ROZDZIAŁ 3. Rekurencja

Tematem tego rozdziału jest jeden z najważniejszych mechanizmów używanych w informatyce, czyli *rekurencja*, zwana również *rekursją*[1]. Mimo że użycie rekurencji nie jest obowiązkowe, jej zalety są oczywiste dla każdego, kto choć raz spróbował tego stylu programowania. Wbrew pozorom nie jest to wcale mechanizm prosty i szereg jego aspektów wymaga dogłębnej analizy. Rekurencja upraszcza jednak opis wielu zagadnień (np. pozwala na łatwe definiowanie struktur opartych na fraktalach), a w wielu kwestiach informatycznych, np. w algorytmach dotyczących struktur „drzewiastych", jest wręcz niezbędna z uwagi na ich charakter.

Rozdział ma kluczowe znaczenie dla pozostałej części książki — o ile jej lektura może być dość swobodna i nieograniczona naturalną kolejnością rozdziałów, o tyle bez dobrego zrozumienia istoty rekurencji nie będzie możliwe swobodne analizowanie wielu zaprezentowanych dalej algorytmów i metod programowania.

Definicja rekurencji

Rekurencja jest często przeciwstawiana podejściu iteracyjnemu, czyli *n*-krotnemu wykonywaniu algorytmów w taki sposób, aby wyniki uzyskane podczas poprzednich iteracji (zwanych też przebiegami) mogły służyć jako dane wejściowe do kolejnych. Sterowanie iteracjami zapewniają instrukcje pętli (np. `for` lub `while`). Rekurencja działa podobnie, tylko funkcję zapętlania pełni wywoływanie się tej samej procedury (funkcji) przez siebie samą z innymi parametrami. Oczywiście w ciele procedury rekurencyjnej też można spotkać klasyczne pętle, ale odgrywają one rolę usługową i nie stanowią kryterium klasyfikacji algorytmu.

> Algorytmy iteracyjne polegają na *n*-krotnym wykonywaniu instrukcji w taki sposób, aby wyniki uzyskane podczas poprzednich iteracji (przebiegów) mogły służyć jako dane wejściowe do kolejnych. Algorytmy rekurencyjne realizują zapętlenie nieco inaczej, a mianowicie poprzez wywoływanie tej samej procedury (funkcji) przez siebie samą z innymi parametrami.

Warto wiedzieć, że programy zapisane w formie rekurencyjnej mogą być przekształcone — z mniejszym lub większym wysiłkiem — na klasyczną postać iteracyjną. Zagadnieniu temu poświęcę cały rozdział 13., na razie jednak zajmę się omówieniem mechanizmu rekurencji, którego zrozumienie nie jest takie proste, jak się na pierwszy rzut oka wydaje.

---

[1] Subtelna różnica między tymi pojęciami w zasadzie już się zatraciła w literaturze, dlatego też nie będę się niepotrzebnie zagłębiał w szczegóły terminologiczne.

Odpowiedzią na nie jest właśnie ten podrozdział. Użyty w nim przykład jest może banalny, niemniej doskonale nadaje się do zilustrowania sposobu wykonywania programu rekurencyjnego.

Już w szkole średniej (a może i podstawowej)[2] na lekcjach matematyki dość często używa się tzw. *silni n*, czyli iloczynu wszystkich liczb naturalnych od 1 do *n* włącznie. Ten użyteczny symbol zdefiniowany jest w następujący sposób:

$$0! = 1,$$

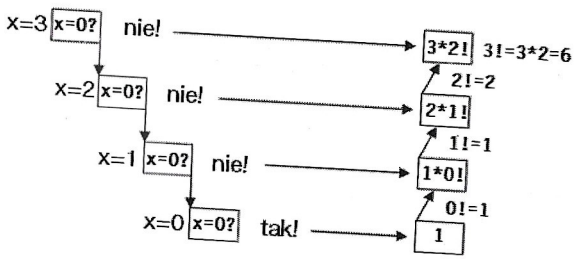
$$n! = n * (n-1)!, \text{ gdzie } n \in N,\ n \geq 1.$$

Podstawiając za *n* wartości dozwolone (czyli większe od 1), łatwo ręcznie wyliczyć początkowe wartości silni. Dla dużych *n* nie jest to już takie proste, ale od czego są komputery — przecież nic nie stoi na przeszkodzie, aby napisać prosty program, który zajmuje się obliczaniem silni w sposób rekurencyjny (*silnia.py*):

```
def silnia(x):
    if (x==0):
        return 1
    else:
        return x*silnia(x-1);
# Testujemy funkcję w zakresie liczb 5 - 10 (zwróci:120, 720, 5040, 40320 i 362880)
for i in range(5, 11):
    print(f"silnia({i})={silnia(i)}")
```

Prześledźmy na przykładzie, jak się wykonuje program, który obliczy 3!. Na rysunku 3.2 przedstawiono kolejne etapy wywoływania procedury rekurencyjnej i badania warunku dla przypadku elementarnego.

RYSUNEK 3.2. Drzewo wywołań funkcji silnia(3)



Konwencje użyte na tym rysunku są następujące:

- Pionowe strzałki w dół oznaczają zagłębianie się programu z poziomu *n* na *n*−1 itd. w celu dotarcia do przypadku elementarnego 0!.
- Pozioma strzałka oznacza obliczanie wyników cząstkowych.
- Ukośna strzałka prezentuje proces przekazywania wyniku cząstkowego z poziomu niższego na wyższy.

[2] Mam nadzieję, że w czasie druku tego wydania pojęcie silni nie zostanie przesunięte na poziom edukacji wyższej przez kolejnego ministra edukacji.

Czymże są jednak owe tajemnicze *poziomy*, *przekazywanie parametrów* itd.? Chwilowo te pojęcia mają prawo brzmieć nieco egzotycznie, ale po zastanowieniu można się zapewne domyślić, co się za nimi kryje.

Scharakteryzujmy zatem nieco dokładniej sposób obliczenia `silnia(1)`, opisując działanie programu.

- Funkcja `silnia()` otrzymuje liczbę 1 jako parametr wywołania i analizuje, „czy 1 równa się 0?". Odpowiedź brzmi: „Nie", zatem funkcja przyjmuje, że jej wynikiem jest wyrażenie `1*silnia(0)`.
- Niestety, wartość `silnia(0)` jest nieznana. Funkcja wywołuje zatem kolejny swój egzemplarz, który zajmie się obliczeniem wartości `silnia(0)`, wstrzymując jednocześnie obliczanie wyrażenia `1*silnia(0)`.
- Wywołanie funkcji `silnia(0)` zwraca już konkretny, liczbowy wynik cząstkowy (1), który może zostać użyty do obliczenia wyrażenia `1*silnia(0)`, czyli `silnia(1)=1`.

I tak dalej. W celu obliczenia `silnia(2)` program wykorzysta wartość `silnia(1)`, czyli 1, używając jej do wyliczenia wyrażenia `2*silnia(1)=2*1=2`.

Technicznie przekazywanie parametrów odbywa się za pośrednictwem tzw. stosu, czyli specjalnego miejsca w pamięci operacyjnej, które jest używane do zapamiętywania informacji potrzebnych podczas wykonywania programów i dynamicznego przydzielania pamięci. Programista ma jednak prawo zupełnie się tym nie przejmować. Fakt, że parametr zostanie zwrócony za pośrednictwem stosu, niewiele się różni od przedyktowania wyniku przez telefon. Końcowy efekt, wyrażony przez stwierdzenie *Wynik jest gotowy!*, jest dokładnie taki sam w każdym przypadku, niezależnie od realizacji.

Gdzież się jednak znajdują wspomniane poziomy rekurencji? Spójrzmy raz jeszcze na rysunek 3.2. Aktualna wartość parametru `x` badanego przez funkcję `silnia()` jest zaznaczona z lewej strony reprezentującego ją „pudełka". Ponieważ dany egzemplarz funkcji `silnia()` czasami wywołuje swój kolejny egzemplarz (w celu obliczenia wyniku cząstkowego), wypadałoby jakoś je zróżnicować. Najprostszą metodą jest wykonywanie tego za pomocą wartości `x`, która jest dla nas punktem odniesienia używanym przy określaniu aktualnej głębokości rekurencji.

Niebezpieczeństwa rekurencji

Z użyciem rekurencji czasami związane są pewne niedogodności. Dwa klasyczne niebezpieczeństwa zaprezentowano w poniższych przykładach.