 2 3))
```

$\Rightarrow$ 6

```
(+ 2 3)
```

$\Rightarrow$ 5

Zmienne i wyrażenie `let`

- Zagnieżdżone wyrażenie `let`

```
(let ((a 4) (b -3))
  (let ((a-squared (* a a))
        (b-squared (* b b)))
    (+ a-squared b-squared)))
```

$\Rightarrow$ 25

- Kiedy zagnieżdżone wyrażenie `let` wiąże tę samą zmienną, tylko wewnętrzne związanie poprzez `let` jest widoczne w wyrażeniu

Zmienne i wyrażenie `let`

Przykład

Jaka jest wartość wyrażenia `(+ x 1)`*?*

Co powiemy o wartości `(+ x x)`*?*

```
(let ((x 1))
  (let ((x (+ x 1)))
    (+ x x)))
```

$\Rightarrow$ `4`

Zmienne i wyrażenie `let`

- Wewnętrzne związanie `x` powoduje tzw. przyciemnianie (przykrycie) zewnętrznego `let`
- Zmienna związana przez `let` jest widoczna gdziekolwiek w ciele rozważanego wyrażenia poza zjawiskiem przykrycia
- Granice dla których związane zmienne są widoczne nazywa się zakresem (zasięgiem) widoczności

Zmienne i wyrażenie `let`

- Zasięg pierwszej wartości `x` jest w ciele zewnętrznego wyrażenia `let`
- W wewnętrznym wyrażeniu `let` jest ona przykryta przez drugą wartość `x`
- Ten sposób zasięgu określany jest jako **leksykalny zasięg**
- Zasięg każdego związania zmiennej może być określany poprzez analizę programu

Zmienne i wyrażenie `let`

- Przykrywanie może być zapobiegnięte poprzez wybranie odmiennych nazw dla zmiennych
- Poprzednie wyrażenie przepisano, tak aby zmienna związana przez wewnętrzne `let` uzyskała nową nazwę `new-x`

```
(let ((x 1))
  (let ((new-x (+ x 1)))
    (+ new-x new-x)))
```

$\Rightarrow$ 4

Wyrażenie `lambda`

- Forma składniowa `lambda` tworzy nową procedurę
- Ogólna forma `lambda` wyrażenia ma postać

  `(lambda (`*var* `...)` exp_1 exp_2 `...)`

- Zmienne *var* ... są formalnymi parametrami procedury `lambda`
- Sekwencja wyrażeń exp_1 ... jest jego ciałem

Wyrażenie `lambda`

- Przykładowo: procedura posiada parametr `x` i ma ciało podobne do wyrażenia `let`

```
(lambda (x) (+ x x))
```

⇒ `#<procedure>`

- Najbardziej znaną operacją do wykonania procedury jest zastosowanie jej do jednej lub wielu wartości

```
((lambda (x) (+ x x)) (* 3 4))
```

⇒ `24`

Wyrażenie `lambda`

- Procedury są obiektami:
  - można określać je jako wartości zmiennych
  - mogą być wywołane więcej niż jeden raz

```
(let ((double (lambda (x) (+ x x))))
  (list (double (* 3 4))
        (double (/ 99 11))
        (double (- 2 7))))
```

$\Rightarrow$ `(24 18 -10)`

Wyrażenie `lambda`

- Parametry mogą być dowolnym obiektem
- Przykładowe zastosowanie `cons` zamiast `+`

```
(let((double-cons
        (lambda (x)(cons x x))))
   (double-cons 'a))
```

$\Rightarrow$ `(a . a)`

Wyrażenie `lambda`

- Zwracając uwagę na podobieństwa `double` and `double-cons`
- Można powyższą procedurę zapisać w postaci

```
(let ((double-any
                (lambda (f x) (f x x))))
    (list (double-any + 13)
          (double-any cons 'a)))
```

$\Rightarrow$ `(26 (a . a))`

Wyrażenie `lambda`

- Wyrażenie `lambda` może być zagnieżdżane wewnątrz wyrażeń `lambda` i `let`

```
(let ((x 'a))
  (let ((f (lambda (y) (list x y))))
       (f 'b)))
```

$\Rightarrow$ `(a b)`

- Wystąpienie `x` w wyrażeniu `lambda` wskazuje na `x` poza ciałem `lambda` które związane jest z zewnętrznym wyrażeniem `let`

Wyrażenie `lambda`

- Zmienna `x` jest określana tutaj jako *zmienna niezwiązana* w wyrażeniu `lambda`
- Zmienna `y` nie występuje tutaj jako wolna w wyrażeniu `lambda` z powodu związania z wyrażeniem `lambda`
- Zmienna występująca jako wolna w wyrażeniu `lambda` powinna być związana przez zewnętrzne wyrażenie `lambda` lub `let`

Wyrażenie `lambda`

- Co w przypadku gdy procedura jest zastosowana gdzieś poza zasięgiem związania dla zmiennych, które występują w procedurze jako wolne?

```
(let ((f (let ((x 'sam))
           (lambda (y z) (list x y z)))))
  (f 'i 'am))
```

$\Rightarrow$`(sam i am)`

- Te same związania, które obowiązują dla określonej procedury są ponownie stosowane gdy procedura jest realizowana

Wyrażenie `lambda`

- Podobny efekty gdy zastosujemy inne związane zmiennej `x` widoczne w zastosowanej procedurze

```
(let ((f (let ((x 'sam))
           (lambda (y z) (list x y z)))))
  (let ((x 'not-sam))
    (f 'i 'am)))
```

$\Rightarrow$ `(sam i am)`

- W obu przypadkach, wartość zmiennej `x` w procedurze `f` jest określone obiektem `sam`

Wyrażenie `lambda`

- Wyrażenie `let` jest bezpośrednim zastosowaniem wyrażenia `lambda` dla zbioru argumentów będących wyrażeniami

- Przykład dwóch równoważnych wyrażeń

```
(let ((x 'a)) (cons x x)) ≡
((lambda (x) (cons x x)) 'a)
```

Wyrażenie `lambda`

- Wyrażenie `let` jest składniowym rozszerzeniem pod względem definicji wyrażenia `lambda`
- `let` i `lambda` stanowią jądro formy składniowej
- Dowolne wyrażenie postaci

  (let ((*var val*) ...) exp_1 exp_2 ...)

  jest równoważne poniższemu

  ((lambda (*var* ...) exp_1 exp_2 ...)
   val ...)

Wyrażenie `lambda`

- Formalna specyfikacja parametrów wyrażenia `lambda`:
  - lista właściwa zmiennych

    $(var_1 \ldots var_n)$
  - pojedyncza zmienna var_r
  - lista niewłaściwa zmiennych

    $(var_1 \ldots var_n . var_r)$

Wyrażenie `lambda`

- Rozważmy kilka poniższych przykładów:

```
(let ((f (lambda x x)))
   (f 1 2 3 4))              ⇒  (1 2 3 4)

(let ((f (lambda x x)))
   (f))                      ⇒  ()

(let ((g (lambda (x . y) (list x y))))
   (g 1 2 3 4))              ⇒  (1 (2 3 4))

(let ((h (lambda (x y . z)(list x y z))))
    (h 'a 'b 'c 'd))         ⇒   (a b (c d))
```

Definicje wyższego poziomu

- Zmienne związane przez `let` oraz `lambda` nie są widoczne poza ciałem tych wyrażeń
- Chcemy zdefiniować obiekt lub procedurę, która będzie widoczna w dowolnym miejscu
- Zastosowanie ***definicji wyższego poziomu*** określonych przez wyrażenie `define`
- Definicje wyższego poziomu są widoczne w każdym wyrażeniu poza wyrażeniami, które zostały przykryte innymi przypisaniami

Definicje wyższego poziomu

- Definicja

  `(define` *`var exp`*`)`

  gdzie *`exp`* - wyrażenie `lambda` (forma skrócona bez `lambda`)

- Składnia zależy od formatu parametrów wyrażenia `lambda` tzn.:
  - lista właściwa zmiennych
  - pojedyncza zmienna
  - lista niewłaściwa zmiennych

Definicje wyższego poziomu

- Definicja postaci

```
(define var0
  (lambda (var1 ... varn)
    e1 e2 ...))
```

może być zapisana skrótowo

```
(define (var0 var1 ... varn)
  e1 e2 ...)
```

Definicje wyższego poziomu

- Definicja postaci

```
(define var0
  (lambda varr
    e1 e2 ...))
```

może być zapisana skrótowo

```
(define (var0 . varr)
  e1 e2 ...)
```

Definicje wyższego poziomu

- Definicja postaci

(define var_0
  (lambda (var_1 ... var_n . var_r)
    e_1 e_2 ...))

może być zapisana skrótowo

(define (var_0 var_1 ... var_n . var_r)
  e_1 e_2 ...)

Definicje wyższego poziomu

- Przykładowo, definicje procedur `cadr, cdar` oraz `list` mogą być zapisane następująco:

```
(define (cadr x)
  (car (cdr x)))

(define (cdar x)
  (cdr (car x)))

(define (list . x) x)
```

Definicje wyższego poziomu

- Określmy definicję wyższego poziomu procedury `double-any`

```
(define double-any
  (lambda (f x)
    (f x x)))
```

- Zmienna `double-any` ma teraz taki sam status jak procedura `cons` lub innej *prymitywnej procedury*

Definicje wyższego poziomu

- Zastosowanie `double-any` jak *prymitywnej procedury:*

```
(double-any + 10)          ⇒  20
(double-any cons 'a)       ⇒  (a . a)
```

- Definicje wyższego poziomu mogą być określane dla dowolnych obiektów

```
(define sandwich "peanut-butter-and-jelly")
     sandwich  "peanut-butter-and-jelly"
```

Definicje wyższego poziomu

- Definicje wyższego poziomu mogą być nadpisane przez wyrażenia `let` lub `lambda`

```
(define xyz '(x y z))
  (let ((xyz '(z y x)))
    xyz)
```

$\Rightarrow$ `(z y x)`

- Zmienne określone w definicjach wyższego poziomu mają taki status jakby były związane z wyrażeniem zamkniętym w ich ciałach

Definicje wyższego poziomu

- Skróty wyrażeń `cadr` i `cddr` są kompozycją (złożeniem) funkcji
  - `car` z `cdr`
  - `cdr` z `cdr`

  Przykładowo:

```
(cadr list)    ≡    (car (cdr list))
(cddr list)    ≡    (cdr (cdr list))
```

Definicje wyższego poziomu

- Można je zdefiniować następująco:

```
(define cadr
  (lambda (x)
    (car (cdr x))))

(define cddr
  (lambda (x)
    (cdr (cdr x))))

(cadr '(a b c))  ⇒  b
(cddr '(a b c))  ⇒  (c)
```

Definicje wyższego poziomu

- Definicje wyższego poziomu sprawiają że:
  - łatwiejsze jest interaktywne eksperymentowanie z procedurą wprowadzania z linii poleceń
  - nie musimy wpisywać procedury za każdym razem w celu jej zastosowania – widoczna globalnie

Definicje wyższego poziomu

- Przykład bardziej skomplikowanego obiektu na podstawie procedury `double-any`:
  - zamiana procedury dwóch argument na procedurę jedenoargumentową

```
(define doubler
   (lambda (f)
     (lambda (x) (f x x))))
```

- Procedura `doubler` akceptuje pojedynczy argument będący procedurą, która musi akceptować dwa argumenty

Definicje wyższego poziomu

- Zdefiniujmy za pomocą procedury `doubler`, proste funkcje:
  - `double`
  - `double-cons`

```
(define double (doubler +))

(double 13/2)          ⇒  13

(define double-cons (doubler cons))

(double-cons 'a)       ⇒  (a . a)
```

Definicje wyższego poziomu

- Możemy również zdefiniować `double-any` za pomocą procedury `doubler`

```
(define double-any
  (lambda (f x)
    ((doubler f) x)))
```

- W procedurze `double` i `double-cons` argument `f` posiada odpowiednią wartość tzn.: `+` lub `cons` mimo że, zastosowane procedury są poza widocznością argumentu `f`

Definicje wyższego poziomu

- Co się stanie, przy próba użycia zmiennej, która nie jest związana z wyrażeniem `lambda` lub `let` i nie ma definicji wyższego poziomie?
- Zastosowanie zmiennej *`i-am-not-defined`*

```
(i-am-not-defined 3)
```

- Interpreter Scheme-a wyświetla informację o błędzie, że zmienna nie jest związana lub jest niezdefinowana