Zadania Kwalifikacyjne

Po pierwsze — wybaczcie za obsuwę z zamieszczeniem zadań; jak już napisałem Luna… znaczy… praca to surowa pani i trzeba było gasić apokalipsę, a przy tych temperaturach to nawet (metaforyczna) woda płonie, czy raczej doznaje spontanicznej fuzji nuklearnej.

Po drugie — nie ma złych odpowiedzi. Bardziej niż na absolutnie poprawnej odpowiedzi zależy mi na poznaniu waszego podejścia i toku myślenia. Także jeżeli nie jesteś pewien (tudzież pewna) czy dasz sobie radę z tymi zadaniami — radzę śmiało próbować. Każdy ma szansę się dostać o ile tylko umie pokazać swój tok rozumowania przy próbie zrozumienia problemu.

Po trzecie — kto pyta nie błądzi. Co prawda nie chcę wam za dużo podpowiadać, coby nie serwować gotowych odpowiedzi (w końcu ważne jest dla mnie to, do czego sami dojdziecie), ale w przypadku jakichś wątpliwości co do treści zadań proszę śmiało pisać pod jaennirin+smoki@gmail.com i postaram się je rozwiać.

And without further ado… ready your pens and verte!

0.1 To proste

Nazwa warsztatów nie jest przypadkowa i jest związana z ich tematem. Znajdowanie informacji to bardzo przydatna umiejętność — odpowiedzią na to pytanie będzie więc podanie tego, do czego nazwa warsztatów się odnosi. Jako bonus powiem, że znalezienie tej odpowiedzi powinno pomóc w rozwiązywaniu dalszych zadań.

0.2 To z drzewem

Wyrażenia — czy to matematyczne, czy to jakiekolwiek inne — można zaprezentować na trzy podstawowe sposoby: w notacji prefiksowej, infiksowej i posfiksowej [1].

Najpopularniejsza notacja to ta infiksowa — przykładem może być wyrażenie `2 + 3 * 4`. Czy widzicie jaki może ona sprawiać problem? Otóż, trudno powiedzieć czy wyrażenie to oznacza `(2 + 3) * 4` czy `2 + (3 * 4)`. Oczywiście ktoś zaraz zakrzyknie „Toć po to kolejność działań jest!", ale właśnie o to chodzi — samo wyrażenie nie jest wystarczająco jasne bez dodatkowej pomocy np. w postaci kolejności działań.

Dlatego pewien polski matematyk, Jan Łukasiewicz, wymyślił notację prefiksową (nazwaną potem notacją polską), aby można było jednoznacznie (tj. bez nawiasów i kolejności wykonywania działań) zapisać wyrażenia. I tak `(2 + 3) * 4` w notacji prefiksowej to `* + 2 3 4`, podczas gdy `2 + (3 * 4)` to `+ 2 * 3 4`. Notacja postfiksowa (aka. odwrotna notacja polska) z kolei stawia to na głowie — postaci tych wyrażeń to odpowiednio `2 3 + 4 *` oraz `2 3 4 * +`. Jeżeli ktoś nie wierzy, że to są te same wyrażenia niech rozrysuje to sobie w postaci drzewek [2].

(a) `(2 + 3) * 4` **(b)** `2 + (3 * 4)`

Rysunek 0.2.1: Drzewka składniowe dla podanych wyrażeń, jakby ktoś chciał sprawdzić czy nie kłamię w powyższym paragrafie.

[1] żeby było śmieszniej — składnia popularnych języków programowania, takich jak C++/Java/JavaScript, miesza wszystkie trzy rodzaj notacji: wyrażenia matematyczne są zapisywane infiksowo, elementy składni takie jak definicje i wywoływanie funkcji czy pętle for są zapisywaneprefiksowo, a do tego istnieją takie operatory jak `++` które mają formy postfiksowe. Oczywiście jak można się domyśleć znacznie komplikuje to pisanie parserów dla takich języków.

Ale do rzeczy: komputery — wbrew temu, co po obserwacji tego co użytkownicy czasem robią licząc iż komputer się domyśli, może się wydawać — nie jest zbyt lotnym stworzeniem.

Niestety nie da się powiedzieć „Komputerze! Uczyń to, o czym myślę, ale w trymiga!" i spodziewać się, że zwoje mózgu elektronowego zaiskrzą, pudło chwilę poduma i zrobi dokładnie to, co trzeba. W związku z tym trzeba mu tłumaczyć wszystko dużymi literami, niczym ogłuszonej tchórzofretce (czyt. stworzyć algorytm).

W związku z tym — wymyślcie w jaki sposób wytłumaczyć komputerowi jak „rozumieć" proste wyrażenia matematyczne — na przykład poprzez zamienienie notacji infiksowej na postfiksową. Istnieje na to pewien algorytm stworzony przez pewnego znanego informatyka, ale nie musicie mi go podać co do joty — wystarczy że napiszecie w jaki sposób waszym zdaniem można komuś o możliwościach umsłowych naszej drogiej ogłuszonej tchórzofretki wytłumaczyć jak zamienić jedną postać na inną.

Za dodatkowe punkty można też napisać jaki widzicie problem w notacji polskiej, dlaczego ktoś wymyślił odwrotną notację polską i co mają z tym wszystkim wspólnego pociągi i matrioszki [3].

0.3 To ze stosem

Podczas wykonywania programu ważną informacją jest miejsce, w jakim program aktualnie się znajduje. Stosowanym do tego zwykle mechanizmem jest tzw. stos. Przy wejściu do jakiejś funkcji na stosie odkładana jest tzw. ramka, która opisuje wszystkie potrzebne informacje — skąd program przyszedł, dokąd ma wrócić, podręczne dane jakie wykorzystuje funkcja i tak dalej. Przy zakończeniu funkcji jej ramka jest zdejmowana ze stosu, a program kontynuuje funkcję, której ramka znajduje się niżej na stosie, w miejscu w którym ją przerwał w celu wywołania innej.

Niektóre funkcjonalności języków programowania — dla przykładu domknięcia [4] i kontynuacje [5] wymagają możliwości odniesienia się do ramki stosu z innego miejsca programu niż aktualnie wykonywane. Być może widzicie problem — wychodząc z funkcji jej ramka zostaje zdjęta ze stosu, a miejsce w którym była może następnie zostać zastąpiona inną ramką, innej funkcji, więc potrzebne nam wartości mogą po prostu „zniknąć".

[2] a jak ktoś wie co-nieco o drzewach, niech pomyśli jak notacje pre-, in- i postfixowa maja sie do przejścia drzewa w kolejności pre-, in- i postorder.

[3] tak, to takie podpowiedzi „na wszelki wypadek", gdyby nic wam nie przychodziło na myśl.

Pytanie brzmi — czy widzicie jakieś rozwiązanie tego problemu i jeżeli tak, to jakie? Czy mogą istnieć jego różne rozwiązania, zależnie od sposobu w jaki „zapamiętamy" stare ramki stosu? A może istnieje jakaś forma stosu, w którym ramki się nigdy nie gubią?

[4] domknięcie (ang. *closure*) — w języku z funkcjami anonimowymi i funkcjami wyższego rzędu może się zdarzyć, że taka funkcja „ucieknie" poza zakres leksykalny w jakim została stworzona, więc do swojego działania potrzebuje w jakiś sposób „pamiętać„ jakie wartości „widziała" w miejscu, w którym została stworzona, co nazywamy „domknięciem się" nad tą wartością.

Przykładowo (w jakimś pseudojęzyku) może wyglądać to następująco:

```
1  makeAdder(a) => { (b) => a + b }
2
3  let addTwo = makeAdder(2)
4
5  addTwo(3) == 5
6  addTwo(12) == 14
7
8  let addFive = makeAdder(5)
9
10 addFive(5) == 10
11 addFive(12) == 17
```

Jak widzicie `makeAdder` zwraca funkcję, która „pamięta" jaką wartość `a` widziała, gdy została stworzona.

[5] kontynuacja (ang. *continuation*) — w pewnych językach można reifikować (z łac. res — rzecz) pewne koncepcje, to znaczy czynić je wyrażalnymi w języku programowania. Przykładowo, wyobrazcie sobie, że musielibyśmy mówić o liczbach posiadając w naszym słowniku `1` oraz `+` — mówienie o rzeczach byłoby niewygodne, a wyrażanie wielkich liczb wręcz niemożliwe. Poprzez reifikację koncepcji `1 + 1 + 1 + 1 + 1` jako `5` zwiększamy moc wyrazu naszego języka i ułatwiamy tym samym komunikację. Tak samo reifikacja koncepcji w językach programowania zwiększa ich siłę wyrazu. Kontynuacje są czymś co reifikuje stan wykonaniu programu — można go następnie zapamiętać w jakiejś wartości i wrócić do tego samego miejsca później, zupełnie jakby program podróżował w czasie.