Wykład 7.
Zmienne i środowiska

Kolejny wykład dotyczy let-wyrażeń, o które rozszerzymy rozważane w poprzednim wykładzie wyrażenia arytmetyczne. Na pierwszy rzut oka let-wyrażenia nie wydają się aż tak piorunująco interesującą konstrukcją. W końcu wyrażenie w Rackecie

```
(let ([x e1]) e2)
```

można traktować jako lukier syntaktyczny wyrażenia

```
((lambda (x) e2) e1)
```

(upewnij się, że rozumiesz dlaczego). A jednak, let-wyrażenia pozwolą nam w kontrolowanych warunkach snuć rozważania o zmiennych, wiązaniach, podstawieniach, przesłanianiu i przechwytywaniu zmiennych, strategiach ewaluacji, środowiskach, reprezentacji składni...

7.1. Zmienne wolne i związane

Zmienna to miejsce w wyrażeniu, w którym (nagle w przyszłości) może pojawić się inne wyrażenie. Operację zamiany zmiennej na wyrażenie nazywamy **podstawieniem**. Co ciekawe, paczkę danych, która nam mówi, co podstawiamy za jakie zmienne, też nazywamy **podstawieniem**. Ze zmiennymi wiąże się kilka pojęć:

- Dany węzeł w składni abstrakcyjnej jest **wiązaniem** (ang. *binder*), jeśli wprowadza nową zmienną, która ma jakiś **zasięg** (ang. *scope*). Dla przykładu, w racketowym wyrażeniu

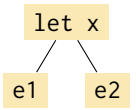
  ```
  (let ([x e1]) e2)
  ```

  konstrukcja `let` **wiąże** zmienną `x` w wyrażeniu `e2`. Innymi słowy, wystąpienia zmiennej `x` na zewnątrz tego `let`-a i w wyrażeniu `e1` oznaczają inną zmienną, która akurat ma taką samą nazwę. Mówi się, że ten `x` zaraz za otwierającym nawiasem kwadratowym to **wystąpienie wiążące**, ale

nie jest to wystąpienie w tradycyjnym znaczeniu tego słowa, bo ten x nie oznacza miejsca, za które coś może zostać podstawione – np. poniższy fragment nie jest poprawnym programem w Rackecie:

```
(let ([(+ 2 2) e1]) e2)
```

Żeby to podkreślić, rysując drzewo składni abstrakcyjnej nie damy wystąpieniom wiążącym swojego własnego węzła, tylko będą wymienione przy wiązaniu. Na przykład:



- Zmienna jest **wolna** (ang. *free*) w jakimś wyrażeniu, jeśli nie jest w nim związana. Terminem **program** będziemy nazywać wyrażenia bez zmiennych wolnych. Proszę zwrócić uwagę, że bycie wolnym jest pojęciem czysto syntaktycznym, tzn. racketowe wyrażenie

  ```
  (+ x y)
  ```

  zawiera trzy zmienne wolne: +, x i y. Prawdą jest, że + jest zdefiniowany w bibliotece standardowej i w związku z tym nie wydaje się aż tak wolny jak x i y, ale to nie ma znaczenia, bo *w tym wyrażeniu* nie jest on nigdzie związany.

- Jeśli podwyrażenie wiąże zmienną, która jest już związana, nazywamy to **przesłanianiem**, np.

  ```
  (let ([x 2]) (let ([x 3]) x))
  ```

  Wartość powyższego wyrażenia to 3, bo zmienna jest zawsze przypisana do *najbliższego* wiązania na ścieżce w drzewie składni abstrakcyjnej od wystąpienia do korzenia.

- Dwa wyrażenia są **α-równoważne** jeśli różnią się tylko nazwami zmiennych związanych.

  Przykładowo,

  ```
  (let ([x 3])
    (+ x
      (let ([y 2])
        y)))
  ```

  oraz

  ```
  (let ([z 3])
    (+ z
      (let ([x 2])
        x)))
  ```

  są α-równoważne. Zwykle uznajemy, że dwa α-równoważne wyrażenia są

nierozróżnialne (czyli że pracujemy *modulo* α-równoważność), ale gdy implementujemy jakieś narzędzie pracujące na składni abstrakcyjnej, musimy sami zadbać o to, żeby tak było (a może nie, patrz sekcja 7.7).

- Gdy zbyt śmiało poczynamy sobie z podstawieniem, może dojść do niebezpiecznego zjawiska nazywanego **przechwytywaniem zmiennych**. Jeśli podstawiamy za zmienną `x` wyrażenie ze zmienną wolną `y` w wyrażeniu, w którym `y` jest zmienną związaną, a `x` jest w zasięgu wiązania zmiennej `y`, wiązanie może przypisać sobie tą zmienną. Przykładowo, rozważmy racketowe wyrażenie

  ```
  (+ y 2)
  ```

  i podstawmy je za zmienną `x` w wyrażeniu

  ```
  (let ([y 2]) (+ x y))
  ```

  to dostaniemy

  ```
  (let ([y 2]) (+ (+ y 2) y))
  ```

  Ale zaraz, zaraz! Przecież na pewno `y` w pierwszym wyrażeniu to nie ten sam `y` w drugim wyrażeniu. Przecież chcemy pracować modulo α-równoważność, więc chcemy, by nie robiło nam różnicy czy podstawiamy w wyrażeniu powyżej czy w wyrażeniu

  ```
  (let ([z 2]) (+ x z))
  ```

  A po podstawieniu, to wyrażenie nie sprawia problemów. Dlatego należy pamiętać, żeby w takich sytuacjach mówić, że robi się **podstawienie unikające przechwytywania zmiennych** (ang. *capture-avoiding substitution*), które, jeśli trzeba, robi odpowiednie α-przemianowanie zmiennych związanych w wyrażeniu, w którym podstawiamy. Proszę uważać: gdy robimy matematykę przy tablicy, można sobie zrobić założenie, że nic nie zostanie przechwycone, a gdy implementujemy, musimy uważać. Na szczęście zwykle będziemy podstawiać programy, więc przechwycenie nie będzie nam grozić.

7.2. Reprezentacja składni abstrakcyjnej z wiązaniem zmiennych

Czas rozbudować wyrażenia arytmetyczne o let-wyrażenia i zmienne. Dla prostoty, nasze let-y będą zawierały zawsze dokładnie jedną lokalną definicję.

Zmienne będziemy reprezentować jako symbole. Z implementacyjnego punktu widzenia nie chcemy wiele więcej od zmiennych niż być w stanie sprawdzać czy dwie zmienne są równe. Do porównywania symboli służy procedura `eq?`.

Samo let-wyrażenie to węzeł w drzewie składni abstrakcyjnej, które przechowuje trzy elementy: nazwę wiązanej zmiennej (czyli jakiś racketowy symbol) i dwa podwyrażenia: to, które definiujemy, i to, w którym możemy użyć wprowadzanej właśnie zmiennej.

Do dotychczasowej definicji dodajemy dwie nowe struktury odpowiadające nowym rodzajom węzłów: `var-expr` i `let-expr`. Nie chcemy ich nazywać `var` i `let`, żeby uniknąć konfliktów z tak nazywającymi się formami specjalnymi w Rackecie. Oczywiście uzupełniamy też predykat, który definiuje, czym jest poprawnie skonstruowane wyrażenie w naszym języku.

let-subst.rkt

```
7  (struct const    (val)      #:transparent)
8  (struct binop    (op l r)   #:transparent)
9  (struct var-expr (id)       #:transparent)
10 (struct let-expr (id e1 e2) #:transparent)
11
12 (define (expr? e)
13   (match e
14     [(const n) (number? n)]
15     [(binop op l r) (and (symbol? op) (expr? l) (expr? r))]
16     [(var-expr x) (symbol? x)]
17     [(let-expr x e1 e2) (and (symbol? x) (expr? e1) (expr?
           e2))]
18     [_ false]))
```

Dla przykładu, wyrażenie, które w Rackecie zapisalibyśmy jako

```
(let ([x 2]) (let ([y 3]) (+ x y)))
```

reprezentowane jest jako następująca wartość:

```
(let-expr 'x (const 2)
  (let-expr 'y (const 3)
    (binop '+ (var-expr 'x) (var-expr 'y))))
```

Proszę zauważyć, że nie stawiamy konstruktora `var-expr` w miejscu wiązania, a jedynie „goły” symbol: tak jak mówiliśmy, to nie jest podwyrażenie!

Znów przydałaby się nam składnia konkretna i znów zrobimy ją tak, by wyglądała jak ta w Rackecie, więc proszę zachować czujność i nie mylić, który

program jest napisany w którym języku. Zrobimy tylko małe uproszczenie: skoro każde let-wyrażenie ma tylko jedną definicję (a nie, jak w Rackecie, całą ich listę), oszczędzimy sobie nawiasów i zamiast `(let ([x e1]) e2)` będziemy pisać `(let [x e1] e2)`:

let-subst.rkt

```
20 (define (parse q)
21   (cond
22     [(number? q) (const q)]
23     [(symbol? q) (var-expr q)]
24     [(and (list? q) (eq? (length q) 3) (eq? (first q) 'let))
25      (let-expr (first (second q))
26                (parse (second (second q)))
27                (parse (third q)))]
28     [(and (list? q) (eq? (length q) 3) (symbol? (first q)))
29      (binop (first q)
30             (parse (second q))
31             (parse (third q)))]))
```

Przykładowo:

```
> (parse '(let (x 2) (let (y 3) (+ x y))))
(let-expr 'x (const 2)
  (let-expr 'y (const 3)
    (binop '+ (var-expr 'x) (var-expr 'y))))
```

Proszę zwrócić uwagę, że procedura `parse` jest trudna do odcyfrowania ze względu na złożone operacje na listach. Właśnie po to definiujemy składnię za pomocą struktur, żeby nie musieć pisać w tym stylu cały czas.

7.3. Let-wyrażenia w modelu podstawieniowym

Czas zadać naszej nowej składni semantykę przez zdefiniowanie ewaluatora. Najpierw musimy zdefiniować czym są wartości. Ale one nie zmieniają się – wyrażenia arytmetyczne z let-wyrażeniami nadal obliczają się do liczb, jedynie obliczenie może spowodować więcej błędów. Wcześniej błąd wywoływało jedynie dzielenie przez zero, teraz także użycie niezdefiniowanej (wolnej!) zmiennej. Zakładając jednak, że ewaluujemy programy, a nie dowolne wyrażenia, ten błąd nie wystąpi.

Jaka jest semantyka let-wyrażenia `(let [x e1] e2)`? Zakładając **gorliwą ewaluację** (co będziemy robić domyślnie), najpierw obliczamy wartość wyrażenia `e1`, a potem podstawiamy wynik za zmienną `x` w wyrażeniu `e2` i obliczamy

wartość tak powstałego wyrażenia – i to jest nasza końcowa wartość. Zaczynamy więc od zdefiniowania procedury, która umie podstawić wyrażenie `e1` za zmienną `x` w wyrażeniu `e2`.

let-subst.rkt

```
39 (define (subst e1 x e2)
40   (match e2
41     [(var-expr y) (if (eq? x y) e1 (var-expr y))]
42     [(const n) (const n)]
43     [(binop op l r)
44      (binop op (subst e1 x l) (subst e1 x r))]
45     [(let-expr y e3 e4)
46      (let-expr y (subst e1 x e3)
47                  (if (eq? x y) e4 (subst e1 x e4)))]))
```

STOP! Uważny czytelnik zauważy natychmiast, że coś tu nie gra! Przecież powiedzieliśmy, że chcemy podstawić **wartość** wyrażenia `e1`, a procedura `subst` służy do podstawiania **wyrażeń**, a nie wartości. Wartości jako takie nie są częścią składni, więc mamy problem. Rozwiązujemy go dokonując **reifikacji**, czyli zanurzając uzyskaną wartość w świecie wyrażeń, tworząc wyrażenie, które w trywialny sposób obliczy się do wartości, którą uzyskaliśmy. Czyli jeśli wyrażenie `e1` oblicza się do wartości `5`, to w wyrażeniu `e2` podstawiamy wyrażenie `(const 5)`.

A jak interpretujemy wyrażenie `var-expr`? Wcale. Jeśli zaczynaliśmy od programu (czyli wyrażenia bez zmiennych wolnych), to nigdy nie będziemy musieli interpretować zmiennej. Skoro zmienna `x` jest związana w wyrażeniu `e2` przez jakieś let-wyrażenie `(let [x e1] e2)`, to zostanie za nią podstawiony wynik ewaluacji wyrażenia `e1` zanim przejdziemy do ewaluacji wyrażenia `e2`. Więc ewaluując `e2`, nigdy nie napotkamy tej zmiennej. Uzbrojeni w tę wiedzę, możemy skreślić ewaluator:

let-subst.rkt

```
58 (define (value? v) (number? v))
59 ...
64 (define (eval e)
65   (match e
66     [(const n) n]
67     [(binop op l r) ((op->proc op) (eval l) (eval r))]
68     [(let-expr x e1 e2)
69      (eval (subst (const (eval e1)) x e2))]))
```

7.4. Let-wyrażenia w modelu środowiskowym

Model podstawieniowy ma pewne wady. Po pierwsze, cała ta zabawa z reifikacją wydaje się mocno podejrzana. W przypadku liczb jest łatwo, ale czy mamy pewność, że to rozwiązanie skaluje się do bardziej skomplikowanych typów danych? Po drugie, za każdym razem jak ewaluujemy jakieś let-wyrażenie, musimy przejrzeć całą resztę programu, żeby wykonać podstawienie. Chociaż sprawami wydajnościowymi raczej nie zajmujemy się na tym przedmiocie, nie możemy sobie pozwolić na takie marnotrawstwo.

Dlatego proponujemy alternatywny model: obliczenia ze środowiskiem. Nie tylko nie trzeba w nim wykonywać podstawień, ale ma też wiele innych zalet, które omówimy w sekcji 7.5.

W tym konkretnym przypadku, **środowisko** (ang. *environment*) to struktura danych, która jest dodatkowym argumentem ewaluacji wyrażeń, a która przechowuje wartości zmiennych wolnych tego wyrażenia. To znaczy: mając dane wyrażenie `(+ x y)` nie wiemy, jaka jest jego wartość, bo nie wiemy, jaka jest wartość zmiennych `x` i `y`. Chyba że towarzyszy nam środowisko, które zwyczajnie powie nam, jakie to wartości.

Środowisko będziemy reprezentować jako listę asocjacyjną, to znaczy listę par klucz–wartość. Zwróćcie Państwo uwagę na to, że gdy dodajemy zduplikowany klucz do środowiska, nie usuwamy starego. Ponieważ wyszukiwanie następuje od lewej do prawej[1], znajdziemy ten element, co trzeba, a fakt, że nie usuwamy starego klucza, a przesłaniamy go, będzie miał kapitalne znaczenie w sekcji 7.7.

let-env.rkt

```
39 (struct environ (xs))
40
41 (define env-empty (environ null))
42 (define (env-add x v env)
43   (environ (cons (cons x v) (environ-xs env))))
44 (define (env-lookup x env)
45   (define (assoc-lookup xs)
46     (cond [(null? xs) (error "Unknown identifier" x)]
47           [(eq? x (car (car xs))) (cdr (car xs))]
48           [else (assoc-lookup (cdr xs))]))
49   (assoc-lookup (environ-xs env)))
```

Jak wykorzystać takie środowisko do ewaluacji wyrażeń? Definiujemy dodatkową procedurę, `eval-env`, która bierze dwa argumenty: wyrażenie i

[1] Czy w kulturach, które używają języków pisanych od prawej do lewej, listy skierowane są w lewo?

środowisko, które mówi nam, jakie wartości przypisane są do zmiennych wolnych wyrażenia. Jak w takim modelu obliczamy wartość wyrażenia `(let [x e1] e2)`? Najpierw liczymy wartość wyrażenia `e1`, a potem wartość wyrażenia `e2`, ale w innym środowisku – dodajemy informację o wartości przypisanej do zmiennej `x`. Tym razem musimy oczywiście interpretować także zmienne poprzez wyszukiwanie przypisanej im wartości w środowisku. Tak więc nasz ewaluator można napisać tak:

let-env.rkt

```
61 (define (eval-env e env)
62   (match e
63     [(const n) n]
64     [(binop op l r) ((op->proc op) (eval-env l env)
65                                    (eval-env r env))]
66     [(let-expr x e1 e2)
67      (eval-env e2 (env-add x (eval-env e1 env) env))]
68     [(var-expr x) (env-lookup x env)]))
69
70 (define (eval e) (eval-env e env-empty))
```

7.5. Inne obliczenia ze środowiskiem

Warto zaznajomić się ze schematem obliczeń ze środowiskiem, bo może on posłużyć do realizacji wielu innych zadań związanych z obróbką programów. Trzeba jednak najpierw zgeneralizować pojęcie środowiska: nie musi ono przechowywać akurat *wartości* przypisanych do zmiennych, a dowolne inne informacje, jakiekolwiek są nam potrzebne. Nie przywiązujmys się też do konkretnej reprezentacji środowisk, traktując pojęcie „obliczenia ze środowiskiem" jako rodzaj wzorca projektowego.

Jako przykład omówimy procedurę, która bierze jako swój argument wyrażenie, a zwraca listę zmiennych wolnych tego wyrażenia. Jest to bardzo istotna operacja, potrzebna do implementacji jakiegokolwiek kompilatora i wydajnego interpretera języka funkcyjnego – dowiemy się dlaczego na kolejnym wykładzie. Do tej operacji będziemy potrzebować środowisko, które nie przechowuje żadnej informacji o zmiennej poza tym, czy dana zmienna jest w środowisku czy nie. Takie środowisko reprezentujemy za pomocą struktury danych o nazwie **zbiór**:

free.rkt

```
3  (require racket/set)
4  ...
41 (define env-empty          (set))
42 (define (env-add x env)    (set-add env x))
43 (define (env-lookup x env) (set-member? env x))
```

Proszę zwrócić uwagę, że choć są w nim te same procedury, interfejs środowiska jest trochę inny. Po pierwsze dlatego, że nie musimy podawać wartości (oczywiście), ale także dlatego, że procedura `env-lookup` w przypadku braku zmiennej w środowisku nie zgłasza globalnego błędu, a wartość `#f`.

Zdefiniujemy procedurę obliczającą listę zmiennych wolnych jak poniżej. Używamy procedury dodatkowej, która używa środowiska, a zwraca zbiór zmiennych. Zbiór jest potem konwertowany na listę w głównej procedurze `free-vars`.

free.rkt

```
45 (define (free-vars-env e env)
46   (match e
47     [(const n) (set)]
48     [(binop op l r)
49      (set-union (free-vars-env l env)
50                 (free-vars-env r env))]
51     [(let-expr x e1 e2)
52      (set-union (free-vars-env e1 env)
53                 (free-vars-env e2 (env-add x env)))]
54     [(var-expr x)
55      (if (env-lookup x env)
56          (set) (list->set (list x)))]))
57
58 (define (free-vars e)
59   (set->list (free-vars-env e env-empty)))
```

Jak działa ta procedura? Z grubsza: w każdym węźle wyrażenia obliczamy zbiór zmiennych wolnych, a potem sumujemy te zbiory. Gdy trafimy na wystąpienie zmiennej, musimy zdecydować, czy jest ona wolna czy nie. Robimy to na podstawie środowiska, które przechowuje właśnie tę informację dzięki temu, że odpowiednio rozszerzamy je w przypadku wyrażenia `let`.

Ważną dla wyrobienia sobie odpowiednich intuicji jest obserwacja, że środowisko nie jest jakąś globalną wartością modyfikowaną przez całe działanie procedury, np.

```
> (free-vars (parse '(+ (let [x 2] y) (let [y 3] x))))
'(y x)
```

Ten przykład ma ilustrować fakt, że każda gałąź operatora + tak naprawdę ma własną „kopię" środowiska, którą modyfikuję niezależnie od innych analizowanych gałęzi drzewa składniowego.

7.6. Ewaluacja gorliwa i leniwa

A teraz odkryjemy, że nie wszystko jest łatwe, a naiwna implementacja potrafi być – co tu dużo mówić – nieprawidłowa. Przykładem będzie próba sprawienia sobie leniwej ewaluacji let-wyrażeń.

Leniwa strategia obliczeń oznacza, że wartość zmiennej liczymy dopiero wtedy, gdy jest ona do czegoś potrzebna. Jak to zrobić w modelu środowiskowym? Spróbujemy tak: skoro środowisko może przechowywać nie tylko wartości, niech przechowuje wyrażenia, których wartość będziemy obliczać dopiero, gdy ktoś poprosi środowisko o wartość tej zmiennej. Przykładowo, żeby obliczyć wartość wyrażenia `(let [x e1] e2)` dodajemy do środowiska pod kluczem `x` wyrażenie `e1`, a wynikiem całości jest wynik ewaluacji wyrażenia `e2` w tak zmienionym środowisku. Żeby obliczyć wartość wyrażenia będącego zmienną, należy pobrać ze środowiska odpowiadające jej wyrażenie, i wartość tego wyrażenia to wartość całości. O tak:

let-lazy.rkt

```
61 (define (eval-env e env)
62   (match e
63     ...
66     [(let-expr x e1 e2)
67      (eval-env e2 (env-add x e1 env))]
68     [(var-expr x) (eval-env (env-lookup x env) env)]))
```

Wygląda jakby działało, np. wyrażenie

```
(eval (parse '(let [x (+ 2 2)]
                (let [y (/ 5 0)]
                  x))))
```

nie powoduje przerwania interpretacji błędem „dzielenie przez zero", bo wartość zmiennej y nie jest do niczego potrzebna, więc nasz interpreter nigdy nie będzie próbował wykonać zakazanego dzielenia. Ale, ale:

```
(eval (parse '(let [x 2]
               (let [y (+ x x)]
               (let [x 0]
                 y)))))
```

To wyrażenie oblicza się do `0`, a nie do spodziewanego `4`. Coś poszło nie tak! Zastanów się, co. Teraz jeszcze nie czas ujawnić, jak rozwiązać ten problem. Wrócimy do tematu na kolejnym wykładzie.

7.7. Notatka o alternatywnych sposobach reprezentacji wiązań

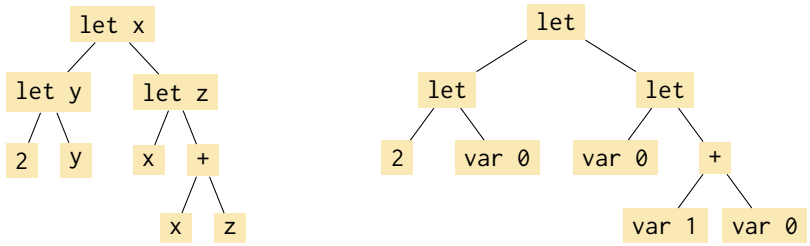
Jedną z podstawowych zasad udanego programowania jest to, że jeśli chcemy, by nasze dane zachowywały jakiś niezmiennik, niech reprezentacja zachowuje go *by construction* (o ile się da). W szczególności, jeśli chcemy programować modulo pewna równoważność danych, niech równoważne dane mają tę samą reprezentację. W naszym przypadku, skoro chcemy nigdy nie rozróżniać α-równoważnych programów, to może da się zrobić tak, by dwa α-równoważne programy miały tę samą składnię abstrakcyjną? Da się, używając **indeksów de Bruijna**[2].

Idea jest dość prosta: wiązanie nie pamięta w żaden sposób nazwy wiązanej zmiennej, a wystąpienie zmiennej związanej reprezentowane jest jako liczba oznaczająca ile konstrukcji wiążących na ścieżce od tej zmiennej do korzenia drzewa trzeba minąć, by trafić na wprowadzające tą zmienną wiązane. Przykładowo, rozważmy wyrażenie, które w naszej składni konkretnej moglibyśmy zapisać tak:

```
'(let [x (let [y 2] y)]
   (let [z x] (+ x y)))
```

Na poniższym rysunku zamieszczono jego reprezentację w formie dotychczas przez nas rozważanej (po lewej) i przy użyciu indeksów de Bruijna (po prawej). By odróżnić stałe liczbowe od zmiennych, zmienne oznaczone są etykietą `var`:

[2]Nicolaas Govert de Bruijn (1918–2012), matematyk holenderski



Proszę zwrócić uwagę na to, że zmienna `x` reprezentowana jest raz przez indeks `0` a raz przez `1`. To dlatego, że `x` w argumencie dodawania oddziela od wiążącego go `let`-a jeszcze jeden `let` (ten wprowadzający zmienną `y` w składni konkretnej).

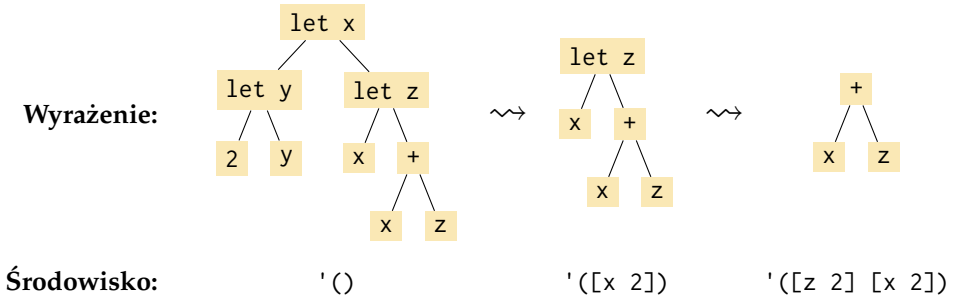
Skoro indeksy de Bruijna są takie fajne, czemu od razu nie formalizowaliśmy składni abstrakcyjnej przy jej użyciu? Cytując Mac Lane'a[3]: *One abstraction at a time!* W reszcie tych notatek będziemy jednak używać składni z nazwami, pozostawiając indeksy de Bruijna jedynie jako ciekawostkę. W każdym razie, formalizujemy:

let-de-bruijn.rkt

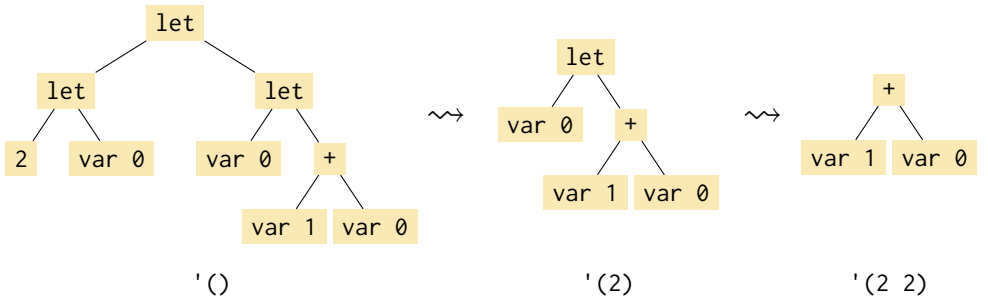
```
7  (struct const    (val)    #:transparent)
8  (struct binop    (op l r) #:transparent)
9  (struct var-expr (n)      #:transparent)
10 (struct let-expr (e1 e2)  #:transparent)
11
12 (define (expr? e)
13   (match e
14     [(const n) (number? n)]
15     [(binop op l r) (and (symbol? op) (expr? l) (expr? r))]
16     [(var-expr n) (number? n)]
17     [(let-expr e1 e2) (and (expr? e1) (expr? e2))]
18     [_ false]))
```

Zmienne wolne można reprezentować jako indeksy większe niż liczba konstrukcji wiążących „nad" zmienną, a środowiska jako listy, które przypisują odpowiednie dane kolejnym takim zmiennym. Przykładowo, ewaluacja we wcześniejszej notacji:

[3]Saunders Mac Lane (1909–2005), amerykański matematyk, współtwórca teorii kategorii



W wersji z indeksami de Bruijna:



Ubierając w słowa, zdejmując najbardziej zewnętrzne wiązanie, wiązane przezeń zmienne zaczynają „wystawać" o jedno oczko, a zmienne wolne wystają o jedno oczko więcej niż wcześniej. Ale dokładamy do środowiska nową wartość, więc wszystkie zmienne wolne wciąż trafiają w swoje miejsca na liście reprezentującej środowisko. Środowisko zatem implementujemy jako listę:

let-de-bruijn.rkt

```
44 (define (n-th n xs)
45   (cond [(null? xs) (error "Unknown variable")]
46         [(eq? n 0) (car xs)]
47         [else (n-th (- n 1) (cdr xs))]))
48
49 (define env-empty  null)
50 (define env-add    cons)
51 (define env-lookup n-th)
```

Ewaluację implementujemy jak wcześniej, z wyjątkiem tego, że nie musimy podawać nazwy zmiennej dodawanej do środowiska. Ponieważ `let` uwalnia

jedną zmienną w wyrażeniu, ale i dodaje jej wartość do środowiska, kolejność zmiennych w środowisku zastępuje nazwy:

let-de-bruijn.rkt

```
63 (define (eval-env e env)
64   (match e
65     ...
68     [(let-expr e1 e2)
69      (eval-env e2 (env-add (eval-env e1 env) env))]
70     [(var-expr n) (env-lookup n env)]))
```