Do tego czasu osoba odpowiedzialna za zmiany w bazie danych sprawiającaj, że imię jest opcjonalne, może pracować w innym zespole niż osoba utrzymująca kod, który wysyła biuletyn. Kod może znajdować się w różnych repozytoriach. Krótko mówiąc, znalezienie wszystkich użyć `Name` może nie być proste.

Dlatego lepiej jest zatem jawnie wskazać, że `Name` jest teraz opcjonalne. Klasa `Subscriber` powinna zostać zmodyfikowana tak:

```
public class Subscriber
{
   public Option<string> Name { get; set; }
   public string Email { get; set; }
}
```

Name jest teraz jawnie oznaczone jako opcjonalne

Taki zapis nie tylko jasno przekazuje fakt, że wartość dla `Name` może nie być dostępna, lecz również sprawia, że `GreetingFor` nie będzie się już kompilować. `GreetingFor` i każdy inny kod, który miał dostęp do własności `Name`, będzie musiał być zmodyfikowany tak, aby uwzględniał możliwość braku wartości. Na przykład możemy zmodyfikować go tak:

```
public string GreetingFor(Subscriber subscriber)
   => subscriber.Name.Match(
      () => "Dear Subscriber, ",
      (name) => $"Dear {name.ToUpper()},");
```

Dzięki wykorzystaniu `Option` zmuszamy użytkowników naszego API do obsługi przypadku, w którym brak jest danych. To nakłada większe wymagania na kod klienta, ale skutecznie usuwa możliwość, że pojawi się wyjątek `NullReferenceException`. Zmiana `string` na `Option<string>` to radykalna zmiana: w ten sposób zamieniamy błędy wykonania na błędy kompilacji, co sprawia, że kompilowanie aplikacji jest bardziej niezawodne.

3.4.5 Option jako naturalny typ wyniku funkcji częściowych

Omawialiśmy, jak funkcje odwzorowują element z jednego zbioru na drugi i jak typy w językach programowania opartych na typach opisują takie zbiory. Ważne jest rozróżnienie między funkcjami *całościowymi* a *częściowymi*:

- *Funkcje całościowe* to odwzorowania zdefiniowane dla *każdego* elementu dziedziny.
- *Funkcje częściowe* to odwzorowania zdefiniowane dla *niektórych*, ale nie wszystkich, elementów dziedziny.

Funkcje częściowe sprawiają kłopoty, gdyż nie jest jasne, co powinny robić po podaniu na wejściu wartości, dla której nie mogą wyznaczyć wyniku. Typ `Option` oferuje idealne rozwiązanie modelowania takich przypadków: jeśli funkcja jest dla danego wejścia zdefiniowana, zwraca `Some` opakowujące wynik. W przeciwnym przypadku zwraca `None`.

Przyjrzyjmy się niektórym przypadkom zastosowań, w których możemy wykorzystać to podejście.

ANALIZA SKŁADNIOWA ŁAŃCUCHÓW

Wyobraźmy sobie funkcję, która analizuje łańcuchową reprezentację liczby całkowitej. Możemy modelować to jako funkcję typu `string → int`. Jest to oczywiście funkcja częściowa. Gdyż nie wszystkie łańcuchy tekstowe reprezentują liczby całkowite. W rzeczywistości jest nieskończenie wiele łańcuchów, których nie da się odwzorować na `int`.

Możemy zapewnić bezpieczniejszą reprezentację analizy za pomocą `Option` dzięki temu, że funkcja analizująca będzie zwracała `Option<int>`. Funkcja zwróci `None`, jeśli danego łańcucha `string` nie da się zanalizować, jak to pokazano na rysunku 3.5.

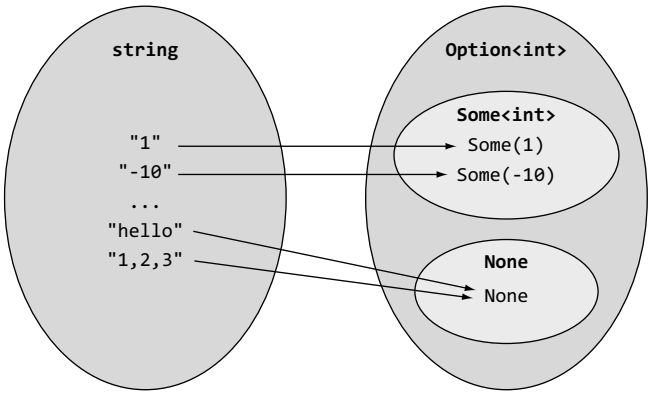
Funkcja zamiany o sygnaturze `string → int` jest częściowa i na podstawie sygnatury nie jest jasne, co się stanie, jeśli dostarczony łańcuch `string` nie będzie mógł być przekształcony na `int`. Z drugiej strony funkcja analizy składniowej o sygnaturze `string → Option<int>` jest całościowa, gdyż dla dowolnego łańcucha da poprawny wynik `Option<int>`.

Oto implementacja wykorzystująca metody tego schematu do wykonania podstawowej pracy, udostępniając API oparte na `Option`:

```
public static class Int
{
   public static Option<int> Parse(string s)
   {
      int result;
      return int.TryParse(s, out result)
         ? Some(result) : None;
   }
}
```

Funkcje pomocnicze w tym podpunkcie znajdują się w `LaYumba.Functional`, możemy więc wypróbować je w REPL:

```
Int.Parse("10") // => Some(10)
Int.Parse("hello") // => None
```



Rysunek 3.5 Przekształcenie łańcucha `string` na liczbę całkowitą `int` to funkcja częściowa