OKIEM EKSPERTA

Kompletny przewodnik po Power Query (M)

Opanuj wykonywanie złożonych przekształceń danych

Greg Deckler, Rick de Groot, Melissa de Korte

Helion

packt

Tytuł oryginału: The Definitive Guide to Power Query (M): Mastering complex data transformation with Power Query

Tłumaczenie: Piotr Rajca

ISBN: 978-83-289-2044-6

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
helion.pl/user/opinie/komprz
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

Parametry i funkcje niestandardowe

Rozdział 9

Język M jest językiem funkcyjnym udostępniającym setki funkcji nadających się do wielu zastosowań. Początkowo standardowa biblioteka języka M prawdopodobnie zaspokoi większość Twoich potrzeb związanych z przekształcaniem danych. Ale gdy napotkasz bardziej wymagające sytuacje, które będziesz chciał rozwiązać, używając języka M, przekonasz się, że umiejętność pisania własnych funkcji niestandardowych otwiera nowe możliwości — tworzenie własnych funkcji może znacznie uprościć proces przekształcania danych i ułatwić wielokrotne stosowanie złożonej logiki. Szczególnie wyraźnie widać to w przypadku tworzenia funkcji umożliwiających przekształcanie istniejących zapytań w dynamiczne funkcje nadające się do wielokrotnego użytku. Największą korzyścią wynikającą z możliwości tworzenia funkcji niestandardowych jest to, że pozwalają one na jednokrotne przygotowanie logiki i późniejsze stosowanie jej wszędzie tam, gdzie będzie potrzebna. Jeśli później zajdzie potrzeba zmodyfikowania tej logiki, wystarczy zaktualizować funkcję. Taka zmiana zostanie automatycznie uwzględniona we wszystkich zapytaniach, w których funkcja jest używana.

Oto zagadnienia, które zostały opisane w tym rozdziale:

- parametry,
- czym są funkcje niestandardowe,
- przekształcanie zapytań w funkcje,
- wywoływanie funkcji niestandardowych,
- wyrażenie `each`,
- poprawianie definicji funkcji,
- odwołania do nazw kolumn i pól,
- debugowanie funkcji niestandardowych,
- zasięg funkcji.

Parametry

W Power Query parametry odgrywają ważną rolę, umożliwiając nadawanie zapytaniom dynamicznego charakteru. Można je sobie wyobrazić jako zmienne, których wartości możemy dostosować, aby zmienić zachowanie zapytań. W trakcie lektury tego rozdziału dowiesz się, dlaczego parametry są przydatnymi elementami, których można używać

do przechowywania wartości i zarządzania nimi. W dalszej części tego podrozdziału dokładnie wyjaśnimy, czym są parametry, oraz opiszemy, gdzie i w jaki sposób można ich używać.

Czym są parametry?

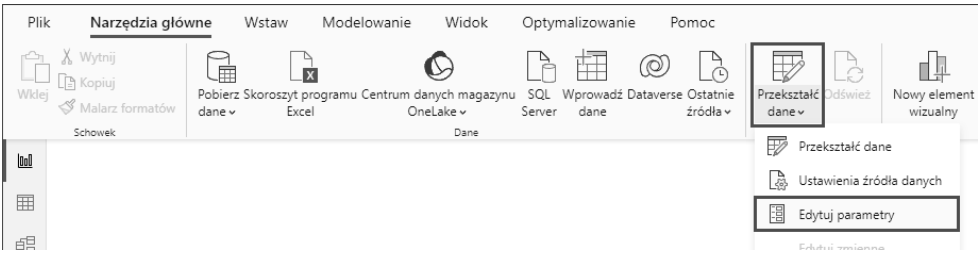
Parametry są swoistymi symbolami zastępczymi, które ułatwiają dostosowywanie zapytań i zarządzanie nimi. Pozwalają one na przekazywanie wartości skalarnych, takich jak daty, liczby lub tekst, bez konieczności podawania ich na stałe w kodzie zapytania. Takie rozwiązanie umożliwia wprowadzanie zewnętrznych modyfikacji pojedynczego parametru, które są następnie automatycznie uwzględniane w wielu zapytaniach lub w wielu krokach zapytania.

Typowe przypadki użycia parametrów zapewniają elastyczność poprzez danie użytkownikom możliwości:

- Wyboru serwera i nazwy bazy danych. Dzięki temu użytkownicy będą mogli wybierać pomiędzy serwerem do prac programistycznych, testowym oraz produkcyjnym.
- Ograniczania liczby rekordów importowanych do zestawu danych. W trakcie prac użytkownik może działać na mniejszym zestawie danych, a po udostępnieniu raportu prosta zmiana wartości parametrów spowoduje kolejne odświeżenie danych i zaimportowanie ich w całości.

W tych scenariuszach parametry zapewniają zarówno elastyczność, jak i łatwość użycia, a także umożliwiają utrzymywanie zapytań. Kolejną zaletą stosowania parametrów jest to, że ich wartości można zmieniać poza edytorem Power Query. Jak można tworzyć parametry i ich używać?

Interfejs użytkownika Power BI Desktop udostępnia opcję pozwalającą na zmienianie parametrów nawet bez otwierania Power Query. Aby to zrobić, należy przejść na Wstążce na kartę *Narzędzia główne* (*H□me*), kliknąć strzałkę widoczną poniżej przycisku *Przekształć dane* (*Transf□rm data*), a następnie wybrać opcję *Edytuj parametry* (*Edit parameters*), jak pokazano na rysunku 9.1.

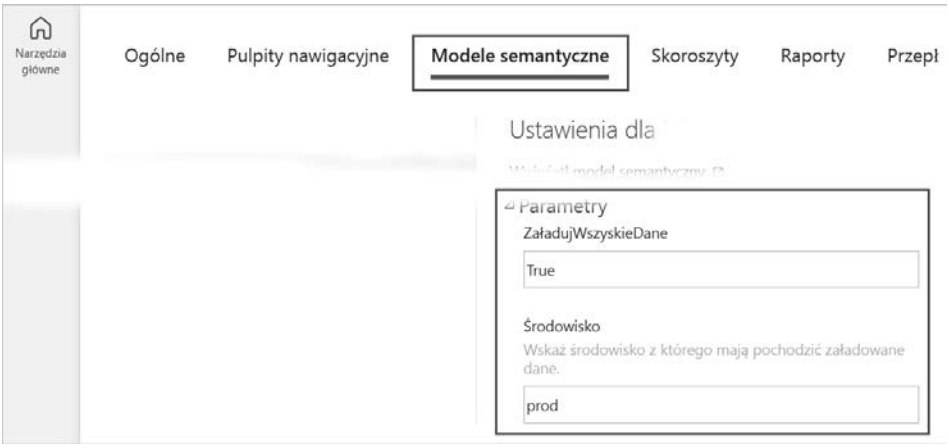


Rysunek 9.1. Zmienianie wartości parametrów w Power BI Desktop

Po dodaniu parametrów użytkownicy mogą wgrywać swoje zestawy danych do usługi Power BI. Jeśli używany zestawu danych będzie mieć parametry, użytkownicy będą mogli wygodnie zmieniać ich wartości z poziomu obszaru roboczego (ang. *w□rkspace*). W tym celu użytkownik może:

1. Wyświetlić w przeglądarce witrynę *https://app.p□werbi.c□m*.
2. Przejść do odpowiedniego obszaru roboczego i zaznaczyć opcję *Ustawienia dla ...* (*Settings*) zestawu danych.
3. Przejść do sekcji *Parametry* (*Parameters*).

Postać okna przeglądarki po wykonaniu tych czynności przedstawia rysunek 9.2.



Rysunek 9.2. Modyfikowanie parametrów w ustawieniach zestawu danych w usłudze Power BI

Później użytkownicy mogą zmienić zdefiniowane parametry. W przypadku modyfikowania parametrów w usłudze Power BI obowiązuje ta sama zasada co w przypadku plików projektu — zmiany staną się widoczne dopiero po odświeżeniu zestawu danych.

O wiele łatwiej jest zmienić parametr w panelu zapytania, niż przeglądać wszystkie kroki zapytania, aby znaleźć odpowiedni kod, który należałoby zmienić. Teraz gdy już wiesz, do czego służą parametry, przyjrzyjmy się, jak można je tworzyć.

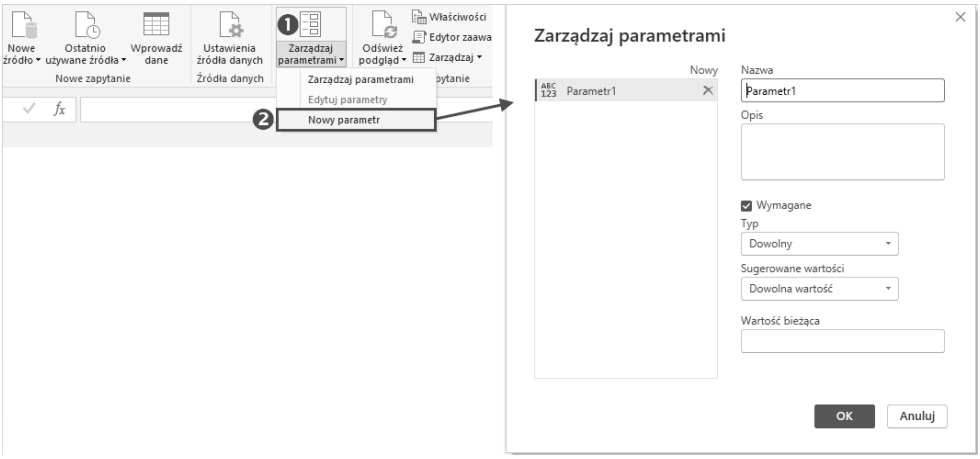
Tworzenie parametrów

Tworzenie parametrów w Power BI jest bardzo proste. Wystarczy otworzyć edytor Power Query i wykonać następujące czynności:

1. Przejść na kartę *Narzędzia główne*.
2. Kliknąć dolną część przycisku *Zarządzaj parametrami* (*Manage Parameters*).
3. Z wyświetlonego menu wybrać opcję *N□wy parametr* (*New Parameter*).

Wykonanie tych czynności spowoduje wyświetlenie menu przedstawionego na rysunku 9.3, w którym można określać oraz modyfikować właściwości parametrów.

Okno dialogowe *Zarządzaj parametrami* pozwala użytkownikom na podawanie dodatkowych informacji o parametrach, a oprócz tego pozwala zagwarantować, że wartość parametru będzie odpowiedniego typu. Poniżej zamieściliśmy przegląd wszystkich pól i opcji dostępnych w tym oknie:



Rysunek 9.3. Tworzenie parametru w oknie dialogowym Zarządzaj parametrami

- *Nazwa* (*Name*). To identyfikator parametru. Nazwa będzie używana w zapytaniach, aby odwołać się do parametru. Określenie nazwy parametru przypomina określenie nazwy zmiennej w kodzie programu — powinna ona być dostatecznie opisowa, by na pierwszy rzut oka można było domyślić się, do czego dany parametr służy.
- *Opis* (*Descripti□n*). Korzystając z tej właściwości, można dodać do parametru krótką notatkę informacyjną lub wyjaśnić jego przeznaczenie. Ten opis służy jako przypomnienie lub porada; wyjaśnia, dlaczego parametr został utworzony i do czego służy.
- *Wymagane* (*Required*). Tym przyciskiem określa się, czy wartość parametru musi zostać podana, by można było wykonać zapytanie.
- *Typ* (*Type*). Z tej listy można wybrać oczekiwany typ wartości parametru. Określając ten typ (taki jak tekst, liczba lub data), zyskamy gwarancję, że w parametrze będzie można zapisać wyłącznie dane tego typu. To ważne dla zapewniania integralności danych oraz unikania nieoczekiwanych błędów podczas wykonywania zapytania.
- *Suger□wane wart□ści* (*Suggested values*). Ta lista udostępnia wskazówki dotyczące wartości, jakie parametr może przyjmować. Udostępnia ona trzy opcje:
  - *D□w□lna wart□ść* (*Any value*). Ta opcja określa, że parametr może przyjąć dowolną wartość wybranego typu, bez żadnych ograniczeń.
  - *Lista wart□ści* (*List □f values*). Ta opcja pozwala zdefiniować z góry określony zestaw wartości, które parametr może przyjąć. Na przykład, jeśli parametr powinien przyjmować tylko wartości: `niska`, `średnia` lub `wysoka`, to należałoby użyć tej opcji.
  - *Zapytanie* (*Query*). Wybór tej opcji pozwala na stosowanie wartości zwracanych przez zapytanie. Przydaje się ona w sytuacjach, gdy dozwolone wartości, które można stosować w parametrze, są wyznaczane dynamicznie i mogą zmieniać się w zależności od używanego źródła danych lub logiki.

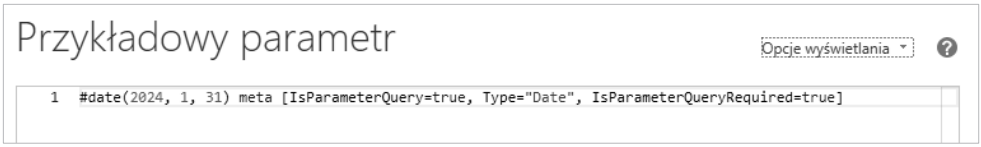
Na przykład w przypadku zastosowania tej opcji dopuszczalnymi wartościami parametru może być unikalna lista kategorii produktów pobrana z bazy danych.

- *Wartość bieżąca* (*Current value*). Określa wartość aktualnie przypisaną parametrowi. Jest to wartość domyślna, która zostanie użyta, jeśli zapytanie zostanie wykonane bez jawnego określenia innej wartości parametru. Jeśli parametr został oznaczony jako wymagany, a jego wartość bieżąca nie będzie określona, to zapytanie nie zostanie wykonane aż do momentu podania wartości parametru.

Po skonfigurowaniu tych ustawień nowy parametr zostanie wyświetlony w panelu *Zapytania*. Interfejs użytkownika automatycznie dodaje do niego trzy pola metadanych:

- `IsParameterQuery` — jest ustawiane na `true`, aby zaznaczyć, że mamy do czynienia z parametrem.
- `IsParameterQueryRequired` — określa, czy parametr jest wymagany.
- `Type` — określa typ danych parametru.

Na przykład podczas wprowadzania wymaganej daty 1 stycznia 2024 r. Power Query tworzy parametr jako nowe zapytanie. Po otwarciu edytora zaawansowanego dla tego zapytania wyświetlony zostanie kod i metadane przedstawione na rysunku 9.4.



Rysunek 9.4. Metadane dodane do parametru

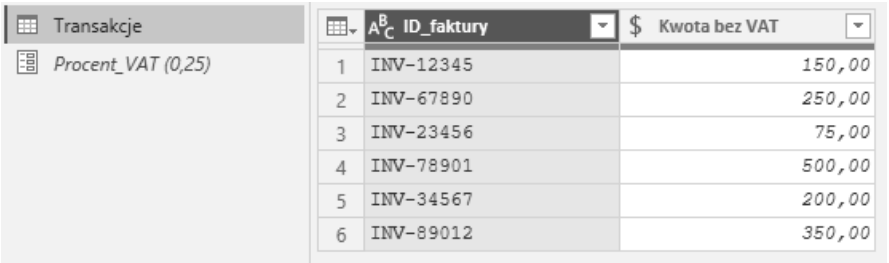
Powyższy kod można sprawdzić w pliku z przykładami do tego rozdziału, dostępnym w materiałach dołączonych do książki. Chociaż parametry można tworzyć i edytować, używając interfejsu użytkownika, to równie dobrze można to robić w edytorze zaawansowanym. Znajomość składni parametrów pozwoli Ci na ich programowe dodawanie (na przykład w aplikacji Tabular Editor 3) lub ręczne wklejenie kodu parametru w edytorze zaawansowanym w celu dodania go do zapytania. Więcej informacji na temat metadanych można znaleźć w rozdziale 7. „Formułowanie języka M".

Skoro już wiesz, jak tworzyć parametry, możemy przejść do sposobów ich używania w zapytaniach.

Stosowanie parametrów w zapytaniach

Parametry można zintegrować z kodem M, odwołując się do nich — analogicznie jak odwołujemy się do tabel lub konkretnych kroków zapytania. Przeanalizujemy to na prostym przykładzie.

W tym przykładzie skorzystamy z prostego zestawu danych przedstawionego rysunku 9.5.



	ID_faktury	Kwota bez VAT
1	INV-12345	150,00
2	INV-67890	250,00
3	INV-23456	75,00
4	INV-78901	500,00
5	INV-34567	200,00
6	INV-89012	350,00

Rysunek 9.5. Przykładowe dane faktur

Aby móc śledzić przykłady i opisy prezentowane w tekście, należy pobrać przykłady dołączone do książki i otworzyć umieszczony w nich plik PBIX dla tego rozdziału.

Zestaw danych przedstawiony na rysunku 9.5 zawiera kolumnę z identyfikatorami faktur oraz z ich wartością bez uwzględniania VAT-u. Naszym celem jest utworzenie nowej kolumny z kwotą uwzględniającą VAT, jednak zależy nam na tym, by użytkownik mógł w prosty sposób zmienić stawkę VAT przy wykorzystaniu parametru.

Aby to zrobić, musimy zacząć od utworzenia odpowiedniego parametru, zgodnie z opisem zamieszczonym w poprzednim punkcie. A to oznacza, że należy wykonać następujące czynności:

1. Wyświetlić menu *Zarządzaj parametrami* i wybrać z niego opcję *Nowy parametr*.
2. W polu *Nazwa* wpisać `Procent_VAT`.
3. Z listy *Typ* wybrać opcję *Liczba dziesiętna* (*Decimal Number*).
4. W polu *Wartość bieżąca* wpisać `0,25` (ta wartość reprezentuje 25% VAT-u).

Po wykonaniu tych czynności okno dialogowe *Zarządzaj parametrami* będzie wyglądać jak na rysunku 9.6.

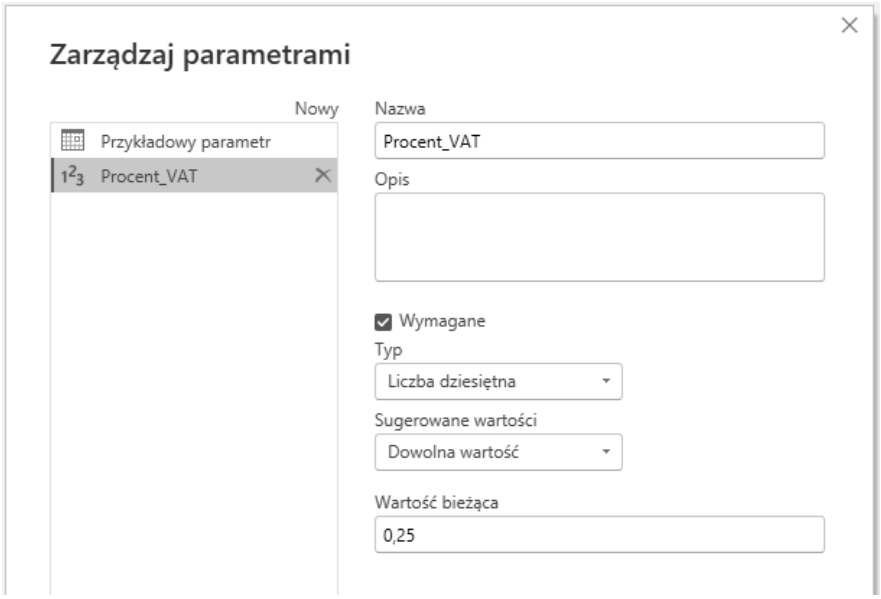
Po dodaniu parametru możemy zająć się utworzeniem kolumny niestandardowej. W tym celu na Wstążce przejdź na kartę *Dodaj kolumnę* (*Add Column*), a następnie kliknij przycisk *Kolumna niestandardowa* (*Custom Column*). W wyświetlonym oknie dialogowym, w polu *Nazwa nowej kolumny* (*New column name*), wpisz `Kwota z VAT`.

A w jaki sposób można użyć parametru w wyrażeniu? Wystarczy odwołać się do niego tak, jak do każdej innej tabeli lub nazwy kroku, jak pokazano na rysunku 9.7.

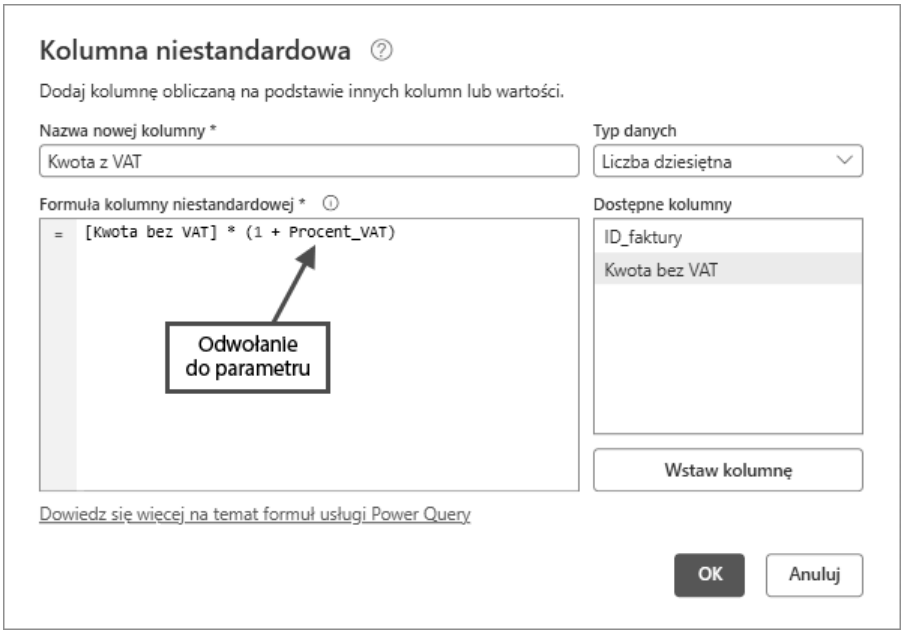
Jeśli nazwa parametru zawiera znaki specjalne lub znaki odstępu, należy zachować ostrożność. W takich przypadkach ważne jest, aby użyć określonego zapisu, dzięki któremu język M poprawnie zinterpretuje nazwę. Konkretnie rzecz biorąc, format zapisu takich nazw ma następującą postać: `#"<nazwa twojego parametru>"`. Na przykład, jeśli parametr ma nazwę `Stawka % VAT`, należy odwołać się do niego, używając zapisu `#"Stawka % VAT"`.

Wskazówka

Rada od mistrza: sugerujemy, aby dla zachowania czystości kodu w nazwach zmiennych używać wyłącznie liter, cyfr i znaków podkreślenia, przy czym nie należy zaczynać ich od cyfry. W przypadku korzystania z tej konwencji stosowanie opisanego wcześniej specjalnego formatu zapisu nazw nie będzie konieczne.



Rysunek 9.6. Właściwości zastosowane podczas tworzenia parametru Procent_VAT



Rysunek 9.7. Odwołanie do parametru umieszczone w formule kolumny

Ponieważ już wiesz, jak odwoływać się do parametru, w następnym punkcie rozdziału, „Łączenie wszystkiego w całość”, przedstawimy kilka praktycznych przykładów zastosowania parametrów.

Łączenie wszystkiego w całość

Parametry z powodzeniem można stosować w wielu różnych scenariuszach. W tym punkcie rozdziału przedstawimy trzy takie często występujące scenariusze i pokażemy, jak można w nich użyć parametrów.

Parametryzacja informacji o połączeniu

Praca z bazami danych często oznacza żonglowanie różnymi środowiskami: programistycznym, testowym oraz produkcyjnym. Związane z tym zmienianie różnych serwerów i ich nazw może być żmudnym zadaniem. W takich przypadkach parametry stanowią eleganckie rozwiązanie, które upraszcza przełączanie się między różnymi serwerami i bazami danych. Przeanalizujmy przykład takiego rozwiązania.

Ten przykład jest przeznaczony tylko do przeczytania; jeśli jednak dysponujesz dostępem do bazy danych, to możesz spróbować wypróbować go u siebie na komputerze.

Nawiązanie połączenia z bazą danych SQL wymaga określenia zarówno serwera, jak i nazwy bazy danych. Typowy kod służący do nawiązania takiego połączenia z bazą danych SQL wygląda następująco:

```
Sql.Database(
  "localhost\sql-dev.database.windows.net",
  "db-staging-dev-westeu-001" )
```

Można go podzielić na dwie poniższe części:

- **serwer**: `localhost\sql-dev.database.windows.net`,
- **bazę danych**: `db-staging-dev-westeu-001`.

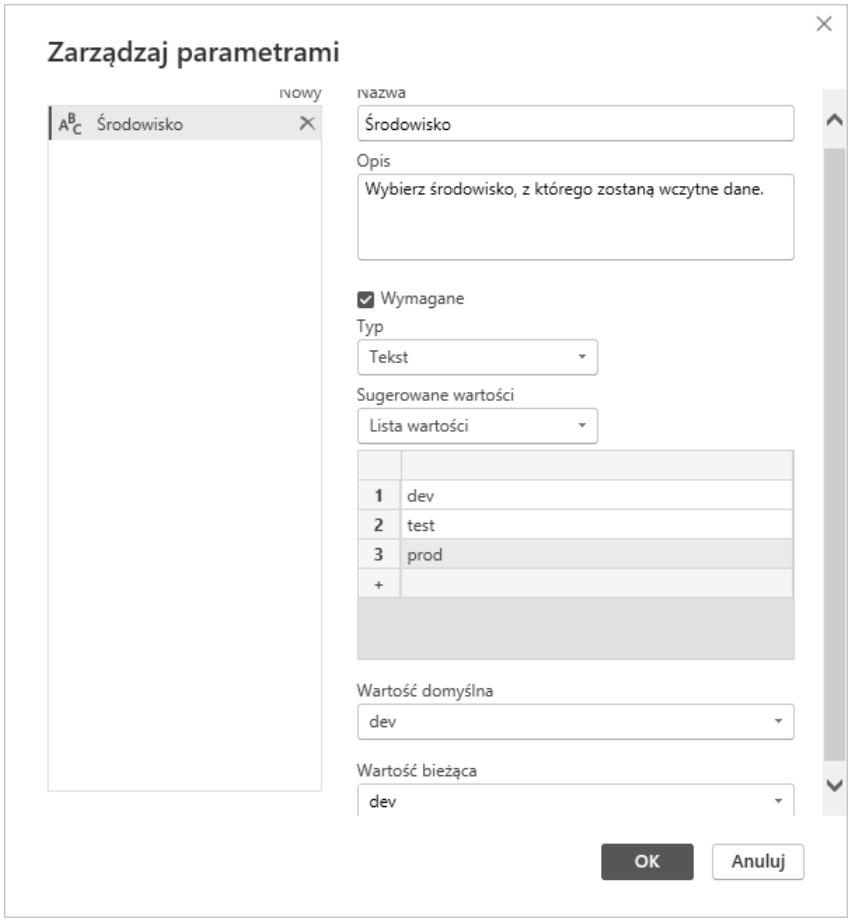
Czy zwróciłeś uwagę na łańcuch znaków `dev` występujący w obu tych częściach? Konfigurując bazy danych, administratorzy starają się jak najbardziej uprościć zmianę środowisk. Na przykład, aby użyć środowiska testowanego, wystarczy zastąpić fragment `dev` łańcuchem `test`, a żeby użyć środowiska produkcyjnego, należałoby użyć łańcucha `prod`.

Zamiast podawać środowisko serwera na stałe, możemy wykorzystać parametry. W tym celu wykonaj następujące czynności:

1. Wyświetl okno dialogowe *Zarządzanie parametrami* i utwórz nowy parametr.
2. Nadaj mu nazwę `Środowisko`, zaznacz pole wyboru *Wymagane*, a z listy *Typ* wybierz opcję *Tekst*. Kliknij listę *Sugerowana wartość* i wybierz z niej opcję *Lista wartości*.
3. Skorzystaj z wyświetlonego poniżej obszaru, by skonfigurować opcje, które będzie można wybrać z rozwijanej listy. Zdefiniuj następujące wartości: `dev`, `test` i `prod`.

Po skonfigurowaniu tych wszystkich ustawień właściwości parametru powinny wyglądać jak na rysunku 9.8.

Czy pamiętasz przedstawiony wcześniej kod do nawiązania połączenia z bazą danych? Teraz dysponujemy już parametrem umożliwiającym dostosowanie środowiska bazy danych do naszych potrzeb, niezależnie od tego, czy potrzebujemy wersji do prac



Rysunek 9.8. Definiowanie parametrów dla połączenia z bazą danych

programistycznych, testowej, czy produkcyjnej. W tym celu wybierz krok zapytania, w którym jest nawiązywane połączenie z bazą danych. W efekcie na pasku formuły zostanie wyświetlony poniższy kod:

```
Sql.Database(
  "localhost\sql-dev.database.windows.net",
  "db-staging-dev-westeu-001" )
```

Aby zastosować teraz parametr, zastąp fragment `dev` parametrem `Środowisko`, jak pokazano poniżej:

```
Sql.Database(
  "localhost\sql-" & Środowisko & ".database.windows.net",
  "db-staging-" & Środowisko & "-westeu-001" )
```

W tym przykładzie parametr `Środowisko` pełni funkcję swoistego symbolu zastępczego, który pozwala na łatwe modyfikowanie łańcucha połączenia. Po zapisaniu tych zmian i wgraniu pliku do usługi Power BI zyskasz możliwość prostego i wygodnego zmieniania wartości parametru bezpośrednio z poziomu menu ustawień zestawu danych. Oznacza

to, że będziesz mógł zmieniać używane połączenie z bazą danych nawet bez otwierania aplikacji Power BI Desktop. Skoro już poznałeś pierwszy bardzo często występujący scenariusz korzystania z parametrów, możemy przejść do drugiego, w którym zajmiemy się dynamicznymi ścieżkami dostępu do plików.

Dynamiczne ścieżki dostępu do plików

Podczas pracy z zapytaniami, które zawierają ścieżki dostępu do plików, ważne jest, aby wziąć pod uwagę, że ścieżki te mogą działać poprawnie na komputerze twórcy zapytania, ale niekoniecznie na komputerach innych osób. Jeśli przewidujesz, że Twój plik PBIX będzie używany przez wiele osób, a w umieszczonych w nim zapytaniach używane są ścieżki dostępu do innych plików, to powinieneś zwrócić uwagę na przygotowanie zapytań w przemyślany sposób. W takich sytuacjach szczególnie ważne i przydatne będzie zapewnienie użytkownikom łatwego modyfikowania ścieżek dostępu do plików, a ten rodzaj elastyczności bardzo łatwo można zapewnić dzięki zastosowaniu parametrów. Przyjrzyjmy się zatem, jak możemy to zrobić. W tym przykładzie wykorzystamy plik *Av□cad□ Prices.csv* dostępny w przykładach dołączonych do książki.

Załóżmy, że tworzysz materiały szkoleniowe, które korzystają z pliku *Av□cad□ Price.csv* dostępnego w przykładach dołączonych do niniejszej książki. Aby zachować porządek, decydujesz się przechowywać ten plik na dysku *C*, w katalogu o nazwie *dane*. Aby połączyć się z tym plikiem za pomocą Power Query, wykonaj następujące czynności:

1. Kliknij przycisk *P□bierz dane* (*Get Data*), a następnie wybierz opcję *Plik tekst□wy lub CSV* (*Text/CSV*) i kliknij przycisk Połącz (*C□nnect*).
2. Przejdź do katalogu *C:\dane* i zaznacz odpowiedni plik CSV.
3. Na ekranie zostanie wyświetlone okno importu danych z pliku CSV; kliknij w nim przycisk *OK*.

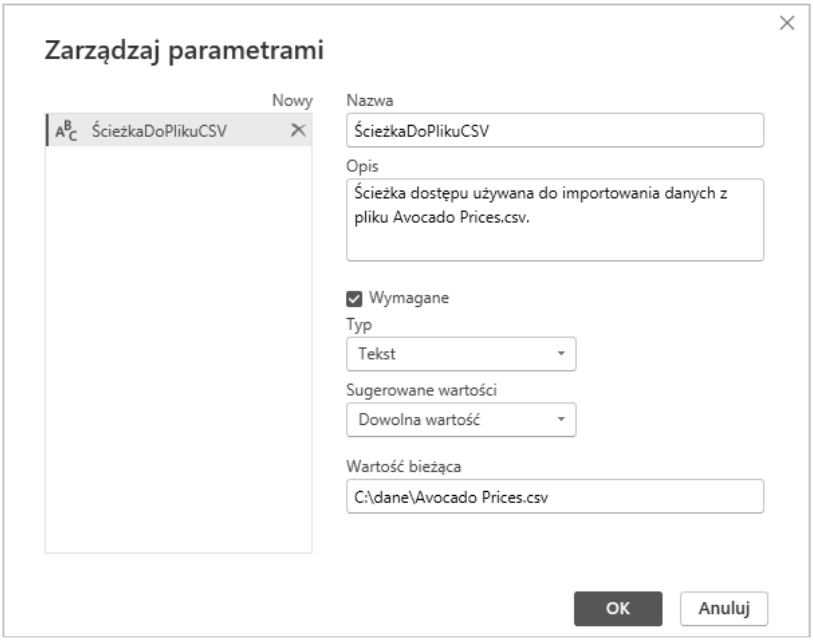
Wykonanie tej sekwencji czynności spowoduje wygenerowanie kodu M o następującej postaci:

```
Csv.Document(
  File.Contents("C:\dane\Avocado Prices.csv"),
  [Delimiter=",", Columns=14,
  Encoding=1252, QuoteStyle=QuoteStyle.None] )
```

Zwróć uwagę na jawnie podaną ścieżkę dostępu do pliku: *C:\dane\Av□cad□ Prices.csv*. Chociaż takie rozwiązanie może być wygodne dla Ciebie, to dla innych może być uciążliwe. Co, jeśli uczestnicy szkolenia przechowują swoje pliki w innych miejscach? Co, jeśli utworzyłeś inne zapytania, które także używają tej ścieżki dostępu do pliku jako swoistego wzoru?

Podanie ścieżki dostępu do używanego pliku z danymi na stałe bezpośrednio w kodzie staje się przeszkodą, która zmusza wszystkich użytkowników do mozolnego, ręcznego wprowadzania zmian. Na szczęście dzięki zastosowaniu parametrów można poprawić elastyczność używanych w kodzie ścieżek dostępu. Poniżej opisaliśmy, jak można to zrobić.

Zacznij od utworzenia nowego parametru. Nadaj mu nazwę `ŚcieżkaDoPlikuCSV`, a z listy *Typ* wybierz opcję *Tekst*. Właściwości tego parametru przedstawiliśmy na rysunku 9.9.



Rysunek 9.9. Tworzenie parametru o nazwie ŚcieżkaDoPlikuCSV

Po utworzeniu parametru można go już zastosować w kodzie. Podczas tworzenia połączenia z plikiem CSV został wygenerowany krok o nazwie `Źródło`, który jest widoczny w panelu *ZASTOSOWANE KROKI* (*Appliled Steps*) z prawej strony edytora Power Query.

Oto bardzo prosty sposób zintegrowania tego parametru z zapytaniem:

1. Kliknij ikonę koła zębatego, widoczną z prawej strony kroku *Źródł*.
2. Z rozwijalnej listy w polu *Ścieżka pliku* wybierz opcję *Parametr*.
3. Z kolejnej rozwijalnej listy, która zostanie wyświetlona z prawej strony, wybierz parametr *ŚcieżkaD PlikuCSV*.

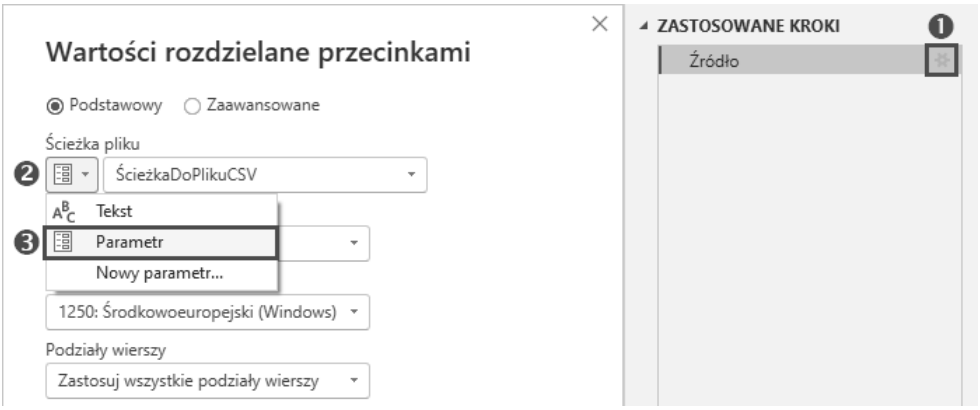
Te kroki przedstawiliśmy także na rysunku 9.10.

Po wprowadzeniu tej modyfikacji kod M służący do nawiązania połączenia z plikiem będzie mieć następującą postać:

```
Źródło = Csv.Document(
  File.Contents( ŚcieżkaDoPlikuCSV ),
  [Delimiter=",", Columns=14,
   Encoding=1252, QuoteStyle=QuoteStyle.None]
)
```

Łatwo zauważyć, że tam, gdzie wcześniej była podana ścieżka dostępu do pliku, teraz jest umieszczony parametr `ŚcieżkaDoPlikuCSV`. Ta zmiana poprawia elastyczność zapytania.

Teraz wystarczy sama zmiana wartości, aby zapewnić elastyczność zapytania — możemy bowiem określać lokalizację pliku bez modyfikowania kodu zapytania. Jeśli kiedyś w przyszłości Ty lub ktoś inny będziecie musieli zmienić ścieżkę dostępu do pliku, wystarczy zmienić wartość parametru.



Rysunek 9.10. Parametru można także używać jako ścieżki dostępu do pliku

Wskazówka

Rada od mistrza: czy już czujesz się komfortowo, pisząc kod w języku M? Zatem pomiń interfejs użytkownika i ręcznie zmodyfikuj kod zapytania, tak by używał parametru `ŚcieżkaDoPlikuCSV`. Przejście ze ścieżek podawanych na stałe w kodzie do wykorzystania parametrów w ogromnym stopniu upraszcza pracę.

Skoro już poznałeś drugi z popularnych scenariuszy wykorzystania parametrów, przejdźmy do trzeciego, w którym zajmiemy się filtrowaniem zakresów dat.

Filtrowanie zakresu dat

Wartości dat są uwzględniane w wielu modelach, niezależnie od tego, czy tworzymy miesięczne raporty sprzedaży, analizujemy historyczne wzorce pogodowe, czy identyfikujemy trendy finansowe. Jednak operowanie w zapytaniu na całym dostępnym zakresie danych historycznych bardzo często prowadzi do znacznego spadku wydajności działania zapytania. Na szczęście jest to jeden ze scenariuszy, w których z powodzeniem można wykorzystać parametry. Zapewniają one bowiem elastyczność w zarządzaniu zakresami dat, pozwalając na obsługę zarówno zakresów stałych, jak i kroczących.

Rozważmy scenariusz, w którym w celu poprawy wydajności działania chcielibyśmy operować wyłącznie na danych z kilku ostatnich kwartałów. Zamiast ręcznie modyfikować zakresy dat w poszczególnych zapytaniach, możemy do tego celu zastosować parametry.

Jeśli nasza tabela transakcji zawiera kolumnę daty, to możemy użyć następującego wyrażenia:

```
Table.SelectRows(
  TabelaDanych,
  each [Data] >= DataPoczątku and [Data] <= DataKońca )
```

W tym kodzie pierwszy argument, `TabelaDanych`, określa tabelę, którą należy przefiltrować, natomiast drugi argument filtruje wiersze tej tabeli na podstawie wartości w kolumnie `Data`.

Za każdym razem, gdy zainteresujemy się nowym kwartałem, wystarczy, że zmienimy parametry `DataPoczątku` i `DataKońca`. Jednak w tym rozwiązaniu tkwi pewien haczyk: choć pozwala ono łatwo modyfikować wartości parametrów, to jednak i tak zawsze trzeba je będzie wpisać ręcznie. Czy nie byłoby wygodnie, gdyby wartości tych parametrów były określane dynamicznie?

Niestety wbudowane parametry tworzone i stosowane przy użyciu przycisków na Wstążce nie obsługują wartości dynamicznych, wyliczanych na podstawie wyrażeń. Nie oznacza to jednak, że nie ma sposobu, by poradzić sobie z tym problemem: zawsze można użyć wartości zwracanej przez wyrażenie w zapytaniu, nawet jeśli nie zostanie ona formalnie zdefiniowana jako parametr.

W ramach przykładu wyobraźmy sobie, że musimy pobrać dane z ostatnio zakończonego kwartału. Utworzenie nowego, pustego zapytania pozwoli nam przygotować następujące wyrażenie, które określi datę końca interesującego nas okresu — odpowiednik naszego parametru `DataKońca`:

```
Date.AddDays(Date.StartOfQuarter( Date.From( DateTime.LocalNow() ) ), -1 )
```

To wyrażenie pobiera wartość aktualnej daty i godziny, konwertuje ją na datę, następnie określa datę początku kwartału zawierającego tę datę, po czym odejmuje od niej jeden dzień. Innymi słowy, powyższe wyrażenie będzie konsekwentnie wyznaczać datę zakończenia poprzedniego kwartału.

Jeśli chodzi o parametr `DataPoczątku`, to możemy wyznaczyć tę datę, określając datę początku kwartału dla daty `DataKońca`:

```
Date.StartOfQuarter( DataKońca )
```

Choć te zapytania nie będą prezentowane w oknie parametrów wyświetlanym z poziomu Wstążki, to jednak działają w podobny sposób i można się do nich odwoływać tak jak do faktycznych parametrów.

Funkcje niestandardowe

Funkcje niestandardowe odgrywają ważną rolę w języku M — pozwalają konsolidować logikę, zapewniać modularność kodu, możliwość jego wielokrotnego użycia i poprawiają wydajność jego działania. Możesz je sobie wyobrazić jako narzędzie zaprojektowane do wykonywania określonego zadania.

I chociaż w standardowej bibliotece języka M dostępne są setki funkcji, to czasami przydatne będzie napisanie własnej. Takie własne funkcje umożliwiają tworzenie rozwiązań dostosowanych do wyzwań, które stawiają dane, na jakich operujesz, i pozwalają na konsolidację wielu niestandardowych kroków oraz wykonywanie przekształceń danych, które nie są obsługiwane przez funkcje standardowe.

Na przykład rozważmy złożone zadania, takie jak tworzenie sum bieżących, pobieranie wartości poprzedniego wiersza lub obliczanie numeru tygodnia według standardu ISO. Ręczne napisanie kodu obsługującego takie przekształcenia może być skomplikowane i podatne na błędy. Natomiast w przypadku udostępnienia funkcji niestandardowej

będzie można osiągnąć pożądany wynik za pomocą jednego wywołania funkcji, którą wystarczy napisać tylko raz. Po utworzeniu takich funkcji można ich łatwo ponownie używać, co znacznie ułatwia mniej doświadczonym członkom zespołu korzystanie z niestandardowych rozwiązań.

W dalszej części tego podrozdziału dowiesz się, czym są funkcje, jak można pisać własne funkcje i jak ich następnie efektywnie używać. Przy okazji poznasz także sposoby usprawniania definicji funkcji, debugowania ich oraz możliwości zarządzania ich zasięgiem.

Czym są funkcje niestandardowe?

Funkcje niestandardowe można sobie wyobrażać jako miniaturowe zapytania. Pobierają one dane wejściowe, nazywane parametrami, i wykonują określone operacje, używając tych parametrów, by wyznaczyć wyniki, które następnie zwracają. Do tworzenia funkcji niestandardowych konieczna jest znajomość ich elementów składowych.

Definicja funkcji niestandardowej zaczyna się od zestawu parametrów zapisanego wewnątrz pary nawiasów, symbolu `=>` oraz wyrażenia, które będzie przetwarzane. To wyrażenie określa sposób, w jaki funkcja będzie wyznaczać swoje wyniki. Przyjrzyjmy się przykładowi takiej funkcji.

Prostą funkcję, która dodaje `10` do przekazanej wartości, można zdefiniować w następujący sposób:

```
(parametr) => parametr + 10
```

Oto jak należy interpretować ten zapis:

- `(parametr)` — ten fragment definicji określa dane wejściowe czy też parametry funkcji. Nazwa tego parametru jest dowolna.
- `=>` — ten symbol oddziela listę parametrów od wyrażenia.
- `parametr + 10` — to wyrażenie, nazywane także ciałem funkcji, które określa, jak funkcja powinna przetworzyć dane wejściowe, aby wyznaczyć wynik.

Do funkcji można przekazywać wiele danych wejściowych — parametrów. Na przykład, gdybyśmy chcieli przygotować funkcję, która mnoży dwie liczby, moglibyśmy zdefiniować ją w następujący sposób:

```
(a, b) => a * b
```

Choć w powyższym przykładzie parametrom nadaliśmy nazwy `a` i `b`, to jednak pisząc własne funkcje, możesz nadawać parametrom dowolne nazwy. Na przykład poniższa definicja funkcji niczym się nie różni od poprzedniej:

```
(lewa, prawa) => lewa * prawa
```

Wskazówka

Rada mistrza: choć parametrom można nadawać dowolne nazwy, to jednak dobrym pomysłem jest wybieranie nazw sensownych i znaczących, zwłaszcza jeśli pisany kod jest przeznaczony dla szerszego grona przyszłych użytkowników. Dzięki temu przeznaczenie parametru od samego początku będzie jasne i zrozumiałe.

W przykładach przedstawionych do tej pory w wyrażeniach stanowiących ciało funkcji nie były używane żadne zmienne. Jeśli jednak chcemy poprawić przejrzystość naszych funkcji, to warto rozważyć zastosowanie w nich wyrażenia `let`.

Oto przykład takiej funkcji:

```
( liczbaTekstowo ) =>
let
    TekstNaLiczbę = Number.FromText( liczbaTekstowo ),
    PomnożoneX3 = TekstNaLiczbę * 3,
    LiczbaNaTekst = Number.ToText( PomnożoneX3 ),
    WynikJakoTekst = "3 razy "& liczbaTekstowo & " = " & LiczbaNaTekst
in
    WynikJakoTekst
```

Ta funkcja przekształca łańcuch znaków na liczbę, mnoży ją przez 3 i zwraca wynik, którym jest kolejny łańcuch znaków. Jeśli nadamy tej funkcji nazwę `fnPomóżTekst` i wywołamy ją w następujący sposób: `fnPomóżTekst("5")`, zwróci ona łańcuch znaków o postaci: `"3 razy 5 = 15"`.

Funkcja ta, w zależności od tego, jak zostanie wywołana, może, choć nie musi, zgłosić błąd. W przypadku wystąpienia błędu sprawdź, czy kod wywołujący funkcję zapisany w pasku formuły ma postać `fnPomóżTekst("5")`, a nie `fnPomóżTekst(5)` (z cyfrą `5` zapisaną bezpośrednio w nawiasach).

Niektóre funkcje w ogóle nie potrzebują danych wejściowych. Przykładem takiej funkcji w bibliotece standardowej języka M może być funkcja `DateTime.LocalNow`. Aby wywołać taką funkcję, za jej nazwą należy zapisać pustą parę nawiasów, jak na poniższym przykładzie:

```
DateTime.LocalNow()
```

Takie wywołanie zwraca wartość określającą bieżącą lokalną datę i godzinę (typu `date ↪time`) na używanym komputerze. Można także tworzyć funkcje niestandardowe, które nie będą korzystać z parametrów. Na przykład poniższa funkcja zwraca promień kuli ziemskiej:

```
() => 6371
```

Ta instrukcja wywołania nie pobiera żadnych parametrów i zwraca wartość liczbową. Jednak takie funkcje spotyka się raczej sporadycznie. Zamiast pisania i stosowaniach takich funkcji zazwyczaj znacznie bardziej praktycznym rozwiązaniem będzie zapisanie wyniku zwracanego przez taką funkcję jako wartości parametru lub w zmiennej w naszym kodzie.

W tym punkcie rozdziału zamieściliśmy podstawowe informacje dotyczące tworzenia funkcji niestandardowych, w tym opisaliśmy definiowanie parametrów i tworzenie ciała funkcji. Podczas jego lektury miałeś także okazję samodzielnie napisać definicję swojej pierwszej funkcji. Jeśli będziesz chciał automatycznie generować kod niestandardowych funkcji, to Power Query udostępnia wydajne, alternatywne rozwiązanie: możesz przekształcić zapytanie na funkcję.

Przekształcanie zapytań na funkcje

Power Query udostępnia funkcjonalność, która pozwala przekształcić zapytanie na funkcję. W przeciwieństwie do ręcznie napisanych funkcji niestandardowych ta metoda sprawia, że wszystkie kroki zapytania będą widoczne i będzie można je edytować. W dalszej części tego punktu rozdziału opisaliśmy, jak można używać tej funkcjonalności do przekształcania zapytań w elastyczne funkcje nadające się do wielokrotnego użytku.

Czym jest funkcjonalność „utwórz funkcję"?

W Power Query funkcjonalność *Utwórz funkcję* umożliwia przekształcenie dowolnego zapytania w funkcję nadającą się do wielokrotnego użytku. Jeśli zapytanie nie ma parametrów, zostanie utworzona funkcja bezparametrowa. W bardziej zaawansowanych przypadkach w kodzie wygenerowanej funkcji będą używane parametry, by umożliwić przekazywanie do niej argumentów.

Załóżmy, że chcemy dodać do naszej funkcji parametry. W tym celu musimy wskazać w kodzie elementy, które się zmieniają, i powiązać je z odpowiednimi parametrami raportu. W takim przypadku mechanizm *Utwórz funkcję* automatycznie zintegruje kod niezbędny dla tych parametrów z kodem funkcji niestandardowej.

Jeśli zdecydujesz się nie używać parametrów, funkcja będzie zwracać pierwotne, niezmienione wyniki zapytania. Jednak tworzenie takich bezparametrowych funkcji jest raczej rzadko spotykane, gdyż równie dobrze można po prostu odwołać się bezpośrednio do zapytania.

Zastosowanie funkcjonalności *Utwórz funkcję* ma trzy podstawowe zalety:

- **Widzialność zapytania**. Pierwotne zapytanie nadal jest w projekcie. Oznacza to, że wciąż możemy przeglądać, modyfikować i wykonywać naszą logikę jako odrębne, niezależne zapytanie.
- **Aktualizacje funkcji**. Zmiany pierwotnego zapytania są automatycznie odzwierciedlane w powiązanej z nim funkcji. Oznacza to, że można w prosty sposób zaktualizować logikę działania bez konieczności wprowadzania zmian w funkcji.
- **Łatwość tworzenia funkcji**. Użytkownicy mogą tworzyć funkcje niestandardowe bez konieczności pisania kodu w języku M.

A zatem w jaki sposób można utworzyć funkcję, używając do tego celu interfejsu użytkownika? Oto czynności pozwalające utworzyć funkcję na podstawie zapytania:

1. Jeśli funkcja ma mieć parametry, to należy ich użyć w zapytaniu. Może to być coś tak prostego jak kryteria wyboru rekordów lub jakaś wartość używana w wyliczeniach.
2. Wyświetlić panel *Zapytania* (*Queries*) umieszczony z lewej strony okna aplikacji i kliknąć zapytanie prawym przyciskiem myszy.
3. Z wyświetlonego menu kontekstowego wybrać opcję *Utwórz funkcję* (*Create Functi□n*). Spowoduje to wyświetlenie okna dialogowego.

4. Nadać funkcji odpowiednio opisową nazwę, dzięki której łatwo będzie zorientować się, jakie jest przeznaczenie funkcji.
5. Kliknąć przycisk *OK*, aby utworzyć funkcję.

Po wykonaniu tych czynności nowa funkcja zostanie wyświetlona w panelu *Zapytania* i będzie można jej używać tak samo, jak każdej innej funkcji dostępnej w Power Query. Przekonajmy się teraz, jak możemy zastosować te kroki w praktyce.

Załóżmy, że naszym zadaniem jest utworzenie logiki dla tabeli kalendarza. Przed utworzeniem funkcji przygotujemy zapytanie o nazwie `LogikaKalendarza`, przedstawione na rysunku 9.11.

```
let
    Źródło = List.Dates(
                #date(2024, 1, 1),
                Duration.Days( #date(2024, 12, 31) - #date(2024, 1, 1) ) + 1,
                #duration( 1, 0, 0, 0)
            ),
    UtworzonoTabelę = Table.FromList (Źródło, Splitter.SplitByNothing( ), type table [Data = date], null, 1),
    DodanoRok = Table.AddColumn(UtworzonoTabelę, "Rok", each Date.Year( [Data] ), Int64.Type),
    DodanoMiesiąc = Table.AddColumn (DodanoRok, "Miesiąc", each Date.Month( [Data] ), Int64.Type)
in
    DodanoMiesiąc
```

Rysunek 9.11. Zapytanie LogikaKalendarza

Zapytanie generuje kolumnę z datami (`Data`), a następnie dodaje do niej dwie kolejne kolumny: `Rok` oraz `Miesiąc`. Chcielibyśmy, by użytkownicy mogli określać długość kalendarza. Z tego względu w kolejnym kroku dodamy do niego dwa parametry: `dataPoczątku` oraz `dataKońca`, oba typu `date`. Po ich utworzeniu i uwzględnieniu w zapytaniu jego kod będzie wyglądał jak na rysunku 9.12.

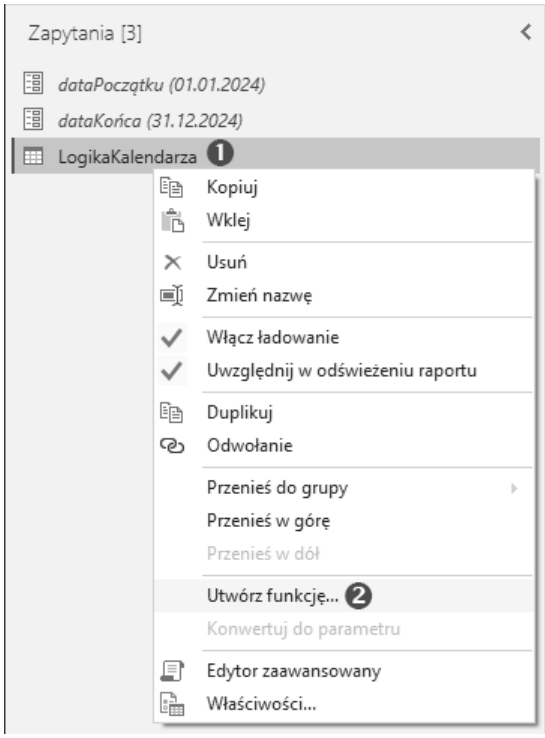
```
let
    Źródło = List.Dates(
                dataPoczątku,
                Duration.Days( dataKońca - dataPoczątku ) + 1,
                #duration( 1, 0, 0, 0)
            ),
    UtworzonoTabelę = Table.FromList (Źródło, Splitter.SplitByNothing( ), type table [Data = date], null, 1),
    DodanoRok = Table.AddColumn(UtworzonoTabelę, "Rok", each Date.Year( [Data] ), Int64.Type),
    DodanoMiesiąc = Table.AddColumn (DodanoRok, "Miesiąc", each Date.Month( [Data] ), Int64.Type)
in
    DodanoMiesiąc
```

Rysunek 9.12. Zapytanie LogikaKalendarza z dodanymi parametrami

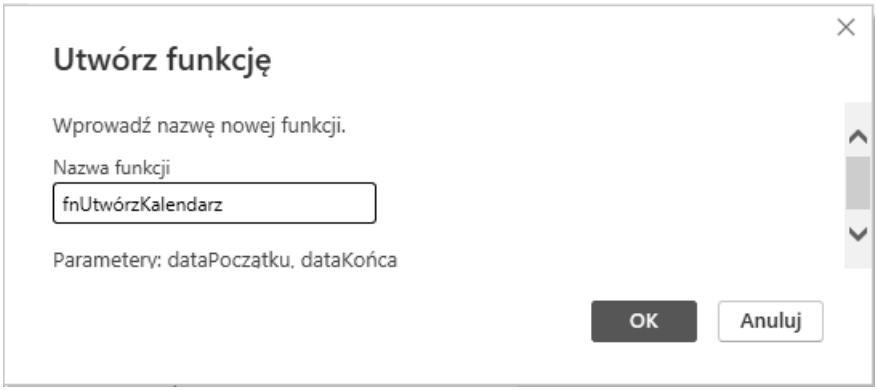
Pu utworzeniu tych parametrów i zastosowaniu ich w kodzie zapytania jesteśmy już gotowi, by przekształcić je w funkcję. Aby to zrobić, wystarczy kliknąć prawym przyciskiem myszy zapytanie i wybrać opcję *Utwórz funkcję* (*Create Functi*n), jak pokazano na rysunku 9.13.

W efekcie na ekranie zostanie wyświetlone okno dialogowe, w którym możemy podać nazwę tworzonej funkcji; przedstawiliśmy je na rysunku 9.14.

Ponieważ nasza funkcja ma tworzyć kalendarz, nadamy jej nazwę `fnUtwórzKalendarz`. Po kliknięciu przycisku *OK* Power Query przekształci nasze zapytanie w funkcję.



Rysunek 9.13. Opcja Utwórz funkcję w menu kontekstowym na panelu Zapytania

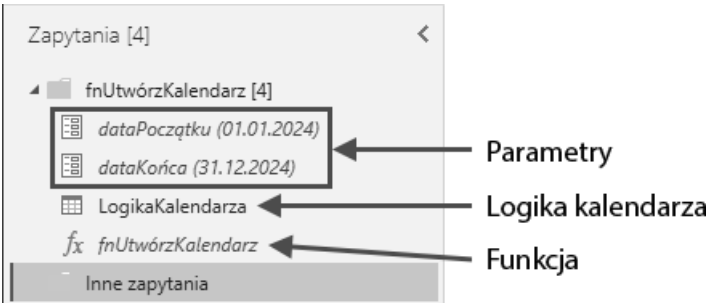


Rysunek 9.14. Okno dialogowe Utwórz funkcję

Po wykonaniu tych wszystkich czynności będziemy dysponować czterema zapytaniami:

- parametrami `dataPoczątku` i `dataKońca`,
- zapytaniem `LogikaKalendarza` implementującym logikę przygotowywania danych na potrzeby kalendarza,
- funkcją `fnUtwórzKalendarz` powiązaną z zapytaniem `LogikaKalendarza`.

Panel *Zapytania* będzie zatem podobny do tego przedstawionego na rysunku 9.15.

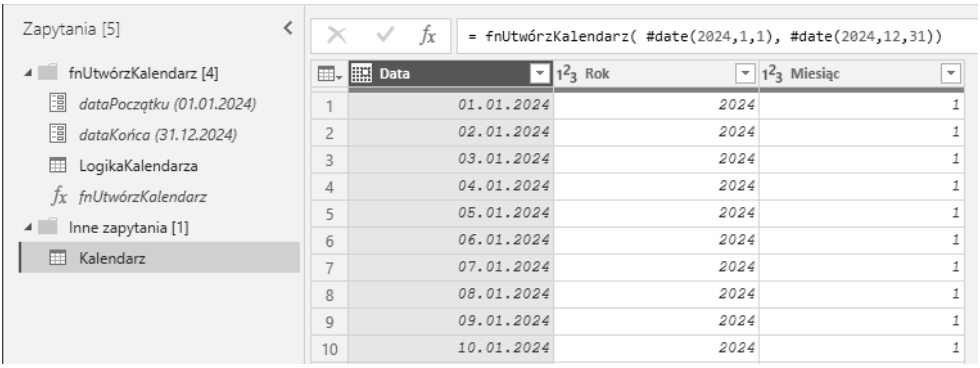


Rysunek 9.15. Komponenty używane do tworzenia kalendarza

Skoro nasza funkcja jest już gotowa, zobaczmy, jak możemy jej efektywnie używać. W tym celu utwórz nowe, puste zapytanie o nazwie `Kalendarz` i wpisz w nim następujący kod:

```
fnUtwórzKalendarz( #date( 2024,1,1 ), #date( 2024,12,3 ) )
```

Wykonanie tego zapytania spowoduje wyświetlenie wyników przedstawionych na rysunku 9.16.



	Data	Rok	Miesiąc
1	01.01.2024	2024	1
2	02.01.2024	2024	1
3	03.01.2024	2024	1
4	04.01.2024	2024	1
5	05.01.2024	2024	1
6	06.01.2024	2024	1
7	07.01.2024	2024	1
8	08.01.2024	2024	1
9	09.01.2024	2024	1
10	10.01.2024	2024	1

Rysunek 9.16. Zastosowanie funkcji do tworzenia kalendarza

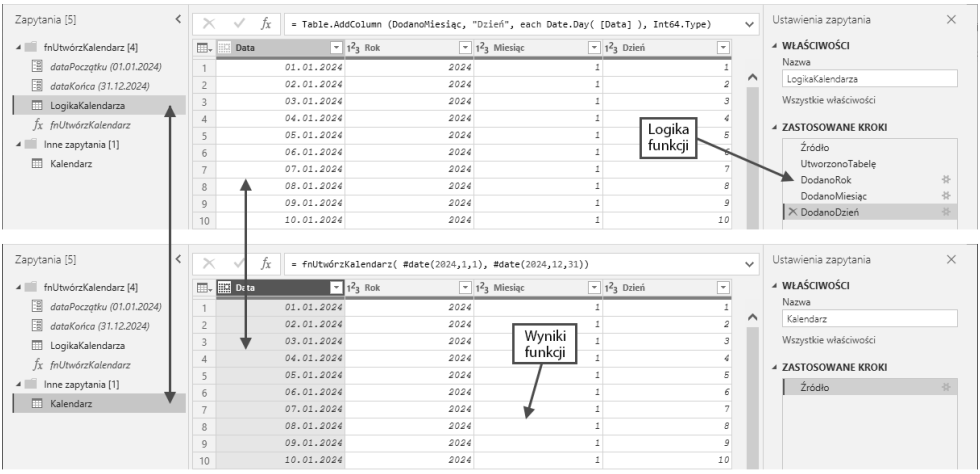
Tak więc dzięki jednemu wierszowi kodu wywołującemu niestandardową funkcję uzyskujemy tabelę kalendarza zawierającą niestandardową logikę; czyż to nie jest wspaniałe? To rozwiązanie można znaleźć w przykładach dołączonych do książki.

Jak na razie mógłbyś sądzić, że główną zaletą korzystania z funkcjonalności *Utwórz funkcję* jest automatyzacja tworzenia funkcji, jednak to nie wszystko. Funkcjonalność tę z powodzeniem można także wykorzystywać w scenariuszach związanych z wprowadzaniem zmian oraz rozwiązywaniem problemów.

Podczas pracy z funkcjami niestandardowymi mogą pojawiać się problemy i błędy; mogą być one związane z błędami danych wejściowych, lecz także mogą je wywoływać nieprzewidziane przypadki brzegowe, których funkcja nie uwzględnia. Także w takich sytuacjach funkcjonalność *Utwórz funkcję* może ułatwić nam życie, ponieważ udostępnia systematyczny sposób sprawdzania każdego kroku funkcji. Dzięki temu, jak pokażemy w kolejnym podpunkcie rozdziału, możemy w prosty sposób odnajdywać i eliminować przyczyny wszelkich problemów.

Upraszczanie rozwiązywania problemów i wprowadzania zmian

Piękno funkcjonalności *Utwórz funkcję* w znacznej mierze wiąże się z łatwością wprowadzania modyfikacji, którą ta funkcjonalność zapewnia. Załóżmy, że do naszej tabeli kalendarza musimy dodać kolumnę określającą numer dnia. Aby wprowadzić tę zmianę, wystarczy przejść do początkowego zapytania `LogikaKalendarza` i wstawić w nim kod tworzący nową kolumnę. Proces ten przedstawiliśmy na rysunku 9.17.



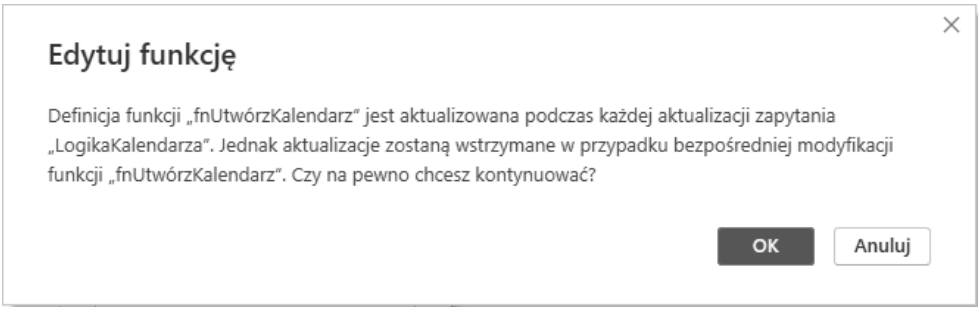
Rysunek 9.17. Dynamiczne połączenie — zapytanie i funkcja niestandardowa są aktualizowane razem

Na rysunku 9.17 jest widoczny nowy krok o nazwie `DodanoDzień` wstawiony do zapytania `LogikaKalendarza`. Dynamiczne połączenie między funkcją `fnUtwórzKalendarz` i zapytaniem `LogikaKalendrza` oznacza, że wszelkie zmiany wprowadzone w zapytaniu — czy to dodanie kolumny, czy też zmiana formuły — są natychmiast synchronizowane, co pozwala uniknąć konieczności ręcznej edycji i zmniejsza ryzyko wystąpienia błędów. Nie tylko upraszcza to proces rozwiązywania problemów — gdyż możemy sprawdzić każdy krok zapytania — ale także ułatwia dodawanie nowej logiki w odpowiedzi na zachodzące zmiany.

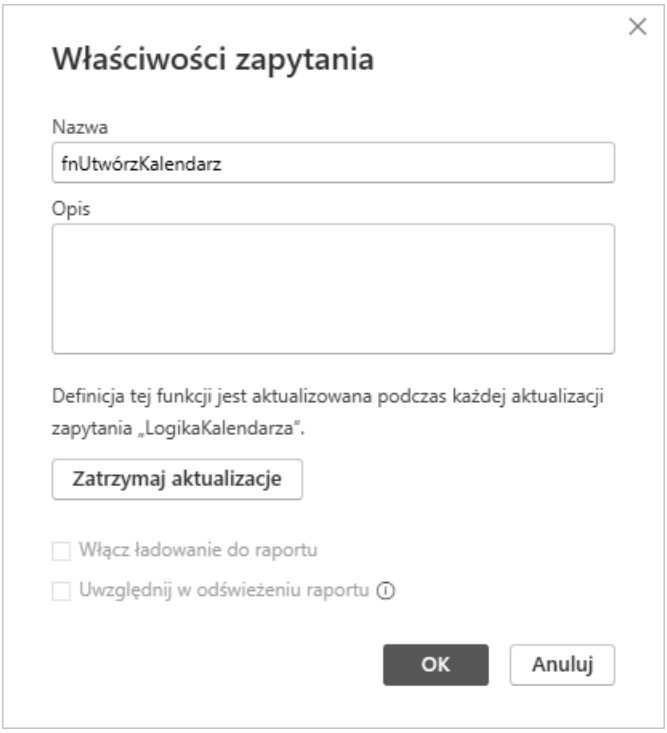
W przypadku próby otworzenia takiej połączonej funkcji w edytorze zaawansowanym zostanie wyświetlony komunikat ostrzegawczy, przedstawiony na rysunku 9.18. Komunikat ten informuje, że jakakolwiek modyfikacja funkcji spowoduje zerwanie połączenia pomiędzy zapytaniem oraz funkcją.

Jeśli jedyną rzeczą, którą chcesz zrobić, jest przejrzenie kodu zapytania, to nic nie stoi na przeszkodzie, byś to zrobił — spokojnie możesz kliknąć przycisk *OK* i obejrzeć ten kod. Pamiętaj tylko, żeby nie zapisywać żadnych zmian, gdyż jeśli to zrobisz, połączenie zapytania z funkcją zostanie zerwane.

Jeśli będziesz chciał zatrzymać aktualizowanie funkcji w przypadku zmiany zapytania, możesz to zrobić, klikając funkcję prawym przyciskiem myszy i wybierające z menu podręcznego opcję *Właściwości* (*Properties*). Spowoduje to wyświetlenie na ekranie okna dialogowego *Właściwości zapytania* (*Query properties*) przedstawionego na rysunku 9.19, w którym wystarczy kliknąć przycisk *Zatrzymaj aktualizacje* (*Stop updates*).



Rysunek 9.18. Komunikat ostrzegawczy wyświetlany podczas próby edycji funkcji połączonej z zapytaniem



Rysunek 9.19. Okno dialogowe Właściwości zapytania dla funkcji, której wyniki są aktualizowane po zmianie zapytania

Wskazówka

Rada mistrza: bądź ostrożny! Kiedy już klikniesz przycisk *Zatrzymaj aktualizacje*, bezpowrotnie stracisz możliwość aktualizowania funkcji po zmianach zapytania. Gdybyś chciał ją odzyskać, musiałbyś utworzyć nową funkcję, używając do tego celu funkcjonalności *Utwórz funkcję* zgodnie z opisem zamieszczonym we wcześniejszej części rozdziału.

Jak pokazaliśmy, funkcjonalność *Utwórz funkcję* udostępnia wygodny sposób przekształcania zapytań w funkcje nadające się do wielokrotnego użycia i zapewniające łatwość wprowadzania zmian w ich logice działania. Jak po utworzeniu takiej niestandardowej funkcji można jej używać we własnych raportach? Sposoby wywoływania takich funkcji zostały opisane w następnym punkcie rozdziału.

Wywoływanie funkcji niestandardowych

Po utworzeniu funkcji można ją wywoływać na dwa sposoby, wymienione na poniższej liście:

- ręcznie — w edytorze zaawansowanym lub na pasku formuły;
- przy użyciu interfejsu użytkownika.

A zatem jak to zrobić w praktyce?

Wywoływanie ręczne, w edytorze zaawansowanym lub na pasku formuły

Większość zaawansowanych użytkowników wywołuje funkcje, odwołując się do nich bezpośrednio. Załóżmy, że dysponujemy funkcją o nazwie `MojaFunkcja`, która pobiera jedną wartość liczbową i zwraca tę liczbę powiększoną o 10. Aby wywołać taką funkcję, wystarczy utworzyć puste zapytanie i wpisać w pasku formuły kod o postaci `= MojaFunkcja( 20 )`, jak pokazaliśmy na rysunku 9.20.

Rysunek 9.20. Wywoływanie funkcji w pasku formuły

Powyższy kod wywołuje naszą funkcję i zwraca wartość `30`. W zależności od wyrażenia, w którym umieszczone jest wywołanie funkcji, jego wynikiem może być niemal wszystko: lista, tabela lub nowa kolumna w już istniejącej tabeli — cokolwiek. Jeśli chcesz wywołać swoją funkcję w nowej kolumnie, to jak się już zaraz przekonasz, możesz to zrobić także w inny sposób.

Zastosowanie interfejsu użytkownika

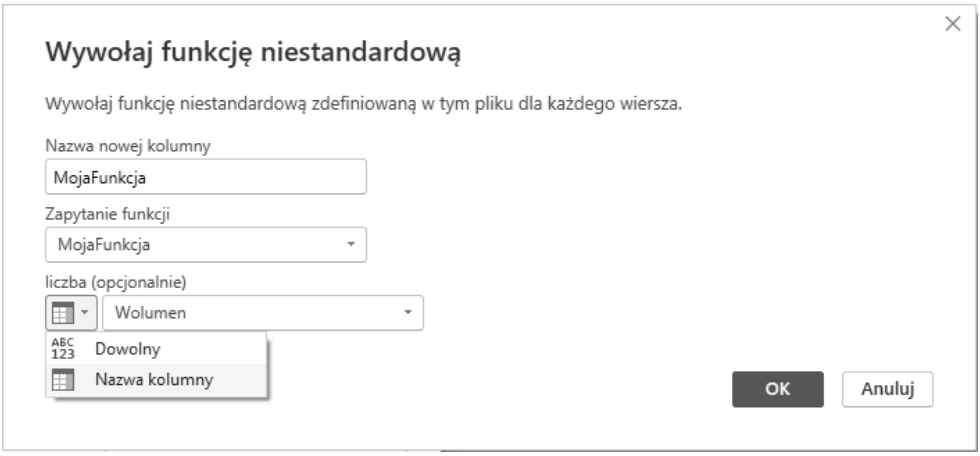
Power Query zapewnia także możliwość utworzenia z poziomu interfejsu użytkownika kolumny wywołującej określoną funkcję. Oto czynności, jakie będziesz musiał w tym celu wykonać:

1. Na Wstążce wyświetl kartę *D□daj k□lumnę* (*Add C□lumn*).
2. Kliknij przycisk *Wyw□łaj funkcję niestandard□wą* (*Inv□ke Cust□m Functi□n*).

Przycisk ten przedstawiliśmy na rysunku 9.21.

Rysunek 9.21. Przycisk pozwalający na wywoływanie funkcji niestandardowej

Kliknięcie tego przycisku spowoduje wyświetlenie okna dialogowego przedstawionego na rysunku 9.22. Należy w nim podać nazwę nowej kolumny oraz wybrać funkcję, która zostanie użyta do wyznaczenia jej wartości. Po wybraniu z listy *Zapytanie funkcji* (*Function name*) nazwy funkcji poniżej zostaną wyświetlone pola zależne od parametrów funkcji i umożliwiające określenie wartości argumentów, które będą przekazywane do funkcji dla każdego wiersza tabeli. Dostępne są tu dwie możliwości:



Rysunek 9.22. Okno dialogowe Wywołaj funkcję niestandardową

- **ustalona wartość** — w przypadku wybrania opcji *Dowolny* (*Enter a value*) będzie można podać konkretną wartość, która będzie używana dla każdego wiersza tabeli;
- **wartość kolumny** — w przypadku wybrania opcji *Nazwa kolumny* (*Use values in a column*) w wywołaniu przekazywana będzie wartość określonej kolumny z tego samego wiersza tabeli.

Ponieważ wszystkie te działania wykonaliśmy na karcie *Dodaj kolumnę*, wyniki wywołania wskazanej funkcji niestandardowej zostaną umieszczone w nowej kolumnie tabeli.

Skoro już dysponujesz podstawowymi informacjami na temat wywoływania funkcji, możemy przyjrzeć się dokładniej wyrażeniu, które w języku M jest najczęściej używane do tworzenia funkcji: wyrażeniu `each`. Jest ono najczęściej używane, gdyż Power Query automatycznie wstawia je w wywołaniach bardzo wielu funkcji podczas wykonywania operacji na danych z poziomu interfejsu użytkownika.

Wyrażenie each

Wyrażenie `each` w języku M jest skrótowym sposobem definiowania funkcji jednoargumentowej, bez konieczności stosowania bardziej rozbudowanej deklaracji.

Ale dlaczego w języku M w ogóle jest dostępne wyrażenie `each`, skoro możemy definiować nasze własne funkcje niestandardowe? Najważniejszym powodem jest to, że wyrażenie `each` jest znacznie bardziej przyjazne dla użytkownika i pozwala na pisanie bardziej zwięzłego kodu.

Zacznijmy od prostego przykładu, który to wyraźnie pokaże. Wyobraź sobie, że chcesz napisać funkcję, która pobiera liczbę i dodaje do niej wartość `5`. Oto w jaki sposób mógłbyś to zrobić, używając normalnego sposobu:

```
( mojaWartość ) => mojaWartość + 5
```

W tym przykładzie `mojaWartość` jest pojedynczym parametrem funkcji. A teraz spróbujmy nieco uprościć tę funkcję:

```
(_) => _ + 5
```

Ta funkcja także ma jeden parametr, choć jest on zapisany jako znak podkreślenia. Ten zapis jest bardzo podobny do sposobu stosowania wyrażenia `each`. Konstrukcja `each` jest tak zwanym „cukrem składniowym" (ang. *syntax suger*) upraszczającym tworzenie jednoargumentowych funkcji, których parametr jest reprezentowany jako znak podkreślenia.

Oznacza to, że poniższe wyrażenie jest jeszcze bardziej uproszczonym sposobem zdefiniowania przedstawionej wcześniej funkcji:

```
each _ + 5
```

Ta uproszczona deklaracja `each` jest używana w celu poprawy czytelności wywołań funkcji wyższego rzędu. Jest to przydatne rozwiązanie, ponieważ przyspiesza pisanie kodu i często pozwala go skracać. Aby zrozumieć, jak to osiągnąć, przyjrzyjmy się kilku typowym przypadkom użycia wyrażeń `each`.

Popularne przypadki użycia

Przykłady zastosowania wyrażenia `each` można znaleźć bardzo często i w różnych funkcjach. Na przykład jest ono powszechnie umieszczane w wywołaniach funkcji `Table.Add`
`↪Column`, `Table.SelectRows` i `List.Transform`. W dalszej części tego podpunktu wykorzystamy te funkcje w kilku przykładach. Pokażemy w nich, jak:

- automatycznie dodawać wyrażenie `each` w niestandardowych kolumnach;
- używać wyrażenia `each` w funkcji `Table.TransformColumns`;
- pomijać znaki podkreślenia w odwołaniach do pól i kolumn;
- upraszczać wywołania funkcji jednoargumentowych.

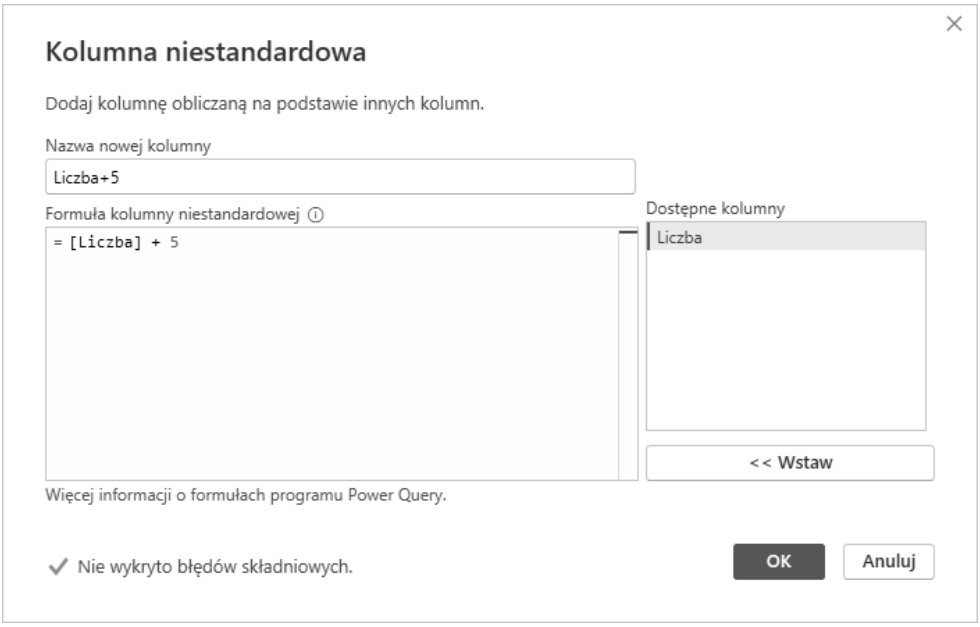
Zobaczmy zatem, jak można używać wyrażenia `each`.

Automatyczne dodawanie wyrażenia each w kolumnach niestandardowych

Wyobraź sobie, że dysponujesz kolumną o nazwie `Liczba` i chcesz do każdej z zapisanych w niej wartości dodać `5`. W tym celu przejdź na kartę *Dodaj kolumnę*, kliknij przycisk *Kolumna niestandardowa* (*Custom Column*), a następnie w wyświetlonym oknie dialogowym wpisz poniższą formułę:

```
[Liczba] + 5
```

Okno dialogowe *Kolumna niestandardowa* (*Custom column*) z wypełnionymi polami nowej kolumny zostało przedstawione na rysunku 9.23.



Rysunek 9.23. Tworzenie nowej kolumny

Po kliknięciu przycisku *OK* w pasku formuły zostanie wyświetlone następujące wywołanie:

```
= Table.AddColumn(
  Źródło,
  "Liczba+5",
  each [Liczba] + 5)
```

Chwila! W ostatnim wierszu tego kodu zostało użyte wyrażenie `each`! Skąd się ono wzięło, skoro nie wpisałeś go w polu formuły kolumny niestandardowej w oknie dialogowym?

Oto wyjaśnienie: kiedy dodajemy kolumnę niestandardową, używając do tego celu interfejsu użytkownika Power Query, aplikacja ta wykonuje część pracy za nas. Automatycznie wywołuje funkcję `Table.AddColumn` i poprzedza wpisane wyrażenie słowem kluczowym `each`. Nie musimy go wpisywać ręcznie. To bardzo przydatne i wygodne rozwiązanie, gdyż koncepcja słowa kluczowego `each` często jest myląca dla początkujących użytkowników Power Query. Przekonajmy się teraz, w jaki sposób ta sama koncepcja jest stosowana dla funkcji `Table.TransformColumns`.

Stosowanie each w wywołaniach funkcji Table.TransformColumns

Funkcja `Table.TransformColumns` służy do manipulowania wartościami kolumn. W przypadku korzystania z niej za pośrednictwem interfejsu użytkownika Power Query w jej wywołaniach stosowane jest słowo kluczowe `each`.

Wyobraź sobie, że chcesz dodać słowo „Pani" do wartości w kolumnie o nazwie `Imię`. W tym celu musisz zaznaczyć odpowiednią kolumnę, wyświetlić na Wstążce kartę *Przekształć* (*Transform*), kliknąć przycisk *Format*, wybrać opcję *Dodaj prefiks* (*Add Prefix*), a następnie w wyświetlonym oknie dialogowym wpisać odpowiednie słowo i wstawić spację (`Pani `). Kiedy to zrobisz, w pasku formuły zostanie wyświetlone następujące wywołanie:

```
Table.TransformColumns(
  Źródło, {{"Imię", each "Pani " & _, type text}} )
```

Pod względem działania powyższe wywołanie odpowiada temu przedstawionemu poniżej:

```
Table.TransformColumns(
  Źródło, {{"Imię", (_) => "Pani " & _, type text}} )
```

Zwróć uwagę na różnice pomiędzy tymi dwoma wywołaniami. W tym przykładzie najważniejsze jest zrozumienie sposobu działania funkcji `Table.TransformColumns`. Wymaga ona zdefiniowania w określonej formie sposobu przekształcania każdej wartości w kolumnie. W tym miejscu na scenę wkracza parametr funkcji, który reprezentuje każdą wartość zapisaną w modyfikowanej kolumnie.

W pierwszym z przedstawionych wywołań używamy wyrażenia `each` — zapisu skrótowego. W drugim przykładzie użyliśmy zapisu `(_) =>`, jawnego sposobu definiowania funkcji. Oba te rozwiązania zapewniają ten sam efekt: poprzedzenie tekstu zapisanego w kolumnie `Imię` słowem `Pani`.

Jak pokazaliśmy na tych przykładach, znak podkreślenia zapewnia możliwość odwoływania się do przekształcanej wartości. Przejdźmy zatem do kolejnego przykładu, który pokaże, w jakich sytuacjach można pomijać ten znak podkreślenia.

Pomijanie znaku podkreślenia w odwołaniach do pól i kolumn

W określonych sytuacjach, kiedy mamy do czynienia z rekordami lub polami tabel, można pominąć znak podkreślenia. Załóżmy, że chcemy dodać do tabeli nową kolumnę, która od wartości zapisanej w kolumnie `Sprzedaż` odejmuje `10`. Wywołanie, które robi to, używając bardziej rozbudowanej definicji funkcji, będzie mieć następującą postać:

```
Table.AddColunn(Źródło, "Niestandardowa", (_) => _[Sprzedaż] - 10)
```

W tym przypadku znak podkreślenia odwołuje się do wiersza tabeli, reprezentowanego w formie rekordu.

W przypadku stosowania bezpośrednich odwołań do pól przy wykorzystaniu nawiasów kwadratowych — czyli procesu określanego jako wybór pola — poprzedzanie nazwy pola znakiem podkreślenia nie jest konieczne. A zatem powyższe wywołanie można zapisać w prostszej postaci:

```
Table.AddColunn(Źródło, "Niestandardowa", (_) => [Sprzedaż] - 10)
```

Interfejs użytkownika Power Query upraszcza to wywołanie jeszcze bardziej, zapisując je w formie wyrażenia `each`:

```
Table.AddColumn(Źródło, "Niestandardowa", each [Sprzedaż] - 10)
```

Wszystkie te trzy wywołania dają te same wyniki: pomniejszenie wszystkich wartości zapisanych w kolumnie `[Sprzedaż]` o `10`.

Dowiedziałeś się zatem, jak można upraszczać pisany kod, pomijając w nim znaki podkreślenia. Jednak na tym możliwości upraszczania kodu jeszcze się nie kończą: dodatkowe możliwości pojawiają się podczas wywoływania funkcji jednoargumentowych. Opisaliśmy je w następnym podpunkcie rozdziału.

Upraszczanie wywołań funkcji jednoargumentowych

Wiele funkcji umożliwia przekazywanie innej funkcji jako argumentu. Sposób, w jaki należy to robić, zależy od liczby argumentów wymaganych przez funkcję zagnieżdżoną. W przypadku funkcji jednoargumentowych rozwiązanie jest bardzo proste. Można całkowicie pominąć nawiasy i odwołać się bezpośrednio do nazwy funkcji.

Jako przykład rozważmy poniższe zapytanie, w którym dodajemy `10` do każdej wartości na liście:

```
let
  fnDodaj10 = (Wartość) => Wartość + 10,
  MojaLista = { 1 .. 5 },
  PrzekształconoListę = List.Transform( MojaLista, fnDodaj10 )
in
  PrzekształconoListę
```

W tym przypadku funkcja `fnDodaj10` została użyta wewnątrz funkcji `List.Transform` do przekształcenia wszystkich elementów listy. Jednak krok `PrzekształconoListę` z powyższego zapytania można zapisać w inny sposób:

```
  PrzekształconoListę = List.Transform( MojaLista, each fnDodaj10(_) )
```

Te przykłady pokazują, jak można upraszczać stosowanie funkcji jednoargumentowych — można w nich pomijać słowo kluczowe `each` oraz nawiasy.

Jednak w przypadku funkcji wymagających przekazania więcej niż jednego argumentu sytuacja się komplikuje. Podczas korzystania z takich funkcji mamy dwie możliwości:

- zastosować słowo kluczowe `each`;
- formalnie zapisać pełną definicję funkcji.

Poniżej przedstawiono przykład pierwszego z tych rozwiązań:

```
PrzekształconoListę = List.Transform( MojaLista, each fnDodaj10( _, 10 ) )
```

A oto przykład użycia pełnej definicji funkcji:

```
PrzekształconoListę = List.Transform( MojaLista, (_) => fnDodaj10( _, 10 ) )
```

Podsumowując, funkcje jednoargumentowe mogą być przekazywane jako argument poprzez proste podanie ich nazwy, bez nawiasów. W przypadku funkcji wieloargumentowych konieczna jest bardziej rozbudowana i jawna składnia, wymagająca użycia słowa kluczowego `each` lub pełnej definicji funkcji niestandardowej.

Do tej pory wyjaśniliśmy, jak można tworzyć i wywoływać funkcje w możliwie najprostszy sposób. Warto jednak także wyjaśnić, jak można udoskonalać tworzone funkcje niestandardowe, aby poprawiać ich niezawodność i zapobiegać występowaniu w nich potencjalnych błędów. Tym zagadnieniem zajmiemy się w następnym punkcie rozdziału.

Udoskonalanie definicji funkcji

Gdy staramy się zaimplementować idealną funkcję niestandardową, nie chodzi tylko o napisanie jej kodu — trzeba także myśleć o jej udoskonalaniu. Ważne jest, aby pamiętać, że dobrze zdefiniowana funkcja ma jasny cel, jest elastyczna w użyciu i odporna na błędy.

Tworzenie niestandardowej funkcji często zaczyna się od napisania jej podstawowej logiki, a następnie podczas kolejnych iteracji dodawane są do niej kolejne kroki, aż do momentu uzyskania pożądanych rezultatów. Zazwyczaj podczas tego etapu prac staramy się także udoskonalać nasze funkcje, tak by były one bardziej niezawodne.

W kolejnych podpunktach rozdziału opiszemy dwie metody udoskonalania funkcji niestandardowych:

- **Definiowanie typów danych**. Pozwala zapewnić spójność i dokładność danych, dzięki czemu dane wejściowe przekazywane do funkcji będą zgodne z naszymi oczekiwaniami.
- **Dodawania parametrów opcjonalnych**. To rozwiązanie zapewnia użytkownikom elastyczność stosowania funkcji i używania jej do zaspokajania różnych potrzeb bez narzucania ścisłych ograniczeń.

Rozwiązania te nie tylko zwiększają użyteczność funkcji, lecz także sprawiają, że stają się one bardziej odporne i przydatne w szerszej gamie scenariuszy. Przyjrzyjmy się zatem, w jaki sposób rozwiązania te mogą poprawiać nasze zapytania.

Określanie typów danych

Podczas tworzenia funkcji niestandardowych naszym celem jest nie tylko zapewnianie ich prawidłowego działania, ale także odpowiedniej niezawodności i odporności na błędy. Jednym ze sposobów, by to zrobić, jest jawne określanie, jakiego rodzaju dane akceptuje każdy parametr funkcji. Dzięki temu możemy zapewnić, że nasze funkcje będą akceptować dane określonego typu, co pozwoli zapobiegać ewentualnym niespodziankom. Na przykład, jeśli zapytanie oczekuje przekazania daty, a zamiast niej otrzyma listę, funkcja prawdopodobnie zwróci błąd.

Typ danych można zdefiniować zarówno dla parametrów wejściowych funkcji, jak i dla zwracanego przez nią wyniku. Parametr bez zdefiniowanego typu danych jest określany jako **parametr niejawny** (ang. *implicit parameter*). Oznacza to, że może on przyjmować wartości dowolnego typu. Określenie typu danych akceptowanych przez konkretny parametr sprawia, że staje się on **parametrem jawnym** (ang. *explicit parameter*).

Po zdefiniowaniu typu danych dla parametru wejściowego funkcje niestandardowe będą akceptować tylko wartości określonego typu. W rozdziale 5., podczas omawiania typów danych, dowiedziałeś się, że istnieją typy podstawowe oraz typy podstawowe

akceptujące wartości `null`. W następnej części rozdziału wrócimy do tego rozróżnienia, gdyż staje się ono istotne w kontekście udoskonalania funkcji niestandardowych.

Podstawowe typy danych

Zacznijmy od pokazania, jak można przypisać parametrowi wartość typu podstawowego. Poniższa funkcja jest dostępna w przykładach dołączonych do niniejszej książki.

Wyobraź sobie, że dysponujesz tabelą osób, jedna z jej kolumn określa ich wiek i chcesz utworzyć funkcję o nazwie `fnPrawidłowyWiek`. Funkcja ta ma sprawdzać, czy podana wartość wieku osoby jest realistyczna, czyli w naszym przypadku: czy mieści się w zakresie od `0` do `125`. Poniżej przedstawiliśmy kod tej funkcji:

```
(wiek as number) as logical =>
  wiek >= 0 and wiek <= 125
```

W tym przykładzie `wiek as number` jest parametrem jawnym. Taki zapis parametru oznacza, że funkcja będzie wymagać, by wartość tego argumentu była liczbą. Za parą nawiasów z parametrem został umieszczony fragment `as logical`; oznacza on, że funkcja zwraca wartość logiczną: `true` (informującą, że wiek jest prawidłowy) lub `false` (informującą, że wartość wieku jest nieprawidłowa).

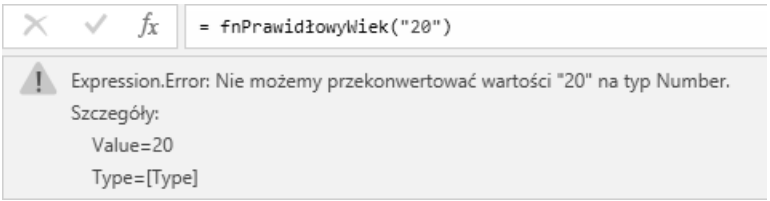
Na rysunku 9.24 przedstawiliśmy tabelę zawierającą dwie kolumny. Kolumna `fnPrawi`
`↪dłowyWiek` zawiera wyniki zwrócone przez wywołanie funkcji `fnPrawidłowyWiek` dla każdej z wartości w kolumnie `Wiek`.

	Wiek	fnPrawidłowyWiek
1	23	TRUE
2	105	TRUE
3	5	TRUE
4	71	TRUE
5	-20	FALSE
6	195	FALSE

Rysunek 9.24. Funkcja sprawdza, czy wartości w kolumnie Wiek są prawidłowe

Ta nowa kolumna zawiera wartości `true` lub `false` informujące, czy wartość kolumny `Wiek` z tego samego wiersza jest prawidłowa. Ze względu na to, że w definicji funkcji określiliśmy typ parametru wejściowego, silnik języka musi sprawdzać, czy przekazywane wartości wejściowe są zgodne z tym typem. Jak widać na rysunku, aktualnie funkcja dla każdego wiersza zwraca prawidłowy wynik, gdyż w każdym z tych wierszy wartość w kolumnie `Wiek` jest liczbą. Jeśli jednak w wywołaniu tej funkcji spróbujemy przekazać jakiś tekst (na przykład `"20"`) lub jakąkolwiek inną wartość, która nie jest liczbą, Power Query wyświetli błąd przedstawiony na rysunku 9.25.

Komunikat ten jest wyświetlany, gdyż nasz parametr jawny wymaga, by funkcja akceptowała wyłącznie wartości liczbowe (typu `number`), jednak w tym przypadku przekazany został łańcuch znaków (wartość typu `text`). Powyższe wyjaśnienia dotyczyły wartości przekazywanych do funkcji; jednak w podobny sposób można także określać typ wyniku zwracanego przez funkcję.



Rysunek 9.25. Przekazanie wartości, która nie jest liczbą, spowoduje wystąpienie błędu

Jeśli chodzi o zwracany wynik, to aktualnie nasza funkcja oczekuje, że będzie zwracana wartość logiczna (`true` lub `false`). W definicji funkcji typ wyniku został określony jako `logical`. Jeśli zmienimy `as logical` na `as text`, lecz wciąż będziemy zwracali z funkcji wynik typu logicznego, to zostanie wyświetlony komunikat o błędzie informujący o tym, że typ danych wyniku nie jest zgodny z oczekiwaniami.

Kolejnym ważnym aspektem tworzenia niezawodnych funkcji niestandardowych jest odpowiednia obsługa wartości `null`. Bardzo często zdarza się, że kolumny tabel będą zawierać zarówno wartości `null` (określane także jako wartości *puste*), jak i liczby. Wartości puste działają w specyficzny sposób i z łatwością mogą doprowadzić do błędów w działaniu zapytań. Funkcja przedstawiona na ostatnim przykładzie nie obsługuje wartości pustych w prawidłowy sposób, dlatego też stosowanie jej w zapytaniach może być ryzykowne, a w razie wystąpienia wartości pustych może doprowadzić do błędów. Na szczęście możemy w prosty sposób zabezpieczyć tę funkcję na taką okoliczność, dodając do parametrów w jej definicji wsparcie dla typów akceptujących wartości puste. W następnym podpunkcie rozdziału opiszemy, jak to zrobić.

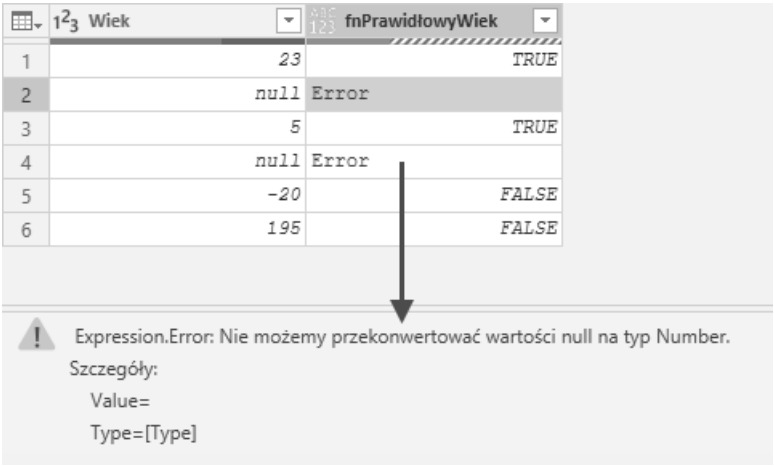
Typy podstawowe akceptujące wartości puste

Czasami w danych może brakować pewnych wartości bądź też mogą się w nich pojawiać wartości nieznane. Są one wspólnie określane jako wartości puste (ang. *null values*). W scenariuszach, w których kolumna oprócz wartości jednego z podstawowych typów danych może także zawierać wartości puste, nieocenione stają się typy podstawowe, które je akceptują (ang. *nullable primitive types*). Typy te są zaprojektowane tak, aby akceptować zarówno wartości określonego typu podstawowego, jak i wartości puste. Po raz pierwszy o tego rodzaju typach danych wspomnieliśmy w rozdziale 5. „Typy danych", jednak są one zagadnieniem złożonym. Zachęcamy, żebyś przed kontynuowaniem lektury ponownie zapoznał się z zamieszczonymi tam informacjami.

Stosowanie ścisłych typów podstawowych (takich jak `text`) w kolumnach, w których mogą pojawiać się wartości puste, może prowadzić do występowania błędów w zapytaniach w przypadkach faktycznego wystąpienia wartości pustych. Z drugiej strony zadeklarowanie typu kolumny jako `any` niejednokrotnie jest zbyt szeroką klasyfikacją. Dobrą praktyką pozwalającą na rozwiązanie tego problemu jest stosowanie typów podstawowych akceptujących wartości puste. Termin „akceptujący wartości puste" w tym przypadku oznacza, że w kolumnie można zapisywać zarówno wartości określonego typu podstawowego, jak i wartości `null`. A w jaki sposób można zastosować to rozwiązanie?

Czy pamiętasz naszą funkcję sprawdzającą poprawność wartości wieku, przedstawioną w poprzednim podpunkcie rozdziału? Załóżmy, że w zestawie danych, którym dysponujesz, może brakować niektórych wartości. Gdybyś na takim zestawie danych zastosował

przedstawioną wcześniej wersję funkcji, to pojawiłyby się w nim błędy, takie jak te przedstawione na rysunku 9.26.



Rysunek 9.26. Kiedy funkcja napotka nieoczekiwane wartości null, pojawiają się błędy

Funkcja `fnPrawidłowyWiek` sprawdza, czy przekazana do niej wartość wejściowa odpowiada oczekiwanemu typowi danych. W swojej obecnej postaci funkcja ta akceptuje wyłącznie liczby (wartości typu `number`). Kiedy napotka wartość, której nie jest w stanie rozpoznać, taką jak wartość pusta (`null`), nie wie, co z nią zrobić, i zgłasza błąd.

Jak widać, poznanie sposobów radzenia sobie w przypadkach występowania wartości pustych może być bardzo przydatne. Wiedza ta pozwala obsługiwać niedoskonałe dane, które bardzo często występują w źródłach danych. Rozwiązanie tego problemu polega na zmianie typu parametru z `number` na `nullable number`, jak pokazano na poniższym przykładzie:

```
(wiek as nullable number) as logical =>
      if wiek >= 0 and wiek <= 125 then true else false
```

W tej postaci funkcja akceptuje wartości wieku, które są liczbami lub wartościami `null`. Po wprowadzeniu tej zmiany funkcja będzie próbowała zwracać wartości `null` dla wszystkich nierozpoznanych wartości. Niestety nawet po wprowadzeniu tej zmiany funkcja wciąż zgłasza błędy, choć tym razem nieco inne: `Expression.Error: Nie możemy przekonwer` ↪`tować wartości null na typ Logical.`

Tym razem przyczyną błędu jest to, że funkcja wymusza zwracanie wyników tylu `logical` — wartości logicznych. A ponieważ funkcja stara się zwracać wartości `null`, zawsze gdy zostanie do niej przekazana wartość `null`, pojawia się błąd.

Także ten problem można łatwo rozwiązać — wystarczy dodać słowo kluczowe `nullable` do typu wartości wynikowej funkcji:

```
(wiek as nullable number) as nullable logical =>
      if wiek >= 0 and wiek <= 125 then true else false
```

Po wprowadzeniu tej zmiany funkcja będzie już zwracać zarówno wartości logiczne, jak i wartości `null`. Wyniki zastosowania tej funkcji dla kolumny zawierającej wartości `null` przedstawiliśmy na rysunku 9.27.

	Wiek	fnPrawidłowyWiek
1	23	TRUE
2	null	null
3	5	TRUE
4	null	null
5	-20	FALSE
6	195	FALSE

Rysunek 9.27. Wyniki zastosowania funkcji akceptującej i zwracającej wartości puste

Po zastosowaniu tej nowej wersji funkcji do kolumny wieku (z wartościami `null`) przekonasz się, że działa ona prawidłowo.

Podsumowując: definiując typy danych, sprawiamy, że pisane przez nas funkcje stają się bardziej precyzyjne i zyskujemy możliwość sprawdzania, czy przekazywane do funkcji wartości wejściowe oraz zwracane przez nią wyniki są odpowiednich, oczekiwanych typów. Taka funkcja wie, jakich danych powinna oczekiwać, i jak ma na nie odpowiadać.

Mogą się także pojawiać sytuacje, w których będziemy chcieli, by nasza funkcja dysponowała możliwością obsługi różnych scenariuszy. Załóżmy, że niektóre scenariusze wymagają zastosowania większej liczby zmiennych niż inne. W takich przypadkach można by skorzystać z możliwości tworzenia argumentów opcjonalnych. Dzięki temu można by używać kilku argumentów wymaganych, aby uprościć sprawę w podstawowych sytuacjach, a jednocześnie udostępnić większe możliwości funkcjonalne, definiując w funkcji jeden lub więcej argumentów opcjonalnych. W kolejnym podpunkcie rozdziału opisaliśmy, jak to można zrobić.

Definiowanie i stosowanie parametrów opcjonalnych

Domyślnie w wywołaniu funkcji dla każdego zdefiniowanego parametru trzeba określić jego wartość. Jednak w przypadku pracy z funkcjami niestandardowymi bardzo ważną cechą jest elastyczność. Skutecznym sposobem zapewnienia tej elastyczności jest zastosowanie w funkcji parametrów opcjonalnych. Oznacza to, że w wywołaniu funkcji nie trzeba będzie podawać wartości każdego z jej parametrów. Na przykład funkcja `Date.StartOfWeek` domyślnie wymaga przekazania jednego argumentu. Jej drugi, opcjonalny argument pozwala określić, który dzień tygodnia ma być traktowany jako jego początek. Jak tworzy się takie parametry opcjonalne?

Podczas definiowania parametrów funkcji można określić je jako opcjonalne, poprzedzając je słowem kluczowym `optional`. Takie opcjonalne parametry często posiadają wartości domyślne, dzięki którym funkcja wciąż będzie mogła działać, choć wartość parametru nie została jawnie określona w wywołaniu. Przeanalizujmy taki parametr opcjonalny na przykładzie.

We wcześniejszym przykładzie sprawdzaliśmy, czy wartość wieku jest prawidłowa, czy nie. Załóżmy, że teraz chcemy zapewnić użytkownikom możliwość określania maksymalnego dopuszczalnego wieku. W tym celu dodamy do naszej wcześniejszej funkcji parametr opcjonalny. Poniżej przedstawiliśmy nową wersję jej kodu:

```
(wiek as nullable number, optional maxWiek as nullable number)
  as nullable logical =>
    wiek >= 0 and wiek <= (maxWiek ?? 125)
```

W tej wersji funkcja definiuje parametr `optional maxWiek` typu `number` akceptującego wartości puste. Przeanalizujmy dokładniej tę funkcję:

- Powyższa funkcja ma dwa argumenty. Pierwszym z nich jest wymagany argument `wiek`, natomiast drugi argument, `maxWiek`, jest opcjonalny.
- Drugi argument jest opcjonalny dzięki poprzedzeniu jego nazwy słowem kluczowym `optional`. Dzięki temu podczas wywoływania tej funkcji określanie wartości drugiego argumentu nie będzie konieczne.
- Oba argumenty mają typy akceptujące wartości puste, co oznacza, że pozwalają na przekazywanie wartości wybranego typu podstawowego lub wartości `null`.
- Operator obsługi wartości `null` (`??`) zastosowany dla parametru `maxWiek` pozwala określić jego wartość domyślną; sposób działania tego rozwiązania opiszemy już niebawem.

Co się stanie, kiedy wywołamy tę funkcję bez określania wartości argumentu opcjonalnego? Jeśli jego wartość zostanie pominięta w wywołaniu funkcji, funkcja domyślnie przypisze mu wartość `null`. Koniecznie należy to zapamiętać, a pamiętając o tym sposobie działania funkcji, należy jakoś obsłużyć potencjalne pojawienie się pustej wartości parametru. I właśnie w tym miejscu przyda się nam operator `??`. Pozwala on na określenie wartości zastępczej. Operator ten w pierwszej kolejności sprawdza wartość lewego operandu. Jeśli będzie ona różna od `null`, zostanie ona zwrócona jako wartość całego wyrażenia; w przeciwnym razie wartością wyrażenia zostanie wartość zastępcza zapisana po prawej stronie operatora. W kontekście naszego przykładu operator `??` zastosowany dla drugiego opcjonalnego argumentu funkcji zwróć wartość `125`, jeśli argument ten przyjmie wartość `null`.

Wskazówka

Rada mistrza: Power Query wymaga, by parametry opcjonalne były zapisywane za wszystkimi parametrami wymaganymi. Oznacza to, że kiedy w definicji funkcji umieścimy pierwszy parametr opcjonalny, wszystkie kolejne parametry także będą musiały być opcjonalne.

Po dodaniu parametru opcjonalnego naszą funkcję można wywoływać, podając jeden lub dwa argumenty:

```
fnPrawidłowyWiek( 100 )      // Wynik: true
fnPrawidłowyWiek( 100, 90 )  // Wynik: false
```

W pierwszym przypadku określona została wyłącznie wartość parametru wymaganego. W drugim przypadku podane zostały wartości obu parametrów — wymaganego

i opcjonalnego. W tym drugim przypadku funkcja sprawdza, czy wiek (którego wartość wynosi 100) mieści się w zakresie od 0 do 90, a ponieważ wartość ta wykracza poza dopuszczalny zakres, funkcja zwraca `false`. Dzięki tej dodatkowej możliwości naszej funkcji użytkownicy będą mogli w jakiś alternatywny sposób obsługiwać wartości odstające; na przykład zdecydować, że w takich przypadkach będą zwracać błąd, bądź też uznać, że takie dane są błędne, i zastąpić je jakąś wartością zastępczą.

Skoro już wiesz, jak tworzyć funkcje, definiować typy ich parametrów, a nawet jak definiować parametry opcjonalne, przyjrzyjmy się, w jaki sposób w kodzie funkcji można odwoływać się do kolumn i pól o określonych nazwach. Na przykład mógłbyś spróbować odwołać się do pola o określonej nazwie, używając składni wyboru pola (czyli zapisać jego nazwę wewnątrz nawiasów kwadratowych). Co zaskakujące, okazałoby się jednak, że takie rozwiązanie nie zadziała. Dowiedzmy się dlaczego.

Odwołania do kolumn i pól o określonych nazwach

Tworzenie niestandardowych funkcji w Power Query obejmuje ważny krok: definiowanie parametrów. Parametry te mają kluczowe znaczenie, gdyż zapewniają użytkownikom możliwość wpływania na wyniki zwracane przez funkcję. Jednak odwoływanie się do obiektów używanych przez logikę funkcji nie zawsze jest proste. Na przykład w standardowej formule możemy odwołać się do kolumny, używając po prostu nazwy `Kolumna1`. Jednak to bezpośrednie podejście jest nieprzydatne, gdy użytkownik wprowadzi nazwę kolumny jako parametr funkcji; w tym przypadku nie można skorzystać z typowej notacji używającej nawiasów kwadratowych.

W dalszej części tego punktu rozdziału zajmiemy się scenariuszami, w których bezpośrednie odwoływanie się do obiektów w edytorze Power Query będzie przysparzać problemów. Przedstawimy w niej alternatywne rozwiązania, koncentrując się szczególnie na wykorzystaniu funkcji. Takie podejście jest niezbędne, gdy standardowe techniki odwoływania się nie zdają egzaminu. Znajomość tych technik pozwoli Ci tworzyć bardziej skuteczne funkcje niestandardowe.

Odwołania do pól rekordów

Załóżmy, że chcesz filtrować wiersze tabeli. Domyślną funkcją służącą do filtrowania wierszy tabel jest `Table.SelectRows`. Na przykład, aby zwrócić wiersze, w których wartość w kolumnie `Sprzedaż` jest mniejsza niż `1000`, należy użyć wywołania o postaci:

```
Table.SelectRows( Źródło, each [Sprzedaż] < 1000 )
```

Ale co zrobić, gdy zechcemy napisać funkcją pozwalającą określać, na podstawie której kolumny chcemy przefiltrować tabelę? W takim przypadku mógłbyś spróbować napisać następującą funkcję, jednak okazałoby się, że ona nie zadziała:

```
( tabela as table, nazwaKolumny as text ) as table =>
  Table.SelectRows( tabela, each [nazwaKolumny] < 1000 )
```

Gdybyś spróbował wykonać tę funkcję, okazałoby się, że faktycznie filtruje ona tabelę przekazaną jako pierwszy argument. Jednak okazałoby się także, że drugi argument funkcji nie działa zgodnie z Twoimi oczekiwaniami. Zamiast pobierać dane z kolumny

o nazwie przekazanej jako argument `nazwaKolumny`, funkcja starałaby się znaleźć w tabeli kolumnę o nazwie `nazwaKolumny`.

A jak można rozwiązać ten problem? Przede wszystkim należy zacząć od zrozumienia, co oznacza zapis `[Sprzedaż]` użyty w pierwszym z przedstawionych wywołań. Może to być nazwa kolumny tabeli lub nazwa pola rekordu.

Jeśli nie jesteś pewien, co reprezentuje takie odwołanie w konkretnym przypadku, to istnieje prosty sposób, by to sprawdzić: spróbuj zastąpić warunek znakiem podkreślenia. Spowoduje to wystąpienie błędu, jednak ten błąd okaże się przydatny. Będzie on miał treść podobną do: `Nie możemy przekonwertować wartości typu Record na typ Logical`. Komunikat ten potwierdza, że mamy do czynienia z rekordem.

Aby dynamicznie odwołać się do kolumny, możemy zatem skorzystać z funkcji `Record.Field`. Funkcja ta wymaga przekazania dwóch argumentów: rekordu oraz nazwy kolumny podanej w formie tekstowej. Używając tej funkcji, możemy więc przepisać funkcję w następujący sposób:

```
( tabela as table, nazwaKolumny as text ) as table =>
  Table.SelectRows( tabela, each Record.Field(_, nazwaKolumny) < 1000 )
```

Po tej zmianie funkcja będzie filtrować wiersze na podstawie nazwy kolumny podanej przez użytkownika w jej wywołaniu, dzięki czemu uzyska oczekiwane dynamiczne zachowanie. Przedstawione rozwiązanie działa w przypadku rekordu, ale co zrobić w przypadku kolumny tabeli?

Odwołania do kolumn tabel

Innymi obiektami, które przysparzają podobnych problemów, są kolumny tabel. Załóżmy, że chcemy sprawdzić, czy wartość w bieżącym wierszu istnieje również w kolumnie innej tabeli. W tym przykładzie zastosujemy tabelę o nazwie `Bonus` zawierającą kolumnę o nazwie `Produkty`.

Gdyby kolumna nie musiała być parametrem funkcji, formuła mogłaby mieć następującą postać:

```
Table.AddColumn(
  Źródło,
  "CzyToProduktBonusowy",
  each if List.Contains(Bonus[Produkty], [Produkt]) then true
    else false )
```

W powyższym wywołaniu dodajemy do tabeli źródłowej nową kolumnę o nazwie `CzyToProduktBonusowy`. Wartości zapisane w tej kolumnie informują, czy dany produkt istnieje również na liście `Bonus[Produkty]`.

Ale co można zrobić, aby poprawić elastyczność tego zapytania? Na przykład co zrobić, gdybyśmy chcieli, by użytkownik mógł podać tabelę do przeszukania i nazwę kolumny? Dzięki temu użytkownicy mogliby przygotować prostą tabelę z wartościami zastępczymi. Mogliby na przykład przygotować tabelę kalendarza i chcieć określić nazwy kolumn dla każdego z używanych języków. Przechowywanie takich nazw w tabeli jest znacznie łatwiejsze niż podawanie ich na stałe w kodzie.

Niestety, w takiej sytuacji bezpośrednie odwołanie do kolumny nie jest możliwe. Zamiast tego należy użyć funkcji `Table.Column`, która zwraca listę wartości z określonej tabeli i kolumny.

Oto jak może wyglądać niestandardowa funkcja implementująca takie rozwiązanie:

```
( tabela as table, nazwaNowejKolumny as text, tabelaWyszukiwania as table,
↪nazwaKolumnyWyszukiwania
as text ) as table =>
  Table.AddColumn(
    tabela,
    nazwaNowejKolumny,
    each if List.Contains(
      Table.Column( tabelaWyszukiwania, nazwaKolumnyWyszukiwania),
        Record.Field( tabela, "Produkt" ) )
      then true else false )
```

W tej funkcji dostępne są cztery parametry:

- `tabela` — główna tabela, na której operujemy;
- `nazwaNowejKolumny` — nazwa nowej kolumny, którą dodajemy;
- `tabelaWyszukiwania` — tabela zawierająca przeglądaną listę;
- `nazwaKolumnyWyszukiwania` — konkretna kolumna tabeli `tabelaWyszukiwania` zawierającej listę.

Zapewnienie prawidłowego działania tej funkcji wymaga użycia dwóch funkcji: `Table.Column`, która pozwala odwoływać się do konkretnej kolumny tabeli, oraz `Record.Filed`, która pozwala odwoływać się do konkretnego pola rekordu.

Najpierw za pomocą funkcji `Table.Column` pobieramy listę wartości z wyznaczonej tabeli wyszukiwania i kolumny. Następnie funkcja sprawdza, czy bieżący produkt, określony przy użyciu wywołania funkcji `Record.Field`, istnieje na tej liście. Na podstawie wyniku tego testu funkcja zwraca wartość `true` lub `false`.

Ważnym aspektem powyższej funkcji jest to, że użytkownicy mogą teraz określić w formie tekstowej zarówno `tabelęWyszukiwania`, jak i `nazwęKolumnyWyszukiwania`, a funkcja pobierze rzeczywiste wartości kolumn. W dalszej części tego rozdziału przeanalizujemy projekt pokazujący, jak można to rozwiązanie zastosować w praktyce.

Funkcje niestandardowe mogą zwiększyć złożoność kodu. W miarę wprowadzania kolejnych iteracji ich kod wydłuża się, czasami osiągając nawet kilka stron długości. Tę dodatkową złożoność można opanować, lecz jedynie do momentu wystąpienia jakiegoś błędu. W takich przypadkach trzeba będzie analizować kod krok po kroku, aby określić miejsce, w którym błąd się pojawia. W przeciwieństwie do zwykłych zapytań funkcje nie zapewniają prostego sposobu pozwalającego na takie szczegółowe analizowanie ich działania. Niezbędne jest zatem opanowanie technik debugowania funkcji niestandardowych. W następnym podrozdziale przedstawimy efektywne sposoby, które pozwalają sprostać tym wyzwaniom.

Debugowanie funkcji niestandardowych

Kiedy już zrozumiemy podstawy funkcji niestandardowych, proces ich tworzenia staje się prosty. Polega on na zdefiniowaniu parametrów, zintegrowaniu ich z kodem zapytania, a następnie przekształceniu zapytania w funkcję. Ale co się stanie, kiedy w kodzie takiej funkcji niestandardowej zaczną pojawiać się błędy? A może próbujesz dostosować do swoich potrzeb kod znaleziony w internecie i masz z tym problemy? Po utworzeniu funkcji zniknęły kroki wyświetlane wcześniej w panelu *ZASTOSOWANE KROKI*, które zwykle pomagają w rozwiązywaniu problemów z kodem zapytań. Cóż, w tym przypadku kluczowa będzie znajomość sposobów debugowania funkcji niestandardowych.

Załóżmy, że pracujemy z konkretną funkcją, taką jak ta przedstawiona na rysunku 9.28.

```
( listaLiczb as list, wartośćMin as number, wartośćMax as number ) as text =>
let
    Lista = List.Select(listaLiczb, each _ >= wartośćMin and _ <= wartośćMax),
    ListaNaTekst = List.Transform(Lista, each Text.From(_)),
    PołączonyTekst = Text.Combine(ListaNaTekst, ", ")
in
    PołączonyTekst
```

Rysunek 9.28. Funkcję niestandardową można rozpoznać na podstawie parametrów zapisanych w pierwszym wierszu jej kodu

Powyższy kod jest wyświetlany w panelu zapytań jako funkcja. Za każdym razem, gdy wykonujemy funkcję, widzimy tylko zwrócone przez nią wyniki. Jeśli jednak dopiero zaczynasz korzystać z funkcji niestandardowych lub jeśli pojawi się jakiś błąd, to zapewne będziesz chciał przeanalizować działanie funkcji krok po kroku.

Na szczęście funkcję można w prosty sposób przekształcić ponownie w zapytanie. W tym celu:

1. Utwórz zmienną dla każdego parametru w definicji zapytania.
2. Definicję funkcji oznacz jako komentarz.

Na rysunku 9.29 pokazaliśmy, jak powinien wyglądać kod „funkcji" po wprowadzeniu tych zmian.

```
//( listaLiczb as list, wartośćMin as number, wartośćMax as number ) as text =>
let
    listaLiczb = { 1 .. 10},
    wartośćMin = 2,
    wartośćMax = 8,
    Lista = List.Select(listaLiczb, each _ >= wartośćMin and _ <= wartośćMax),
    ListaNaTekst = List.Transform(Lista, each Text.From(_)),
    PołączonyTekst = Text.Combine(ListaNaTekst, ", ")
in
    PołączonyTekst
```

Rysunek 9.29. Funkcja niestandardowa przekształcona na zwyczajne zapytanie poprzez zastąpienie parametrów odpowiednimi zmiennymi

Każdy parametr zdefiniowany w pierwszym wierszu kodu został przekształcony na zmienną zdefiniowaną wewnątrz wyrażenia `let`, w wierszach 3 – 5. Dzięki temu pozostała część zapytania nadal może odwoływać się do używanych wcześniej nazw parametrów, choć tym razem ich wartości są określone w kodzie.

W ten sposób przekształciliśmy funkcję z powrotem w zapytanie. Dzięki temu możemy dokładnie zbadać każdy krok jej logiki. Po zakończeniu debugowania możemy przywrócić funkcję do początkowej postaci — w tym celu wystarczy usunąć komentarz z pierwszego wiersza funkcji oraz oznaczyć jako komentarz zmienne zastępujące parametry.

Jak na razie funkcje zapisywaliśmy jako odrębne zapytania. Robiliśmy to, bądź to pisząc ich kod ręcznie, bądź też tworząc zapytania, które następnie przekształcaliśmy na funkcje. Należy jednak pamiętać, że całkowicie prawidłowym rozwiązaniem jest także definiowanie funkcji bezpośrednio w zapytaniu. Tak zdefiniowanej funkcji można następnie używać w dalszej części tego samego zapytania. W następnym punkcie rozdziału zajmiemy się dokładniej koncepcją zasięgu funkcji. Dowiesz się w nim, dlaczego znajomość zagadnienia zasięgu umożliwia efektywne wywoływanie funkcji w ramach jednego zapytania.

Zasięg funkcji

Pojęcie zasięgu (ang. *sc□pe*) odnosi się nie tylko do parametrów i zapytań, lecz także do definiowania funkcji. Funkcje można definiować jako:

- wyrażenia najwyższego poziomu,
- funkcje wpisane umieszczane wewnątrz zapytań.

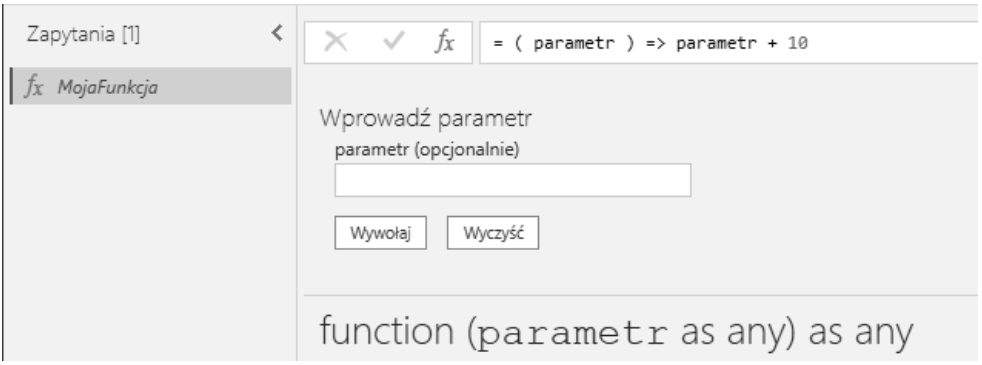
Na czym polegają różnice pomiędzy tymi dwoma rodzajami funkcji oraz w jaki sposób można różnice wykorzystać?

Wyrażenia najwyższego poziomu

Większość osób używa funkcji niestandardowych jako wyrażeń najwyższego poziomu. Aby utworzyć taką funkcję, można wkleić kod funkcji do edytora zaawansowanego i kliknąć przycisk *OK*. W efekcie Power Query automatycznie utworzy funkcję i doda ją do zapytań wyświetlonych w panelu *Zapytania* (*Queries*). Do takich funkcji można odwoływać się w innych zapytaniach i funkcjach.

Rysunek 9.30 pokazuje, jak wygląda prosta funkcja o nazwie `MojaFunkcja` oraz jak została zdefiniowana.

To, że mamy do czynienia z funkcją, można rozpoznać po symbolu *fx* wyświetlonym w panelu *Zapytania* z lewej strony jej nazwy. Dowolna nazwa nadana zapytaniu stanie się nazwą funkcji. Na przykład funkcja z rysunku 9.30 będzie nosić nazwę `MojaFunkcja`. Zaletą definiowania funkcji na najwyższym poziomie jest to, że można się do nich odwoływać w innych zapytaniach — wystarczy podać ich nazwę. To właśnie tego sposobu definiowania funkcji niestandardowych używaliśmy wcześniej w tym rozdziale. Jednak funkcje niestandardowe można także tworzyć wewnątrz zapytań; ten sposób definiowania funkcji opisaliśmy dokładniej w następnym podpunkcie rozdziału.



Rysunek 9.30. Tworzenie prostej funkcji niestandardowej

Definiowanie funkcji wewnątrz zapytań

Definiowanie funkcji wewnątrz zapytania jest przydatną techniką. Rozwiązanie to możesz wyobrazić sobie w następujący sposób: pomyśl, że funkcja jest wyrażeniem, a zapytanie składa się z sekwencji wyrażeń. I tak samo jak jest w przypadku każdego innego wyrażenia, również w zapytaniu można umieścić definicję funkcji niestandardowej.

Załóżmy na przykład, że utworzyliśmy następującą logikę i zapisaliśmy ją w zapytaniu o nazwie `Pierwiastki`:

```
let
    Wartość1 = 5,
    Wartość2 = 10,
    dzielPierwiastki = ( Liczba1, Liczba2 ) =>
        let
            Pierwiastek1 = Number.Sqrt( Liczba1 ),
            Pierwiastek2 = Number.Sqrt( Liczba2 ),
            Result = Pierwiastek1 / Pierwiastek2
        in
            Result,
    WynikFunkcji = dzielPierwiastki( Wartość1, Wartość2 )
in
    WynikFunkcji
```

Powyższy przykład, jak doskonale wiesz na podstawie wielu innych kodów przedstawionych w tej książce, jest wyrażeniem `let`. W trzecim kroku tego wyrażenia definiujemy funkcję niestandardową o nazwie `dzielPierwiastki`. Funkcję tę wywołujemy w kolejnym kroku zapytania — `WynikFunkcji`. Ten przykład pokazuje, że funkcje niestandardowe można tworzyć, nawet jeśli nie będą one zapisywane jako pojedyncze, odrębne zapytania.

Koniecznie należy zwrócić uwagę, że funkcje tworzone wewnątrz zapytań są dostępne wyłącznie w zasięgu danego zapytania. Naszej przykładowej funkcji możemy użyć w kroku `WynikFunkcji`, gdyż stanowi ona element tego samego zapytania. Jeśli przejdziemy do dowolnego innego zapytania, takiego jak `Kalendarz`, nie będziemy mogli z niej skorzystać. Dzieje się tak, gdyż możliwość odwoływania się do funkcji ogranicza się wyłącznie do miejsca, w którym została ona utworzona, a nasza funkcja `dzielPierwiastki` została utworzona poza zapytaniem `Kalendarz`.

Zastanawiając się nad sposobem utworzenia funkcji, pomyśl, jak chcesz jej używać. Jeśli chciałbyś używać jej w kilku różnych zapytaniach, zdefiniuj ją jako funkcję najwyższego poziomu. Jeśli planujesz używać funkcji tylko w jednym zapytaniu, to definiując ją wewnątrz niego, ograniczysz liczbę elementów wyświetlanych w panelu *Zapytania* i poprawisz jego organizację. Oczywiście ta decyzja należy do Ciebie. A skoro już zdobyłeś tę całą nową wiedzę o funkcjach niestandardowych, to w jaki sposób możesz ją zastosować w praktyce? Tego dowiesz się w następnym punkcie rozdziału.

Łączenie wszystkiego w całość

Czytając ten rozdział, poznałeś różne sposoby tworzenia funkcji niestandardowych i stosowania parametrów. Opisaliśmy w nim elementy funkcji niestandardowej i wyjaśniliśmy, że wyrażenie `each` jest tak zwanym „cukrem składniowym" ułatwiającym tworzenie funkcji jednoargumentowych. Następnie szczegółowo opisaliśmy zagadnienie definiowania typów danych, zarówno dla parametrów funkcji, jak i zwracanych przez nią wyników. I w końcu pokazaliśmy, w jaki sposób parametry opcjonalne zapewniają funkcjom dodatkowy poziom elastyczności.

Zachęcamy, byś korzystał z tych koncepcji we własnych projektach. Jeśli musisz używać pewnej logiki w wielu miejscach, spróbuj zaimplementować ją, choćby częściowo, w formie funkcji niestandardowej. Kluczem do zrozumienia tych wszystkich koncepcji jest stosowanie ich w praktyce. W dalszej części rozdziału postaramy się pomóc Ci w utrwaleniu tej całej wiedzy poprzez przeanalizowanie kilku praktycznych przykładów.

Przekształcanie wszystkich kolumn na tekst

Pierwszym scenariuszem, którym się zajmiemy, będzie przekształcanie wszystkich kolumn tabeli na tekst.

Przedstawienie problemu typu kolumn

W ramach pierwszego scenariusza przyjrzymy się sytuacji, w której nowe dane będą pochodzić z pliku niestrukturalnego. Problem związany z takimi niestrukturalnymi plikami, do których zaliczamy między innymi pliki CSV, Excel lub JSON, polega na tym, że nie zawierają one metadanych z informacjami o typach danych. Brak takich metadanych może nas zmuszać do wielokrotnego powtarzania działań polegających na ustawianiu odpowiednich typów danych. Aby ułatwić sobie realizację tych czynności, spróbujemy stworzyć funkcję niestandardową, która będzie to robić za nas. Zachęcamy, byś podczas lektury opisu tworzenia tej funkcji korzystał z zestawu danych dostępnego w przykładach dołączonych do książki.

Oto sytuacja, z myślą o której tworzymy naszą funkcję. Dysponujemy plikiem zawierającym ponad 20 zapytań. Większość z tych zapytań zawiera kolumny z danymi tekstowymi. Od czasu do czasu do tych danych dodawane są nowe kolumny, także tekstowe. Naszym celem jest stworzenie narzędzia, które pozwoli nam uniknąć konieczności samodzielnego ustawiania typu tych nowych kolumn. Niektóre kolumny powinny jednak pozostać niezmienione — nie chcemy konwertować ich na dane tekstowe.

Naszym zadaniem jest stworzenie funkcji, która przekształci wszystkie kolumny w zapytaniu do typu `text`. Dysponując taką funkcją, będziemy mogli po prostu wywołać ją jako ostatni krok każdego zapytania. A nawet jeśli później zostaną dodane jakieś kolumny, funkcja zmieni ich typ na tekstowy. Jak zaimplementować taką funkcję?

Wyjaśnienie przekształcania typu danych

Załóżmy, że tabela, na której pracujemy, ma postać taką jak przedstawiona na rysunku 9.31.

	Cel	Kraj	Popularne atrakcje	Optymalny termin	Opis
1	Paryż	Francja	Wieża Eiffela, Luwr, Katedra Notre…	wiosna i jesień	Znane jako „Miasto Mi
2	Tokio	Japonia	Disneyland w Tokio, świątynia Sens…	wiosna i jesień	Tętniąca życiem metro
3	Wenecja	Włochy	Grand Canal, Plac św. Marka	wiosna i lato	Słynące z romantyczny
4	Nowy Jork	USA	Times Square, Central Park, Tower	wiosna i jesień	Miasto, które nigdy n
5	Sydney	Australia	Opera w Sydney, plaża Bondi	wiosna i lato	Dom ikonicznych zabyt
6	Santorini	Grecja	Oia Fira	lato	Oszałamiająca wyspa n
7	Kair	Egipt	Piramidy w Gizie, Muzeum Egipskie	jesień i wiosna	Odkrywaj starożytną h
8	Rio de Janeiro	Brazylia	Figura Chrystusa, plaża Copacabana	lato	Słynące z karnawału,

Rysunek 9.31. Tabela z kolumnami typu any — dowolnego

Obecnie każda z kolumn tej tabeli jest typu `any`. Kiedy zmienimy typ pierwszych dwóch kolumn na `text`, Power Query zrobi to, używając funkcji `Table.TransformColumnTypes`:

```
Table.TransformColumnTypes(
  Źródło,
  {{"Cel", type text}, {"Kraj", type text}} )
```

Ten wygenerowany kod pokazuje, że funkcja wymaga dwóch rzeczy. Pierwszą jest tabela, na której będzie operować — w naszym przypadku jest to tabela `Źródło`. Drugim niezbędnym elementem jest lista zawierająca przynajmniej jedną listę zagnieżdżoną. Każda z tych list zagnieżdżonych musi zawierać dwa elementy: nazwę kolumny, której typ chcemy określić, oraz ten typ. W naszym przykładzie ta lista list ma następującą postać:

```
{{"Cel", type text}, {"Kraj", type text}}
```

Chociaż ten przykład pokazuje format, do którego dążymy, to jednak nie jest to rozwiązanie elastyczne. W końcu nazwy kolumn zostały podane na stałe w wyrażeniu. A to oznacza, że gdyby nazwy kolumn uległy zmianie lub kolumny zostały usunięte, pojawiłyby się błędy i problemy. Co więcej, takie rozwiązanie nie zapewnia możliwości obsługi kolumn, które mogą zostać dodane do tabeli w przyszłości.

To oznacza, że ten automatycznie wygenerowany kod wymagałby ciągłego ręcznego wprowadzania modyfikacji. Nie jest to praca, na którą inteligentny programista chciałby poświęcać swój czas. Przekonajmy się, czy uda się nam napisać funkcję, która będzie to robić za nas.

Implementacja wstępnej wersji funkcji

Oto co zrobimy w ramach próby napisania wstępnej wersji naszej funkcji:

1. Pobierzemy *nazwy wszystkich k□lumn* dostępnych w tabeli i zapiszemy je na liście.
2. Przekształcimy każdą nazwę kolumny z tej listy na parę składającą się z nazwy kolumny oraz typu danych.

3. Użyjemy tej przekształconej listy jako argumentu wywołania funkcji `Table.TransformColumnTypes`.

Do pobrania nazw wszystkich kolumn tabeli możemy użyć funkcji `Table.ColumnNames`. Poniższe wywołanie zwraca nazwy wszystkich kolumn tabeli określonej w kroku `Źródło`:

```
Table.ColumnNames( Źródło )
```

W naszym przykładzie zwróci ono następujące wyniki:

```
{"Cel", "Kraj", "Popularne atrakcje", "Optymalna pora", "Opis"}
```

Uzyskaliśmy więc listę nazw kolumn. Teraz chcielibyśmy przekształcić każdy z tekstów zapisanych na tej liście na listę zawierającą nazwę kolumny oraz typ danych. Do tego celu wykorzystamy funkcję `List.Transform`. Poniżej pokazaliśmy, jak to zrobić:

```
List.Transform(
  Table.ColumnNames( Źródło ),
  each {_, type text } )
```

W tym fragmencie kodu pierwszy argument wywołania funkcji `List.Transform` pobiera nazwy wszystkich kolumn tabeli `Źródło`, natomiast drugi argument określa logikę przekształcenia każdej z tych nazw na listę o pożądanej postaci.

Znak podkreślenia _ zastosowany w drugim argumencie reprezentuje kolejno każdą z nazw kolumn na liście. W zastosowanym wyrażeniu za znakiem podkreślenia umieszczony jest kod o postaci: `, type text`, a całość została zapisana wewnątrz nawiasów klamrowych. Powyższe wywołanie funkcji `List.Transform` zwróci następujące wyniki:

```
{ {"Cel", type text },
  {"Kraj", type text },
  {"Popularne atrakcje", type text },
  {"Optymalna pora", type text },
  {"Opis", type text }
}
```

Jedyne, co nam jeszcze pozostaje do zrobienia, to przekazanie tych wyników jako drugiego argumentu wywołania funkcji `Tabel.TransformColumns`. Wszystkie wykonane do tej pory kroki zapytania zostały przedstawione na rysunku 9.32.

```
1  let
2      Źródło = MiejscaIKraje,
3      Kolumny = Table.ColumnNames(Źródło),
4      PrzekształcenieTypów = List.Transform(Kolumny, each {_, type text}),
5      TypNaTekst = Table.TransformColumnTypes(Źródło, PrzekształcenieTypów)
6  in
7      TypNaTekst
```

Rysunek 9.32. Zapytanie konwertujące typ kolumn na text

Aby przekształcić ten kod w funkcję, przede wszystkim musimy określić, jakich parametrów będzie ona potrzebować. W naszym przypadku celem funkcji jest zmiana typu wszystkich kolumn tabeli na tekstowy (`text`). Zatem na pewno będziemy musieli przekazywać do funkcji tabelę. Utwórzmy odpowiedni parametr, a następnie zastosujmy go na samym początku zapytania (w wierszu nr 3).

W efekcie uzyskamy funkcję, której kod został przedstawiony na rysunku 9.33.

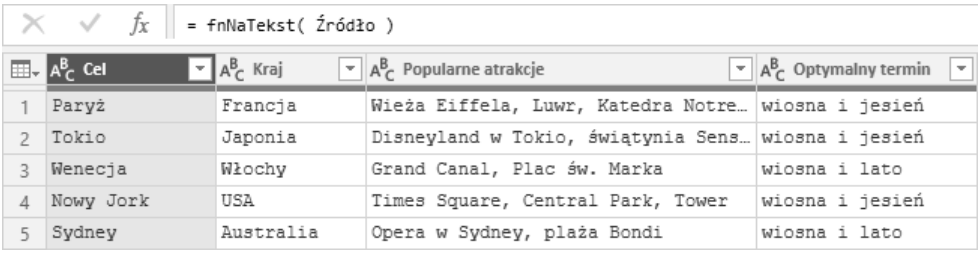
```
( TabelaWejściowa as table ) as table =>
let
    Źródło = TabelaWejściowa,
    Kolumny = Table.ColumnNames(Źródło),
    PrzekształcenieTypów = List.Transform(Kolumny, each {_, type text}),
    TypNaTekst = Table.TransformColumnTypes(Źródło, PrzekształcenieTypów)
in
    TypNaTekst
```

Rysunek 9.33. Funkcja konwertująca typ kolumn na tekstowy

Po utworzeniu tej pierwszej wersji funkcji nadamy jej nazwę `fnNaTeskt`. Od tej chwili za każdym razem, gdy będziesz chciał skonwertować typ wszystkich kolumn tabeli na `text`, wystarczy dodać krok o następującej postaci:

```
fnNaTekst( <poprzedni krok> )
```

Dodanie tego kroku poniżej kroku `Źródło` sprawi, że uzyskamy tabelę, której wszystkie kolumny będą typu `text`, jak pokazano na rysunku 9.34.



	Cel	Kraj	Popularne atrakcje	Optymalny termin
1	Paryż	Francja	Wieża Eiffela, Luwr, Katedra Notre…	wiosna i jesień
2	Tokio	Japonia	Disneyland w Tokio, świątynia Sens…	wiosna i jesień
3	Wenecja	Włochy	Grand Canal, Plac św. Marka	wiosna i lato
4	Nowy Jork	USA	Times Square, Central Park, Tower	wiosna i jesień
5	Sydney	Australia	Opera w Sydney, plaża Bondi	wiosna i lato

Rysunek 9.34. Wywołanie funkcji fnNaTekst zmienia typ wszystkich kolumn tabeli na tekstowy

To świetny punkt wyjścia do dalszych prac. Przekonajmy się, czy będziemy w stanie rozszerzyć naszą funkcję o możliwość pomijania podczas konwersji niektórych kolumn.

Rozszerzenie funkcji poprzez dodanie parametrów opcjonalnych

Powyższy kod działa i można go z powodzeniem używać. Jednak ostatnim wymogiem naszego scenariusza jest zapewnienie możliwości pominięcia konwersji typu niektórych kolumn. Ponieważ możliwość ta nie będzie używana w każdym wywołaniu funkcji, implementując ją, zastosujemy parametr opcjonalny. Dzięki temu nasza funkcja nie będzie zbytnio komplikować prostych przypadków użycia, a jednocześnie wciąż będzie pozwalała na pomijanie określonych kolumn podczas konwersji, jeśli będzie to konieczne.

Co musimy zrobić, by zaimplementować w naszej funkcji tę nową, wymaganą funkcjonalność?

1. Zaczniemy od dodania parametru opcjonalnego, który pozwoli nam przekazywać do funkcji listę nazw kolumn, które należy pominąć podczas konwersji typów.
2. Następnie zmodyfikujemy kod funkcji, tak by podczas konwersji pomijał te kolumny.

Zacznijmy zatem od dodania do funkcji parametru opcjonalnego o nazwie `Pominięcia`. Bardzo często, kiedy funkcja wymaga przekazania wielu wartości wejściowych, wygodnym rozwiązaniem jest pobieranie ich w formie listy. Dzięki temu użytkownik korzystający z takiej funkcji będzie mógł wpisać taką listę ręcznie, bądź też odwołać się do kolumny, z której zostaną pobrane wartości.

```
( TabelaWejściowa as table, optional Pominięcia as list) as table
```

Czy pamiętasz, co się dzieje, kiedy użytkownik nie poda wartości parametru opcjonalnego? Funkcja domyślnie przyjmie, że ma on wartość `null`. Zatem aby funkcja działała niezawodnie, musimy uwzględnić możliwość przyjęcia przez parametr tej wartości i odpowiednio ją obsłużyć.

Zastanówmy się więc, w jaki sposób reszta kodu używa parametru `Pominięcia`. Najprostszym sposobem pominięcia określonych nazw kolumn jest skorzystanie z funkcji `List.RemoveItems`. Można to zrobić w następujący sposób:

```
List.RemoveItems( WszystkieKolumny, Pominięcia )
```

Jeśli jednak parametr `Pominięcia` przyjmie wartość `null`, to takie wywołanie funkcji `List.RemoveItems` zgłosi błąd. Aby zapobiec takiej ewentualności, musimy użyć pustej listy jako wartości zastępczej. Możemy to zrobić przy użyciu operatora obsługi wartości `null` (`??`):

```
Pominięcia ?? {}
```

To wyrażenie zwróci pustą listę, jeśli parametr `Pominięcia` będzie miał wartość `null`. Ponieważ usunięcie pustej listy w żaden sposób nie zmienia zawartości pierwotnej listy, to rozwiązanie doskonale pasuje do naszych potrzeb. Kiedy połączymy te wszystkie fragmenty kodu, uzyskamy końcową wersję naszej funkcji, przedstawioną na rysunku 9.35.

```
1  ( TabelaWejściowa as table, optional Pominięcia as list ) as table =>
2  let
3      Źródło = TabelaWejściowa,
4      WszystkieKolumny = Table.ColumnNames(Źródło),
5      Pominięcia = Pominięcia ?? {},
6      KolumnyDoKonwersji = List.RemoveItems( WszystkieKolumny, Pominięcia),
7      PrzekształcenieTypów = List.Transform(KolumnyDoKonwersji, each {_, type text}),
8      TypNaTekst = Table.TransformColumnTypes(Źródło, PrzekształcenieTypów)
9  in
10     TypNaTekst
```

Rysunek 9.35. Dodanie do funkcji parametru opcjonalnego

Ponieważ usunięcie pustej listy nie zmienia zawartości listy początkowej, takie rozwiązanie spełnia nasze potrzeby. Na rysunku 9.36 pokazaliśmy, jak można zastosować tę funkcję do konwersji typów w naszej tabeli wejściowej, `MiejscaIKraje`.

Zwróć uwagę, że kolumny `Cel` oraz `Kraj` nie zostały skonwertowane na typ `text`. To wszystko — zadanie wykonane. Udało nam się napisać funkcję, która konwertuje kolumny na typ tekstowy. Teraz wszystko, co musimy zrobić, sprowadza się do dodania wywołania tej funkcji jako ostatniego kroku i odwołania się w nim do kroku poprzedniego. To wywołanie będzie działać zarówno na już istniejących kolumnach tabeli, jak

Rysunek 9.36. Funkcja, która konwertuje typy wszystkich kolumn poza kilkoma

i na kolumnach, które zostaną do niej dodane w przyszłości. Gdyby pojawiła się konieczność wykluczenia określonych kolumn z konwersji, możesz to zrobić, podając ich nazwy w drugim, opcjonalnym argumencie wywołania.

W następnym podpunkcie rozdziału przedstawimy kolejny użyteczny przykład, który rozwiązuje zupełnie inny problem. W tym przypadku konieczna będzie nie tylko znajomość sposobu odwoływania się do obiektów wewnątrz funkcji, lecz także zagadnienia zasięgu.

Scalanie tabel na podstawie zakresu dat

W tym podpunkcie rozdziału zajmiemy się wyzwaniem łączenia tabel na podstawie zakresu dat. Zastosujemy w nim różne pojęcia, które poznałeś podczas lektury tego rozdziału. Będziesz musiał znać wyrażenie `each` oraz pojęcie zasięgu, w którym jest wykonywany nasz kod. Wiedzę tę wykorzystamy do tworzenia odwołań do różnych wartości. Oprócz tego zastosowanie funkcji niestandardowej będzie wymuszać znajomość sposobów odwoływania się do kolumn i pól, by można je było określać przy wykorzystaniu parametrów wejściowych. Dane, których będziemy używać w tym przykładzie, są dostępne w kodach dołączonych do książki, które można pobrać z serwera FTP wydawnictwa Helion.

Wyobraźmy sobie, że mamy do czynienia z transakcjami klientów powiązanymi z określonymi umowami, ale nie dysponujemy unikalnym identyfikatorem umowy, który pozwoliłby nam je połączyć. Chociaż Power Query oferuje różne typy złączeń, raczej nie nadają się one do użycia w sytuacji, gdy tabele trzeba połączyć na podstawie zakresu dat. Na szczęście jest to doskonała okazja, by poćwiczyć swoje umiejętności rozwiązywania problemów i przekonać się, czy potrafisz napisać funkcję niestandardową, która pozwoli rozwiązać ten problem.

Nasza funkcja niestandardowa będzie musiała spełniać trzy wymagania:

- tabela główna będzie zawierać kolumnę zawierającą daty;
- tabela `Kontrakty` będzie zawierać datę początku oraz zakończenia kontraktu;
- po połączeniu tabel typy danych ze wszystkich kolumn pozostaną niezmienione.

Zaczniemy od przygotowania formuły, która wykona to specjalistyczne złączenie. Później przekształcimy ją w funkcję nadającą się do wielokrotnego użytku.

Wykonanie złączenia na podstawie zakresów dat

W tym przykładzie zastosujemy uproszczony zestaw danych, przedstawiony na rysunku 9.37. Składa się on z dwóch tabel. Pierwsza z nich, `Transakcje`, zawiera daty i kwoty transakcji. Druga tabela, `Kontrakt`, zawiera poszczególne kontrakty i dla każdego określa datę początku i końca.

Tabela: Transakcje

	Id	Data	Kwota
1	SI-1401	07.02.2023	237
2	SI-1402	03.11.2023	489
3	SI-1403	22.09.2023	712
4	SI-1404	09.05.2023	56
5	SI-1405	29.12.2023	901

Tabela: Kontrakt

	IDKontraktu	Początek	Koniec
1	VI-2023-A1	1/1/2023	31/3/2023
2	VI-2023-B2	1/4/2023	30/6/2023
3	VI-2023-C3	1/7/2023	31/8/2023
4	VI-2023-D4	1/9/2023	31/12/2023

Rysunek 9.37. Zestaw danych do testowania łączenia tabel na podstawie zakresu dat

Naszym celem jest wzbogacenie tabeli `Transakcje` o dane z tabeli `Kontrakty`. Prostym sposobem, by to zrobić, jest dodanie nowej kolumny i zastosowanie funkcji `Table.SelectRows`.

W ramach pierwszego etapu prac wykonaj następujące czynności:

1. Na Wstążce przejdź na kartę *Dodaj kolumnę* (*Add Column*) i kliknij przycisk *Kolumna niestandardowa* (*Custom Column*).
2. Nadaj nowej kolumnie nazwę `Kontrakty`.
3. Wpisz formułę: `Table.SelectRows( Kontrakt, each true )`.

Po wykonaniu tych czynności do tabeli `Transakcje` zostanie dodana nowa kolumna, a w każdym jej wierszu zostanie umieszczona cała tabela `Kontrakt`. Kod wygenerowanego zapytania będzie mieć następującą postać:

```
Table.AddColumn(
  Źródło,
  "Kontrakty",
  each Table.SelectRows( Kontrakt, each true ) )
```

Ten kod dodaje do tabeli `Źródło` nową kolumnę o nazwie `Kontrakty`. Ta nowa kolumna zawiera kopię tabeli `Kontrakt` wraz ze wszystkimi jej wierszami. Po tych przygotowaniach jesteśmy już gotowi, by umieścić wszystkie dane kontraktu w zasięgu każdej transakcji. Ponieważ mamy teraz dostęp do danych z tabeli `Kontrakt`, naszym następnym zadaniem będzie przefiltrowanie tabeli kontraktów na podstawie daty.

Modyfikacja formuły poprzez dodanie warunku określającego datę

Warunek, który podajemy w wywołaniu funkcji `Table.SelectRows`, ma wpływ wyłącznie na wiersze wybierane z tabeli `Kontrakt`. Aby wybrać tylko te kontrakty, które rozpoczęły się 1 lipca 2023 roku, moglibyśmy użyć kodu takiego jak ten przedstawiony poniżej:

```
Table.AddColumn(
  Źródło,
  "Kontrakty",
```

```
    each Table.SelectRows( Kontrakt,
      each [Początek] = #date(2023,7,1) )
)
```

Zwróć uwagę na zmianę kodu przekazanego jako drugi argument funkcji `Table.Select ↪Rows`. Teraz określa on, że interesują nas tylko wiersze z określoną datą.

Powyższy kod przedstawia niezbędny krok służący do odfiltrowania niepotrzebnych wierszy, jednak data jest w nim podana na stałe. Aby zapewnić odpowiednie działanie naszej funkcji, musimy sprawić, by podczas filtrowania wierszy był uwzględniany zakres dat oraz by funkcja określała go dynamicznie. Przekonajmy się, jak możemy to zrobić.

Implementacja dynamicznego filtra daty

Na tym etapie prac spróbujemy zastąpić datę podaną na stałe dynamicznym odwołaniem do kolumny `Data` tabeli `Transakcje`. Przy okazji ponownie będziemy musieli zmierzyć się z problemem zasięgów — tematem który został opisany w rozdziale 7. „Formułowanie języka M".

Jeśli spróbujemy zastąpić wyrażenie `#date(2023, 7, 1)` odwołaniem do pola o postaci `[Data]`, Power Query zwróci błąd, gdyż nie będzie w stanie odnaleźć pola `Data` w kontekście bieżącego rekordu.

Kluczem do rozwiązania tego problemu jest zrozumienie roli słowa kluczowego `each`. Jego pierwsze wystąpienie znajduje się w wywołaniu funkcji `Table.AddColumn`, gdzie kolumna `Data` z tabeli `Transakcje` jest nadal dostępna w zasięgu. Kolejny raz słowo kluczowe `each` pojawia się w wywołaniu funkcji `Table.SelectRows`. I w tym miejscu występuje konflikt.

Pamiętaj, że każde wystąpienie słowa kluczowego `each` jest skrótowym zapisem definicji funkcji jednoargumentowej, reprezentowanej jako `(_) =>`. Oznacza to, że kod zawiera dwie zmienne o takiej samej nazwie: `_`. A za każdym razem, gdy odwołujemy się do nazwy zmiennej zdefiniowanej wiele razy, pierwszeństwo będzie mieć ta zdefiniowana w zasięgu wewnętrznym, a nie zewnętrznym.

Aby móc odwoływać się zarówno do kolumn dostępnych w zasięgu wewnętrznym (`Początek` i `Koniec`), jak również tych z zasięgu zewnętrznego (`Data`), musimy przekształcić jedno z wyrażeń `each` w odrębną funkcję niestandardową. Dzięki temu będziemy mogli wprowadzić nową nazwę zmiennej, która pozwoli nam zarządzać tymi różnymi zasięgami. Zobaczmy zatem, jak dodać taką funkcję do naszego kodu.

Obsługa zasięgów przy wykorzystaniu funkcji niestandardowej

W wywołaniu funkcji `Table.AddColumn` można przekazywać funkcję niestandardową służącą do rozwiązywania problemów z zasięgiem. Ta funkcja niestandardowa umożliwia dostęp zarówno do tabel `Transakcje`, jak i `Kontrakt`:

```
Table.AddColumn(
  Źródło,
  "Kontrakty",
  (x) => Table.SelectRows( Kontrakt,
    each x[Data] <= [Początek] )
)
```

W tym przypadku parametr `x` przekazywanej funkcji zapewnia dostęp do zewnętrznego kontekstu, w którym widoczna jest tabela `Transakcje`. Zwróć uwagę, że parametr ten istnieje w zasięgu wywołania funkcji `Table.AddColumn`. Korzystając z tego rozwiązania, możemy przygotować warunek porównujący obie daty — początku i końca:

```
Table.AddColumn(
  Źródło,
  "Kontrakty",
  (x) => Table.SelectRows( Kontrakt,
    each x[Data] >= [Początek] and x[Data] <= Koniec)
)
```

Warunek filtrowania odwołuje się teraz do kolumny `x[Data]`, czyli do kolumny `Data` tabeli `Transakcje`. Świetnie, w ten sposób udało się nam rozwiązać problem zasięgów. Okazuje się jednak, że kiedy rozwiniemy kolumny po wykonaniu powyższych operacji, utracimy informacje o ich typach danych. Zobaczmy co możemy zrobić, żeby je zachować.

Rozwiązanie problemu typów

Teraz w każdym wierszu nowej kolumny dysponujemy już odpowiednio przefiltrowanymi wierszami kontraktów. Pozostaje nam jednak do rozwiązania jeszcze jeden problem: jak widać na rysunku 9.38, kolumna `Kontrakty` nie ma określonego typu danych.

	Id	Data	Kwota	Kontrakty
1	SI-1401	07.02.2023	237	Table
2	SI-1402	03.11.2023	489	Table
3	SI-1403	22.09.2023	712	Table
4	SI-1404	09.05.2023	56	Table
5	SI-1405	29.12.2023	901	Table

Rysunek 9.38. Kolumna powiązania nie ma określonego typu danych

Co możemy zrobić, aby określić prawidłowe typy danych w tej kolumnie? Określanie ich ręcznie w kodzie przekreślałoby sens stosowania funkcji niestandardowej. Ale przecież tabela `Kontrakt` już ma swoje określone typy danych. Być może będziemy mogli odwołać się do nich i zastosować je w nowej kolumnie?

Najprostszym rozwiązaniem pozwalającym odwołać się do typów danych innej tabeli jest skorzystanie z funkcji `Value.Type`. Możemy jej użyć, by odwołać się do tabeli `Kontrakt` i pobrać definicję jej typu danych.

Poniżej przedstawiliśmy kod, który przygotowuje dane i określa także ich typ:

```
Table.AddColumn(
  Źródło,
  "Kontrakty",
  (x) => Table.SelectRows( Kontrakt,
    each x[Data] >= [Początek] and x[Data] <= Koniec),
  Value.Type( Kontrakt )
)
```

Powyższe wywołanie funkcji `Table.AddColumn` zawiera teraz czwarty argument, który dynamicznie pobiera typy danych z tabeli `Kontrakt`. Teraz nowa kolumna nie tylko poprawnie filtruje dane, ale także przenosi ich typ z tabeli `Kontrakt`.

Ta logika sprawia, że powiązanie tabel transakcji i kontraktów spełnia już wszystkie nasze wymagania. Kolejnym etapem prac będzie zatem przekształcenie tego zapytania w funkcję.

Przekształcanie logiki w funkcję

Aby nasz kod nadawał się do wielokrotnego użytku, musimy go przekształcić w funkcję niestandardową. Chcielibyśmy, żeby ta funkcja w możliwie największym stopniu przypominała funkcję standardową `Table.NestedJoin`. Zaczniemy od utworzenia dwóch parametrów określających odpowiednio:

- tabelę, do której chcemy dodać kolumnę (`Tabela`);
- kolumnę, której należy użyć do porównywania danych (`Data`).

Jak już wspominaliśmy wcześniej w tym rozdziale, używając funkcji `Record.Field`, jesteśmy w stanie pobrać wartość tekstową, użyć jej jako danych wejściowych dla kolumny zawierającej datę i nadal odwoływać się do kolumny. Zaimplementowanie tego rozwiązania w kodzie naszego zapytania sprawi, że przyjmie on następującą postać:

```
( Tabela as table, Data as text ) as table =>
  Table.AddColumn(
  Tabela,
  "Kontrakty",
  (x)=> Table.SelectRows( Kontrakt,
    each Record.Field(x, Data) >= [Początek]
      and Record.Field(x, Data) <= [Koniec] ),
  Value.Type( Kontrakt )
)
```

Choć ten kod jest nieco bardziej rozbudowany, to jednak prawidłowo odwołuje się on do pola daty z tabeli transakcji. Naszym kolejnym krokiem będzie dodanie parametrów do określenia:

- dołączanej tabeli (`dołączanaTabela`),
- nazwy nowej kolumny połączenia (`nazwaNowejKolumny`),
- daty początku i końca kontraktu, których należy użyć podczas łączenia tabel (`dataPoczątku` i `dataKońca`).

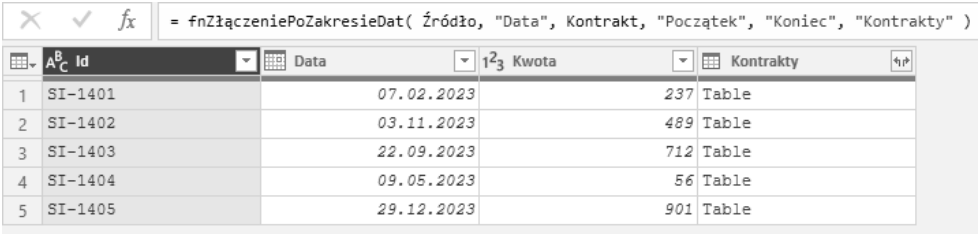
W tym celu ponownie rozszerzymy listę parametrów funkcji i dodamy do niej parametry opisane na powyższej liście:

```
( Tabela as table, Data as text, dołączanaTabela as table, dataPoczątku as
↪text, dataKońca as text, nazwaNowejKolumny as text ) as table =>
  Table.AddColumn(
  Tabela,
  nazwaNowejKolumny,
  (x)=> Table.SelectRows( dołączanaTabela, each
    Record.Field(x, Data) >= Record.Field(_, dataPoczątku) and
```

```
      Record.Field(x, Data) <= Record.Field(_, dataKońca) ),
    Value.Type( Kontrakt )
  )
```

Jak widać, w tym zapytaniu zastąpiliśmy nazwę tabeli `Kontrakt` parametrem `dołączanaTabela`, a podaną na stałe nazwę kolumny zastąpiliśmy parametrem `nazwaNowejKolumny`. Oprócz tego zastosowaliśmy parametry `dataPoczątku` i `dataKońca` oraz funkcję `Record.Field`, aby odwoływać się do odpowiednich kolumn.

Teraz dysponujemy już kompletną funkcją niestandardową, `fnZłączeniePoZakresie↪Dat`. Kiedy zastosujemy ją na tabeli `Transakcje`, uzyskamy wyniki przedstawione na rysunku 9.39.



= fnZłączeniePoZakresieDat(Źródło, "Data", Kontrakt, "Początek", "Koniec", "Kontrakty")

	Id	Data	Kwota	Kontrakty
1	SI-1401	07.02.2023	237	Table
2	SI-1402	03.11.2023	489	Table
3	SI-1403	22.09.2023	712	Table
4	SI-1404	09.05.2023	56	Table
5	SI-1405	29.12.2023	901	Table

Rysunek 9.39. Funkcja niestandardowa do łączenia tabel na podstawie zakresu dat

Ostatnią rzeczą, jaka nam pozostaje do zrobienia, jest rozszerzenie kolumny `Kontrakty`, tak by jej elementy były dostępne w danym wierszu wraz z trzema polami z tabeli `Transakcje`. Po wykonaniu tej operacji uzyskamy tabelę przedstawioną na rysunku 9.40.

	Id	Data	Kwota	IDKontraktu	Początek	Koniec
1	SI-1401	07.02.2023	237	VI-2023-A1	01.01.2023	31.03.2023
2	SI-1402	03.11.2023	489	VI-2023-D4	01.09.2023	31.12.2023
3	SI-1403	22.09.2023	712	VI-2023-D4	01.09.2023	31.12.2023
4	SI-1404	09.05.2023	56	VI-2023-B2	01.04.2023	30.06.2023
5	SI-1405	29.12.2023	901	VI-2023-D4	01.09.2023	31.12.2023

Rysunek 9.40. Rozszerzenie pól po wykonaniu złączenia tabel na podstawie zakresów data zapisanych w określonych polach

I to wszystko — uzyskaliśmy zamierzony efekt. Od teraz zawsze, gdy będziesz musiał połączyć tabele na podstawie zakresów dat, będziesz mógł zastosować przedstawioną tu uproszczoną funkcję, zamiast tworzyć złożone formuły.

Jak mogłeś się przekonać podczas analizy tych dwóch ostatnich projektów, przekształcenie zapytania w niezawodną funkcję niestandardową jest procesem iteracyjnym. Zaczynamy od przygotowania logiki zapytania. Kiedy to już zrobimy, musimy zastanowić się, które elementy tego zapytania powinny być dynamiczne. Kolejnym krokiem jest utworzenie parametrów i zastosowanie ich w kodzie zapytania. Początkowo proces ten może być nieco przytłaczający, jednak kiedy nabierzesz już więcej doświadczenia, stanie się potężnym rozwiązaniem, które rozszerzy Twoje możliwości stosowania zapytań.

Podsumowanie

Pierwszym zagadnieniem przedstawionym w tym rozdziale były parametry. Jak zapewne zauważyłeś, parametry w Power Query są czymś więcej niż tylko symbolami zastępczymi. Zapewniają one elastyczność i pozwalają obsługiwać różne scenariusze. W miarę zdobywania doświadczeń w korzystaniu z Power Query będziesz coraz częściej sięgać po parametry, aby ułatwić sobie dostosowywanie zapytań.

Następnym zagadnieniem, którym zajęliśmy się w tym rozdziale, były funkcje niestandardowe. Pozwalają one przekształcać zapytania w logikę nadającą się do wielokrotnego użytku. W tym rozdziale dowiedziałeś się, jak można przekształcać zapytania w funkcje bez konieczności samodzielnego pisania ich kodu. Następnie opisaliśmy tajniki wywoływania funkcji, ze szczególnym uwzględnieniem wyrażenia `each`, i wyjaśniliśmy, jak upraszcza ono tworzenie funkcji niestandardowych.

Omówiliśmy także różnice pomiędzy składnią wywołań funkcji jedno- oraz wieloargumentowych. W bardziej skomplikowanych sytuacjach, w których mogą pojawiać się problemy związane z zasięgiem, stosowanie słowa kluczowego `each` może być niewystarczające. W takich przypadkach konieczne jest tworzenie własnych funkcji niestandardowych. Oprócz tego w rozdziale wróciliśmy do zagadnień związanych z typami danych oraz przedstawiliśmy parametry opcjonalne, które zapewniają elastyczność naszym funkcjom. Omówiliśmy także sposoby debugowania funkcji niestandardowych, z których można skorzystać w przypadku, gdy pojawią się w nich jakieś błędy lub problemy.

Dzięki tej nowej wiedzy z zakresu tworzenia i stosowania funkcji niestandardowych zyskałeś możliwość ponownego stosowania logiki w wielu zapytaniach przy jednoczesnym zachowaniu porządku w kodzie. Z kolei Twoje funkcje niestandardowe ułatwią mniej doświadczonym współpracownikom wykonywanie złożonych przekształceń.

W następnym rozdziale zajmiemy się kolejnym ważnym tematem. Dowiesz się, jak pracować z datami, czasami i czasami trwania. Te typy wartości nie tylko często pojawiają się w importowanych danych — są one niezbędne w dobrych tabelach kalendarza, których wymaga praktycznie każdy model danych Power BI. Poznanie sposobów pracy z wartościami tych typów jest zatem nieocenione i okaże się przydatne w wielu scenariuszach.

Skorowidz

E

F

R

S

Ś

T

U

V

W

X

Z

ź

Rozpocznij fascynującą podróż do świata zaawansowanej analizy danych!

Profesjonalna analiza danych ułatwia podejmowanie opłacalnych decyzji, a przekształcanie danych jest krytycznym etapem tego procesu. Nieocenionym narzędziem dla każdego, kto chce przetwarzać dane na profesjonalnym poziomie, jest Power Query. Zdobycie wysoce zaawansowanych umiejętności wymaga dogłębnego poznania M — języka zaimplementowanego w Power Query.

Dzięki temu praktycznemu przewodnikowi w pierwszej kolejności opanujesz podstawy języka M. Po zapoznaniu się z kluczowymi koncepcjami stopniowo będziesz stosować tę wiedzę do coraz bardziej zaawansowanych przekształceń danych. Zajmiesz się także takimi zagadnieniami jak optymalizacja wydajności zapytań, obsługa błędów i implementacja efektywnych technik przetwarzania danych. W każdym rozdziale znajdziesz wiele przydatnych przykładów, za sprawą których rozwiniesz umiejętności pozwalające używać Power Query do radzenia sobie z realnymi wyzwaniami i poprawiania swoich umiejętności analitycznych. Przekonasz się, że dzięki zdobytym w ten sposób umiejętnościom z łatwością zastosujesz funkcje Power Query do przekształcania danych!

W książce:

- solidne podstawy Power Query i M
- funkcje służące do wykonywania złożonych przekształceń danych
- wartości, typy i struktury sterujące w Power Query
- wydajna obsługa błędów
- strategie optymalizacji wydajności w Power Query i M
- przykłady rzeczywistych scenariuszy

Greg Deckler posiada kilka tytułów Microsoft Data Platform MVP. Jest ekspertem i autorem licznych rozwiązań dla Power Query.
Rick de Groot jest konsultantem do spraw Power Query, blogerem, youtuberem i trenerem. Przez dwa lata z rzędu był Microsoft Data Platform MVP.
Melissa de Korte jest pasjonatką rozwiązywania problemów, ekspertką i autorką treści edukacyjnych ułatwiających poznawanie Power Query i języka M.

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-2044-6
9 788328 920446
Cena: 169,00 zł
packt