Wykład 9.
Stan i rekursja

Wykład nr 9 poświęcony jest dwóm na pozór niezwiązanym tematom: mutowalnemu stanowi i procedurom rekurencyjnym. Podczas gdy w językach imperatywnych i obiektowych mutowalny stan jest bardzo powszechny, my do tej pory go unikaliśmy, zajmując się głównie programowaniem **czysto funkcyjnym** (ang. *purely functional programming*). Czystość oznacza, że wszystko, co robimy, to obliczamy wartości, nie wykonując w tle żadnych **efektów ubocznych** (ang. *side effects*), traktując procedury jako funkcje matematyczne, których wartość zależy jedynie od wartości argumentów. Język programowania może udostępniać programiście kilka rodzajów efektów ubocznych – nazywanych także często bardziej ogólnie **efektami obliczeniowymi** – na przykład wejście–wyjście, mutowalny stan, wyjątki.

Programowanie czysto funkcyjne może dla niewprawnego oka jawić się pomysłem nieco dziwnym, bo jest to krępowanie sobie rąk bez wyraźnej przyczyny. Przyczyna jednak jest: ograniczając efekty, zyskujemy gwarancje pewnych pożądanych własności. Najważniejszą z nich jest to, że elementy programów czysto funkcyjnych nie są ze sobą silnie powiązane (czyli mamy tzw. *loose coupling*). Każdy z nich może działać w izolacji, bez potrzeby istnienia całej reszty systemu, co koncepcyjnie upraszcza projektowanie takich systemów, bardzo upraszcza testowanie, świetnie nadaje się do automatycznego optymalizowania przez kompilator[1], ułatwia ponowne użycie kodu w tym samym lub innym systemie, niweluje wiele upierdliwych problemów przy współbieżności i aż prosi się o zrównoleglenie.

To prawda, że gdy dużo dzieje się za kulisami, to trudno zrozumieć całość, ale trzeba też znać proporcje. Przecież są sytuacje, w których sensowność i czytelność kodu zyskuje, jeśli pewne rzeczy wyrazimy przy użyciu wyjątków albo mutowalngo stanu. W szczególności, podczas tego wykładu znajdziemy ciekawe zastosowanie stanu w naszym interpreterze języka czysto funkcyjnego!

A skoro jesteśmy już przy stanie, to napiszemy też interpreter mikroskopijnego języka z mutowalnym stanem.

[1] Pan Paweł Dietrich sprowokował mnie do tego, żebym rzucił okiem na optymalizacje robione przez kompilator Racketa i tutaj niestety nie możemy liczyć na cuda. W tym celu muszą Państwo zerknąć na bardzo silnie optymalizujące kompilatory Haskella czy OCamla.

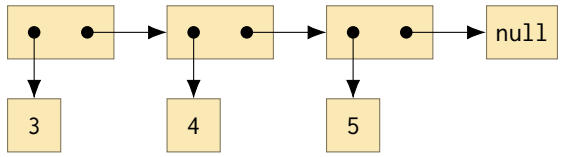
9.1. Mutowalny stan w Rackecie

Racket to język funkyjny, ale daje programiście możliwość pracy z mutowalnym stanem. Mechanizmy do tego służące opisane są szczegółowo w podręczniku w rozdziale 3.

Tym, co będzie nas interesowało tutaj, są mutowalne listy. Na pewno pamiętają Państwo „model pudełkowy" dla zwykłych list, w którym każde wywołanie procedury `cons` tworzy w pamięci pudełko złożone z dwóch wskaźników. Rozważmy następujące wyrażenie:

```
(cons 3 (cons 4 (cons 5 null)))
```

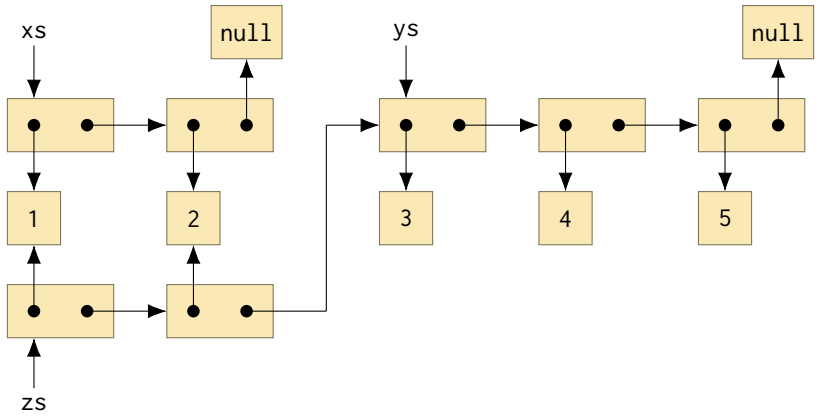
Stworzy ono w pamięci następującą strukturę:



Listy są **trwałe** (ang. *persistent*), co znaczy, że raz stworzona lista nigdy nie ulegnie zmianie. Przykładowo, patrząc na definicję procedury `(append xs ys)`, wywołujemy procedurę `cons` dla każdego elementu `xs`, a na `ys` w ogóle nie patrzymy. To znaczy, że pierwszą listę musimy w całości skopiować tylko po to, żeby w ostatnim `cons`-ie zamiast wskaźnika na `null` zrobić wskaźnik na (bloczek rozpoczynający) listę `ys`. Z drugiej strony, bardzo ważnym elementem tego podejścia jest **współdzielenie** (ang. *sharing*). Znaczy to, że listy `xs` i `(append xs ys)` mają wspólne komórki w pamięci. Rozważmy następujące definicje:

```
(let* ([xs (list 1 2)]
       [ys (list 3 4 5)]
       [zs (append xs ys)]) … )
```

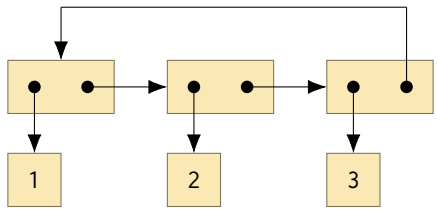
Te definicje tworzą w pamięci następujący pudełkowy graf:



Właśnie dlatego, że ys i zs nigdy nie ulegną zmianie, współdzielenie jest bezpieczne. Szczególnie efektownym przykładem jest wstawianie elementu do drzewa BST, gdzie musimy skopiować tylko ścieżkę od korzenia do nowego elementu, a jednocześnie, właśnie dzięki współdzieleniu, nie tracimy w ogóle dostępu do wersji drzewa przed wstawieniem. Z trwałości struktur danych korzystaliśmy już w naszym interpreterze ze środowiskiem: np. obliczając wartość wyrażenia `(binop op l r)`, podczas obliczania wartości wyrażenia `l` być może modyfikujemy środowisko (dodając do niego nowe zmienne), a gdy przychodzi czas obliczenia wartości wyrażenia `r`, po prostu używamy starej wersji środowiska, która wciąż istnieje w pamięci.

Wadą trwałych struktur danych jest to, że (w znanym nam podzbiorze Racketa) w pamięci możemy tworzyć tylko pudełkowe grafy acykliczne. Jest tak dlatego, że wywołując cons oba argumenty muszą być już policzone, więc wszystkie wskaźniki zawsze wskazują „w przeszłość".

Czasem chcielibyśmy mieć bardziej skomplikowaną strukturę w pamięci. Najprostszą jest **lista cykliczna**, np.



Takie struktury uzyskamy używając typu danych **mutowalnych list** (ogólniej: **mutowalnych par**), które zachowują się bardzo podobnie do zwykłych list (par),

ale pozwalają zmienić wartość pierwszego i drugiego elementu w pudełku. Listę taką tworzymy przy użyciu procedury `mcons`, a łańcuszek `mcons`-ów kończymy starym dobrym `null`-em. Selektory nazywają się `mcar` i `mcdr`, z oczywistą semantyką. Mutowanie odbywa się przy użyciu procedur `set-mcar!` i `set-mcdr!`.[2] Bardziej szczegółowo, jeśli wyrażenie `p` oblicza się do mutowalnej pary (w modelu pudełkowym: do wskaźnika na mutowalną parę), to

```
(set-mcar! p e)
```

podmieni pierwszy element tej pary na wartość wyrażenia `e`. Analogicznie z `set-mcdr!`. Procedury te nie zwracają żadnej interesującej dla nas wartości, jedynie modyfikują rzeczy w pamięci. Jest też oczywiście predykat `mpair?`.

Procedurę, która ze zwykłej listy produkuje jej zacykloną mutowalną wersję definiujemy poniżej. Posiada ona procedurę pomocniczą `list->mlist`, której jedynym zadaniem jest odtworzyć listę, ale tym razem zbudowawszy ją z `mcons`-ów. Druga procedura pomocnicza, `firt-to-last!` pamięta całą mutowalną listę (czyli wskaźnik na jej pierwsze pudełko) w argumencie `mxs`, idąc używając procedury `aux` aż do końca listy, w którym podmienia wskaźnik na `null`-a na wskaźnik na `mxs`.

mutable.rkt

```
3  (define (cycle xs)
4    (define (list->mlist xs)
5      (cond [(null? xs) null]
6            [else (mcons (car xs) (list->mlist (cdr xs)))]))
7    (define (first-to-last! mxs)
8      (define (aux mys)
9        (cond [(null? mys) (error "Can't cycle empty list")]
10             [(and (mpair? mys) (null? (mcdr mys)))
11              (set-mcdr! mys mxs)]
12             [else (aux (mcdr mys))]))
13     (aux mxs))
14   (let ([mxs (list->mlist xs)])
15     (begin (first-to-last! mxs)
16            mxs)))
```

[2] Tradycyjnie w lispowym świecie procedury, które mutują stan, mają nazwy kończące się wykrzyknikiem.

9.2. Rekurencyjne let-wyrażenia – koncepcja

> But by then the ship's screen had turned and something came into view that was not a star and not a galaxy. It was a dim mass of pale-blue light, mottled, immense, and terrifying at the first glimpse. I knew it was not a sun. No sun can be so big and so dim. It hurt the eyes to look at it, not because of brightness. It hurt inside the eyes, up far into the optic nerve. The pain was in the brain itself.
> Metchnikov switched off the radio, and in the silence that followed I heard Danny A. say prayerfully, "Dearest God, we've had it. That thing is a black hole."
>
> Frederik Pohl, *Gateway*

Język z poprzedniego wykładu zdefiniowany w pliku `fun.rkt` wydaje się bardzo zaawansowany (w końcu ma pary, listy, lambdy), ale trudno w nim pisać ciekawe programy, bo brakuje w nim procedur rekurencyjnych. Może i można zdefiniować listę, ale nie da się napisać procedur `length`, `reverse` ani `append` w sposób, jaki znamy z Racketa. Czas dodać procedury rekurencyjne, a właściwie bardziej ogólną konstrukcję: rekurencyjne let-wyrażenia.

Takie wyrażenie zapisane w składni konkretnej będzie miało postać

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

W takim wyrażeniu zmienna `x` związana jest **zarówno** w `e1` jak i `e2`. W obu tych wyrażeniach intuicyjnie oznacza ona wynik wyrażenia `e1`. Trzeba więc być ostrożnym: obliczając wartość wyrażenia `e1`, lepiej nie potrzebować znać wartości zmiennej `x`, bo to oznaczałoby, że do obliczenia wartości wyrażenia `e1` potrzebujemy wartości wyrażenia `e1`. Taka sytuacja powinna kończyć się zapętleniem programu albo przerwaniem obliczeń z odpowiednim komunikatem o błędzie.

Na co więc nam zmienna `x`, której wartości w ogóle nie wolno użyć? Nie możemy jej użyć, ale możemy zamknąć ją w domknięciu! Przykładowo, następujące obliczenie nie powinno powodować żadnego błędu:

```
(letrec [f (lambda (x) (if (= x 0) 1 (* x (f (- x 1)))))]
  (f 5))
```

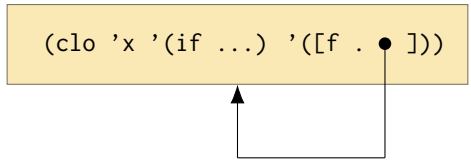
Dlaczego? Bo `f` jest lambdą, a lambdy obliczają się od razu do wartości, nie obliczając w ogóle wartości ciała funkcji (w którym występuje zmienna `f`) w momencie definicji. Na pierwszy rzut oka nie rozwiązuje to jednak naszych problemów, bo może nie używamy wartości zmiennej `f` podczas obliczania

wartości `e1`, ale musimy w tworzonym domknięciu **zapisać** jej wartość. A przecież nie znamy tej wartości, bo jesteśmy w trakcie jej obliczania.

Cofnijmy się więc o krok i zamiast pytać, jak obliczyć wartość w ogólnym przypadku, zadajmy sobie pytanie, jaka to powinna być wartość w tym konkretnym przypadku. Powinno to oczywiście być domknięcie o poniższym kształcie:

```
(clo 'x '(if ...) '([f . ?]))
```

Natomiast wartość wskazywana przez `f` (oznaczona przez „?") to... to samo domknięcie. Innymi słowy chcemy utworzyć w pamięci następującą strukturę, naszkicowaną w bardziej abstrakcyjnej notacji pudełkowej:



Teraz już Państwo rozumiecie, po co nam był cały ten mutowalny stan: żeby zacyklić strukturę danych. Podczas obliczania wartości wyrażenia nie można odwołać się do niej samej, ale strukturę można zacyklić *post factum*.

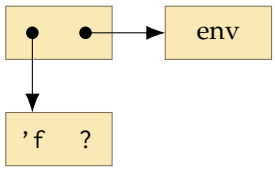
Wciąż nie rozwiązuje to naszych problemów w przypadku ogólnym, bo nikt nie obiecał nam, że wartością definiowanego wyrażenia będzie jakaś lambda. Może być to np. para lambd definiująca dwie wzajemnie rekurencyjne procedury:

letrec.rkt

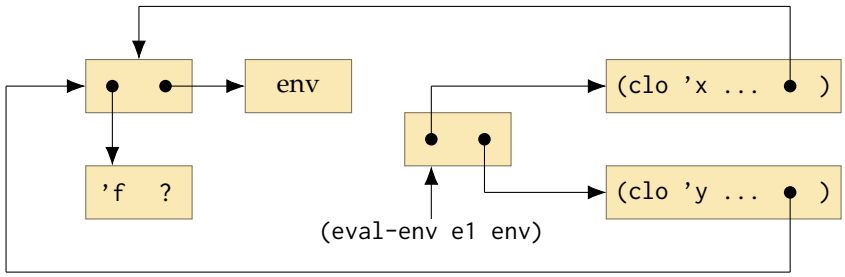
```
174 '(letrec
175    [fact (lambda (n) (if (= n 0) 1 (* n (fact (- n 1)))))]
176  (letrec
177    [f
178      (cons
179        (lambda (x) (if (= x 0) true  ((cdr f) (- x 1))))
180        (lambda (y) (if (= y 0) false ((car f) (- y 1)))))]
181  (let [even (car f)]
182  (let [odd  (cdr f)]
183  (even (fact 6)))))
```

Prześledźmy, jak będzie postępowało tworzenie wartości reprezentującej parę `f` (całe wyrażenie `(cons ...)` będziemy nazywać `e1`). Zaczynamy od aktualnego środowiska:

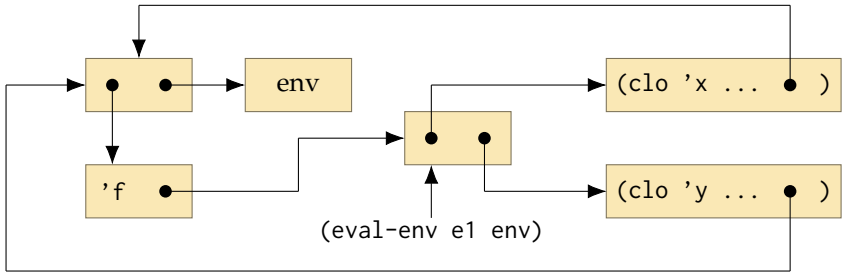
Do środowiska dokładamy nową zmienną `f`, której wartością jest tajemnicza wartość ?:



Następnie budujemy interesującą nas strukturę dokładnie tak jak do tej pory, czyli normalnie obliczając wartości, zapisując w tworzonych domknięciach aktualne środowisko:



Dopiero ostatni krok to mutowanie pamięci. „Przepinamy" wartość zmiennej `f` tak, by pokazywała na stworzoną przed chwilą wartość:



Proszę zwrócić uwagę na pewne subtelności tego rozwiązania. Po pierwsze, musimy stworzyć komórkę `'f ?` zanim zaczniemy obliczać wartość pary `f`, żeby była ona obecna w tworzonych domknięciach. Po drugie musimy

uważać, żeby nigdzie indziej w interpreterze nie skopiować komórki `'f ?` przed „zawiązaniem węzła" – **jest to dobry przykład, jak mutowalny stan wprowadza nielokalność**. Na przykład, ktoś mógłby w interpreterze zamiast

```
[(lam x e) (clo x e env)]
```

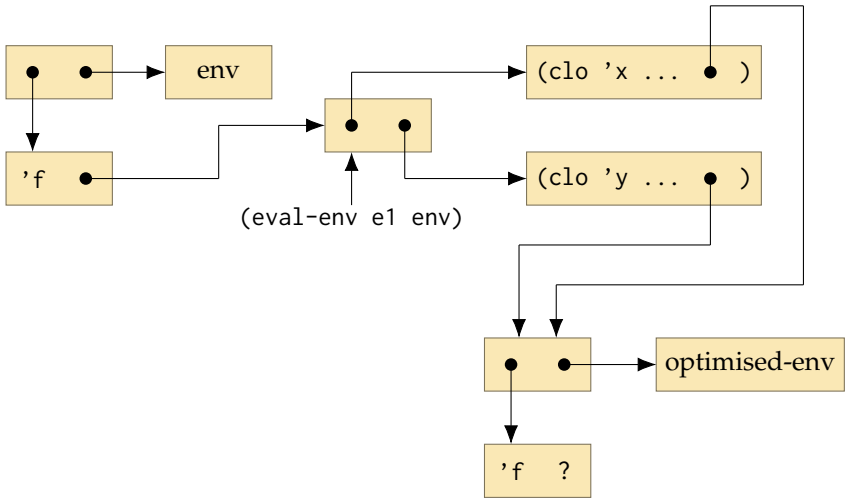
w procedurze `eval-env` spróbować wprowadzić optymalizację i zapisać nie całe środowisko, ale tylko wartości zmiennych wolnych występujących w `e`:

```
[(lam x e) (clo x e (build-env x (free-vars e) env))]
```

Zakładając, że `free-vars` zwraca listę zmiennych wolnych argumentu, procedurę `build-env` definiujemy następująco (pobiera ona jeszcze argument funkcji, żeby nie uznać go za zmienną wolną):

```
(define (build-env x vars env)
  (cond [(null? vars) (env-empty)]
        [(eq? x (car vars)) (build-env x (cdr vars) env)]
        [else (env-add (car vars)
                       (env-lookup (car vars) env)
                       (build-env x (cdr vars) env)]
```

Bardzo szybko okazuje się, że ta optymalizacja, dokonana w gałęzi wyrażenia `match` dla struktury `lam`, psuje nam zupełnie kod, który napisaliśmy w gałęzi dla struktury `letrec`. Dzieje się tak dlatego, że dodawanie zmiennych do środowiska tworzy nowy bloczek, na nowo parując zmienną `f` z wartością aktualną w momencie tworzenia, czyli `?`, a zawiązując węzeł modyfikujemy tylko oryginalny bloczek. Innymi słowy, ewaluacja przykładu powyżej stworzy w pamięci następującą strukturę:



To zdecydowanie nie jest struktura, o jaką nam chodziło! Mamy nadzieję, że będzie to dla Państwa przestroga, że ze stanem trzeba uważać.

Czym jest tajemnicza wartość ?? Reprezentuje ona wartość wyrażenia `e1` w trakcie obliczania jego wartości, więc jest to coś, czego nie chcemy nigdy dotknąć. Taką wartość nazywamy **czarną dziurą**, bo jeśli choć raz spróbujemy poznać jej wartość, to znaczy, że obliczenie się zapętliło i już nie ma z tej pętli ucieczki. Tak więc ? to wartość, której próba poznania kończy się błędem.

9.3. Rekurencyjne let-wyrażenia – implementacja

Rozbudowujemy składnię o rekurencyjne let-wyrażenia. W składni konkretnej przypominają one oczywiście zwykłe let-wyrażenia, poza tym, że wprowadzane są formą specjalną `letrec`. W składni abstrakcyjnej rekurencyjne let-wyrażenia reprezentujemy przy pomocy struktury `letrec-expr`:

letrec.rkt

```
15 (struct letrec-expr (id e1 e2) #:transparent)
```

Duże zmiany zachodzą w środowiskach. Pierwszą z nich jest to, że nie są one już reprezentowane przez listę par, ale przez listę **mutowalnych** par. Zmiana ta wymaga jedynie dopisania litery `m` do kilku `cons`-ów, `car`-ów i `cdr`-ów. Drugą zmianą jest to, że jedną z możliwych wartości jest czarna dziura, którą definiujemy jako bezargumentową strukturę `blackhole`. Do tej pory środowiskom było zupełnie obojętne, jaki rodzaj wartości przechowywały. Teraz

próba odczytania ze środowiska wartości zmiennej, której wartością jest czarna dziura, kończy się błędem:

letrec.rkt

```
98  (struct blackhole () #:transparent)
99  (struct environ (xs) #:transparent)
100
101 (define env-empty (environ null))
102 (define (env-add x v env)
103   (environ (cons (mcons x v) (environ-xs env))))
104 (define (env-lookup x env)
105   (define (assoc-lookup xs)
106     (cond [(null? xs) (error "Unknown identifier" x)]
107           [(eq? x (mcar (car xs)))
108              (match (mcdr (car xs))
109                [(blackhole) (error "Stuck in a black hole!")]
110                [x x])]
111           [else (assoc-lookup (cdr xs))]))
112   (assoc-lookup (environ-xs env)))
```

Potrzebujemy także procedury, która umożliwia nam zmianę wartości przechowywanej w środowisku. Wyszukuje ona odpowiednią zmienną w środowisku i używa `set-mcdr!`, żeby zapisać nową wartość:

letrec.rkt

```
98  (define (env-update! x v xs)
99    (define (assoc-update! xs)
100     (cond [(null? xs) (error "Unknown identifier" x)]
101           [(eq? x (mcar (car xs))) (set-mcdr! (car xs) v)]
102           [else (env-update! x v (cdr xs))]))
103   (assoc-update! (environ-xs xs)))
```

Z tak przygotowaną infrastrukturą implementacja strategii opisanej nieformalnie w poprzedniej sekcji jest bardzo prosta. Najpierw rozbudowujemy predykat `value?` o czarne dziury:

letrec.rkt

```
126 (define (value? v)
127   (or ...
133       (blackhole? v)))
```

Teraz wystarczy zaimplementować schemat z poprzedniej sekcji. Najpierw (wiersz 149) tworzymy nową wartość w środowisku, potem (wiersz 150) obliczamy wartość wyrażenia `e1` w tym środowisku. Po uzyskaniu wyniku, zawiązujemy węzeł poprzez przepięcie wskaźnika (wiersz 152) i kontynuujemy obliczanie `e2` w tak zmodyfikowanym środowisku (wiersz 153):

letrec.rkt

```
178 (define (eval-env e env)
179   (match e
180     ...
148     [(letrec-expr x e1 e2)
149      (let* ([new-env (env-add x (blackhole) env)]
150             [v (eval-env e1 new-env)])
151        (begin
152           (env-update! x v new-env)
153           (eval-env e2 new-env)))] ... ))
```

Sprawdźmy, czy działa:

letrec.rkt

```
187 (define PROGRAM
188   '(letrec [from-to (lambda (n k)
189                       (if (> n k)
190                           null
191                           (cons n (from-to (+ n 1) k))))]
192    (letrec [sum (lambda (xs)
193                   (if (null? xs)
194                       0
195                       (+ (car xs) (sum (cdr xs)))))]
196    (sum (from-to 1 36)))))
```

A następnie:

```
> (eval (parse PROGRAM))
666
```

9.4. Programujemy!

Przy okazji poprzedniego wykładu wspominaliśmy, że chcemy mieć interpreter języka programowania, w którym można napisać interpreter języka programowania. Skoro mamy już funkcje rekurencyjne, bierzmy się do dzieła. Pisanie

całego interpretera byłoby katorgą, bo mamy bardzo mało lukru syntaktycznego (chyba że ktoś dzielnie rozwiązywał zadania z list zadań), więc poprzestaniemy na ewaluatorze wyrażeń arytmetycznych. Niech starczy on Państwu za argument, że można.

Najpierw jednak rozbudujmy nasz język o ciągi znakowe (stringi) i operator binarny eq?. Na tym etapie Państwo na tyle dobrze wiedzą, jak to zrobić (składnia – wartości – ewaluator), że nie będziemy tu zamieszczać szczegółów.

Reprezentację wyrażeń arytmetycznych zbudujemy przy użyciu list, tak jak robiliśmy to w Rackecie zanim poznaliśmy struktury. Jednak zamiast symboli użyjemy stringów (rozbudowanie języka o symbole → lista zadań). Na przykład:

- stałe: `(cons "const" (cons 6 null))`
- operatory: `(cons "binop" (cons "+" (cons l (cons r null))))`

Ewaluator i przykładowe wyrażenie arytmetyczne definiujemy jak poniżej:

programming.rkt

```
98  (define (EVAL p)
99    `(let [op->proc (lambda (op)
100                      (if (eq? op "+") (lambda (x y) (+ x y))
101                      (if (eq? op "*") (lambda (x y) (* x y))
102                      (if (eq? op "-") (lambda (x y) (- x y))
103                      false))))]
104     (letrec [eval (lambda (e)
105                     (if (eq? (car e) "binop")
106                         (let [op (car (cdr e))]
107                         (let [l  (car (cdr (cdr e)))]
108                         (let [r  (car (cdr (cdr (cdr e))))]
109                         (op->proc op (eval l) (eval r)))))
110                     (if (eq? (car e) "const")
111                         (car (cdr e))
112                     false)))]
113     (eval ,p))))
114
115 (define EXPRESSION
116     '(let [const (lambda (n) (cons "const" (cons n null)))]
117      (let [binop (lambda (op l r) (cons "binop" (cons op
            (cons l (cons r null)))))]
118      (binop "+" (const 2) (binop "*" (const 2) (const 2))))))
```

Testujemy:

```
(eval (parse (EVAL EXPRESSION)))
6
```

9.5. Alternatywnie: Kombinator punktu stałego

Korzystając z faktu, że nasz język, podobnie jak Racket, nie jest typowany, możemy zrobić funkcje rekurencyjne używając tzw. **kombinatora punktu stałego** (ang. *fixed-point combinator*). Takich kombinatorów jest dość nieskończenie wiele, my użyjemy kombinatora Y:

fixpoint.rkt

```
98  (define PROGRAM
99    '(let [fix (lambda (f)
100               ((lambda (x) (f (lambda (y) (x x y))))
101                (lambda (x) (f (lambda (y) (x x y))))))]
102    (let [from-to (fix (lambda (From-to n k)
103                         (if (> n k)
104                           null
105                           (cons n (From-to (+ n 1) k)))))]
106    (let [sum (fix (lambda (Sum xs)
107                   (if (null? xs)
108                       0
109                       (+ (car xs) (Sum (cdr xs))))))]
110    (sum (from-to 1 36))))))
```

Testujemy:

```
> (eval (parse PROGRAM))
666
```

Taki kombinator to prawdziwy łamacz głowy. Żeby zrozumieć jego działanie, nie ma chyba innego sposobu niż zapisać wyrażenie w stylu `(fix f)` dla jakieś funkcji `f` (która nie jest rekurencyjna, ale jako swój pierwszy argument przyjmuje funkcję, której używa do robienia wywołań „rekurencyjnych", a kombinator już zadba o to, żeby dostała „samą siebie") i rozpisać kolejne kroki obliczeń, np. w modelu podstawieniowym.

Zwracamy uwagę, że ten trik nie działa w językach typowanych, bo `fix` zazwyczaj nie chce mieć typu. Np. jeśli próbujemy zdefiniować równoważne wyrażenie w Haskellu:

```
> λf → (λx → (f (λy → x x y))) (λx → (f (λy → x x y)))

<interactive>:1:26: error:
    * Occurs check: cannot construct the infinite type:
        t0 ~ t0 -> t1 -> t
      Expected type: t0 -> t1 -> t
        Actual type: (t0 -> t1 -> t) -> t1 -> t
```

9.6. Coś z zupełnie innej beczki: WHILE

Racket to język funkcyjny, więc bez wyraźnej potrzeby nie używamy mutowalnego stanu. Po drugiej stronie spektrum znajdują się języki imperatywne, których celem egzystencji jest mutowanie stanu. Często są to języki niskopoziomowe (np. C), które służą do modyfikowania stanu fizycznej maszyny – więc nie można mieć do nich o to pretensji.

W tej sekcji (jako przedsmak tego, co będzie się działo w przyszłym tygodniu), napiszemy interpreter minimalistycznego imperatywnego języka, który zwyczajowo nazywa się WHILE. Oczywiście nazwa ta pochodzi od rodzaju pętli.

Ideą języka WHILE jest to, że mamy pewien potencjalnie nieskończony globalny zbiór **aktywnych** zmiennych. Aktywnym zmiennym przypisane są wartości w środowisku, które nazywamy **pamięcią**. Pamięć jest modyfikowana (a raczej „modyfikowana") w trakcie interpretacji programu. Sam język składa się z dwóch kategorii składniowych:

- **Wyrażenia** – to takie same wyrażenia, jak rozważaliśmy do tej pory. Każde wyrażenie może zawierać zmienne wolne pochodzące ze zbioru aktywnych zmiennych. Wyrażenie oblicza się do wartości.
- **Instrukcje** (ang. *commands*) – cały program jest instrukcją, być może złożoną z mniejszych instrukcji. Instrukcja może modyfikować pamięć. W szczególności rodzajem instrukcji jest instrukcja przypisania, która zmienia wartość aktywnej zmiennej albo „uaktywnia" zmienną do tej pory nieużywaną nadając jej wartość. Innymi instrukcjami są instrukcje kontroli, np. pętle i `if`-y.

W języku zdefiniowanym w pliku **while.rkt** znajdziemy pięć instrukcji, które podsumowujemy w poniższej tabeli. Argument `x` to zmienna (symbol), argumenty, których nazwy rozpoczynają się literą `c` to instrukcje, a literą `e` to

wyrażenia (te same wyrażenia, których używaliśmy w pliku **letrec.rkt**):

Składnia abstrakcyjna	**składnia konkretna**
`(struct skip ())`	`skip`
`(struct assign (x e))`	`(x := e)`
`(struct if-cmd (eb ct cf))`	`(if eb ct cf)`
`(struct while (eb cb))`	`(while eb cb)`
`(struct comp (c1 c2))`	`(c1 c2)`

Intuicyjnie, `skip` to instrukcja, która nic nie robi (przydatna np. do zdefiniowania `if`-a bez gałęzi „else"). Instrukcja `(comp c1 c2)` to sekwencyjne złożenie instrukcji (w C-podobieństwie napisalibyśmy coś w stylu `c1; c2`). Dodatkowo wprowadzamy lukier: Instrukcja `(c1 c2 ... cn-1 cn)` oznacza `(comp c1 (comp c2 ( ...(comp cn-1 cn)... )))`.

Jak interpretujemy język WHILE? Co to w ogóle znaczy go interpretować? Interpreter języka funkcyjnego przeprowadza wyrażenia w wartości, a języka imperatywnego? Skoro język jest imperatywny, to znaczy, że jego *modus operandi* to modyfikowanie stanu. Nasz interpreter będzie więc rodzajem przetwornika, który przetwarza stan początkowy w stan końcowy. Ten stan to oczywiście pamięć.

Pamięć przypisuje zmiennym wartości. Mamy już całkiem sensowną strukturę do przechowywania i przetwarzania takich danych, środowiska. Ponieważ chcemy modyfikować wartości zmiennych, potrzebujemy do tego dedykowaną procedurę. Co prawda mamy już `env-update!`, którą zdefiniowaliśmy przy okazji rekurencyjnych let-wyrażeń. Jednak działa ona przez fizyczne mutowanie komórki pamięci. W tym wypadku nie ma takiej potrzeby: ewaluator języka WHILE to procedura, która jako argumenty bierze program (instrukcję) i pamięć początkową (środowisko), a jako wynik zwraca pamięć końcową. Dlatego definiujemy modyfikowanie zmiennych czysto funkcyjnie, w poniższej procedurze `env-update`:

while.rkt

```
118 (define (env-update x v xs)
119   (define (assoc-update xs)
120     (cond [(null? xs) (list (mcons x v))]
121           [(eq? x (mcar (car xs))) (cons (mcons x v) (cdr
                xs))]
122           [else (cons (car xs) (assoc-update (cdr xs)))]))
123   (environ (assoc-update (environ-xs xs))))
```

Pozostaje zdefiniować ewaluator, który na podstawie instrukcji przetwarza pamięć początkową w pamięć końcową. Działa on przez analizę tego, jaką instrukcją jest program:

- Instrukcja `skip` nie zmienia stanu, więc jej ewaluacja w pamięci początkowej zwraca jako wynik tę samą pamięć.
- Ewaluacja instrukcji `(assign x e)` (przypisania) najpierw oblicza **wartość** wyrażenia `e` przy użyciu ewaluatora wyrażeń (procedury `eval-env`, gdzie środowiskiem jest oczywiście pamięć początkowa), a jako rezultat zwraca pamięć początkową, gdzie zaktualizowaliśmy zmienną `x` nadając jej wyliczoną wartość wyrażenia `e`.
- Ewaluacja instrukcji `(if eb ct cf)` najpierw oblicza wartość wyrażenia `eb`. Na jej podstawie wybiera instrukcję `ct` albo `cf`. Końcową pamięcią dla interpretacji tego `if`-a jest końcowa pamięć ewaluacji wybranej instrukcji.
- Ewaluacja instrukcji `(while eb cb)` także rozpoczyna się od obliczenia wartości wyrażenia `eb`. Jeśli wynik to `#f`, to znaczy, że nie wykonujemy ciała pętli, więc pamięć końcowa jest taka sama jak początkowa. Jeśli jednak `eb` oblicza się do `#t` to najpierw wykonujemy instrukcję `cb` (ciało pętli) uzyskując jako rezultat pewną pamięć pośrednią. Następnie wykonujemy całe obliczenie instrukcji `while` jeszcze raz, ale tym razem jako pamięć początkową używamy owej pamięci pośredniej. Pamięć końcowa tego ponownego wyliczenia jest pamięcią końcową całej instrukcji.
- Ewaluacja instrukcji `(comp c1 c2)` najpierw ewaluuje `c1` używając pamięci początkowej. Jako rezultat uzyskujemy pamięć pośrednią, którą używamy jako pamięci początkowej dla ewaluacji `c2`. Pamięć będąca wynikiem jest ostatecznym wynikiem ewaluacji złożenia.

Zapisawszy to w Rackecie, otrzymujemy poniższy, dość zwięzły ewaluator:

while.rkt

```
218 (define (eval-while e env)
219   (match e
220     [(skip) env]
221     [(assign x e)
222      (env-update x (eval-env e env) env)]
223     [(if-cmd eb ct cf)
224      (if (eval-env eb env)
225          (eval-while ct env)
```

```
226           (eval-while cf env))]
227      [(while eb cb)
228       (if (eval-env eb env)
229           (eval-while e (eval-while cb env))
230           env)]
231       [(comp c1 c2) (eval-while c2 (eval-while c1 env))]))
```

Jak wykorzystać nasz ewaluator? Gdyby ktoś z Państwa miał kłopot z napisaniem silni z użyciem Racketa i choćby się wytężał nie umiał ani rekurencyjnie, ani iteracyjnie, może napisać silnię w WHILE...

while.rkt

```
233 ; zakladamy, ze program startuje z pamiecia w ktorej
234 ; aktywna jest zmienna t
235 (define WHILE_FACT
236   '{(i := 1)
237     (while (> t 0)
238       {(i := (* i t))
239        (t := (- t 1))})})
```

...a potem zinterpretować go w odpowiedniej pamięci początkowej i odczytać z wynikowej pamięci interesującą nas wartość, która przypisana jest do zmiennej `i`:

while.rkt

```
241 (define (fact n)
242   (let* ([init-env (env-add 't n env-empty)]
243          [final-env
244            (eval-while (parse-while WHILE_FACT) init-env)])
245     (env-lookup 'i final-env)))
```

Przykładowo:

```
> (fact 6)
720
```