Wykład 6.
Wyrażenia i wartości

Szanowni Państwo! Rozpoczynamy drugą tercję przedmiotu metody programowania. Ze względu na ograniczenia dydaktyczne wynikające z wysiłków podejmowanych w walce z pandemią koronawirusa, wykłady w tradycyjnej formie nie mogą się odbywać, a my, prowadzący przedmiot, zmuszeni jesteśmy szukać form zastępczych. Stąd te notatki – lakoniczne, sklecone naprędce, spisane ołówkiem na opakowaniu po papierosach. Znaleźli się więc Państwo między młotem ciężaru abstrakcji omawianych tematów, a kowadłem naszego braku czasu i środków na przygotowanie w pełni satysfakcjonujących materiałów dydaktycznych. Pozostaje nam życzyć Państwu powodzenia!

Pierwsza tercja przedmiotu była zorganizowana wokół podręcznika. Druga tercja odbiega od podręcznika, chociaż większość tematów podejmowanych tutaj znajdą Państwo także tam, głównie w rozdziale 4. Materiał w podręczniku jest jednak mocno skondensowany, a naciski położone są na treści inne niż te, które prowadzący uznają na najważniejsze na tym przedmiocie. Stąd decyzja o trochę innym spojrzeniu na temat, a co za tym idzie – sporządzeniu tych notatek.

W razie znalezienia błędów w notatkach, proszę informować prowadzących. W razie wątpliwości i niejasności, warto dyskutować na forum na SKOS-ie. Ta część przedmiotu jest dużo bardziej implementacyjna, a to znaczy, że będą musieli Państwo pisać i czytać ze zrozumieniem większe kawałki kodu. Fragmenty kodu omawiane w tych notatkach znajdą Państwo na SKOS-ie w postaci plików `.rkt`.

* * *

Na Drugiej Tercji będziemy zajomować się składnią i semantyką języków programowania. Chcemy, by zrozumieli Państwo narzędzia, których będziecie używać na co dzień podczas studiów, a część z Państwa także potem w pracy zawodowej. Bardzo oględnie zobaczymy, jak język programowania działa od środka, prześledzimy drogę od programu wprowadzonego przez programistę do uzyskania opisywanej przez niego wartości. Chcemy wyposażyć Państwa w wiedzę niezbędną do przeprowadzenia merytorycznej rozmowy na temat

języków programowania, by mogli Państwo sami zdecydować, czy język C jest funkcyjny, skoro można w nim używać wskaźników do funkcji, a także wypowiadać się na internetowym forum w wątku „Język programowania, którego używam, jest lepszy niż język programowania, którego ty używasz". Nie będziemy pisać kompilatorów (z jednym małym wyjątkiem), ale będziemy pisać interpretery (ewaluatory).

Ten wykład obejmuje reprezentację i ewaluację wyrażeń arytmetycznych, a także *divertimento* na temat kwestii, którymi będziemy zajmować się mniej: kompilacji i analizy składniowej (czyli tzw. parsowania). Jest to cała nasza „Składnia i semantyka" w pigułce, bo przez całą tercję będziemy zajmować się właśnie reprezentacją i ewaluacją, za to coraz bardziej rozbudowanych języków programowania.

W tej sekcji zaczniemy od obserwacji, że Racketowe wyrażenie `(+ 2 (* 2 2))` oraz infiksowo zapisane wyrażenia używalne w większości języków programowania `2 + 2 * 2` oraz `2 + (2 * 2)` są w gruncie rzeczy tym samym wyrażeniem. Sformalizujemy tę intuicję, odkrywając, że z matematycznego punktu widzenia wyrażenia arytmetyczne to odpowiednio etykietowane drzewa, które tworzą tzw. **składnię abstrakcyjną** (widzieli już to Państwo przy okazji reprezentowania formuł rachunku zdań na wykładzie 5.). A skoro drzewa, to możemy reprezentować je w postaci struktur danych w Rackecie. Następnie napiszemy **ewaluator**, czyli program, który oblicza wartość wyrażenia reprezentowanego przez takie drzewo.

Najpierw poznamy jednak **struktury**, czyli konstrukcję w Rackecie, która zaoszczędzi nam trochę kodowania.

6.1. Abstrakcja danych przez struktury

Zapewne zauważyli już Państwo, że na tym przedmiocie słowo „abstrakcja" odmieniane jest przez wszystkie przypadki i wyskakuje zza każdego winkla. Jest tak dlatego, że język programowania to narzędzie, które oferuje nam właśnie to: abstrakcje. Pozwalają nam one nie tylko odejść od fizycznych szczegółów działania maszyn liczących, ale przede wszystkim zbliżają nasz język opisu jakiegoś algorytmu do bliższego nam, istotom ludzkim, sposobu postrzegania rzeczywistości. Do tej pory zidentyfikowaliśmy dwa rodzaje abstrakcji:

- **Abstrakcja proceduralna.** Pozwala nam ona na wyizolowanie pewnego powtarzalnego, sparametryzowanego kawałka programu jako oddzielnej procedury, która na domiar wszystkiego może mieć własną nazwę. Funkcje wyższych rzędów ułatwiają nam też podział fragmentów programu na

powtarzalny kod szkieletowy i konkretne działanie. Przykładem takich szkieletowych procedur są `map` i `fold`.

- **Abstrakcja danych.** Pozwala nam oddzielić konkretną **reprezentację** danych od **interfejsu**, którego używa kod zewnętrzny (*kliencki*) do obsługi tych danych. Prostym przykładem jest słownik (kolekcja par klucz-wartość), która w środku jest zwykle reprezentowana przez zbalansowane binarne drzewo poszukiwań (BST) lub tablicę haszującą, a na zewnątrz udostępnia operacje kolekcji: wstaw, wyszukaj, usuń, itp.

Abstrakcję danych zwykle realizujemy przez zdefiniowanie konstruktorów, predykatów i selektorów. Przykładowo, drzewa binarne z etykietami w wierzchołkach wewnętrznych możemy reprezentować następująco:

conventional.rkt

```
3  (define (node data l r) (list 'node data l r))
4  (define (node? t) (and (pair? t) (eq? (car t) 'node)))
5  (define (node-data t) (second t))
6  (define (node-l t) (third t))
7  (define (node-r t) (fourth t))
8
9  (define (leaf) (list 'leaf))
10 (define (leaf? t) (and (pair? t) (eq? (car t) 'leaf)))
```

Zdefiniowane w ten sposób procedury tworzą interfejs, który oddziela klienta od konkretnej reprezentacji. Możemy użyć interfejsu, by zdefiniować procedurę zdradzającą wysokość drzewa:

conventional.rkt

```
12 (define (height t)
13   (cond [(leaf? t) 0]
14         [(node? t) (+ 1 (max (height (node-l t))
15                              (height (node-r t))))]))
```

Jest to abstrakcja **konwencjonalna**. Znaczy to, że bariera abstrakcji jest czysto umowna i tak naprawdę można zajrzeć, co siedzi w czarnej skrzynce:

conventional.rkt

```
17 (define (size t)
18   (cond [(leaf? t) 0]
19         [(node? t) (+ 1 (size (third t))
20                         (size (fourth t)))]))
```

Definiowanie struktur danych w ten sposób ma oczywistą wadę: konstruktory, predykaty i selektory mają zawsze podobną definicję. Jest to tzw. *boilerplate code*, który trzeba napisać dla każdej struktury oddzielnie, chociaż ma się wrażenie, że kompilator umiałby go napisać za nas. I rzeczywiście tak jest! Do tego właśnie służą **struktury**. Wystarczy zdefiniować:

struct.rkt

```
3  (struct node (data l r))
4  (struct leaf ())
```

Kompilator sam utworzy za nas procedury `node`, `node?`, `node-data`, `node-l`, `node-r`, `leaf` oraz `leaf?`. Możemy użyć ich w programie:

struct.rkt

```
6  (define (height t)
7    (cond [(leaf? t) 0]
8          [(node? t) (+ 1 (max (height (node-l t))
9                               (height (node-r t))))]))
10
11 ; Tego lepiej nie probowac:
12 ; (define (size t)
13 ;   (cond [(leaf? t) 0]
14 ;         [(node? t) (+ 1 (size (third t))
15 ;                         (size (fourth t)))]))
```

Dwie zalety struktur to uszczelnienie abstrakcji i dopasowanie wzorca.

Uszczelnienie abstrakcji oznacza, że struktury z pliku `struct.rkt` to nie tylko skrót notacyjny dla tego, co dzieje się w `conventional.rkt`. Struktury są oddzielnym rodzajem wartości w Rackecie, więc nie ma innej możliwości, by zajrzeć „do środka" niż użycie interfejsu.

Dopasowanie wzorca (ang. *pattern matching*) to mechanizm w Rackecie, który łączy wyrażenie `cond` z rozbiorem struktury na części. Dla przykładu:

struct.rkt

```
17 (define (height-pm t)
18   (match t [(leaf) 0]
19            [(node d l r) (+ 1 (max (height-pm l)
20                                    (height-pm r)))]))
```

Wyrażenie `(match t ...)` pozwala sprawdzić, jaką strukturą jest wartość wyrażenia `t`, a następnie nazwać argumenty konstruktora użytego, by tę wartość

stworzyć. W szczególności w powyższym fragmencie kodu zmiennej `l` (związanej z drugą klauzulą, czyli `(node ...)`) przypisane jest lewe poddrzewo, a zmiennej `r` – prawe poddrzewo. Jest to wygodne, bo w ciele klauzuli nie musimy już używać selektorów.

Dodajmy, że domyślnie abstrakcja jest tak silna, że interpreter Racketa nawet nie chce pokazać, co taki wierzchołek ma w środku. Przykładowo:

```
> (node 2 (leaf) (leaf))
#<node>
```

Dla łatwiejszego testowania, możemy jednak zmusić interpreter do bardziej obszernego pokazywania wartości poprzez dodanie flagi `#:transparent`:

struct.rkt

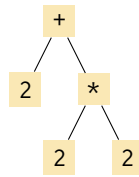
```
3 (struct node (data l r) #:transparent)
4 (struct leaf ()         #:transparent)
```

Teraz możemy zobaczyć, co struktura ma w środku:

```
> (node 2 (leaf) (leaf))
(node 2 (leaf) (leaf))
```

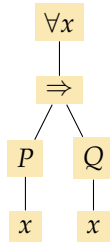
6.2. Składnia konkretna i abstrakcyjna

Kod źródłowy programu komputerowego zwykle żyje w pliku tekstowym, jest więc ciągiem znaków. A jednak rzut oka na to, jak wygląda składnia większości języków programowania, ujawnia, że programy mają strukturę. W programach znajdziemy wyrażenia arytmetyczne[1], nawiasy, bloki itp. Pisząc lub analizując program, bez wątpienia mamy w głowie właśnie takie strukturalne rozumienie kodu. Widząc Racketowe wyrażenie `(+ 2 (* 2 2))`, oczami wyobraźni widzimy drzewo:



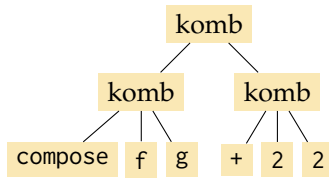
[1] Pierwszym językiem, który realizował pomysł, by zamiast `PUSH 2; PUSH 2; MULT; PUSH 2; ADD` pisać `2*2+2`, był Fortran stworzony w latach 50. XX wieku przez ekipę pod dowództwem Johna Backusa.

Ponieważ podczas drugiej tercji metod programowania będziemy pisać programy, które robią coś z innymi programami (głównie obliczały ich wartość), okazuje się, że strukturalne rozumienie programu bardzo ułatwi nam pracę. Tak samo jest z innymi obiektami w matematyce. Np. formułę logiczną $\forall x.P(x) \Rightarrow Q(x)$ można (a nawet należy) rozumieć jako następujące drzewo:



Sposób zapisu programu jako napisu „`(+ 2 (* 2 2))`" albo formuły jako napisu „$\forall x.P(x) \Rightarrow Q(x)$" nazywamy **składnią konkretną** (ang. *concrete syntax*). Natomiast jej strukturalne rozumienie w postaci drzewa nazywamy **składnią abstrakcyjną** (ang. *abstract syntax*). Jak widać, składnia abstrakcyjna uwalnia nas od konieczności myślenia o priorytecie operatorów (o ile składnia konkretna używa operatorów infiksowych), nawiasów, komentarzy, układu tekstu programu itd. Często też można natknąć się na skrót **AST** (*abstract syntax tree*) na określenie takich drzew.

Warto pamiętać, że format składni abstrakcyjnej wcale nie jest ustalony raz na zawsze, ani nie zawsze jest tak, że wszystkie etykiety w drzewie to fragmenty programu. Mamy tu pewną dowolność. Np. być może wygodniej jest myśleć o programie Racketowym `((compose f g) (+ 2 2))` jako o drzewie, w którym węzły nie reprezentują operatorów, a kombinację, której wszystkie elementy (w tym operator) są kolejnymi dziećmi:

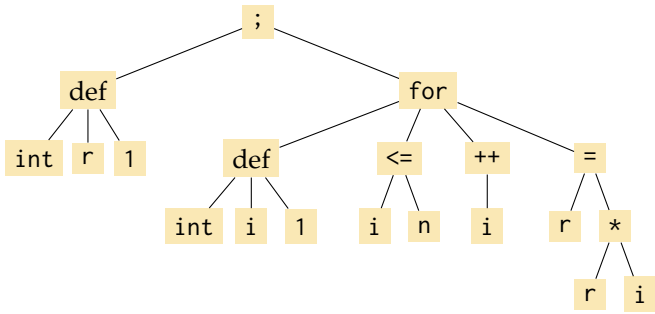


Jako inny przykład, rozważmy następujący program w C++:

```
int r = 1;
for(int i = 1; i <= n; i++)
{
  r = r * i;
```

```
}
```

Można go przedstawić jako następujące drzewo:



Nie oznacza to, że składnia konkretna nie jest ważna. Dobrze pomyślana może pozytywnie wpłynąć na czytelność i zwięzłość programów (np. Haskell), a nieprzemyślana może doprowadzić do katastrofy (np. nazwy procedur `car` i `cdr` w Rackecie, albo definicje typów w C). Częstą praktyką jest to, by język programowania składał się z małego, prostego do zrozumienia i zweryfikowania **rdzenia** (ang. *core*) oraz **lukru syntaktycznego** (ang. *syntactic sugar*[2]). Lukier to konstrukcje, które wprowadzają nowe środki wyrazu, ale na tyle proste, że można je wyeliminować już na etapie analizy składniowej. Np. w Rackecie wyrażenie

```
(let* ([x e1] [y e2] [z e3]) f)
```

jest równoważne wyrażeniu

```
(let ([x e1]) (let ([y e2]) (let ([z e3]) f)))
```

więc tłumacząc komuś, jak działa konstrukcja `let*`, albo pisząc kompilator Racketa, wystarczy wyrazić ją za pomocą kaskady wyrażeń `let`. Krańcowym przykładem lukru syntaktycznego jest popularny język Elixir, który w całości jest jedynie lukrem syntaktycznym dla języka Erlang.

Racket ma bardzo prostą składnię konkretną, bez dużej ilości lukru. Ma to swoje wady i zalety, do których wrócimy w sekcji 6.7.

[2]Kalkując z języka angielskiego, niektórzy mówią „cukier syntaktyczny”. Poprawną wersję („lukier”) wprowadził Andrzej Blikle, który – jako prominentny cukiernik-celebryta – wiedział, co mówi.

6.3. Reprezentacja składni abstrakcyjnej

Są przynajmniej dwie perspektywy pozwalające zrozumieć, jak działa język programowania. Jedna to formalna matematyczna semantyka, z którą – w bardzo szczątkowej postaci – spotkali się Państwo przy okazji indukcyjnych dowodów równoważności programów na listach i drzewach. Drugim podejściem, tym, które będziemy uprawiać tutaj, jest napisanie kilku programów, które implementują różne narzędzia związane z językiem. W szczególności przydatne jest napisanie interpretera, który zadaje językowi semantykę.

Pierwszym krokiem jest zdefiniowanie, jak reprezentujemy programy wewnątrz naszego interpretera. Oczywiście, reprezentowanie go w postaci tekstu programu jest złym pomysłem: nie po to odkryliśmy, że program jest drzewem, żeby nie skorzystać z tego faktu. I rzeczywiście, skoro wyrażenia arytmetyczne to drzewa, możemy je reprezentować przy użyciu typu danych zdefiniowanego przy użyciu następujących struktur:

arith.rkt

```
3 (struct const (val)    #:transparent)
4 (struct binop (op l r) #:transparent)
```

Struktura `const` reprezentuje stałą numeryczną, a `binop` reprezentuje operator `op` (reprezentowany jako symbol) zaaplikowany do dwóch argumentów, które też są reprezentacją wyrażeń. Przykładowo, wartość reprezentującą nasze wyrażenie `(+ 2 (* 2 2))` możemy zdefiniować w Rackecie tak:

arith.rkt

```
7 (define 2+2*2 (binop '+ (const 2)
8                          (binop '* (const 2)
9                                    (const 2))))
```

Jak mówiliśmy, jest pewna dowolność w tym, jaką składnię abstrakcyjną przyjmiemy dla danego języka programowania. Oczywiście, jest też pewna dowolność w tym, jaką wybierzemy reprezentację – to w końcu prywatna sprawa danego kompilatora/interpretera. Na metodach programowania będzie trochę inaczej. Dla uproszczenia sprawy założymy, że to właśnie składnia abstrakcyjna jest naszym wejściem, a nie składnia konkretna. W ten sposób ominiemy proces analizy składniowej (parsowania), czyli zamiany składni konkretnej na abstrakcyjną. Nie jest tak dlatego, że jest to faza mało ciekawa, ale dlatego, że na wszystko nie starczy nam czasu. Pisanie programów przez bezpośrednie wyrażenie składni abstrakcyjnej jak powyżej jest stanowczo niewygodne, dlatego uprościmy sobie sprawę w sekcji 6.7 używając cytowania.

Innym ciekawym aspektem reprezentacji składni abstrakcyjnej jest to, że takie drzewo może być dodatkowo udekorowane informacjami przydatnymi kompilatorowi/interpreterowi. Może to być pozycja danego węzła w kodzie źródłowym (przydatne, gdy próbujemy zgłosić programiście błąd) albo oznaczenie typu danego (pod)wyrażenia, nie pochodzącego z samego kodu źródłowego, a wywnioskowanego podczas fazy rekonstrukcji typów.

6.4. Świat wyrażeń i świat wartości

Kolejnym etapem konstrukcji interpretera jest zastanowienie się, co jest **wartością** (ang. *value*), czyli możliwym wynikiem ewaluacji programu. W wypadku wyrażeń arytmetycznych odpowiedź jest prosta: są to liczby. Należy jednak pamiętać, że w ogólnym przypadku wartości mogą być skomplikowane, a są też sytuacje, gdy interpreter używa jakiś wartości pośrednich, które mogą być wynikiem obliczania jakiegoś podwyrażenia, a nie mogą być wynikiem całego programu.

W ten sposób znaleźliśmy się w sytuacji, która może powodować lekką dezorientację na temat tego, co żyje na którym poziomie. W przypadku wyrażeń arytmetycznych ciężko o jakieś mocne pogmatwanie, ale proszę zauważyć, że mamy już trzy odrębne byty. Dla przykładu, jeśli chcemy wyposażyć nasz język programowania w listy (co zrobimy w okolicach wykładu 7.), mamy do czynienia z trzema różnymi pojęciami list: listy w Rackecie, wyrażenia w interpretowanym języku, które obliczają się do listy, oraz wartości reprezentujące listy. A ponieważ część z tych rzeczy ma wspólną reprezentację w Rackecie (np. listy jako wartości mogą być reprezentowane po prostu przez listy Racketowe), może dojść do pomyłek. Dlatego zawsze będziemy wyraźnie zaznaczać, co jest wyrażeniem, a co wartością poprzez zdefiniowanie predykatów. Wyrażenia arytmetyczne to:

arith.rkt

```
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     [_ false])))
```

A wartości, do których takie wyrażenia się obliczają to:

arith.rkt

```
19 (define (value? v)
20   (number? v))
```

Interpreter to program (dla nas: procedura w Rackecie), który przyjmuje na wejściu tekst programu, a na wyjściu daje wartość (w znaczeniu zadanym powyżej). Tak jak już mówiliśmy, interpreter dla nas składa się z dwóch faz: **analizy składniowej** i **ewaluacji**:

Składnia konkretna	Analiza składniowa	Składnia abstrakcyjna	Ewaluacja	Wartość

`"(+ 2 (* 2 2))"`	⇝	`(binop '+ (const 2)`
		`(binop '* (const 2)`
		`(const 2)))`
			⇝	6

Nas interesować będzie głównie druga faza.

6.5. Ewaluacja wyrażeń arytmetycznych

Ewaluator to procedura `eval`, która jako swój argument bierze wyrażenie arytmetyczne (czyli Racketową wartość spełniającą predykat `expr?`) i oblicza jego wartość (czyli produkuje coś, co spełnia predykat `value?`).

- Wartością stałej jest ona sama, czyli `(eval (const n))` oblicza się do `n`.
- Wartość wyrażenia będącego operatorem binarnym obliczamy tak, jak robiliśmy to w modelu podstawieniowym: wyliczamy wartości argumentów, a potem aplikujemy do nich interpretację(!) operatora. Czyli jeśli w wyrażeniu `(binop '+ e1 e2)`, wyrażenie `e1` oblicza się do `v1`, a `e2` oblicza się do `v2`, to całość oblicza się do `(+ v1 v2)` (bo interpretacją operatora `'+` jest procedura `+`).

Teraz wystarczy to sformalizować w postaci procedury rekurencyjnej. Najpierw definiujemy interpretację operatorów, a potem interpretację wyrażeń zgodnie z powyższym schematem:

arith.rkt

```
22 (define (op->proc op)
23   (match op ['+ +] ['- -] ['* *] ['/ /]))
24
```

```
25 (define (eval e)
26   (match e
27     [(const n) n]
28     [(binop op l r) ((op->proc op) (eval l) (eval r))]))
```

Możemy przetestować:

```
> (eval 2+2*2)
6
```

6.6. Maszyna abstrakcyjna i kompilacja do RPN

Kompilacja to kolejne pojęcie, o którego istnieniu warto wiedzieć, a w przyszłości zapisać się na dedykowany kompilatorom przedmiot. **Kompilator** to program, który tłumaczy program w jakimś języku programowania do innego języka programowania, np. kodu maszynowego, *bytecode*'u jakieś maszyny wirtualnej (Java Bytecode, WebAssembly) albo po prostu innego języka programowania ogólnego przeznaczenia (cała masa kompilatorów z narzędzi front-endowych do Javascriptu). W tej sekcji napiszemy prosty kompilator.

Skoro kompilator, to potrzebujemy aż dwóch języków: **źródłowego** (ang. *source*) i **docelowego** (ang. *target*). Język źródłowy już mamy; to język naszych wyrażeń arytmetycznych. Językiem docelowym będą wyrażenia w **odwrotnej notacji polskiej** (ang. *reverse Polish notation*, *RPN*[3]). Programy w tym języku to niepuste listy, których każdy element jest albo stałą liczbową, albo operatorem:

rpn.rkt

```
6  (define (rpn-expr? e)
7    (and (list? e)
8         (pair? e)
9         (andmap (lambda (x) (or (number? x) (member x '(+ - * /))))
10                e)))
```

Wyrażeniom arytmetycznym zadaliśmy semantykę przy użyciu ewaluatora. Nie tylko służy on do wyliczenia wartości programów, ale także jest sposobem zadania formalnej semantyki. Wyrażeniom RPN damy semantykę używając innego formalizmu, **maszyny abstrakcyjnej**. Jest to Racketowa procedura definiująca proces iteracyjny, używająca dodatkowego argumentu, opisującego

[3]Polskość notacji pochodzi od narodowości jej twórcy – Jana Łukasiewicza – który urodził się we Lwowie, w cesarstwie Austro-Węgierskim.

stan modelujący (na odpowiednio abstrakcyjnym poziomie) maszynę fizyczną. Maszyna abstrakcyjna dla RPN powinna być Wam znana: używa ona stosu do przechowywania obliczonych już argumentów. Stos implementujemy przy użyciu listy (stara prawda mówi, że różnica między stosem a listą jest taka, że stos stoi, a lista leży). Ponieważ wyrażenia RPN też są reprezentowane jako lista, dla czytelności szczelnie zamkniemy listową reprezentację stosu za barierą abstrakcji przy pomocy struktury:

rpn.rkt

```
12 (struct stack (xs))
13
14 (define empty-stack (stack null))
15 (define (empty-stack? s) (null? (stack-xs s)))
16 (define (top s) (car (stack-xs s)))
17 (define (push a s) (stack (cons a (stack-xs s))))
18 (define (pop s) (stack (cdr (stack-xs s))))
```

Teraz możemy już napisać naszą maszynę. Widząc stałą liczbową na przedzie wyrażenia, wrzucamy ją na stos. Widząc operator – aplikujemy jego interpretację do dwóch pierwszych elementów na stosie:

rpn.rkt

```
23 (define (eval-am e s)
24   (cond [(null? e)
25          (top s)]
26         [(number? (car e))
27          (eval-am (cdr e) (push (car e) s))]
28         [(symbol? (car e))
29          (eval-am (cdr e)
30                   (push ((op->proc (car e)) (top s) (top
                          (pop s)))
31                         (pop (pop s))))]))
32
33 (define (eval-rpn e) (eval-am e empty-stack))
```

Kompilacja wyrazeń arytemtycznych do RPN nie jest trudna: w końcu RPN to postfiksowy zapis wyrażeń. Ponieważ liczba argumentów (*krotność*) operatorów jest stała, nie potrzebujemy nawiasów:

rpn.rkt

```
35 (define (arith->rpn e)
36   (match e
37     [(const n) (list n)]
38     [(binop op l r) (append (arith->rpn l)
39                             (arith->rpn r)
40                             (list op))]))
```

Najważniejszą obserwacją jest to, że struktura procedury `arith->rpn` jest bardzo podobna do struktury procedury `eval`. Można myśleć o `arith->rpn` jako o ewaluatorze, ale z innym typem wartości. Wartościami tutaj są nie liczby, a wyrażenia RPN. Istnieje cała działka teorii języków programowania, *abstrakcyjna interpretacja*[4], która eksploruje przestrzeń pomiędzy ewaluacją a kompilacją w celu analizy i wnioskowania o programach.

Warto uruchomić nasz kompilator:

```
> (eval-rpn (arith->rpn 2+2*2))
6
```

Bardziej dociekliwi i zainteresowani znajdą w pliku **arith-am.rkt** implementację maszyny abstrakcyjnej obliczającej wartość wyrażenia arytmetycznego bez potrzeby kompilacji do RPN – jest to alternatywna (ale równoważna) semantyka dla wyrażeń arytmetycznych.

6.7. Składnia konkretna przez cytowanie

Umiemy już ewaluować wyrażenia arytmetyczne, ale pozostał pewien niedosyt. Chcąc zdefiniować reprezentację wyrażenia $2 * 3 + 4 * 5$, musimy napisać tak:

```
(binop '+ (binop '* (const 2) (const 3))
           (binop '* (const 4) (const 5)))
```

Skoro reprezentacja wyrażeń jest taka rozwlekła, bardzo ciężko porządnie przetestować nasz ewaluator, a już na pewno utrudnia to próbę pochwalenia się nim przed kolegami. Potrzebujemy przynajmniej zalążka składni konkretnej!

Najpierw sprawimy sobie jedną przy użyciu **cytowania**. Cytowanie bardzo dobrze współgra z ascetyczną składnią Racketa, która zapewnia symetrię pomiędzy prostotą składni, a prostotą podstawowych struktur danych (pary, listy). Dla nas cytowanie to przydatne narzędzie, z którego teraz skorzystamy. Czym właściwie jest cytowanie? To zamiana wyrażenia na strukturę danych oddającą

[4] Czy ktoś mówił, że słowo „abstrakcja" wyskakuje na tym przedmiocie zza każdego winkla?

jego składnię abstrakcyjną – proszę zwrócić uwagę, że już czerpiemy zysk z nauczenia się o składni konkretnej i abstrakcyjnej, mogąc napisać tak zwięzłą definicję cytowania.

Zamiana składni konkretnej na abstrakcyjną to oczywiście pierwsza faza interpretera, z tą różnicą, że nie zaczynamy od *tekstu* programu, a od wyrażenia w Rackecie. Teraz wystarczy przetłumaczyć surowe listy będące wynikiem cytowania na składnię abstrakcyjną naszych wyrażeń arytmetycznych. Procedurę, która to robi, nazwiemy trochę na wyrost `parse`:

arith.rkt

```
34 (define (parse q)
35   (cond [(number? q) (const q)]
36         [(and (list? q) (eq? (length q) 3) (symbol? (first
               q)))
37          (binop (first q) (parse (second q)) (parse (third
                q)))]))
```

Teraz już możemy pisać wyrażenia arytmetyczne korzystając ze składni Racketa:

```
> (parse '(+ (* 2 3) (* 4 5)))
(binop '+ (binop '* (const 2) (const 3)) (binop '* (const 4)
    (const 5)))
> (eval (parse '(+ (* 2 3) (* 4 5))))
26
```

To ogranicza nas do racketopodobnej składni konkretnej, ale na nasze potrzeby to wystarczy. Oczywiście nie oznacza to, że jesteśmy ograniczeni do racketopodobnych języków. Np. możliwym przedstawieniem przy użyciu S-wyrażeń fragmentu programu w C ze strony 7 mogłoby być:

```
'(block
   (def int r 1)
   (for (def int i 1) (<= i n) (post++ i)
     (= r (* r i))))
```