Definicje wyższego poziomu

- Interpreter Scheme-a nie będzie „narzekać” w przypadku wystąpienia niezdefionowanej zmiennej w ciele wyrażenia `lambda` dopóki nie będzie zastosowana
- Przykład braku informacji o błędzie

```
(define proc1
  (lambda (x y)
    (proc2 y x))
```

Definicje wyższego poziomu

- Próba zastosowania `proc1` przed definicją `proc2` spowoduje wystąpienie błędu
- Przykładowe utworzenie `proc2` i wywołanie funkcji `proc1`

```
(define proc2 cons)

(proc1 'a 'b)  ⇒  (b . a)
```

Definicje wyższego poziomu

- Scheme akceptuje założenie o zdefiniowaniu `proc2` w innym miejscu kodu programu
- Nie uruchamianie `proc1` przed zdefiniowaniem `proc2` nie generuje błędu
- Definicje funkcji mogą być umieszczane w dowolnej kolejności
- Użyteczny mechanizm do bardziej czytelnego pisania programów i organizacji w pliku definicji funkcji

Wyrażenia warunkowe

- Wyrażenie `if` ma postać:

  (`if` *`test consequent alternative`*)

  gdzie:

  – *`consequent`* – wyznaczane wyrażenie jeżeli *`test`* jest prawdą

  – *`alternative`* – oceniane wyrażenie jeżeli *`test`* jest fałszem

Wyrażenia warunkowe

- Rozważmy funkcje `modulus`
- Jeżeli argument `n` jest ujemny, `modulus` zwraca `-n`, w przeciwnym przypadku zwracane jest `n`

```
(define modulus (lambda (n)
    (if (< n 0)
        (- 0 n)
        n)))
```

```
(modulus 77)  ⇒  77
(modulus -77) ⇒  77
```

Wyrażenia warunkowe

- Inne sposoby definicji funkcji `modulus`

```
(define  modulus (lambda (n)
    (if (>= n 0)
        n
        (- 0 n))))

(define  modulus (lambda (n)
    (if (not (< n 0))
        n
        (- 0 n))))
```

Wyrażenia warunkowe

```
(define  modulus (lambda (n)
  (if (or (> n 0) (= n 0)) n (- 0 n))))
```

```
(define  modulus (lambda (n)
  (if (= n 0) 0
          (if (< n 0)(- 0 n) n))))
```

```
(define  modulus (lambda (n)
      ((if (>= n 0) + -)0 n)))
```

Wyrażenia warunkowe

- `if` jest formą składniową – nie procedurą
- Rozważmy funkcję `reciprocal`

```
(define reciprocal
  (lambda (n)
    (if (= n 0)
        "oops!"
        (/ 1 n))))
```

Wyrażenia warunkowe

- Forma składniowa `or` działa podobnie jak `if`
- Ogólna postać wyrażenia `or`

`(or` *exp* `...)`

- Wartość wyrażenia `(or)` jest fałszem
- Każde *exp* jest oceniane do jednej z wartości:
  a) jedno z wyrażeń jest prawdą (`#t`)
  b) brak wyrażeń w ciele formy `if` (`#f`)

Wyrażenia warunkowe

- Wartość wyrażenia `or` jest wartością ostatniego podwyrażenia - (a)
- Wiele możliwych obiektów dla wartości prawdy
- Zwykle, wartość wyrażenia testowanego jest jedną z dwóch obiektów `#t` dla prawdy lub `#f` dla fałszu

```
(< -1 0)   ⇒   #t
(> -1 0)   ⇒   #f
```

Wyrażenia warunkowe

- Obiekty Scheme-a mogą przyjmować wartość prawdy lub fałszu:
  - w wyrażeniach warunkowych
  - w przypadku stosowania procedury `not`
- Tylko obiekt `#f` posiada wartość fałszu
- Pozostałe obiektu mają wartość prawdy

```
(if #t 'true 'false)  ⇒  true
(if #f 'true 'false)  ⇒  false
```

Wyrażenia warunkowe

```
(if '() 'true 'false)          ⇒  true
(if 1 'true 'false)            ⇒  true
(if '(a b c) 'true 'false)     ⇒  true

(not #t)          ⇒  #f
(not "false")     ⇒  #f
(not #f)          ⇒  #t

(or)              ⇒  #f
(or #f)           ⇒  #f
(or #f #t)        ⇒  #t
(or #f 'a #f)     ⇒  a
```

Wyrażenia warunkowe

- `and` forma składniowa podobna do formy `or`
- Wyrażenie `and` jest prawdą jeżeli wszystkie podwyrażenia mają wartość `#t`
- Wyrażenie `(and)` jest zawsze prawdziwe
- Podwyrażenia są oceniane do jednej z wartości:
  - brak podwyrażeń w ciele formy - `#t`
  - wartość podwyrażenia jest fałszem
- Wartość wyrażenia `and` jest wartością ostatniego ocenianego podwyrażenia

Wyrażenia warunkowe

- *Przykład. Użycie* `and` *w funkcji* `reciprocal`

```
(define reciprocal
  (lambda (n)
    (and (not (= n 0))
         (/ 1 n))))

(reciprocal 3)      ⇒  1/3
(reciprocal 0.5)    ⇒  2.0
(reciprocal 0)      ⇒  #f
```

Wartość - `#f` *jeżeli* `n` *jest zerem*
w przeciwnym wypadku `1/n`

Wyrażenia warunkowe

- *Predykaty* `=, <, >, <=, >=`
- Predykat jest procedurą, która odpowiada na specyficzne pytania dotyczące argumentu i zwraca jedną z dwóch wartości `#t` or `#f`
- Nazwy predykatów zakończone są znakiem (`?`)
- Zwykłe predykaty relacji są wyjątkiem od powyższej reguły

Wyrażenia warunkowe

- Nie wszystkie predykaty wymagają argumentów w postaci liczb
- Predykat `null?` jest prawdą jeżeli jego argument jest listą pustą `()`
- Przykładowo:

```
(null? '())                    ⇒  #t
(null? 'abc)                   ⇒  #f
(null? '(x y z))               ⇒  #f
(null? (cdddr '(x y z)))       ⇒  #t
```

Wyrażenia warunkowe

- Problem z wyrażeniem `(cdr '())`
- Interpreter Scheme-z informuje o błędzie
- Definicja `lisp-cdr` rozwiązuje ów problem

```
(define lisp-cdr (lambda (x)
    (if (null? x)
        '()
        (cdr x))))
```

```
(lisp-cdr '(a b c))      ⇒  (b c)
(lisp-cdr '(c))          ⇒  ()
(lisp-cdr '())           ⇒  ()
```

Wyrażenia warunkowe

- Predykat `eqv?` wymaga dwóch argumentów
- Jeżeli dwa argumenty są równoważne ⇒ prawda
- W przeciwnym wypadku `eqv?` zwraca fałsz
- Przykładowo:

```
(eqv? 'a 'a)        ⇒  #t
(eqv? 'a 'b)        ⇒  #f
(eqv? #f #f)        ⇒  #t
(eqv? #t #t)        ⇒  #t
(eqv? #f #t)        ⇒  #f
(eqv? 3 3)          ⇒  #t
(eqv? 3 2)          ⇒  #f
```

Wyrażenia warunkowe

- Inne przykłady

```
(let ((x "Hi Mom!"))
  (eqv? x x))
                    ⇒  #t

(let ((x (cons 'a 'b)))
  (eqv? x x))
                    ⇒  #t

(eqv? (cons 'a 'b) (cons 'a 'b))
                    ⇒  #f
```

Wyrażenia warunkowe

- ***Predykaty typu*** zwracające prawdę lub fałsz:
  - `pair?, symbol?, number?, string?`
- Predykat `pair?` zwraca prawdę tylko jeżeli jego argument jest parą

```
(pair? '(a . c))          ⇒   #t
(pair? '(a b c))          ⇒   #t
(pair? '())               ⇒   #f
(pair? 'abc)              ⇒   #f
(pair? "Hi Mom!")         ⇒   #f
(pair? 1234567890)        ⇒   #f
```

Wyrażenia warunkowe

- ***Predykat typu*** jest użyteczny do określania czy stosowany argument jest odpowiedniego typu
- Przykładowo:

```
(define reciprocal  (lambda (n)
  (if (and (number? n) (not (= n 0)))
      (/ 1 n)
      "oops!")))

(reciprocal 2/3)  ⇒  3/2
(reciprocal 'a)   ⇒ "oops!"
```

Wyrażenia warunkowe

- Rozważmy inne wyrażenie warunkowe
- `cond` często stosowane zamiast `if`
- `cond` (podobne do `if`) pozwala na użycie wielu testów i alternatywnych wyrażeń
- Wyrażenie `cond` posiada ogólną formę:

```
(cond (test exp) ... (else exp))
```

Wyrażenia warunkowe

- Rozważmy definicje funkcji `sign` zwracającej:
  - -1 dla ujemnych argumentów
  - +1 dla dodatnich argumentów
  - 0 dla zera

```
(define sign
  (lambda (n)
    (if (< n 0)
        -1
        (if (> n 0)
            +1
            0))))
```

Wyrażenia warunkowe

- Dwa wyrażenia `if` mogą być zastąpione przez pojedyncze wyrażenie `cond`:

```
(define sign
  (lambda (n)
    (cond
      ((< n 0) -1)
      ((> n 0) +1)
      (else 0))))
```

Wyrażenia warunkowe

- Pominięcie `else` zwiększa przejrzystość kodu
- Nowa wersja funkcji `sign`

```
(define sign
  (lambda (n)
    (cond
      ((< n 0) -1)
      ((> n 0) +1)
      ((= n 0) 0))))
```

- Definicja `sign` nie zależy od kolejności testowanych warunków

Wyrażenia warunkowe

- Przykładowa procedura obliczająca podatek dla danych dochodów w skali progresywnej z progami 10000, 20000, and 30000 euro

```
(define income-tax
 (lambda (income)
   (cond
    ((<= income 10000) (* income .05))
    ((<= income 20000) (+ (* (- income 10000) .08) 500.00))
    ((<= income 30000) (+ (* (- income 20000) .13) 1300.00))
    (else (+ (* (- income 30000) .21) 2600.00)))))
```

Wyrażenia warunkowe

- Przykładowe wywołania:

```
(income-tax 5000)   ⇒  250.0
(income-tax 15000)  ⇒  900.0
(income-tax 25000)  ⇒  1950.0
(income-tax 50000)  ⇒  6800.0
```

- Ważny kierunek przeprowadzania testu – od lewej do prawej i od góry do dołu

Prosta rekurencja

- **Q**: Jak wykonać powtórnie wyrażenie:
  - dla wszystkich elementów listy
  - dla liczb np. od 1 do 10?
- **A**: Realizacja za pomocą rekurencji
- **Rekurencja** jest prostą koncepcją zastosowania procedury w procedurze

Prosta rekurencja

- ***Rekurencyjna procedura*** jest procedurą zastosowaną do samej siebie
- Najprostsza rekurencyjna procedura:

```
(define goodbye
  (lambda ()
    (goodbye)))

(goodbye)  ⇒
```

- Procedura nie zwraca wyniku ze względu na nieskończone wywołanie

Prosta rekurencja

- Rekurencyjne procedury powinny mieć co najmniej dwa podstawowe elementy:
  - ***przypadek bazowy***
  - ***krok rekurencyjny***
- ***Przypadek bazowy*** kończy rekurencję dając procedurze wartość
- ***Krok rekurencyjny*** daje wartość pod względem wartości zastosowanej procedury do innego argumentu

Prosta rekurencja

- Problem: znalezienie funkcji wyznaczającej długość listy w sposób rekurencyjny
- Argument bazowy rekursji to lista pusta
- Długość pustej listy wynosi zero
- *Przypadek bazowy* powinien zwrócić zerową wartość dla listy pustej
- W celu zbliżania się do listy pustej *krok rekurencji* dotyczy ogona listy `(cdr)`

Prosta rekurencja

- Krok rekurencji daje wartość o jeden więcej niż długość ogona listy `(cdr)`

```
(define length
  (lambda (ls)
    (if (null? ls)
        0
        (+ (length (cdr ls)) 1))))
(length '())          ⇒  0
(length '(a))         ⇒  1
(length '(a b))       ⇒  2
```

Prosta rekurencja

- Śledzenie kolejnych kroków procedury rekurencyjnej

```
length '(a b c d))

|(length (a b c d))
| (length (b c d))
| |(length (c d))
| | (length (d))
| | |(length ())
| | |0
| | 1
| |2
| 3
|4
```

Prosta rekurencja

- Wcięcia reprezentują poziom zagnieżdżenia procedury
- Linie pionowe odpowiadają graficznie wartościom wywołań rekurencyjnych
- Każde wywołanie funkcji długość powoduje zmniejszenie listy aż do listy pustej
- Wartość dla listy pustej wynosi 0 a następnie każdy zewnętrzny poziom dodaje 1 do otrzymania ostatecznej wartości

Prosta rekurencja

- Rozważmy procerdurę `list-copy` zwracająca kopie listy będącej jej argumentem
- `list-copy` zwraca nową listę zawierającą elementy (nie pary) starej listy
- Tworzenie kopi listy ma na celu zachowanie oryginału, który może podlegać dalej zmianie

```
(list-copy '())          ⇒  ()
(list-copy '(a b c))     ⇒  (a b c)
```

Prosta rekurencja

- Definicja `list-copy`

```
(define list-copy
  (lambda (ls)
    (if (null? ls)
        '()
        (cons (car ls)
              (list-copy (cdr ls))))))
```

- `list-copy` łączy głowę listy z wartościami rekurencyjnego wywołania

Prosta rekurencja

- Możliwość zdefiniowania więcej niż jednego przypadku bazowego
- Procedura `memv` ma 2 argumenty - obiekt i lista
- `memv` zwraca:
  - podlistę (ogon) którego pierwszy element jest równy obiektowi
  - `#f` jeżeli obiekt nie został znaleziony
- Wartości `memv` – używane jako lista lub jako wartość prawdy w wyrażeniach warunkowych

Prosta rekurencja

```
(define memv
  (lambda (x ls)
    (cond
      ((null? ls) #f)
      ((eqv? (car ls) x) ls)
      (else (memv x (cdr ls))))))
```

- Pierwszy `cond` testuje bazowy przypadek $\Rightarrow$ ()

  Obiekt nie jest elementem listy $\Rightarrow$ `#f`
- Drugi `cond` pyta czy głowa listy jest obiektem
- Krok rekurencji kontynuuje sprawdzanie dla pozostałych elementów

Prosta rekurencja

- Przykładowe wywołanie:

```
(memv 'a '(a b b d))
                                ⇒  (a b b d)
(memv 'b '(a b b d))
                                ⇒  (b b d)
(memv 'c '(a b b d))
                                ⇒  #f
(memv 'd '(a b b d))
                                ⇒  (d)
(if (memv 'b '(a b b d)) "yes" no")
                                ⇒  "yes"
```

Prosta rekurencja

- Procedura `remv` posiada dwa argumenty - obiekt i listę
- `remv` zwraca nową listę obiektów bez ich powtórzeń

```
(define remv
  (lambda (x ls)
    (cond
      ((null? ls) '())
      ((eqv? (car ls) x) (remv x (cdr ls)))
      (else (cons (car ls) (remv x (cdr ls)
))))))
```

Prosta rekurencja

```
(remv 'a '(a b b d))  ⇒  (b b d)
(remv 'b '(a b b d))  ⇒  (a d)
(remv 'c '(a b b d))  ⇒  (a b b d)
(remv 'd '(a b b d))  ⇒  (a b b)
```

- `remv` nie zatrzymuje się tylko na elementach głowy listy – kontynuuje rekurencję poprzez zignorowanie obiektów pojedynczych
- Jeżeli obiekt nie został znaleziony jako `car` listy, `remv` łączy ten element z rekurencyjną wartością

Prosta rekurencja

- Inna wersja procedury `tree-copy` traktującej strukturę drzewa argumentu jako parę

```
(define tree-copy
  (lambda (tr)
    (if (not (pair? tr))
        tr
        (cons (tree-copy (car tr))
              (tree-copy (cdr tr))))))
(tree-copy '((a . b) . c))
                    ⇒  ((a . b) . c)
```

Prosta rekurencja

- Lewe podrzewo jest głową pary
- Prawe podrzewo jest ogonem pary
- Podobna operacja do `list-copy` budująca nowe pary pozostawiając tylko liście
- Przypadek bazowy sprawdza czy elementem drzewa jest wszystko poza parą
- Krok rekurencyjny jest *podwójnie rekursywny* znajdujący wartość rekurencyjnie dla głowy jak i ogona argumentu

Prosta rekurencja

- Iteracje w Scheme-a wyrażone za pomocą rekurencji są bardziej czytelne i treściwe
- Rekurencja jest eliminuje dodatkowe zmienne i przypisania wymagane w innych językach zawierające struktury iteracyjne
- W rezultacie kod programu jest bardziej niezawodny i łatwiejszy do śledzenia
- Niektóre rekurencje są w istocie iteracją

Prosta rekurencja

- Rozważmy specjalną formę składniową powtórzeń zwanych *odwzorowaniem*
- Procedura `modulus-all` posiada argument będący listą liczb całkowitych i zwraca ich moduł

```
(define modulus-all
  (lambda (ls)
    (if (null? ls)
        '()
        (cons (modulus (car ls))
              (modulus-all (cdr ls))))))
```

```
(abs-all '(1 -2 3 -4 5 -6))  ⇒ (1 2 3 4 5 6)
```

Przypisanie

- 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)
```