Przypisania

- Przypisanie nie tworzy nowego związania jak wyrażenia `let` lub `lambda`
- Przypisanie zmienia wartość istniejącego związania
- Przypisanie realizowane wyrażeniem `set!`

```
(define abcde '(a b c d e))
   abcde  ⇒  (a b c d e)
(set! abcde (cdr abcde))
   abcde  ⇒  (b c d e)
```

Przypisania

- Wiele języków programowania wymaga użycia przypisania do:
  - inicjalizacji zmiennych
  - odseparowania deklaracji od związania ze zmiennymi
- W Scheme-ie zmienne lokalne są natychmiast związane z wartościami
- Składnia Scheme-a powoduje, że programista nie może zapomnieć o związaniu zmiennych lokalnych z danymi wartościami

Przypisania

- Przypisania w językach imperatywnych są konieczne lub też wygodne
- W praktyce algorytmy w tych językach są wyrażane za pomocą sekwencji przypisań
- Przypisania są niepotrzebne oraz uciążliwe w językach funkcyjnych (np. Scheme)
- Nie stosowanie przypisań daje szanse programiście zdefiniowania algorytmu w bardziej przejrzysty sposób

Przypisania

- Przykładowa procedura wyznaczająca pierwiastki równania kwadratowego

```
(define quadratic-formula
  (lambda (a b c)
    (let ((root1 0) (root2 0) (minusb 0) (radical 0)
          (divisor 0))
      (set! minusb (- 0 b))
      (set! radical (sqrt (-(* b b) (* 4 (* a c)))))
      (set! divisor (* 2 a))
      (set! root1 (/ (+ minusb radical) divisor))
      (set! root2 (/ (- minusb radical) divisor))
      (cons root1 root2))))
```

Przypisania

- Powyższa procedura bez przypisań

```
(define quadratic-formula
  (lambda (a b c)
   (let ((minusb (- 0 b))
         (radical (sqrt ((* b b) (* 4 (* a c)))))
         (divisor (* 2 a)))
     (let ((root1 (/ (+ minusb radical) divisor))
           (root2 (/ (- minusb radical) divisor)))
       (cons root1 root2)))))
```

Przypisania

- Procedura wyjaśnia zależność pierwiastków `root1` i `root2` od wartości `minusb`, `radical`, oraz `divisor`
- Wyrażenie `let` określa brak zależności pomiędzy `minusb`, `radical` oraz `divisor` a pierwiastkami `root1` i `root2`

Przypisania

- Funkcja `cons-count` zliczająca liczbę wywołań
- Liczba wywołań zapisana w `cons-count`
- Funkcja wykorzystuje formę `set!`
- Brak innego sposobu w przypadku przypisań

```
(define cons-count 0)
(define cons
  (let ((old-cons cons))
    (lambda (x y)
      (set! cons-count (+ cons-count 1))
      (old-cons x y))))
```

Przypisania

- Przykładowe wywołanie:

```
(cons 'a '(b c))   ⇒   (a b c)
cons-count  ⇒  1

(cons 'a (cons 'b (cons 'c '())))
                    ⇒   (a b c)
cons-count  ⇒  4
```

- `set!` jest stosowany do:
  - określenia nowej wartości procedury `cons`
  - nadpisania `cons-count` po każdym wywołaniu co`ns`

Przypisania

- Procedura zwracająca 0 po pierwszym uruchomieniu, 1 po drugim, 2 po trzecim itd.

```
(define next 0)
(define count
  (lambda ()
    (let ((v next))
      (set! next (+ next 1))
      v)))
(count)  ⇒  0
(count)  ⇒  1
```

Przypisania

- Rozwiązanie to jest nieco niepożądane – zmienna `next` jest widoczna na najwyższym poziomie (tak nie musi)
- Każdy kod w systemie może zmienić jej wartość
- Zastosowanie `let`-związanie `next` poza `lambda`
- ```
  (define count
    (let ((next 0))
      (lambda ()
        (let ((v next))
          (set! next (+ next 1))
          v))))
  ```
  ```

Przypisania

- Procedura `make-counter` zwracająca nowy licznik przypisań po każdym jej wywołaniu

```
(define make-counter
  (lambda ()
    (let ((next 0))
      (lambda ()
        (let ((v next))
          (set! next (+ next 1))
          v)))))
```

Przypisania

- Każda nowa procedura zliczająca swoje wywołania zwraca i utrzymuje swój unikalny licznik

```
(define count1 (make-counter))
(define count2 (make-counter))
(count1)  0
(count2)  0
(count1)  1
(count1)  2
(count2)  1
```

Przypisania

- Formy składniowe `set-car!` i `set-cdr!` przypisują nowe wartości odpowiednio dla głowy i ogona listy
- Przykładowo:

```
(define p (list 1 2 3))

(set-car! (cdr p) 'two)
p ⇒ (1 two 3)

(set-cdr! p '())
p  ⇒  (1)
```

Rozszerzenia składniowe

- Rozszerzenia składniowe:
  - posiadają ten sam status jak formy podstawowe
  - muszą ostatecznie pochodzić od form podstawowych
- Scheme:
  - rozwija rozszerzenia składniowe w formy podstawowe w kroku będącym kompilacją kodu lub interpretacją wyrażenia
  - pozwala dalej skupić uwagę kompilatora lub interpretera na formach podstawowych

Rozszerzenia składniowe

- `define-syntax` – forma definiująca rozszerzenia składniowe
- Podobna do formy `define` – powiązana z składniami przekształcającymi procedury (***procedura transformującą***)
- Przykładowa definicja formy `let`

```
(define-syntax let
  (syntax-rules ()
    ((_ ((x e) ...) b1 b2 ...)
     ((lambda (x ...) b1 b2 ...) e ...))))
```

Rozszerzenia składniowe

- Identyfikator pojawiający się po formie `define-syntax` oznacza ***słowo kluczowe***
- Forma `syntax-rules` jest wyrażeniem, której wynikiem jest ***procedura transformująca***
- Wyrażenie `syntax-rules` zawiera ***listę pomocniczych słów kluczowych***
- *Lista pomocniczych słów kluczowych* jest sekwencją jednej lub wielu ***reguł***, oraz ***par wzorców/szablonów***

Rozszerzenia składniowe

- Wzorzec reguły określa formę, która musi pobrać dane wejściowe wyrażenie
- Szablon reguły definiuje w co powinno być przekształcone wyrażenie
- Wzorzec powinien mieć strukturę wyrażenia z pierwszym elementem – podkreśleniem (_)
- Odpowiednio jedna reguła jest wybierana poprzez dopasowanie wzorca w celu rozwijania danego wyrażenia

Rozszerzenia składniowe

- Naruszenie zasad składni – jeżeli żaden wzorzec nie pasuje do danego wyrażenia wejściowego
- Identyfikatory inne niż podkreślenia lub wielokropek występujące w strukturze reprezentują ***zmienne wzorca***
- Oznacznie `pat...` we wzorcu pozwala na dopasowanie zerowej lub większej liczby wyrażeń do niego

Rozszerzenia składniowe

- Podobnie oznaczenie *expr*`...` w szablonie powoduje uzyskanie zerowej lub większej liczby wyrażeń wynikowych dla *expr*`...`
- Liczba zmiennych wzorca *pats* określa liczbę wynikowych wyrażeń *exprs*
- Jakiekolwiek wielokropki we wzorcu muszą zawierać co najmniej jedną zmienną wzorca w wyrażeniu

Rozszerzenia składniowe

- Składnia `let` wymaga, aby ciało zwierało co najmniej jedną formę tzn. `b1 b2 ...`
- `let` nie wymaga, aby zawierało co najmniej jedną parę zmienna-wartość - `(x e) ....`
- Zmienne wzorca `x` i `e` oddzielone w szablonie
- Zmienne wzorca `x`, `e` i `b2` , które pojawiają się przy wielokropkach w strukturze pojawiają się również w szablonie

Rozszerzenia składniowe

- Bardziej złożony przykład definicji formy `and`

```
(define-syntax and
  (syntax-rules ()
    ((_) #t)
    ((_ e) e)
    ((_ e1 e2 e3 ...)
     (if e1 (and e2 e3 ...) #f))))
```

Rozszerzenia składniowe

- Powyższa definicja jest rekurencyjna i dotyczy więcej niż jednej reguły
- Wyrażenie (`and`) ocenianie jako wartość `#t` (pierwsza reguła z definicji)
- Druga i trzecia reguła stanowią przypadek bazowy rekurencji
- Czwarta reguła reprezentuje krok rekurencji tłumaczony na dwa lub więcej podwyrażeń w postaci zagnieżdżonej formy `if`

Rozszerzenia składniowe

- Przykładowe rozwinięcie `(and a b c)`

```
(if a (and b c) #f)
↓
(if a (if b (and c) #f) #f)
↓
(if a (if b c #f) #f)
```

Jeżeli `a` i `b` przyjmują wartość prawdy, wówczas wartością wyrażenia jest wartość `c`,
w przeciwnym przypadku `#f`

Rozszerzenia składniowe

- W formie `or` należy zastosować tymczasową zmienną dla każdej pośredniej wartości
- Jednoczesne testowanie wartości i zwracanie jej jeżeli jest ona prawdą (`#t`)
- Tymczasowa zmienna nie potrzebna dla formy `and` (tylko jeden obiekt o wartości fałszu (`#f`)

Rozszerzenia składniowe

```
(define-syntax or
   (syntax-rules ()
     ((_) #f)
     ((_ e) e)
     ((_ e1 e2 e3 ...)
      (let ((t e1))
        (if t t (or e2 e3 ...))))))
```

Rozszerzenia składniowe

- Identyfikatory wprowadzne przez szablony mają zasięg leksykalny (widoczne tylko wewnątrz wyrażenia szablonu)
- Jeżeli jedno z wyrażeń `e2 e3` ... zawiera odniesienie do zmiennej `t` to zastosowane związanie dla `t` „nie wychwytuje” tego odniesienia
- Zwykle jest to realizowane poprzez automatyczną zmianę nazwy wprowadzonego identyfikatora

Rekurencja

- `let`-związanie nie można zastosować do rekurencji, np.

```
(let ((sum (lambda (ls)
             (if (null? ls)
                 0
                 (+ (car ls)
                    (sum (cdr ls)))))))
  (sum '(1 2 3 4 5)))
```

⇒ **reference to undefined identifier: sum**

Rekurencja

- Rozwiązanie problemu poprzez zastosowanie formy składniowej `letrec`
- `letrec` zawiera zestaw par zmienna-wartość, wraz z ciągiem wyrażeń określonych jako ciało tego wyrażenia

  `(letrec ((`*`var expr`*`) ...)` $body_1$ `...)`

- Zmienne *`var`* `...` nie są widoczne tylko w ciele `letrec` ale także wewnątrz *`expr`* `....`

Rekurencja

- Powyższa procedura z użyciem `letrec`

```
(letrec ((sum (lambda (ls)
                (if (null? ls)
                    0
                    (+ (car ls) (sum
                        (cdr ls)))))))
  (sum '(1 2 3 4 5)))
                        ⇒ 15
```

Rekurencja

- Definicja wzajemnie rekurencyjnych procedur (predykatów) `even?` i `odd?`

```
(letrec ((even?
          (lambda (x)
            (or (= x 0)
                (odd? (- x 1)))))
         (odd?
          (lambda (x)
            (and (not (= x 0))
                 (even? (- x 1))))))
  (list (even? 20) (odd? 20)))
```

⇒ `(#t #f)`

Rekurencja

- W wyrażeniu `letrec` , *`expr`* `...` są zwykle wyrażeniem `lambda`
- Każde wyrażenie *`expr`* musi być ocenione bez oceny jakiejkolwiek zmiennej *`var`*
- Ograniczenie zawsze spełnione przez `lambda`
- Zmienne w `lambda` nie mogą zostać ocenione dopóki procedura nie zostanie zastosowana w ciele wyrażenia `letrec`

Rekurencja

- Wyrażenie `letrec` nie naruszające ograniczenie

```
(letrec ((f
          (lambda()
            (+ x 2)))(x 1))(f))
```

$\Rightarrow$ 3

- Wyrażenie `letrec` naruszające ograniczenie

```
(letrec ((y (+ x 2)) (x 1)) y)
```

Rekurencja

- Jeżeli rekurencja jest wywoływana tylko w jednym miejscu poza procedurą, wówczas zastosować można *nazwane wyrażenie* `let`
- *Nazwane wyrażenie* `let` posiada formę:

  `(let` *name*`((`*var expr*`)...)` $body_1$ $body_2$`...)`
- Zmienna *name* jest związana w ciele procedury wywołującej rekurencję
- Argumenty procedury stają się nowymi wartościami dla zmiennych *var* `....`

Rekurencja

- Nazwany `let`

  (let *name* ((*var expr*)...) $body_1$ $body_2$ ...)

- Zapisane za pomocą `letrec`

```
((letrec ((name (lambda (var ...)
                  body1 body2 ...)))
   name)
 expr ...)
```

- Alternatywna postać

```
(letrec ((name (lambda (var ...)
                 body1 body2 ...)))
  (name expr ...))
```

Rekurencja

- Niektóre rekurencje są w istocie powtórzeniami
- ***Rekrencja ogonowa*** - procedura jest wywoływana dla ogona argumentu w odniesieniu do wyrażenia `lambda`
- Scheme traktuje rekurencję ogonową jak procedurę "goto" lub „jump”
- Zastosowanie rekurencji ogonowej zapobiega przepełnieniu pamięci stosu

Rekurencja

- Przykładowo, wywołanie ogonowe występuje jeżeli ostatnim wyrażeniem w ciele wyrażenia `lambda` jest:
  - częścią *konsekwencji, alternatywy* wyrażenia `if`
  - ostatnim podwyrażenim wyrażenia `and` lub `or`
  - ostatnim wyrażeniem w ciele `let` or `letrec`

```
(lambda () (f (g)))
(lambda () (if (g) (f) (f)))
(lambda () (let ([x 4]) (f)))
(lambda () (or (g) (f)))
```

Rekurencja

- Definicja silni `n!` dla liczb nieujemnych `n` z zastosowaniem nazwanego wyrażenia `let`
- Definicja rekurencyjna $n! = n \times (n-1)!$, dla przypadku `n=0` silnia wynosi `1`

```
(define factorial
  (lambda (n)
    (let fact ((i n))
      (if (= i 0)
          1
          (* i (fact (- i 1)))))))
```

Rekurencja

- Definicja iteracyjna silni

$n! = n \times (n-1) \times (n-2) \times \ldots \times 1$

- Zastosowanie akumulatora do zapamiętania pośredniego wyniku

```
(define factorial
  (lambda (n)
    (let fact ((i n) (a 1))
      (if (= i 0)
          a
          (fact (- i 1) (* a i))))))
```