

Wykład 8.

Dane i domknięcia

Do tej pory rozważaliśmy reprezentację i interpretację programów, które składały się z wyrażeń arytmetycznych, zmiennych i let-wyrażeń. To bardzo proste konstrukcje, ale, jak przekonamy się teraz, zawierają w sobie większość koncepcji potrzebnych do zbudowania interpretera prawdziwego języka programowania. Brakuje nam tylko kilku elementów. Zdecydowanie brakuje nam **typów danych** wykraczających poza liczby i to im poświęcony jest ten wykład.

W przypadku każdego typu danych, o który rozszerzymy nasz język, będziemy postępować według tego samego schematu:

1. Rozbudujemy składnię abstrakcyjną i konkretną o nowe konstrukcje wprowadzające, eliminujące i operujące na danym typie danych, np. stałe `true` i `false` to konstruktory typu boolowskiego, a konstrukcja `if` to jego eliminator.
2. Ustalimy, jak reprezentować wartości danego typu, czyli jakie dodatkowe rodzaje odpowiedzi może wyprodukować nasz ewaluator z okazji rozszerzenia języka o nowy typ danych.
3. Rozbudujemy ewaluator tak, by konstrukcje wprowadzające interpretował jako ów nowy rodzaj wartości, a przy okazji konstrukcji eliminujących działał na podstawie tych wartości.

Tak postąpimy przy okazji omawianych poniżej wartości boolowskich, par, list, funkcji. A gdy będziemy już mieć język z liczbami, boolami, parami, listami i funkcjami, to będziemy mieli już prawie całkiem dorosły język programowania!

8.1. Racket w Rackecie (w Rackecie?)

Celem dwóch poprzednich, obecnego i dwóch następnych wykładów jest zabawa interpreterami języków programowania. Ale jakich języków właściwie? Skupimy się na tym, co znamy, i będziemy krok po kroku implementować interpreter języka dość podobnego do Racketa. Ponieważ używamy cytowania do uzyskania „za darmo” składni konkretnej, język będzie przypominał Racketa nawet składniowo.

Może też się wydawać, że pisząc interpreter trochę oszukujemy: będziemy interpretować liczby jako racketowe liczby, wartości boolowskie jako racketowe wartości boolowskie, konstrukcję `if` zaimplementujemy przy użyciu racketowej konstrukcji `if` itd. Czy to oznacza, że jesteśmy ograniczeni do racketopodobnych języków pod względem semantyki i nie nauczymy się pisać interpreterów innych języków, np. imperatywnych albo obiektowych?

Na szczęście nie. Żeby to zademonstrować, na tym i następnych wykładach zrobimy kilka rzeczy wykraczających poza podzbiór Racketa zaprezentowany na pierwszej tercji metod programowania: leniwość, dynamiczne wiązanie zmiennych, język z niedeterminizmem. Napišemy też interpreter minimalistycznego języka imperatywnego WHILE. Wierzmy, że po takiej dawce będą Państwo w stanie sami wymyślić jak zinterpretować język obiektowy.

Więc dlaczego Racket? Naszym celem jest napisanie interpretera języka programowania, w którym dałoby się napisać... interpreter języka programowania. A do pisania interpreterów podobno najlepiej nadają się języki funkcyjne, więc taki interpretujemy.

8.2. Wartości boolowskie

Rozpoczynamy od wyrażeń boolowskich. Ktoś mógłby pomyśleć, że nie są one zbyt skomplikowane w porównaniu z wyrażeniami arytmetycznymi, bo można o nich myśleć jako o jednobitowych liczbach. Ale, w przeciwieństwie do wyrażeń arytmetycznych, tym razem potrzebujemy też **eliminatory** tego typu, czyli konstrukcji `if`. Żeby dodać do języka wartości boolowskie, potrzebujemy następujących konstrukcji:

- Stałych `true` i `false` w składni konkretnej i abstrakcyjnej,
- Operatorów porównania (`=`, `<`, `>`, `<=`, `>=`, itp.) i logicznych (`and`, `or`, `not`),
- Wyrażeń `if`.

Zacznijmy od składni abstrakcyjnej. Żeby reprezentować stałe `true` i `false`, użyjemy tej samej konstrukcji, co w przypadku liczb: skoro stałe liczbowe jako element składni reprezentujemy przy użyciu racketowych liczb, zadziałajmy analogicznie w przypadku stałych boolowskich i użyjmy znów struktury `const`, tylko tym razem pozwólmy argumentowi być albo liczbą (`number?`), albo boolem (`boolean?`). Operatory porównania i logiczne (z wyjątkiem `not`, do którego wrócimy na liście zadań) możemy reprezentować tak samo jak binarne operatory arytmetyczne: przy użyciu struktury `binop`. Dla `if`-a definiujemy nową strukturę, `if-expr`, która nie jest bardzo zaskakująca. Ma ona trzy argumenty,

które reprezentują kolejno: wyrażenie obliczające się do wartości boolowskiej, gałąź „then” i gałąź „else”:

boolean.rkt

```

11 (struct const (val) #:transparent)
12 (struct binop (op l r) #:transparent)
13 (struct var-expr (id) #:transparent)
14 (struct let-expr (id e1 e2) #:transparent)
15 (struct if-expr (eb et ef) #:transparent)
16
17 (define (expr? e)
18   (match e
19     [(const n) (or (number? n) (boolean? n))]
20     [(binop op l r) (and (symbol? op) (expr? l) (expr? r))]
21     [(var-expr x) (symbol? x)]
22     [(let-expr x e1 e2)
23      (and (symbol? x) (expr? e1) (expr? e2))]
24     [(if-expr eb et ef)
25      (and (expr? eb) (expr? et) (expr? ef))]
26     [_ false]))

```

Składnię konkretną bierzemy wprost z Racketa:

boolean.rkt

```

11 (define (parse q)
12   (cond
13     ...
31     [(eq? q 'true) (const true)]
32     [(eq? q 'false) (const false)]
38     [(and (list? q) (eq? (length q) 4) (eq? (first q) 'if))
39      (if-expr (parse (second q))
40               (parse (third q))
41               (parse (fourth q)))]
45     ... ))

```

Działa? Przetestujmy:

```

> (parse '(if (> 3 2) false 17))
(if-expr (binop '> (const 3) (const 2))
  (const #f)
  (const 17))

```

Czas na semantykę. Zaczynamy, tak jak mówiliśmy powyżej, od zastanowienia się, jakie są możliwe wartości. Skoro w wyrażeniach arytmetycznych reprezentowaliśmy wartości liczbowe jako racketowe liczby, to może wartości boolowskie reprezentujemy jako boole:

boolean.rkt

```
69 (define (value? v)
70   (or (number? v)
71       (boolean? v)))
```

Pora na ewaluator. Jeśli chodzi o stałe, to proszę zwrócić uwagę, że już przy okazji wyrażen arytmetycznych zrobiliśmy sprytny zabieg: reprezentujemy wartość stałej w składni abstrakcyjnej w taki sam sposób, jak wartość będącą wynikiem (trywialnego) obliczenia tej stałej. Innymi słowy, wyrażenie `(const n)` zawsze interpretujemy jako `n`, niezależnie od tego, jakiego typu jest `n`. Proszę jednak pamiętać, że koncepcyjnie reprezentacja stałych w składni i reprezentacja wartości to dwie różne rzeczy i mogą być zaimplementowane przy użyciu dwóch różnych typów danych (patrz: lista zadań).

Musimy też być w stanie interpretować większą liczbę operatorów binarnych. Wystarczy rozbudować procedurę `op->proc` o odpowiednie operatory, które działają na odpowiednich rodzajach wartości.

Pozostaje interpretacja wyrażen `if`. Nie jest ona trudna: obliczamy wartość pierwszego argumentu struktury reprezentującej `if`-wyrażenie i na podstawie tej wartości wybieramy, czy obliczamy drugie czy trzecie wyrażenie. Ponieważ sprytnie wybraliśmy reprezentację wartości przez boole, możemy interpretować nasze `if`-wyrażenia przy użyciu racketowych `if`-wyrażen.

Nie trzeba zmieniać pozostałych elementów ewaluatora. W szczególności implementacja środowisk pozostaje bez zmian, bo wiemy, że środowiska przechowują wartości, jakiegokolwiek typu te wartości by nie były.

boolean.rkt

```
72 (define (op->proc op)
73   (match op ['+ +] ['- -] ['* *] ['/ /] ['% modulo]
74           ['= =] ['> >] ['>= >=] ['< <] ['<= <=]
75           ['and (lambda (x y) (and x y))]
76           ['or  (lambda (x y) (or  x y))]))
77
78 (define (eval-env e env)
79   (match e
80     [(const n) n]
81     [(binop op l r) ((op->proc op) (eval-env l env)
```

```

82                                     (eval-env r env))]
83   [(let-expr x e1 e2)
84     (eval-env e2 (env-add x (eval-env e1 env) env))]
85   [(var-expr x) (env-lookup x env)]
86   [(if-expr eb et ef) (if (eval-env eb env)
87                           (eval-env et env)
88                           (eval-env ef env)))]

```

W działaniu:

```

> (define program
  '(if (or (< (% 123 10) 5)
        true)
      (+ 2 3)
      (/ 2 0)))
> (eval (parse program))
5

```

8.3. Pary

Znów postąpimy według schematu „rozbuduj składnię – zdefiniuj wartości – rozbuduj ewaluator”. Oczywiście będziemy rozbudowywać plik **boolean.rkt**, żeby dokładać do naszego języka więcej i więcej. Nasze pary będą działać jak te w Rackecie.

Składnia:¹

pair.rkt

```

16 (struct cons-expr (e1 e2) #:transparent)
17 (struct car-expr  (e)      #:transparent)
18 (struct cdr-expr  (e)      #:transparent)
19
20 (define (expr? e)
21   (match e
22     ...
29     [(cons-expr e1 e2) (and (expr? e1) (expr? e2))]
30     [(car-expr e) (expr? e)]
31     [(cdr-expr e) (expr? e)] ... ))

```

¹Jeśli składnia konkretna nie różni się od składni Racketa, pomijamy w tych notatkach definicje procedury `parse` (jest ona oczywiście zaimplementowana w odpowiednim pliku `.rkt` dostępnym na SKOS-ie).

Wartości:

pair.rkt

```
82 (define (value? v)
83   (or (number? v)
84       (boolean? v)
85       (and (pair? v) (value? (car v)) (value? (cdr v)))))
```

Ewaluator:

pair.rkt

```
93 (define (eval-env e env)
94   (match e
95     ...
104   [(cons-expr e1 e2) (cons (eval-env e1 env)
105                             (eval-env e2 env))]
106   [(car-expr e) (car (eval-env e env))]
107   [(cdr-expr e) (cdr (eval-env e env))])
```

8.4. Listy

Nikogo chyba nie zaskoczy definicja list, bo robimy wszystko na tzw. jedno kopyto. Znów zwyczajnie implementujemy fragment Racketa używając racketowych typów danych. Ponieważ w Rackecie listy budujemy z par i nulla, wystarczy dodać do języka wartość null. Dodatkowo, zdecydowanie potrzebujemy umieć odróżnić parę od nulla, więc dodajemy także konstrukcję null?.

Składnia:

list.rkt

```
19 (struct null-expr () #:transparent)
20 (struct null?-expr (e) #:transparent)
21
22 (define (expr? e)
23   (match e
24     ...
34   [(null-expr) true]
35   [(null?-expr e) (expr? e)] ... ))
```

Wartości:

list.rkt

```
87 (define (value? v)
88   (or ...
91     (null? v)))
```

Ewaluator:

list.rkt

```
99 (define (eval-env e env)
100   (match e
101     ...
114     [(null-expr) null]
115     [(null?-expr e) (null? (eval-env e env))]))
```

Dla przykładu:

```
> (eval (parse '(car (cdr (cons 1 (cons 2 (cons 3 null)))))))
2
```

Test trzeźwości: czemu zamiast powyższego nie możemy napisać prościej:

```
> (eval (parse '(cadr (cons 1 (cons 2 (cons 3 null))))))
2
```




Odpowiedź: Bo w naszym języku **nie ma** konstrukcji cadr. Nie mylmy naszego języka z Racketem!

8.5. Interludium: Sortowanie przez generowanie sortowania przez wstawianie

Ale nuda ten nasz język! Nie mamy w nim funkcji, nie mówiąc już o funkcjach rekurencyjnych. Co nam po listach, skoro nawet nie możemy rekurencyjnie przejść się po liście!

Zarzut ten odpieramy mówiąc, że może i język jest prosty, ale ma wspaniałą zaletę: jest to język zanurzony w Rackecie! To znaczy, że możemy (niektóre) wartości racketowe zmieniać na programy w naszym języku, a wynik działania naszego ewaluatora to poprawna racketowa wartość. Poćwiczmy więc zrozumienie, co żyje na którym poziomie i w którym świecie.

Najpierw trochę utrudnimy sobie życie wprowadzając nowy rodzaj cytowania: **kwazicytowanie**. Kwazicytowanie to takie cytowanie z dziurką albo

kilkoma dziurkami, w które możemy wstawić racketowe wyrażenia. Takie cytowanie rozpoczyna się od grawisu, czyli znaku ``` (U+0060), który znajduje się na klawiszu , który na klawiaturze znajdują Państwo zwykle na lewo od klawisza  albo (na maku) od . To inny znak niż używany do zwykłego cytowania apostrof `'`. Dziurkę robimy stawiając przed wyrażeniem przecinek, np.

```
> `(dzien-dobry 1 ,(+ 2 3) 8 ,(car '(1 2 3)))
'(dzien-dobry 1 5 8 1)
```

Dzięki kwazycytowaniu możemy napisać procedurę, która wyrazi racketową listę jako kawałek składni konkretnej naszego języka:

list.rkt

```
130 (define (reify xs)
131   (cond [(null? xs) 'null]
132         [else `(cons ,(car xs) ,(reify (cdr xs)))]))
```

Wówczas:

```
> (reify '(1 2 3 4))
'(cons 1 (cons 2 (cons 3 (cons 4 null))))
```

Teraz użyjemy naszego ewaluatora, żeby napisać niebanalną procedurę sortowania list w Rackecie. Będzie to *insertion sort* zaimplementowany w naszym języku. Ale skoro nie mamy do dyspozycji funkcji rekurencyjnych, w naszym języku będzie zaimplementowane **pełne rozwinięcie** sortującej procedury w formie składni konkretnej generowanej przez procedurę w Rackecie. Ponieważ możemy sprawdzić (w Rackecie) długość listy wejściowej, wiemy na jaką głębokość trzeba odwinąć rekursję. Poniżej znajduje się implementacja generatorów programu do wstawiania i całego sortowania, gdzie n to głębokość rekursji, x to wstawiany element, a xs to lista (a raczej zmienna reprezentująca listę):

list.rkt

```
134 (define (make-insert n x xs)
135   (cond [(= n 0) `(cons ,x null)]
136         [else `(if (< ,x (car ,xs))
137                   (cons ,x ,xs)
138                   (cons
139                     (car ,xs)
140                     ,(make-insert (- n 1) x `(cdr ,xs)))))]))
```


list.rkt

```

142 (define (make-insertion-sort n xs)
143   (cond [(= n 0) xs]
144         [else (make-insert (- n 1)
145                             `(car ,xs)
146                             (make-insertion-sort
147                               (- n 1)
148                               `(cdr ,xs))))]))

```

Proszę zwrócić uwagę, że gdybyśmy pozbyli się wszystkich grawisów i przecinków, a także wywoływali procedurę sortującą z n równym długości listy xs, dostalibyśmy zwykłą implementację sortowania przez wstawianie! Testujemy:

```

> (make-insert 1 'x 'xs)
'(if (< x (car xs)) (cons x xs) (cons (car xs) (cons x
null)))
> (make-insert 2 'x 'xs)
'(if (< x (car xs))
    (cons x xs)
    (cons
     (car xs)
     (if (< x (car (cdr xs))) (cons x (cdr xs)) (cons (car
(cdr xs)) (cons x null))))))
> (make-insertion-sort 2 'xs)
'(if (< (car xs) (car (cons (car (cdr xs)) null)))
    (cons (car xs) (cons (car (cdr xs)) null))
    (cons (car (cons (car (cdr xs)) null)) (cons (car xs)
null)))

```

Nasze programy dość szybko rosną, np. (make-insertion-sort 4 'xs) generuje program o ponad 2300 węzłach! Ale nie przejmujemy się tym i implementujemy naszą procedurę sortującą w Rackecie. Proszę zwrócić uwagę, że (make-insertion-sort n 'xs) produkuje wyrażenie ze zmienną wolną xs, której przypisujemy zreifikowaną listę podlegającą sortowaniu:

list.rkt

```

150 (define (cool-sort xs)
151   (eval (parse
152         `(let [xs ,(reify xs)]
153           ,(make-insertion-sort (length xs) 'xs))))

```

Podsumowując, napisaliśmy procedurę, która generuje w naszym języku kod sortujący zreifikowaną listę, a następnie uruchamiającą nasz ewaluator, by

uzyskać posortowaną listę. Testujemy:

```
> (cool-sort '(3 2 1 4))
'(1 2 3 4)
> (cool-sort '(3 2 1 5 4))
'(1 2 3 4 5)
> (cool-sort '(3 2 1 5 6 4))
'(1 2 3 4 5 6)
> (cool-sort '(3 7 2 1 5 6 4))
'(1 2 3 4 5 6 7)
> (cool-sort '(3 7 2 1 8 5 6 4))
Interactions disabled, out of memory
```

8.6. Procedury

Ponieważ implementujemy język funkcyjny, na pewno spodziewają się Państwo, że trzeba do niego dodać funkcje — zwane w nomenklaturze racketowej procedurami. Procedury w Rackecie to też typ danych, i my nie potraktujemy go inaczej, czyli wrócimy do naszego schematu „składnia – wartości – interpreter”.

Sztuka projektowania języków programowania to sztuka wyboru. W wypadku procedur wyższych rzędów mamy wybór między automatycznymi częściowymi aplikacjami a procedurami o zmiennej liczbie argumentów. Racket implementuje to drugie podejście, a my wybierzemy pierwsze. Jest to decyzja dość pragmatyczna: procedury z częściową aplikacją mają prostszą składnię, bo wystarczą procedury i aplikacje jednoargumentowe.

Składniową formą wprowadzającą procedurę jest oczywiście lambda-wyrażenie `(lambda (x) e)`. Eliminatorem jest aplikacja do argumentu `(e1 e2)`.

fun.rkt

```
21 (struct app (f e) #:transparent) ; aplikacja funkcji
22 (struct lam (id e) #:transparent) ; lambda
23
24 (define (expr? e)
25   (match e
26     ...
38     [(app f e) (and (expr? f) (expr? e))]
39     [(lam id e) (and (symbol? id) (expr? e))] ... ))
```

Dodatkowo w procedurze `parse` w pliku `fun.rkt` definiujemy lukier syntaktyczny. Definicję

```
(lambda (x y z) e)
```

rozumiemy jako procedurę

```
(lambda (x) (lambda (y) (lambda (z) e)))
```

a aplikację

```
(f x y z)
```

jako

```
((f x) y) z)
```

By zrozumieć, jak działają procedury wyższych rzędów, kluczowa jest definicja wartości. W modelu podstawieniowym Racketa mówiliśmy, że procedury to wartości i już. Ale muszą Państwo uważać, żeby nie wpaść w tę samą pułapkę, w którą wpadliśmy przy okazji próby zdefiniowania leniwych let-wyrażeń. Ewaluator próbowaliśmy napisać tak:

let-lazy.rkt

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

Ale próba obliczenia wyrażenia

```
(let [x 2]
  (let [y (+ x x)]
    (let [x 0]
      y))))
```

dawała nam 0 a nie 4. Dlaczego? Oczywiście dlatego, że próbowaliśmy obliczać ciało definicji zmiennej *y* w środowisku aktualnym w *momencie użycia* zmiennej, a chcielibyśmy użyć środowiska z *momentu definicji*.

To samo dotyczy procedur, których ciała zawierają zmienne wolne. Przykładowo, spójrzmy jaka jest wartość następującego wyrażenia w Rackecie:

```
> (let ([x 4])
      (let ([f (lambda (y) (+ x y))])
        (let ([x 0])
          (f 10)))))
14
```

Gdy obliczamy wartość aplikacji funkcji *f* do argumentu 10, musimy obliczyć wartość ciała procedury wiedząc, że *y* przyjmuje właśnie tę wartość 10. Ale jaką wartość przyjmuje zmienna *x*? Jeśli założymy, że rzeczywiście pracujemy modulo α -równoważność, powyższe wyrażenie jest równoważne poniższemu wyrażeniu:

```
> (let ([x 4])
      (let ([f (lambda (y) (+ x y))])
        (let ([z 0])
          (f 10)))))
14
```

To rozwiązuje problem, bo mamy tylko jedną zmienną *x* w całym programie, więc wiadomo, że w ciele procedury *f* zmienna *x* musi przyjąć wartość taką, jaką miała w środowisku *w momencie definiowania* procedury, czyli 4.

Ale skąd wiadomo w momencie wywołania procedury, jakie wartości miały jej zmienne wolne w momencie definiowania tej procedury? Odpowiedź jest genialna w swej prostocie: zapamiętać te wartości razem z ciałem procedury. W ten sposób, gdy wywołujemy procedurę, możemy sobie „przypomnieć” co oznaczają zmienne wolne w jej ciele. Dlatego lambda-wyrażenia obliczają się do paczek zawierających ciało procedury i środowisko aktualne w momencie definicji. Paczkę taką nazywamy **domknięciem** (ang. *closure*).

Prócz zmiennych zdefiniowanych w środowisku, ciało procedury ma jeszcze jedną zmienną wolną: argument procedury. Nazwę argumentu też zamykamy w domknięciu, żeby móc podczas interpretacji zbudować środowisko, w którym ewaluujemy ciało. Domknięcie reprezentujemy przy użyciu poniższej struktury *clo*, gdzie *id* to nazwa zmiennej formalnej będącej argumentem procedury, *e* to wyrażenie będące ciałem procedury, a *env* to środowisko z momentu definicji:

fun.rkt

```
107 (struct clo (id e env) #:transparent)
108
109 (define (value? v)
110   (or ...
114     (clo? v)))
```

Przykładowo, gdybyśmy przetłumaczyli wyrażenie w Rackecie z poprzedniej strony na nasz język, do zmiennej *f* powinna być przypisana następująca *wartość*:

```
(clo 'y                                ; argument
  (binop '+' (var-expr 'x) (var-expr 'y)) ; ciało procedury
  (environ '([x . 4])))                 ; środowisko
```

Teraz wystarczy rozbudować ewaluator. Interpretacja lambda-wyrażenia jedynie tworzy domknięcie, paczkując argument, ciało procedury i aktualne środowisko. Interpretacja aplikacji (*f e*) jest bardziej skomplikowana. Najpierw obliczamy wartość wyrażenia *f* (uzyskując jakieś domknięcie) oraz wyrażenia *e*. Wynikiem jest wynik obliczenia ciała procedury z domknięcia w odpowiednio spreparowanym środowisku: jest to środowisko z domknięcia rozbudowane o informację, że wartością argumentu jest obliczona wartość wyrażenia *e*:

fun.rkt

```
123 (define (eval-env e env)
124   (match e
125     ...
140     [(lam x e) (clo x e env)]
141     [(app f e)
142       (let ([vf (eval-env f env)]
143             [ve (eval-env e env)])
144         (match vf [(clo x body fun-env)
145                   (eval-env body
146                             (env-add x ve fun-env))]))]))]
```

Proszę zwrócić uwagę, że ewaluując ciało procedury, w ogóle nie patrzymy na aktualne środowisko. A to dlatego, że ciało procedury w ogóle nie potrzebuje aktualnego środowiska – potrzebuje tylko znać wartość swojego argumentu i zmiennych wolnych zapisanych w domknięciu.

Oczywiście w prawdziwych implementacjach nie opłacałoby się zapamiętywać całego środowiska, bo wystarczy znać wartości zmiennych, które rzeczywiście występują w ciele funkcji. Właśnie dlatego analiza wyszukująca zmienne wolne w wyrażeniu przedstawiona na poprzednim wykładzie jest tak ważna.²

²Proszę zwrócić uwagę, że dodanie funkcji wyższych rzędów do języka zajęło nam 9 wierszy kodu (pominawszy parsowanie i predykaty, które nie biorą przecież udziału w procesie interpretacji). Dodanie funkcji wyższych rzędów do języka Java zajęło kolejnym jego wydawcom 18 lat (pierwsza wersja JDK to 1996 rok, Java 8 z lambda to rok 2014, LISP z procedurami wyższych rzędów to rok 1958). Oczywiście nie sugerujemy tutaj, że twórcy Javy nie wiedzieli, co robią, a jedynie próbujemy uświadomić Państwu, z jak szeroką gamą możliwych spojrzeń będą się Państwo spotykać w życiu – coś, co dla jednych jest podstawowym narzędziem ekspresji programistycznej, dla innych jest czymś zupełnie obcym.

Teraz już powinno być jasne, jak zaimplementować leniwe let-wyrażenie: prócz zapamiętywania wyrażeń zdefiniowanych w let-ach, trzeba jeszcze zapamiętać środowisko aktualne w momencie definicji – albo przynajmniej wartości zmiennych wolnych tych wyrażeń.

Możemy przetestować nasz interpreter i przy okazji przekonać się czy dobrze rozumiemy częściowe aplikacje:

```
> (eval (parse
      '(let [twice (lambda (f x) (f (f x)))]
          (let [inc (lambda (x) (+ 1 x))]
              (twice twice twice twice inc 1))))
65537
```

8.7. Domknięcia w językach imperatywnych

Żeby mieć prawdziwe procedury wyższych rzędów, chcemy móc przyjmować procedury jako argumenty i zwracać procedury jako wyniki innych procedur (np. procedura `op->proc` w naszym interpreterze). To jednak nie wszystko – potrzebujemy też, żeby język umiał automatycznie tworzyć domknięcia. Np. w języku C możemy użyć wskaźników do funkcji, żeby przyjmować i zwracać funkcje, ale poniższy program nie jest dobrym pomysłem:

fun.c

```
1  #include <stdio.h>
2
3  typedef int (*int2int)(int);
4
5  int2int twice(int2int f)
6  {
7      int new_f(int x) { return f(f(x)); }
8      return new_f;
9  }
10
11 int inc(int x) { return x + 1; }
12
13 int main()
14 {
15     printf("%d", twice(inc)(1));
16     return 0;
17 }
```

Po uruchomieniu mogą się Państwo spodziewać błędu *segmentation fault*, bo i owszem możemy stworzyć lokalną funkcję `new_f` i zwrócić wskaźnik do niej, ale nie zostało utworzone domknięcie, więc zmienna `f` w ciele `new_f` pokazuje na w miarę losowe komórki pamięci.

Jeśli ktoś z Państwa programuje w C++, na pewno wie, że lambda-wyrażenia (począwszy od C++20) mają składnię, która jest poezją wśród składni, wykorzystującą wszystkie rodzaje nawiasów:

```
[ ]<>() {}
```

gdzie:

- W nawisach kwadratowych `[]` znajduje się specyfikacja zmiennych z zewnątrz, które można używać wewnątrz ciała funkcji, mówiąca czy w domknięciu ma znaleźć się wartość danej zmiennej czy referencja.
- W nawiasach kątowych³ `<>` znajduje się lista argumentów typowych.
- W nawiasach okrągłych `()` znajduje się lista argumentów funkcji.
- W nawiasach klamrowych `{ }` znajduje się ciało funkcji.

Dla przykładu, polimorficzną identyczność, która zlicza w zmiennej `x`, ile razy została użyta, można zdefiniować tak:

fun.cpp

```
1  #include <iostream>
2
3  int main()
4  {
5      int x = 0;
6      auto id = [&]<class T>(T t) { x++; return t; };
7      std::cout << id(5) << id(" xxx ") << x;
8      return 0;
9  }
```

Wynikiem uruchomienia tego programu będzie wypisanie na wyjściu poniższego ciągu znaków:

```
5 xxx 2
```

³Znak mniejszości `<` i większości `>` to **nie** to samo, co nawiasy kątowne `< i >`, co czyni programowanie w C++ albo HTML-u bardzo uciążliwe dla ludzi o dużej wrażliwości typograficznej.

8.8. Dynamiczne wiązanie zmiennych



Uwaga! Treści zawarte w tej sekcji, jeśli użyte bez głębokiego zrozumienia, mogą prowadzić do wielu nieporozumień. Zalecamy ostrożność!

Dzięki domknięciom realizujemy **statyczne wiązanie zmiennych**. Jeśli zrezygnujemy z domknięć, uzyskamy **dynamiczne wiązanie zmiennych**. Z perspektywy programisty współczesnego sama myśl o dynamicznym wiązaniu zmiennych jeży włos na głowie, ale historycznie (i współcześnie w EmacsLISP-ie) sposoby interpretacji zmiennych wolnych w ciele funkcji były przedmiotem zażartej dyskusji.

Za dynamicznym wiązaniem przemawia fakt, że umożliwia ono robienie różnego rodzaju szemranych trików. Sprawdźmy to na przykładzie naszego języka. Najpierw wprowadzamy dynamiczne wiązanie. Uzyskujemy je, modyfikując domknięcia tak, by nie zapamiętywały środowiska, a podczas ewaluacji ciała funkcji używamy środowiska z momentu wywołania:

dynamic.rkt

```

105 (struct clo (id e) #:transparent)
106
121 (define (eval-env e env)
122   (match e
123     ...
138     [(lam x e) (clo x e)]
139     [(app f e)
140      (let ([vf (eval-env f env)]
141            [ve (eval-env e env)])
142        (match vf [(clo x body)
143                   (eval-env body (env-add x ve env))]]))]

```

Jednym z trików, które są możliwe dzięki dynamicznemu wiązaniu zmiennych są funkcje rekurencyjne. Nasze let-wyrażenia nie są rekurencyjne, ale skoro obliczamy wartość ciała funkcji w momencie jej użycia, więc już po jej zdefiniowaniu...

dynamic.rkt

```

147 (define program-fact
148   '(let [fact (lambda (n) (if (<= n 0)
149                               1
150                               (* n (fact (- n 1)))]))]
151     (fact 6)))

```


I rzeczywiście:

```
> (eval (parse program-fact))  
720
```

Następny wykład poświęcony będzie funkcjom rekurencyjnym – głównie temu jak zrobić je bezpiecznie bez potrzeby dynamicznego wiązania.