Poetyckie aspekty programowania

Panicz Maciej Godek
godek.maciek@gmail.com

7 kwietnia 2016

Streszczenie

Niniejszy tekst stanowi rekonstrukcję i uzupełnienie prezentacji wygłoszonej przeze mnie 14 listopada 2014 na spotkaniu Geek Girls Carrots w Pomorskim Parku Naukowo-Technologicznym w Gdyni. Prezentuję w nim najogólniejsze rozważania dotyczące programowania z perspektywy filozofii języka i teorii literatury, starając się przy tym wyciągać wnioski natury praktycznej.

1 Wprowadzenie

Nazwa „poetyka" wywodzi się od greckiego słowa *poiesis*, oznaczającego wytwór, wytwarzanie lub po prostu twórczość. W tradycji przyjęło się za Arystotelesem używać tego słowa na określenie teorii dzieła literackiego. Podążając za tą tradycją, wybitny literaturoznawca dwudziestowieczny Roman Jakobson w ramach swojej teorii komunikatu określił mianem „funkcji poetyckiej" tę funkcję komunikatu, która polega na skupianiu uwagi na sobie, w przeciwieństwie do innych funkcji, polegających m.in. na skupianiu uwagi na nadawcy (funkcja ekspresywna), odbiorcy (funkcja impresywna) czy kontekście (funkcja referencyjna).

W szczególności nie należy mylić poetyki (jako teorii) z poezją (jako wytworem działalności poetów). Przykładowo, kod we wstawce 1 na następnej stronie jest wierszem napisanym w języku Perl[1]. Jest to wprawdzie poprawny kod Perla, jednak z punktu widzenia języka Perl nie ma w nim nic ciekawego – jego jedyną wartością jest wrażenie, jakie wywiera na czytelniku – owo wrażenie zawdzięcza jednak jedynie temu, że jest wypowiedzią w języku angielskim.

Istnieją dużo ciekawsze formy poetyckie wyrażane w językach programowania. Przykładowo, wstawka 2 na stronie 3 zawiera kod programu (napisany w języku C), którego wykonanie generuje ten sam kod. Programy tego rodzaju noszą nazwę *quine'ów*, na cześć amerykańskiego filozofa Willarda van Ormana Quine'a. Nimi również nie będziemy się jednak zajmować, ponieważ – jakkolwiek

[1]Perl jest bardzo ciekawym językiem programowania, stworzonym przez amerykańskiego lingwistę i informatyka (w klasycznym sensie, tzn. noszącego sweterki i wąsy) Larry'ego Walla. Osoby zainteresowane tą jakże barwną postacią mogą dowiedzieć się więcej ze strony internetowej *http://wall.org/~larry/*

Algorytm 1 Poemat w języku Perl autorstwa Sharon Hopkins

```
#!/usr/bin/perl
APPEAL:
listen (please, please);
    open yourself, wide;
         join (you, me),
    connect (us, together),

  tell me.

   do something if distressed;

           @dawn, dance;
           @evening, sing;
           read (books, $poems, stories) until peacful;
           study if able;

sort your feelings, reset goals, seek (friends, family, anyone);

             do*not*die (like this)
             if sin abounds;

keys (hidden), open (locks, doors), tell secrets;
   do not, I-beg-you, close them, yet.

                                     accept (yourself, changes),
                                     bind (grief, despair);
      require truth, goodness if-you-will, each moment;
select (always), length(of-days)
```

rozwijające intelektualnie – jako czyste formy poetyckie są one bezużyteczne dla codziennej praktyki programistycznej.

Zagadnieniem, które pragnę podjąć w niniejszej rozprawce, jest spojrzenie na program komputerowy jako na tekst, w taki sposób, w jaki na tekst patrzą strukturaliści, oraz rozważenie języków programowania w taki sposób, w jaki rozważa się inne języki – tzn. jako media komunikacji i ekspresji myśli.

Moją motywacją jest silne przekonanie dotyczące tego, że programy komputerowe nie tylko pozwalają nam robić wiele rzeczy szybciej i dokładniej, ale również – a może przede wszystkim – mają duży wpływ na nasz sposób myślenia. Jest bowiem faktem udowodnionym naukowo[2], że już nawet sama umiejętność

[2] przez amerykańskich naukowców[3]

Algorytm 2 Przykładowy *quine* języka C

```
char*f="char*f=%c%s%c;main()
{printf(f,34,f,34,10);}%c";
main(){printf(f,34,f,34,10);}
```

pisania i czytania ma istotny wpływ na nasz sposób myślenia.

2 Struktura

> Myślisz, że wiesz, gdy się nauczysz,
> masz większą pewność, gdy zdasz egzamin,
> jeszcze większą nawet, gdy nauczasz innych,
> lecz całkowitą dopiero wtedy,
> gdy potrafisz to zaprogramować
>
> Alan Perlis

Niezależnie od tego, z jakim językiem programowania nie mielibyśmy do czynienia, każdy program jest tekstem i posiada pewne własności inherentne dla tekstu. Każdy tekst jest zaś pewnym skończonym ciągiem znaków zbudowanym w oparciu o pewne reguły. W szczególności, wszelki inteligibilny tekst zorganizowany jest w pewną strukturę: z pierwotnych znaków budujemy większe całościowe struktury, będące elementarnymi składnikami jeszcze większych struktur (dla przykładu, z liter budujemy słowa, ze słów zdania, ze zdań akapity itd.)

Oprócz pełnienia roli jednostek bazowych dla większych struktur, złożone struktury posiadają swoją *strukturę wewnętrzną*, czyli to, *w jaki sposób* są złożone. To, jakie sposoby składania są dopuszczalne, określają odpowiednie reguły – na przykład reguły gramatyczne albo logiczne.

Znajomość reguł kompozycji tekstu pozwala nam nie tylko budować zdania, ale również analizować je, a tym samym – rozumieć (to znaczy: sprowadzać z postaci złożonej do postaci prostej i zrozumiałej).

Reguły gramatyczne w szczególności służą do określania relacji między słowami. Tekst wzięty sam w sobie jest po prostu sekwencją słów. Relacje gramatyczne kodują jednak pewną tajemniczą własność, która nie dałaby się wywnioskować z samego położenia słów względem siebie. Rozważmy zdanie

(1) *Samochód, który został skradziony dziś rano, stoi na parkingu.*

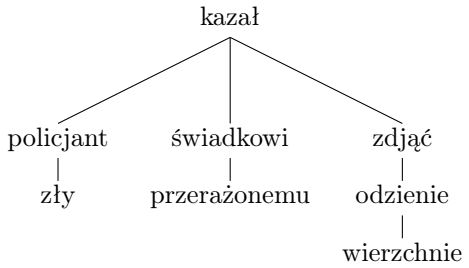
Jasne jest, że słowo „stoi" odnosi się do słowa „samochód", i to pomimo tego, że słowa te znajdują się dość daleko od siebie. Z tego powodu gramatycy (nawet

nauczyciele w szkołach podstawowych) posługują się nieco inną reprezentacją zdań, mianowicie – drzewami rozbioru składniowego.

Przykładowo, dla zdania

(2) *Zły policjant kazał przerażonemu świadkowi zdjąć odzienie wierzchnie.*

struktura rozbioru składniowego mogłaby wyglądać następująco:



Struktura składniowa zdania nie zawsze jest jednoznaczna. Na przykład w zdaniu

(3) *Zwiedziłem wieś koleżanki, która przywiązana była do tradycji.*

nie wiadomo, czy wyraz „która" odnosi się do wyrazu „wieś", czy „koleżanka". Jasne jest za to, że do zdekodowania struktury zdania niezbędna jest wiedza niezawarta w tekście.

Można byłoby jednak przyjąć odpowiednie konwencje notacyjne, dzięki którym struktura tekstu byłaby możliwa do wyczytania z samego tekstu, bez znajomości żadnych dodatkowych reguł gramatycznych. Na przykład, drzewo rozbioru zdania (2) moglibyśmy zakodować następująco:

```
(kazał (policjant zły)
       (świadkowi przerażonemu)
       (zdjąć (odzienie wierzchnie)))
```

W tekście tym użyłem wcięć, żeby uzyskać większą czytelność. Nic nie stałoby jednak na przeszkodzie, żeby zapisać ową sekwencję znaków ciurkiem – struktura zdania (2) byłaby w dalszym ciągu zachowana.

Zastosowana tu konwencja notacyjna pozwala zakodować dowolne drzewo. Czytelnik z pewnością zdążył już zauważyć ogólną prawidłowość, że każdy węzeł albo jest liściem (tak jak słowa „zły", „przerażonemu" i „wierzchnie"), albo rozgałęzieniem, i w tym ostatnim przypadku ma postać:

```
(nazwa węzeł-1 węzeł-2 ...)
```

gdzie poszczególne węzły same mogą być albo liśćmi, albo rozgałęzieniami.

Nie jest to jedyna możliwa konwencja. W istocie, niemal wszystkie języki programowania stosują różne konwencje, np. Python używa zamiennie wcięć i różnego rodzaju nawiasów; języki, których składnia wywodzi się z C (np. C++,

Java, JavaScript, PHP) używają w różnych kontekstach różnego rodzaju nawiasów ({}, (), [], <>), a niektóre z nich dodatkowo pozwalają w pewnych kontekstach je opuszczać. Takie komplikacje nie są jednak potrzebne, i w pewnym sensie mogą nawet być szkodliwe. Jedną z zalet wprowadzonej przez nas konwencji jest niewątpliwie prostota. Można się jej łatwo nauczyć, i można też łatwo nauczyć jej komputera.

3 Interpretacja

Żaden tekst nie miałby jakiejkolwiek wartości, gdyby nie wyrażał jakiegoś znaczenia. Czym jednak miałoby być owo mistyczne znaczenie?

Filozofia obfituje w różne próby odpowiedzi na to pytanie. Próby takie noszą nazwę *teoryj znaczenia*. Niektóre z nich próbują twierdzić, że znaczeniami wyrażeń są różne rzeczywiste obiekty, do których te wyrażenia miałyby się odnosić. Inne postulują, że owymi znaczeniami są myśli, które wspomniane wypowiedzi inspirują w umysłach ich odbiorców. Jeszcze inne – że są nimi zamieszkujące „platoński świat" idee. Na potrzeby niniejszej pracy przyjmiemy jednak dużo prostszą teorię znaczenia, kojarzoną z nurtem strukturalizmu, mianowicie – że znaczeniami tekstów są inne teksty.

Warto się zastanowić, czy przyjmując taką koncepcję znaczenia, nie popadamy w błędne koło: czy mianowicie nie zakładamy tego, co chcemy wyjaśnić.

Nasz domniemany strukturalista mógłby jednak bronić swojej teorii następująco. Wiemy z doświadczenia, że pewne wypowiedzi są proste i niejako zrozumiałe same przez się, zaś te bardziej złożone możemy wyjaśnić w oparciu o te prostsze – w rzeczy samej, na tym właśnie polega definiowanie! Rozumienie wypowiedzi będzie zatem polegało na sprowadzeniu jej do najprostszych terminów.

Z samej tej koncepcji wynika już kilka ważnych dla nas wniosków. Mianowicie, do budowania znaczenia tekstu będziemy potrzebować jakichś środków kompozycji, oraz – analogicznie – do analizowania znaczenia będziemy potrzebowali odpowiednich środków dekompozycji. Zanim zajmiemy się samymi tymi środkami, warto się przyjrzeć pewnym ogólnym własnościom semantycznym wypowiedzi językowych.

Jedną z bardziej zdumiewających cech języka było dla lingiwstów XX-wiecznych to, że my, ludzie, potrafimy wypowiadać i rozumieć nieskończenie wiele wypowiedzi językowych, których nigdy wcześniej nie słyszeliśmy.

Wśród hipotez starających się wyjaśnić ten fenomen warto wymienić tezę o *kompozycjonalności* języka. Głosi ona mianowicie, że

> znaczenia wyrażeń językowych zależą od (i) znaczeń ich elementów składowych oraz (ii) sposobów połączenia tych elementów.

Wprawdzie można wskazać na wyrażenia językowe (tzw. idiomy) do których owa teza się nie stosuje – na przykład znaczenia zwrotu „świnka morska" nie sposób wydedukować z samej znajomości słów „świnka" i „morska" – jednak wydaje się, że w przypadku bardzo wielu wyrażeń ów postulat nie budzi większych

zastrzeżeń (np. w sformułowaniu „zielony balon" wystarczy, że wiemy, czym jest balon i jak wygląda kolor zielony).

Tym niemniej, nawet jeśli pominiemy idiomy, w naszym języku będziemy mogli znaleźć wyrażenia w pewien sposób zaburzające ową zasadę kompozycjonalności.

Rozważmy na przykład zdanie:

(4) *Louis Lane wie, że Superman umie latać.*

Wiemy przy tym, że Superman i Klark Kent to dwie nazwy odnoszące się do tej samej osoby. Stąd też gotowi jesteśmy potraktować następujące zdania jako równowazne:

(5) *Superman mieszka w Smallville.*

(6) *Klark Kent mieszka w Smallville.*

Mimo tego, nie możemy uznać zdania

(7) *Louis Lane wie, że Klark Kent umie latać.*

za zdanie równoważne zdaniu (4). Wynika to stąd, że nazwy „Superman" i „Klark Kent" występują w nich w tzw. kontekście *intensjonalnym* (w przeciwieństwie do tzw. kontekstu *ekstensjonalnego*, w którym owe nazwy pojawiają się w zdaniach (5) i (6)). Mówiąc w skrócie – chodzi o to, że w zdaniach (4) i (7) nie mówimy o rzeczywistym Supermanie i Klarku Kencie, tylko o przekonaniach Louis Lane dotyczących tych (potencjalnie różnych) osób.

Innymi słowy, nazwy „Superman" i „Klark Kent" zostały użyte w zdaniach (4) i (7) w sposób przypominający tzw. mowę zależną. Moglibyśmy np. sparafrazować zdanie (4) do następującej postaci:

(8) *Louis Lane byłaby gotowa uznać zdanie „Superman umie latać".*

W zdaniu tym użyliśmy cudzysłowu, dzięki czemu mogliśmy zaakcentować to, że słowo „Superman" nie jest w nim użyte w normalnej supozycji, tylko jest wzmiankowane, czy inaczej – użyte w tzw. supozycji materialnej[3].

Ktoś, kto zna się nieco na rzeczy, mógłby jednak zakwestionować poprawność zdania (8), ponieważ przytoczona w niej fraza „Superman umie latać" jest wyrażona w języku polskim, zaś Louis Lane umie mówić jedynie po amerykańsku. Moglibyśmy poradzić sobie z tym problemem, pisząc:

(9) *Louis Lane byłaby gotowa uznać zdanie „Superman can fly".*

Zdanie to jednak może być trudne do zrozumienia dla osób nie posługujących się językiem amerykańskim. Również ten problem dałoby się jakoś obejść:

(10) *Louis Lane byłaby gotowa uznać zdanie „Superman umie latać" wyrażone w języku amerykańskim.*

[3]Teoria supozycji pochodzi od XIII-wiecznego filozowa angielskiego, Williama Ockhama, znanego szerzej dzięki postulatowi minimalizmu ontologicznego teorii naukowych, zwanemu „brzytwą Ockhama".

Tym, co jest znamienne dla zdań (8), (9) i (10) jest to, że stanowią one zdania o zdaniach, zatem pełnią one funkcję metajęzykową względem zacytowanych w nich zdań.

Owa możliwość odnoszenia się do wyrażeń językowych (w opozycji do ich normalnego użycia) nosi nazwę *abstrakcji metajęzykowej*, i stanowi warunek możliwości uprawiania poetyki i filozofii języka.[4] Stanowi też wyjasnienie tego, dlaczego niektóre wyrażenia językowe nie zachowują się kompozycjonalnie.

Z pojęciem kompozycjonalności związane jest inne, na swój sposób dualne pojęcie – mianowicie pojęcie „przezroczystości odniesieniowej" („referential transparency"). Jest to idea, według której pewne złożone wyrażenie językowe możemy zastąpić jego desygnatem (rozumianym tutaj jednak znów – w świetle przyjętej przez nas teorii znaczenia – jako inne wyrażenie językowe). Na przykład w zdaniu

(11) *Suma liczb dwa i trzy jest nieparzysta.*

moglibyśmy zastąpić wyrażenie „suma liczb dwa i trzy" wyrażeniem „liczba *pięć*", ponieważ to właśnie do liczby 5 odnosi się to pierwsze wyrażenie.

Pojęcie przezroczystości odniesieniowej jest szczególnie istotne przy rozważaniu języków programowania – głównie z tego powodu, że wyrażenia w językach etnicznych są przezroczyste odniesieniowo ze swojej natury. Najlepszym przykładem nieprzezroczystości odniesieniowej w języku naturalnym, jaki udało mi się znaleźć, jest zdanie:

(12) *Ty, ty i ty – do dyrektorki!*

Zakładamy przy tym, że osoba, która wypowiada to zdanie, przy każdym wypowiedzeniu słowa „ty" wskazuje palcem inną osobę. W takim przypadku użyte kilkukrotnie słowo „ty" przy każdym użyciu odnosi się do innego obiektu.

4 Programy komputerowe

W przypadku programów komputerowych wyrażenia językowe mogą jednocześnie wykonywać pewne operacje, które modyfikują kontekst wykonania programu. Takie operacje nazywane są skutkami ubocznymi.

Rozważmy definicję funkcji wyrażoną w pseudo-kodzie we wstawce 3 na następnej stronie.

[4] Jest też fundamentem dla teorii prawdy sformułowanej przez wybitnego polskiego logika Alfreda Tarskiego w rozprawie *Pojęcie prawdy w językach nauk dedukcyjncyh* z 1933 roku[6].

Alfed Tarski rozpoczął swoją uwagę od następującego spostrzeżenia:

> Zdanie „Śnieg jest biały" jest prawdziwe wtedy i tylko wtedy, gdy śnieg jest biały.

Następnie przeszedł do sformułowania ogólnego schematu, pod który podpadają wszystkie zdania tego typu (schemat ten nazywany jest w literaturze „T-schematem"):

> Zdanie «ϕ» jest prawdziwe wtedy i tylko wtedy, gdy ϕ.

Owe rozważania miały doprowadzić Tarskiego do pewnych wniosków dotyczących definiowalności prawdy w językach formalnych, jednak przytaczam je tutaj tylko w celu zilustrowania tego, jakie możliwości daje abstrakcja metajęzykowa.

Algorytm 3 Funkcja nieprzezroczysta odniesieniowo

```
globalValue = 0;

integer function rq(integer x)
begin
   globalValue = globalValue + 1;
   return x + globalValue;
end
```

Gdybyśmy chcieli policzyć wartość wyrażenia

```
rq(2) + (rq(2) * rq(2))
```

napotkalibyśmy na pewną trudność – mianowicie każde wywołanie `rq(2)` dawałoby w wyniku inną wartość – za pierwszym razem byłoby to 3, za drugim 4, a za trzecim 5. Dodatkowo, ostateczny wynik zależy od kolejności ewaluacji. Gdybyśmy liczyli wartości wyrażeń od lewej do prawej, wynikiem byłoby 3 + (4 * 5), czyli 23. Gdybyśmy jednak liczyli wartości w odwrotnej kolejności, otrzymalibyśmy 5 + (4 * 3), czyli 17.

Dla odmiany, analiza działania kodu używającego funkcji ze wstawki 4 byłaby dużo prostsza, ponieważ funkcja `rt` wywołana z tym samym argumentem daje zawsze ten sam wynik, dlatego w wyrażeniu

```
rt(2) + (rt(2) * rt(2))
```

możemy po prostu za kazde z wystąpień `rt(2)` podstawić wartość 3.

Algorytm 4 Funkcja przezroczysta odniesieniowo

```
integer function rt(integer x)
begin
   return x + 1;
end
```

Programy, w których nie występują skutki uboczne, nazywa się *programami funkcyjnymi*.

Dysponując tą wiedzą, możemy spróbować scharakteryzować nieco bardziej formalnie to, czym jest znaczenie programu:

$$\langle v, c^* \rangle = m(e, c).$$

Powyższy prosty wzór stwierdza tyle, że znaczenie (m – od „meaning”) jest funkcją od dwóch zmiennych, e – kodu programu oraz c – kontekstu (środowiska)

wykonania. Wartością tej funkcji jest zaś para uporządkowana, której pierwszy element, v, jest wartością wyrażenia e, zaś drugi element – c^* – jest nowym kontekstem (po wykonaniu programu).

W szczególności, jeżeli mamy do czynienia z programem funkcyjnym, ów wzór redukuje się do postaci:

$$\langle v, c\rangle = m(e, c).$$

Z drugiej strony, jeżeli mamy do czynienia z programem czysto imperatywnym, w którym wartości wyrażeń są nieistotne, bo wykonanie programu stanowi wyłącznie serię skutków ubocznych[5], możemy zapisać[6]

$$\langle \oslash, c^*\rangle = m(e, c).$$

Jednak większość programów znajduje się gdzieś pomiędzy tymi dwiema skrajnościami.

Jeżeli przywołamy nasze wcześniejsze rozważania dotyczące tekstu, przypomnimy sobie, że każdy tekst stanowi (wewnętrznie ustrukturyzowany) ciąg słów. Żebyśmy mogli ów tekst rozumieć, użyte w nim słowa muszą być nam znane. Jedynym wyjątkiem są słowa, które wzmiankujemy (np. w formie cytatu) albo takie, które dopiero definiujemy.

Doszliśmy też do wniosku, że interpretacja tekstu programu będzie polegała na wyrażeniu go w prostszych terminach – czyli na zredukowaniu go, bądź też sprowadzeniu do innej postaci.

Chcielibyśmy zatem jako programiści mieć możliwość budowania rzeczy złożonych z rzeczy prostych, nazywania tych nowych złożonych rzeczy (definiowania) oraz komponowania tych rzeczy (zarówno prostych jak i złożonych) w taki sposób, żeby wyrażały to, co chcemy wyrażać. Z drugiej strony, oczekiwalibyśmy, żeby zbudowane w ten sposób wypowiedzi dało się zrozumieć – to znaczy żeby istniały jakieś środki analizy (czy to intelektualne, czy mechaniczne), które pozwalałyby redukować terminy złożone do prostych.

Okazuje się, że istnieje pewien prosty system formalny, który spełnia wszystkie te wymagania. Nazywa się on *rachunkiem* λ (lambda), choć być może lepszą nazwą byłby „rachunek podstawieniowy". Wyrażenia rachunku λ mogą przyjmować postać stałych, zmiennych, λ-abstrakcji albo kombinacji.

Przykładowa λ-abstrakcja może mieć postać:

$$(\lambda \quad \underbrace{(x\ y)}_{argumenty} \quad \underbrace{x \quad \text{ma} \quad y}_{ciało})$$

λ-abstrakcja składa się z trzech elementów: specjalnego symbolu λ, listy dowolnie wielu zmiennych (którymi zasadniczo mogą być dowolne różne symbole)

[5]Sytuacja taka ma miejsce głównie w przypadku języka *asembler* albo tzw. programów na maszynę Turinga. Wówczas poszczególne wyrażenia interpretujemy jako „instrukcje" bądź „rozkazy" dla komputera.

[6]Symbol $\oslash$ oznacza, że dana wartość jest ignorowana, albo że jej po prostu nie ma.

oraz ciała, które również może stanowić dowolne λ-wyrażenie. W ciele wyrażenia w powyższym przykładzie występują dwie zmienne, x i y, które zostają związane argumentami λ-abstrakcji.

Kombinacje zapisujemy w postaci

$$(\textit{wyrażenie argumenty ...}),$$

gdzie *wyrażenie* jest dowolnym λ-wyrażeniem redukującym się do λ-abstrakcji (może być zatem albo λ-abstrakcją, albo kombinacją, która – być może po wielu krokach redukcji – redukuje się do λ-abstrakcji).

Reguły redukcji możemy określić następująco: stałe, zmienne i λ-abstrakcje redukują się do samych siebie (czyli nie redukują się wcale), natomiast w przypadku kombinacji najpierw dokonujemy redukcji *wyrażenia*, a kiedy zostanie ono zredukowane do λ-abstrakcji, zastępujemy całą kombinację ciałem tej λ-abstrakcji, w którym za wszystkie zmienne podstawiamy wartości odpowiednich argumentów.

Jakkolwiek skomplikowanie by to nie brzmiało, reguła ta jest całkiem prosta. Oto przykładowa redukcja kombinacji:

$$((\lambda\ (x\ y)\ x \text{ ma } y) \text{ Dorota kota})$$

To, że wyrażenie jest kombinacją, nie powinno budzić zastrzeżeń. *Wyrażenie* kombinacji jest λ-abstrakcją, w związku z czym redukuje się do siebie samego. Ciałem wyrażenia jest „x ma y”. Występują w nim dwie zmienne, x oraz y. Odpowiadają im kolejno argumenty Dorota oraz kota. W wyniku redukcji kombinacji otrzymamy zatem wyrażenie[7]

Dorota ma kota

Chociaż prosta operacja podstawiania argumentów za zmienne w ciałach λ-abstrakcji może wydać się czymś banalnym, pozwala ona na zdefiniowanie działań arytmetycznych na liczbach naturalnych oraz instrukcji warunkowej `if-then-else`[8]. Osoby zainteresowane odsyłam do znakomitego skryptu Johna Harrisona dotyczącego programowania funkcyjnego [4]. W ramach zachęty proponuję zaś dokonanie redukcji następującej kombinacji (sugerowałbym jednak nie próbować zbyt długo!):

$$((\lambda\ (x)\ (x\ x))\quad (\lambda\ (x)\ (x\ x)))$$

[7] Uważna czytelniczka z pewnością zwróciła uwagę, że wynik redukcji nie jest już λ-wyrażeniem: nie jest bowiem ani stałą, ani zmienną, ani λ-abstrakcją, ani kombinacją – jest za to układem trzech stałych. Nie powinno to jednak przeszkadzać w zrozumieniu przykładu. Wystarczy, że uznamy wynik redukcji za nieredukowalny, czyli wyrażony w pierwotnych terminach.

[8] Tak naprawdę, rachunek λ pozwala na zakodowanie dowolnej funkcji obliczalnej – jego moc wyrazu jest zatem równoważna Uniwersalnej Maszynie Turinga albo maszynie RAM. Przewagą rachunku λ jest jednak to, że oprócz wyrażania różnych pojęć i operacji, pozwala także na uchwycenie ich, ponieważ obecność argumentów w λ-abstrakcjach umożliwia nam dodatkowo nadawanie nazw różnym wartościom.

Dla naszych potrzeb wprowadzimy do naszego języka jeszcze dwie specjalne konstrukcje[9]:

`(define symbol wartość)`

działa w ten sposób, że wiąże `wartość` z `symbolem` – od tej pory każdy `symbol` (niebędący argumentem λ-abstrakcji) będzie redukował się do swojej `wartości`;

`(if warunek działanie alternatywa)`

działa w ten sposób, że najpierw dokonuje redukcji `warunku`. Jeżeli `warunek` redukuje się do specjalnej wartości logicznej „fałsz", to cała instrukcja redukuje się do `alternatywy`. W przeciwnym razie instrukcja redukuje się do `działania`.

Oprócz tych środków łączenia potrzebujemy również jakichś pojęć pierwotnych. Tradycyjnie za pojęcia pierwotne w informatyce teoretycznej uznaje się liczby i podstawowe działania na nich[10] – dodawanie, odejmowanie, mnożenie i dzielenie. Dla spójności, nasz język będzie stosował onawiasowaną notację polską – działania arytmetyczne będziemy zawsze pisać przed argumentami. Przykładowo, zamiast pisać

$$2 + 2,$$

zapiszemy

`(+ 2 2)`

– dzięki temu możemy traktować działania arytmetyczne tak, jak do tej pory traktowaliśmy kombinacje i λ-abstrakcje.

Opisanego w ten sposób języka możemy użyć do zdefiniowania funkcji *silnia*. Silnię z liczby n zapisujemy $n!$ i czytamy „n-silnia"[11]. Funkcję tę można zdefiniować nieformalnie jako

$$n! = 1 \cdot 2 \cdot ... \cdot (n-1) \cdot n$$

dla dowolnego naturalnego $n > 0$. Można też zdefiniować silnię rekurencyjnie:

$$n! = \begin{cases} 1 & dla\ n = 0 \\ n \cdot (n-1)! & dla\ n > 0 \end{cases}$$

[9] Konstrukcje te wprawdzie daje się wyrazić w rachunku λ, ale jest to dość skomplikowane, i z tego powodu na potrzeby prezentacji wygodniej uznać je za wyrażenia pierwotne. Przytoczę tutaj – bez uzasadnienia ani żadnych wyjaśnień – wyrażenie służące do uzyskania efektowi równoważnemu użyciu `define`. Jest to tzw. kombinator paradoksalny albo Y-kombinator, i przytaczam go tu ze względu na jego osobliwe piękno:

$$(\lambda\ (f)\ ((\lambda\ (x)\ (f\ (x\ x)))\ (\lambda\ (x)\ (f\ (x\ x)))))$$

[10] Chociaż nie chciałbym, żeby użycie przykładu matematycznego mogło kogokolwiek zniechęcić – istnieje wiele programów, które w ogóle nie stosują działań arytmetycznych – przyznam, że trudno mi zerwać z tą tradycją – głównie dlatego, że przykłady odnoszące się do liczb są prostsze w analizie od innych przykładów, jakie mógłbym podać.

[11] Jeżeli mamy w danym pokoju n krzeseł, to $n!$ oznacza np. ilość sposobów, na które możemy posadzić na tych krzesłach n osób (zakładając, że każda z tych osób będzie siedzieć).

Definicję tę możemy bezpośrednio przełożyć na „naš jezik”[12]:

```
(define ! (lambda (n)
            (if (= n 0)
                1
                (* n (! (- n 1))))))
```

Możemy sprawdzić, czy ta definicja w świetle naszej reguły redukcji rzeczywiście działa. Spróbujmy zobaczyć, co się stanie, jeśli napiszemy

```
(! 5)
```

Zgodnie z regułą podstawiania wartości za symbole, powyższe wyrażenie zostanie „zredukowane” do postaci:

```
((lambda(n)(if (= n 0) 1 (* n (! (- n 1))))) 5)
```

W dalszym kroku, zgodnie z opisaną wcześniej regułą redukcji dla kombinacji, zastępujemy λ-abstrakcję jej ciałem, w którym za wszystkie użycia zmiennej `n` podstawiamy wartość 5:

```
(if (= 5 0) 1 (* 5 (! (- 5 1))))
```

Teraz zgodnie z regułą redukcji wyrażenia warunkowego, najpierw dokonujemy ewaluacji `warunku`. Ponieważ liczba 5 jest różna od liczby 0, wartością `warunku` jest fałsz logiczny. W związku z tym całe wyrażenie redukuje się do `alternatywy` wyrażenia `if`:

```
(* 5 (! (- 5 1)))
```

Zgodnie z regułą redukcji kombinacji, musimy najpierw dokonać redukcji wszystkich argumentów. Jedynym argumentem jest wyrażenie `(! (- 5 1))`. Jest ono kombinacją, zatem aby dokonać jego redukcji, musimy najpierw dokonać redukcji wszystkich argumentów. Jedynym argumentem jest wyrażenie `(- 5 1)`. Jest ono działaniem pierwotnym, które – zgodnie z tym, czego uczono nas w szkole – redukuje się do liczby `4`. Ostatnie wyrażenie możemy zatem przepisać jako:

```
(* 5 (! 4))
```

Wyrażenie `(! 4)` możemy zredukować analogicznie do tego, w jaki sposób redukowaliśmy wyrażenie `(! 5)`. W wyniku dostaniemy:

```
(* 5 (* 4 (! 3)))
```

Następnie, po redukcji wyrażenia `(! 3)`:

```
(* 5 (* 4 (* 3 (! 2))))
```

[12] widać, że symbol λ zastąpiliśmy słowem „lambda”. To dlatego, że niepostrzeżenie przeszliśmy już od czystego rachunku λ do języka programowania Scheme.

Następnie, po redukcji wyrażenia `(! 2)`:

```
(* 5 (* 4 (* 3 (* 2 (! 1)))))
```

Następnie, po redukcji wyrażenia `(! 1)`:

```
(* 5 (* 4 (* 3 (* 2 (* 1 (! 0))))))
```

Wyrażenie `(! 0)` zostanie zredukowane pośrednio do wyrażenia `(if (= 0 0) 1 ...)`[13], warunek `(= 0 0)` przyjmie wartość prawdy logicznej, w związku z czym wartością tego wyrażenia będzie ostatecznie liczba `1`. Ostatecznie nasze wyrażenie w najbardziej rozwiniętej postaci przyjmie wartość:

```
(* 5 (* 4 (* 3 (* 2 (* 1 1)))))
```

Przeprowadzając mnożenia (od środka do zewnątrz) uzyskamy pożądany wynik.

Zaprezentowane powyżej rozumowanie stanowi przykład tzw. podstawieniowego modelu obliczeń [1]. Tym, co jest dla niego charakterystyczne, jest to, że wykonanie programu polega na transformacji wystepujących w nim wyrażeń pierwotnych.

Nie jest to jedyny model obliczeń, jednak posiada taką zaletę, że ewaluację programu można z łatwością przeprowadzić na papierze. Inne popularne modele obliczeń wymagają od osoby analizującej kod zapamiętania całego stanu środowiska w każdym kroku wykonania obliczenia, co jest bardzo obciążające kognitywnie.

Powyższy program dokonywał operacji na liczbach. Oczywiście, liczby nie są jedynym typem danych, na których programy mogą operować. W szczególności – tak jak wcześniej przedstawialiśmy przykłady wypowiedzi odnoszących się do innych wypowiedzi – programy mogą manipulować kodem innych programów. Na przykład, wyrażenie

```
(lambda (n) (if (= n 0) 1 (* n (! (- n 1)))))
```

możemy traktować jako program, ale moglibyśmy też potraktować je jako listę trójelementową, której pierwszym elementem jest symbol `lambda`, drugim lista jednoelementowa `(n)`, a trzecim lista czteroelementowa, której pierwszym elementem jest symbol `if`, i tak dalej.

Do zapobiegania redukcji (czy też interpretacji) wyrażenia służy specjalny operator `quote`. Działa w taki sposób, że wyrażenie

`(quote x)`

redukuje się do `x` (przy czym `x` uprzednio nie zostaje poddane redukcji). Uświadamia nam to, że program, który wykonujemy, w istocie stanowi szczególne spojrzenie na dane, zaś wszelkie dane – jeśli tylko spojrzeć na nie odpowiednio – są również programami.

[13] `Alternatywa` wyrażenia warunkowego została zastąpiona symbolem `...`, ponieważ jej postaćnie ma w tym kontekście znaczenia.

5 Czy programowanie jest literaturą?[14]

Pomimo zaproponowanego tu przeze mnie nieco teorioliterackiego spojrzenia na programy komputerowe, naiwnością byłoby twierdzenie, że programowanie to taka sama literatura, jak każda inna. Z drugiej strony, to stwierdzenie w podobnym stopniu dotyczy każdej innej literatury, ponieważ każda literatura jest inna. Być może więc lepszym pytaniem byłoby: „do jakiego innego rodzaju literatury programowanie jest najbardziej podobne?". Pytanie to również jest kłopotliwe, ponieważ nawet wśród programów komputerowych występuje spora różnorodność.

Tym, co odróżnia programy komputerowe od form fabularnych, jest stosunek ilości definicji do stwierdzeń. W formach beletrystycznych definicji pojęć jest albo bardzo mało, albo nie ma ich wcale – stanowią one po prostu różne użycia pojęć powszechnie znanych. Programy komputerowe składają się zaś głównie z definicji pojęć, a stwierdzeń (czyli użyć pojęć poza kontekstem definicji) jest w nich stosunkowo mało albo nie ma ich wcale.

Pod tym względem programy komputerowe są najbardziej podobne do tekstów matematycznych albo logicznych. Celem tekstów matematycznych jest przede wszystkim prezentacja pojęć i twierdzeń dotyczących tych pojęć. Z tego względu teksty matematyczne składają się z definicji, przykładów, twierdzeń, dowodów (rozumowań) i ćwiczeń, a niekiedy również aksjomatów. (Twierdzenie w tym różni się od stwierdzenia, że ma charakter ogólny, a nie jednostkowy. W kontekście tekstu matematycznego za przykłady stwierdzeń można uznać przykłady użycia definicji)

Spojrzenie na program jako na tekst matematyczny wydaje się najbardziej obiecującym kierunkiem pod względem możliwości dalszego rozwoju. O ile jednak w dobrze napisanym programie definicje funkcji bądź obiektów można traktować po prostu jako definicje pojęć, testy jednostkowe – jako przykłady użycia pojęć, zaś asercje – jako swego rodzaju cząstkowe twierdzenia bądź aksjomaty – o tyle całkowicie brakuje w nich reprezentacji rozumowań, na podstawie których programiści dochodzą do określonych sformułowań.

Być może użycie jakiegoś systemu dowodowego do dowodzenia różnych własności programów, na których programista polega, mogłoby zwiększyć niezawodność wytwarzanego oprogramowania, w podobny sposób, w jaki użycie statycznego systemu typów (w językach takich, jak C, Java, ML czy Haskell) pozwala wyeliminować całe klasy błędów, na jakie narażone są języki dynamicznie typowane.

Warto przy tej okazji zwrócić uwagę na jedną rzecz. Niekiedy programiści uważają, że sposobem na poprawienie czytelności kodu jest umieszczenie w nim komentarzy i tzw. docstringów. Chociaż tego rodzaju działania mogą stanowić

[14] W sekcji tej prezentuję pewne ogólne wnioski odnoszące się do zagadnień, o których wcześniej nie wspominałem. Z tego powodu będą one zrozumiałe raczej dla osób, które miały styczność z tekstami matematycznymi, systemami dowodowymi oraz statycznie typowanymi językami programowania. Mam jednak nadzieję, że zdołam przynajmniej podsycić ciekawość pozostałych osób. W szczególności polecam podręcznik [5], ponieważ spełnia wszystkie powyższe kryteria.

doraźną pomoc i na przykład generować z kodu źródłowego użyteczną dokumentację, na dłuższą metę takie rozwiązanie wydaje się wadliwe, ponieważ jest sposobem obejścia niedostatków języków programowania. Komentarz w kodzie źródłowym świadczy o tym, że programista nie jest w stanie przekazać tej samej informacji w języku programowania.

Z drugiej strony, istnieje cały nurt tzw. programowania piśmiennego (literate programming), zapoczątkowany przez prominentnego amerykańskiego matematyka i programistę, Donalda Knutha. Polega on na tym, że przeznaczony do czytania przez człowieka tekst wyjaśniający istotę programu przeplata się z kodem zrozumiałym dla komputera. W taki sposób został napisany m.in. system składania tekstu TEX (z którego pośrednio korzystałem przy tworzeniu niniejszego dokumentu).

Podejście takie budzi jednak pewne kontrowersje. Wprawdzie każdy artykuł wyjasniający działanie jakiegoś programu można postrzegać jako „program piśmienny", wydaje się, że poza sferą dydaktyki styl ten raczej się nie przyjął.

Inny ciekawy nurt w programowaniu związany jest z tworzeniem tzw. interaktywnej fikcji, czyli swego rodzaju gier przygodowych opartych na tekście. Pomimo pozornej archaiczności interfejsu, istnieją całe środowiska autorów tworzących tego typu programy. Powstały również interesujące narzędzia ułatwiające autorom pracę. Jednym z najciekawszych takich narzędzi jest język programowania Inform 7, skonstruowany w ten sposób, że programy w nim same przypominają prozę.

Algorytm 5 Funkcja implementująca silnię w języku Inform 7

```
To decide what number is the factorial of (n - a number):
    if n is zero, decide on one;
    otherwise decide on the factorial of (n minus one) times n.
```

Wstawka 5 zawiera definicję funkcji implementującej silnię w tym języku[15]. Z pewnością wygląda ona interesująco, choć niewątpliwą wadą upodobniania języka programowania do języka naturalnego jest to, że ludzie są raczej przyzwyczajeni do tego, że język naturalny jest nieprecyzyjny i wieloznaczny – własnie z tych powodów chętniej używają w różnych sytuacjach notacji formalnych.

6 Podsumowanie

W prezentacji niniejszej chciałem przedstawić alternatywę wobec utartego poglądu, jakoby programowanie było równoznaczne warunkowaniu i wydawaniu komputerowi rozkazów. Moim celem było wykazanie, że programowanie w istocie rzeczy polega na tworzeniu, nazywaniu i używaniu pojęć – na fabrykacji i inhabitacji świata idei. Pod tym względem programowanie może stanowić jedną z najszlachetniejszych form ludzkiego myślenia, ponieważ uczy systematyczności i klarownego formułowania myśli. Zaakcentowanie aspektu programowania

[15] Źródło: *http://stackoverflow.com/questions/23930/factorial-algorithms-in-different-languages/91001#91001*

związanego z myśleniem – pomysł, żeby w kodzie źródłowym zapisywać nie tylko wyniki rozumowań, ale również same rozumowania – tak, jak w podręcznikach matematyki, w których obok definicji prezentuje się przykłady, twierdzenia i dowody – może wpłynąć pozytywnie zarówno na zrozumiałość, jak i niezawodność oraz wydajność tworzonego kodu.

Ważne jest przy tym, żeby programowanie – podobnie jak wszelkie inne formy pisarstwa – było traktowane jako aktywność społeczna.

Literatura

[1] Abelson Harold, Sussman Gerald, *Struktura i Interpretacja Programów Komputerowych,* WNT, Warszawa 2002. Pełny tekst książki w języku angielskim dostępny jest na stronie: *http://mitpress.mit.edu/sicp/full-text/book/book.html*

[2] Frege Gottlob, *Pisma Semantyczne*, PWN, Warszawa 1977

[3] Gleick James, *Informacja. Bit, wszechświat, rewolucja.* Wydawnictwo Znak, Kraków 2012

[4] Harrison John, *Introduction to Functional Programming*, skrypt do wykładu na uniwersytecie Cambridge. Tekst dostępny pod adresem: *http://www.cl.cam.ac.uk/teaching/Lectures/funprog-jrh-1996/all.pdf*

[5] Harrison John, *Handbook of Practical Logic and Automated Reasoning*, Cambridge University Press, Cambridge 2009

[6] Tarski Alfred, *Pisma logiczno-filozoficzne*, PWN, Warszawa 1995