27

Kompilator Roslyn

Kompilator C# także jest napisany w C# i ma postać zbioru modułowych bibliotek o nazwie Roslyn. Odwołując się do nich, można korzystać z różnych elementów funkcjonalności kompilatora, a nie ograniczać się jedynie do kompilacji kodu źródłowego do postaci zestawu. Można np. tworzyć narzędzia do statycznej analizy kodu i refaktoryzacji, edytory z podświetlaniem składni i uzupełnianiem kodu oraz wtyczki dla Visual Studio, które obsługują kod C#.

Biblioteki Roslyn można pobrać za pomocą menedżera NuGet — dostępne są pakiety zarówno dla C#, jak i VB. Oba wymienione języki współdzielą pewne aspekty architektury, więc charakteryzują się wspólnymi zależnościami. Identyfikator pakietu NuGet dla bibliotek kompilatora C# to `Microsoft.CodeAnalysis.CSharp`.

Strona Roslyn w portalu GitHub (*https://github.com/dotnet/roslyn*) zawiera także dokumentację, przykłady oraz różne inne dokumenty przedstawiające analizę kodu i refaktoryzację.

Architektura Roslyn

Architektura Roslyn powoduje podział kompilacji na trzy fazy:

1. Przetworzenie kodu na postać drzew składni (warstwa *syntaktyczna*).
2. Połączenie identyfikatorów z symbolami (warstwa *semantyczna*).
3. Wygenerowanie kodu IL.

W pierwszej fazie *analizator składni* odczytuje kod C# i na jego podstawie przygotowuje *drzewa składni*. To drzewo składni jest modelem DOM (ang. *Document Object Model*) opisującym kod źródłowy w strukturze drzewa.

W drugiej fazie zachodzi *statyczne wiązanie* C#. Odczytywane są odwołania do zestawów i kompilator ustala np., że `Console` odwołuje się do `System.Console` w *System.Console.dll*. Ustalanie typów i obsługa przeciążenia również zachodzą w tej fazie.

W trzeciej fazie następuje wygenerowanie danych wyjściowych w postaci zestawu. Jeżeli planujemy użycie Roslyn do analizy kodu lub refaktoryzacji, nie będziemy korzystać z tej funkcjonalności.

Edytor w Visual Studio wykorzystuje dane wyjściowe warstwy syntaktycznej do kolorowania słów kluczowych, ciągów tekstowych, komentarzy i wyłączonego kodu (za pomocą kolorów odpowiednio: niebieskiego, czerwonego, zielonego i szarego). Natomiast dane wyjściowe warstwy semantycznej wykorzystuje do kolorowania ustalonych nazw typów (w kolorze turkusowym).

Przestrzenie robocze

W tym rozdziale przedstawimy kompilator i oferowane przez niego funkcje. Warto pamiętać, że nad kompilatorem znajdują się jeszcze dodatkowe warstwy, w tym **przestrzenie robocze** i **funkcje**.

Warstwa przestrzeni roboczych, zawierająca API do pracy z rozwiązaniami, projektami i dokumentami, znajduje się w pakiecie NuGet *Microsoft.CodeAnalysis.CSharp.Workspaces*.

Warstwa funkcji znajduje się w pakiecie *Microsoft.CodeAnalysis.CSharp.Features* i zawiera wiele API do analizy kodu i refaktoryzacji.

Skrypty

Za pomocą pakietu NuGet *Microsoft.CodeAnalysis.CSharp.Scripting* można pisać kod podobny do poniższego:

```
int result = (int) await CSharpScript.EvaluateAsync ("1 + 2");
```

API skryptów kompiluje „1 + 2” do postaci programu, który następnie wykonuje, więc charakteryzuje się mniejszą wydajnością niż rozwiązanie, które opisaliśmy w rozdziale 19. (zobacz punkt „Współpraca z językami dynamicznymi”). Na stronie *https://github.com/dotnet/roslyn/wiki/Scripting-API-Samples* znajduje się więcej przykładów skryptów API Roslyn.

Drzewa składni

Drzewo składni jest modelem DOM dla kodu źródłowego. API drzewa składni jest zupełnie odmienne od API `System.Linq.Expressions` omówionego w rozdziale 8., choć pod względem koncepcyjnym oba rodzaje API są podobne. Wymienione API mogą przedstawiać wyrażenia C# w modelu DOM, jednak drzewo składni Roslyn charakteryzuje się następującymi unikatowymi cechami:

- możliwość przedstawienia całego języka C#, a nie tylko wyrażeń;
- możliwość uwzględnienia komentarzy, białych znaków oraz innych „drobiazgów”, a także zachowania pełnej wierności podczas powrotu do pierwotnego kodu źródłowego;
- dostępność metody `ParseText()` przeznaczonej do przetwarzania kodu źródłowego w drzewie składni.

Z kolei API `System.Linq.Expressions` charakteryzuje się następującymi unikatowymi cechami:

- Jest wbudowane w platformę .NET Core, a kompilator C# sam jest zaprogramowany do emisji typów `System.Linq.Expression` po napotkaniu wyrażenia lambda wraz z przypisaną konwersją na `Expression<T>`.
- Ma szybką i lekką metodę `Compile()`, która emituje delegat. Dla porównania: semantyczna warstwa kompilująca drzewa składni Roslyn oferuje jedynie cięższą opcję kompilacji całego programu do postaci zestawu.

Cechą łączącą oba omawiane rodzaje API jest niemodyfikowalność drzew składni, więc po utworzeniu żaden z elementów drzewa nie może być zmieniony. Oznacza to, że aplikacje takie jak Visual Studio i LINQPad muszą tworzyć zupełnie nowe drzewo składni w trakcie każdego naciśnięcia klawisza w edytorze, aby móc uaktualnić podświetlanie składni oraz zapewnić obsługę usług automatycznego uzupełniania kodu. Taka operacja jest mniej kosztowna niż się wydaje, ponieważ nowe drzewo składni może wykorzystać większość elementów starego (zob. sekcję „Transformacja drzewa składni" w dalszej części rozdziału). Skoro wiadomo, że obiekt nie może ulec zmianie, praca z API jest prostsza. Ponadto łatwiejsza i szybsza staje się współbieżność, ponieważ wielowątkowy kod może bezpiecznie uzyskać dostęp do wszystkich elementów drzewa składni bez konieczności nakładania blokad.

Struktura SyntaxTree

Struktura `SyntaxTree` składa się z trzech podstawowych elementów:

Węzły

(Klasa abstrakcyjna `SyntaxNode`). Przedstawia konstrukcje C# takie jak: wyrażenia, polecenia i deklaracje metod. Węzeł zawsze ma przynajmniej jeden element potomny, więc nigdy nie może być liściem na drzewie. Elementami potomnymi węzła mogą być zarówno węzły, jak i tokeny.

Tokeny

(Struktura `SyntaxToken`). Przedstawia identyfikatory, słowa kluczowe, operatory i znaki przestankowe tworzące kod źródłowy. Jedynym elementem potomnym, jaki może mieć token, są opcjonalne drobiazgi na początku i na końcu. Elementem nadrzędnym tokena zawsze będzie węzeł.

Drobiazgi

(Struktura `SyntaxTrivia`). Drobiazgi są związane z białymi znakami, komentarzami, dyrektywami preprocesora oraz kodem nieaktywnym ze względu na kompilację warunkową. Drobiazgi zawsze są powiązane z tokenem znajdującym się po lewej lub prawej stronie i wyrażone za pomocą właściwości odpowiednio `TrailingTrivia` i `LeadingTrivia` tokena.

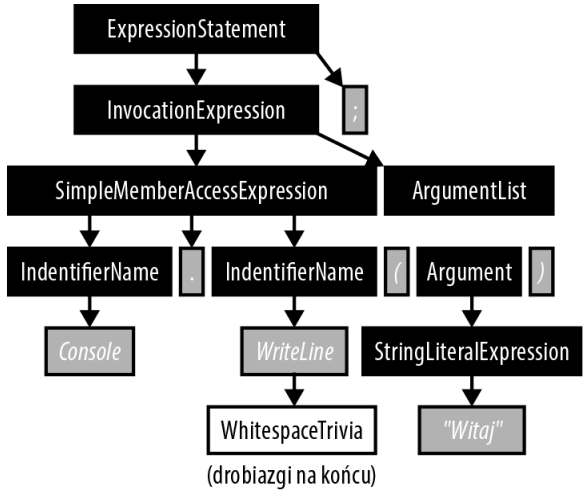
Na rysunku 27.1 pokazano strukturę poniższego wiersza kodu (węzły są oznaczone kolorem czarnym, tokeny szarym, a drobiazgi białym):

```
Console.WriteLine("Witaj")
```

Klasa `SyntaxNode` jest klasą abstrakcyjną, która ma charakterystyczne dla C# podklasy dla każdego rodzaju elementu syntaktycznego, np. `VariablleDeclarationSyntax` lub `TryStatementSyntax`.

`SyntaxToken` i `SyntaxTrivia` to struktury, a pojedynczy typ przedstawia każdy rodzaj tokena lub drobiazgu. W celu odróżniania poszczególnych rodzajów tokenów lub drobiazgów konieczne jest użycie właściwości `RawKind` lub rozszerzenia metody `Kind()`, czym zajmiemy się w kolejnej sekcji.

Najlepszym sposobem poznania drzewa składni jest wykorzystanie wizualizacji. Dla oprogramowania Visual Studio jest dostępny do pobrania komponent wizualizacji przeznaczony do użycia wraz z debuggerem. Z kolei LINQPad ma już wbudowany tego rodzaju komponent. Ponadto LINQPad wyświetla wizualizację automatycznie dla kodu w edytorze tekstu po kliknięciu przycisku *Tree* w oknie danych wyjściowych. Istnieje również możliwość wyświetlenia przez LINQPad wizualizacji dla drzewa składni utworzonego w sposób programowy przez wywołanie `DumpSyntaxTree()` w drzewie (lub `DumpSyntaxNode()` w węźle).



Rysunek 27.1. Drzewa składni

Poznajemy typy węzłów

Podklasy klasy `SyntaxtNode` zostały zaprojektowane w celu odzwierciedlenia syntaktycznego przetwarzania i pozostają niewrażliwe na typ semantyczny oraz na informacje o symbolach pobierane z zachodzącej później operacji wiązania. Na przykład rozważmy wynik przetworzenia przedstawionego poniżej kodu:

```
using System;

class Foo : PewnaKlasaBazowa
{
  void Test() { Console.WriteLine(); }
}
```

Być może oczekujesz, że wywołanie `Console.WriteLine()` będzie przedstawione przez klasę o nazwie `MethodCallExpressionSyntax`, ale tego rodzaju klasa nie istnieje. Zamiast tego jest przedstawiane przez klasę `InvocationExpressSyntax`, w której została zdefiniowana metoda `SimpleMemberAccessExpression()`. Wynika to z faktu ignorowania typów przez analizator składni, który tym samym nie wie, że `Console` to typ, a `WriteLine()` to metoda. Istnieje wiele innych możliwości, np. `Console` mogłoby być właściwością `PewnejKlasyBazowej`, natomiast `WriteLine` mogłoby być zdarzeniem, elementem składowym lub właściwością typu delegatu. Na podstawie składni wiemy jedynie, że otrzymujemy dostęp do elementu składowego (*identyfikator.identyfikator*), a następnie pewnego rodzaju *wywołanie* bez jakiegokolwiek argumentu.

Najczęściej używane właściwości i metody

Węzły, tokeny i drobiazgi mają wiele ważnych najczęściej używanych właściwości oraz metod:

Właściwość `SyntaxTree`

Zwraca drzewo składni, do którego należy dany obiekt.

Właściwość `Span`

Zwraca położenie obiektu w kodzie źródłowym (zob. sekcję „Wyszukiwanie elementu potomnego na podstawie jego położenia" w dalszej części rozdziału).

Rozszerzenie metody `Kind()`

Zwraca typ wyliczeniowy `SyntaxKind` klasyfikujący węzeł, token lub drobiazg na jedną z setek wartości, np.: `IntKeyword`, `CommaToken`, `WhitespaceTrivia`. Ten sam typ wyliczeniowy `SyntaxKind` zapewnia obsługę węzłów, tokenów i drobiazgów.

Metoda `ToString()`

Zwraca tekst (kod źródłowy) dla węzła, tokena lub drobiazgu. W przypadku tokena odpowiednikiem tej metody jest właściwość `Text`.

Metoda `GetDiagnostics()`

Zwraca błędy lub ostrzeżenia wygenerowane podczas analizy składni.

Metoda `IsEquivalentTo()`

Zwraca wartość `true`, jeśli obiekt jest identyczny z innym egzemplarzem węzła, tokena lub drobiazgu. Różnice w zakresie białych znaków są ważne (w celu ignorowania białych znaków należy przed operacją porównania wywołać `NormalizeWhitespace()`).

Węzły i tokeny mają również właściwość `FullSpan` i metodę `ToFullString()`. W trakcie ich działania drobiazgi są brane pod uwagę, natomiast właściwość `Span` i metoda `ToString()` nie uwzględniają drobiazgów.

Rozszerzenie metody `Kind()` jest skrótem dla rzutowania właściwości `RawKind` typu `int` na typ `Microsoft.CodeAnalysis.CSharp.SyntaxKind`. Powodem, dla którego nie ma właściwości `Kind` typu `SyntaxKind`, jest fakt, że typy drobiazgów i tokeny są używane również w drzewach składni VB mających inny typ wyliczeniowy dla `SyntaxKind`.

Uzyskanie drzewa składni

Metoda statyczna `ParseText()` w `CSharpSyntaxTree` przetwarza kod C# na postać `SyntaxTree`:

```
SyntaxTree tree = CSharpSyntaxTree.ParseText (@"class Test
{
  static void Main() => Console.WriteLine (""Witaj"");
}");

Console.WriteLine (tree.ToString());

tree.DumpSyntaxTree();      // wyświetla wizualizację drzewa składni w LINQPad
```

Aby uruchomić ten kod w projekcie Visual Studio, należy zainstalować pakiet NuGet *Microsoft.*
↪CodeAnalysis.CSharp oraz zaimportować wymienione poniżej przestrzenie nazw:

```
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
```

Opcjonalnie można przekazać obiekt `CSharpParseOptions` w celu podania wersji języka C#, symboli preprocesora oraz `DocumentationMode` do wskazania, czy komentarze XML powinny być

przetwarzane (zob. sekcję „Strukturalne drobiazgi” w dalszej części rozdziału). Istnieje nawet możliwość podania `SourceCodeKind`. Wybór `Script` nakazuje analizatorowi składni akceptację pojedynczego wyrażenia lub polecenia (bądź poleceń) zamiast wymagania całego programu (obsługiwane od Roslyn 2).

Innym sposobem uzyskania drzewa składni jest wywołanie `CSharpSyntaxTree.Create()` i przekazanie obiektu wykresu węzłów i tokenów. Sposób tworzenia wspomnianych obiektów będzie przedstawiony w sekcji „Transformacja drzewa składni” w dalszej części rozdziału.

Po przeanalizowaniu drzewa ewentualne błędy i ostrzeżenia można pobrać za pomocą wywołania `GetDiagnostics()`. (Tę metodę można wywołać także dla określonego węzła lub tokena).

Jeżeli analiza składni zakończy się wygenerowaniem nieoczekiwanych błędów, struktura drzewa może być inna od oczekiwanej. Z tego powodu warto wywołać `GetDiagnostics()` przed przejściem dalej.

Użyteczną cechą jest to, że drzewo wraz z błędami zostanie przywrócone do pierwotnego tekstu (wraz z tymi samymi błędami). W takich przypadkach analizator składni stara się dostarczyć drzewo składni, które będzie jak najbardziej przydatne w warstwie semantycznej, tworząc przy okazji „fikcyjne węzły”, jeśli zajdzie potrzeba. Dzięki temu narzędzia takie jak przeznaczone do uzupełniania kodu będą mogły działać wraz z niekompletnym kodem. (Za pomocą właściwości `IsMissing` można sprawdzić, czy węzeł jest fikcyjny).

Wywołanie `GetDiagnostics()` w drzewie składni utworzonym w poprzedniej sekcji nie wskazuje na żadne błędy, mimo wywołania `Console.WriteLine()` bez zaimportowania przestrzeni nazw. Jest to dobry przykład porównania analizy syntaktycznej i semantycznej. Pod względem syntaktycznym program jest prawidłowy, a błąd nie zostanie ujawniony aż do chwili kompilacji, dodania odwołań zestawu i sprawdzenia *modelu semantycznego*, gdzie zachodzi wiązanie.

Poruszanie się po drzewie i jego przeszukiwanie

Klasa `SyntaxTree` działa w charakterze opakowania dla struktury drzewa. Zawiera odwołanie do jednego węzła głównego, który można otrzymać za pomocą wywołania `GetRoot()`:

```
var tree = CSharpSyntaxTree.ParseText (@"class Test
{
  static void Main() => Console.WriteLine (""Witaj"");
}");

SyntaxNode root = tree.GetRoot();
```

Węzłem głównym w programie C# jest `CompilationUnitSyntax`:

```
Console.WriteLine (root.GetType().Name);    // CompilationUnitSyntax
```

Poruszanie się po elementach potomnych

Klasa `SyntaxNode` udostępnia przyjazne dla LINQ metody przeznaczone do poruszania się po węzłach potomnych oraz tokenach. Poniżej wymieniono najprostsze z tych metod:

```
IEnumerable<SyntaxNode> ChildNodes()
IEnumerable<SyntaxToken> ChildTokens()
```

Kontynuujemy pracę z wcześniejszym przykładem. Węzeł główny ma jeden węzeł potomny typu `ClassDeclarationSyntax`:

```
var cds = (ClassDeclarationSyntax) root.ChildNodes().Single();
```

Istnieje możliwość wyświetlenia elementów składowych `cds` za pomocą metody `ChildNodes()` lub właściwości `Members` obiektu `ClassDeclarationSyntax`:

```
foreach (MemberDeclarationSyntax member in cds.Members)
  Console.WriteLine (member.ToString());
```

Oto otrzymany wynik:

```
static void Main() => Console.WriteLine (""Witaj"");
```

Dostępne są również metody `Descendant*()` pozwalające na rekurencyjne poruszanie się po elementach potomnych. Za pomocą poniższego fragmentu kodu możemy wymienić tokeny wykorzystane w programie:

```
foreach (var token in root.DescendantTokens())
  Console.WriteLine ($"{token.Kind(),-30} {token.Text}");
```

Powyższy kod powoduje wygenerowanie następujących danych wyjściowych:

```
ClassKeyword                   class
IdentifierToken                Test
OpenBraceToken                 {
StaticKeyword                  static
VoidKeyword                    void
IdentifierToken                Main
OpenParenToken                 (
CloseParenToken                )
EqualsGreaterThanToken         =>
IdentifierToken                Console
DotToken                       .
IdentifierToken                WriteLine
OpenParenToken                 (
StringLiteralToken             "Witaj"
CloseParenToken                )
SemicolonToken                 ;
CloseBraceToken                }
EndOfFileToken
```

Zwróć uwagę na brak w wygenerowanym wyniku białych znaków. Podanie białych znaków (oraz wszelkich pozostałych drobiazgów) można uzyskać dzięki zastąpieniu `token.Text` przez `token.ToFullString()`.

W poniższym fragmencie kodu wykorzystujemy metodę `DescendantNodes()` do odszukania węzła składni dla podanej deklaracji metody:

```
var ourMethod = root.DescendantNodes()
                    .First (m => m.Kind() == SyntaxKind.MethodDeclaration);
```

Poniżej przedstawiono podejście alternatywne:

```
var ourMethod = root.DescendantNodes()
                    .OfType<MethodDeclarationSyntax>()
                    .Single();
```

W tym drugim podejściu egzemplarz ourMethod jest typu MethodDeclarationSyntax i udostępnia użyteczne właściwości charakterystyczne dla deklaracji metod. Na przykład jeśli kod zawiera więcej niż tylko jedną definicję metody i chcemy znaleźć jedynie metodę o nazwie Main(), wówczas możemy użyć następującego rozwiązania:

```
var mainMethod = root.DescendantNodes()
                     .OfType<MethodDeclarationSyntax>()
                     .Single (m => m.Identifier.Text == "Main");
```

W powyższym kodzie Identifier to właściwość w MethodDeclarationSyntax zwracająca token odpowiadający identyfikatorowi metody (np. jej nazwę). Ten sam efekt można uzyskać większym nakładem pracy, jak w poniższym fragmencie kodu:

```
root.DescendantNodes().First (m =>
  m.Kind() == SyntaxKind.MethodDeclaration &&
  m.ChildTokens().Any (t =>
    t.Kind() == SyntaxKind.IdentifierToken && t.Text == "Main"));
```

Klasa SyntaxNode ma również metody GetFirstToken() i GetLastToken() będące odpowiednikami wywołań DescendantTokens.First() i DescendantTokens.Last().

Metoda GetLastToken() działa szybciej niż DescendantTokens.Last(), ponieważ zwraca bezpośrednie łącze zamiast przeprowadzać operację przejścia przez wszystkie elementy potomne.

Ponieważ węzły mogą zawierać zarówno węzły potomne, jak i tokeny, których względna kolejność jest ważna, dostępne są także metody pozwalające na jednoczesne podanie węzłów potomnych i tokenów:

```
ChildSyntaxList ChildNodesAndTokens()
IEnumerable<SyntaxNodeOrToken> DescendantNodesAndTokens()
IEnumerable<SyntaxNodeOrToken> DescendantNodesAndTokensAndSelf()
```

(ChildSyntaxList implementuje IEnumerable<SyntaxNodeOrToken> i jednocześnie udostępnia właściwość Count oraz indeks umożliwiający uzyskanie dostępu do elementu na podstawie jego położenia).

Po drobiazgach można poruszać się bezpośrednio z węzła za pomocą metod: GetLeadingTrivia(), GetTrailingTrivia() i DescendantTrivia(). Jednak znacznie częściej dostęp do drobiazgu będziemy uzyskiwali za pomocą tokena, do którego został dołączony, co wymaga użycia właściwości LeadingTrivia i TrailingTrivia tego tokena. W celu przeprowadzenia konwersji na postać tekstu należy skorzystać z metody ToFullString(), która uwzględnia drobiazg w wygenerowanych danych wyjściowych.

Poruszanie się po elementach nadrzędnych

Węzły i tokeny mają właściwość Parent typu SyntaxNode.

W przypadku SyntaxTrivia „elementem nadrzędnym" jest token dostępny za pomocą właściwości Token.

Węzły mają również metody pozwalające na poruszanie się w górę drzewa. Nazwy tych metod zawierają prefiks Ancestor.

Wyszukiwanie elementu potomnego na podstawie jego położenia

Wszystkie węzły, tokeny i drobiazgi mają właściwość `Span` typu `TextSpan` pozwalającą wskazać położenie początkowe i końcowe w kodzie źródłowym. Węzły i tokeny mają ponadto właściwość o nazwie `FullSpan`, która zawiera drobiazgi na początku i na końcu (nieuwzględniane przez właściwość `Span`). Z kolei właściwość `Span` węzła uwzględnia węzły potomne i tokeny.

Praca ze strukturą TextSpan

Struktura `TextSpan` ma właściwości: `Start`, `Length` i `End` wskazujące położenie znaku w kodzie źródłowym. Ponadto ma metody takie jak: `Overlap()`, `OverlapsWith()`, `Intersection()` i `IntersectsWith()`. Różnica między nakładaniem się i intersekcją to kwestia jednego znaku. Dwa obszary *nakładają się*, gdy jeden rozpoczyna się przed końcem drugiego (`<`), natomiast z *intersekcją* między nimi mamy do czynienia, gdy się zaledwie stykają (`<=`).

Klasa `SyntaxTree` udostępnia metodę `GetLineSpan()`, która konwertuje strukturę `TextSpan` na wartość wskazującą wiersz i miejsce położenia znaku. Ta metoda ignoruje efekt działania wszelkich dyrektyw `#line` umieszczonych w kodzie źródłowym. Istnieje również metoda `GetMappedLineSpan()` uwzględniająca te dyrektywy.

Obiekt potomny można znaleźć na podstawie jego położenia. Do tego celu służą metody: `FindNode()`, `FindToken()` i `FindTrivia()` w klasie `SyntaxNode`. Wymienione metody zwracają obiekt potomny wraz z najmniejszym możliwym obszarem zawierającym wskazany obszar. Istnieje również metoda o nazwie `ChildThatContainsPosition()` odpowiedzialna za wyszukiwanie zarówno węzłów potomnych, jak i tokenów.

Jeżeli wynik wyszukiwania w dwóch węzłach to identyczny obszar (zwykle węzeł potomny i jego węzeł potomny), wówczas metoda `FindNode()` zwróci węzeł zewnętrzny (nadrzędny). To zachowanie można zmienić przez przekazanie wartości `true` opcjonalnemu argumentowi `getInnermostNodeForTie`.

Metody `Find*()` mają również opcjonalny parametr `findInsideTrivia` typu `bool`. Jeżeli przyjmie on wartość `true`, wówczas przeprowadzane będzie wyszukiwanie węzłów lub tokenów wewnątrz *strukturalnego drobiazgu* (zob. sekcję „Drobiazgi” nieco dalej w rozdziale).

Klasa CSharpSyntaxWalker

Innym sposobem poruszania się po drzewie jest tworzenie podklas klasy `CSharpSyntaxWalker`, nadpisując jedną lub więcej z jej setek wirtualnych metod. Przedstawiona poniżej klasa służy do zliczania poleceń `if`:

```
class IfCounter : CSharpSyntaxWalker
{
  public int IfCount { get; private set; }

  public override void VisitIfStatement (IfStatementSyntax node)
  {
    IfCount++;
    // jeżeli chcesz otrzymać węzeł potomny, wywołaj metodę bazową
    base.VisitIfStatement (node);
  }
}
```

Poniżej pokazano przykład jej użycia:

```
var ifCounter = new IfCounter ();
ifCounter.Visit (root);
Console.WriteLine ($"Znaleziono {ifCounter.IfCount} poleceń if.");
```

Wynik jest odpowiednikiem następującego wywołania:

```
root.DescendantNodes().OfType<IfStatementSyntax>().Count()
```

Zastosowanie powyższego rozwiązania może być łatwiejsze niż użycie metod `Descendant*()`, zwłaszcza w bardziej skomplikowanych przypadkach, gdy zachodzi potrzeba nadpisania wielu metod (po części wynika to z faktu, że C# nie ma takich jak F# możliwości w zakresie dopasowania wzorca).

Domyślnie `CSharpSyntaxWalker` uwzględnia jedynie węzły. W celu uwzględnienia tokenów lub drobiazgów konieczne jest wywołanie konstruktora bazowego wraz z właściwością `SyntaxWalkerDepth`, wskazując oczekiwany poziom zagłębienia (węzeł→token→drobiazg). Następnie można już nadpisać metodę `VisitToken()` lub `VisitTrivia()`:

```
class WhiteWalker : CSharpSyntaxWalker    // zlicza spacje
{
  public int SpaceCount { get; private set; }

  public WhiteWalker() : base (SyntaxWalkerDepth.Trivia) { }

  public override void VisitTrivia (SyntaxTrivia trivia)
  {
    SpaceCount += trivia.ToString().Count (char.IsWhiteSpace);
    base.VisitTrivia (trivia);
  }
}
```

Po usunięciu w klasie `WhiteWalker` wywołania konstruktora klasy bazowej metoda `VisitTrivia()` nie będzie wywołana.

Drobiazgi

Drobiazgi to kod, który po przeanalizowaniu może być w całości zignorowany przez kompilator w kategoriach wygenerowania zestawu. Do drobiazgów zaliczamy: białe znaki, komentarze, dokumentację XML, dyrektywy preprocesora oraz kod nieaktywny ze względu na kompilację warunkową.

Obowiązkowe białe znaki w kodzie również są uznawane za drobiazgi. Wprawdzie mają istotne znaczenie podczas analizy składniowej, ale nie są niezbędne (przynajmniej kompilatorowi) po wygenerowaniu drzewa składni. Drobiazgi nadal są ważne podczas przywracania pierwotnej postaci kodu źródłowego.

Drobiazgi należą do tokena, do którego przylegają. Zgodnie z konwencją analizator składni uznaje białe znaki i komentarze znajdujące się po tokenie aż do końca wiersza za drobiazgi końcowe. Wszystko następne jest traktowane jako drobiazgi początkowe kolejnego tokena. (Mamy tutaj pewne wyjątki dotyczące początku i końca pliku). Jeżeli tokeny tworzymy w sposób programowy (zob. sekcję „Transformacja drzewa składni” nieco dalej w rozdziale), wówczas białe znaki możemy umieścić w dowolnym z wymienionych miejsc (lub w ogóle ich nie potrzebujemy, jeśli nie zamierzamy przywracać pierwotnej postaci kodu źródłowego):

```
var tree = CSharpSyntaxTree.ParseText (@"class Program
{
    static /*komentarz*/ void Main() {}
}");

SyntaxNode root = tree.GetRoot();

// wyszukanie tokena statycznego słowa kluczowego
var method = root.DescendantTokens().Single (t =>
  t.Kind() == SyntaxKind.StaticKeyword);

// umieszczenie drobiazgów wokół tokena statycznego słowa kluczowego
foreach (SyntaxTrivia t in method.LeadingTrivia)
  Console.WriteLine (new { Kind = "Leading " + t.Kind(), t.Span.Length });

foreach (SyntaxTrivia t in method.TrailingTrivia)
  Console.WriteLine (new { Kind = "Trailing " + t.Kind(), t.Span.Length });
```

Oto otrzymane dane wyjściowe:

```
{ Kind = Leading WhitespaceTrivia, Length = 1 }
{ Kind = Trailing WhitespaceTrivia, Length = 1 }
{ Kind = Trailing MultiLineCommentTrivia, Length = 11 }
{ Kind = Trailing WhitespaceTrivia, Length = 1 }
```

Dyrektywy preprocesora

Może wydawać się dziwne uznawanie dyrektyw preprocesora za drobiazgi, biorąc pod uwagę fakt, że niektóre z nich (zwłaszcza dotyczące kompilacji warunkowej) mają niemały wpływ na sposób wygenerowania danych wyjściowych.

Jednak takie podejście wynika z tego, że dyrektywy preprocesora są semantycznie przetwarzane przez analizator składni, którego zadaniem jest przecież wstępne przetworzenie. Po tej operacji kompilatorowi nie pozostaje już nic do wyraźnego rozważenia (poza dyrektywą `#pragma`). Aby to zilustrować, poznamy teraz sposób działania analizatora składni po napotkaniu dyrektyw kompilacji warunkowej:

```
#define FOO

#if FOO
    Console.WriteLine ("Zdefiniowano FOO");
#else
    Console.WriteLine ("Nie zdefiniowano FOO");
#endif
```

Po odczytaniu dyrektywy `#if FOO` analizator składni wie o zdefiniowaniu `FOO`, więc kolejny wiersz zostanie przetworzony normalnie (jako węzły i tokeny), podczas gdy wiersz znajdujący się po dyrektywie `#else` będzie przetworzony na postać egzemplarza `DisabledTextTrivia`.

Podczas wywoływania `CSharpSyntaxTree.Parse()` można podać kolejne symbole preprocesora przez utworzenie i przekazanie egzemplarza `CSharpParseOptions`.

Dlatego też w przypadku kompilacji warunkowej egzemplarz `DisabledTextTrivia` zawiera zignorowany tekst, który został uznany za drobiazg (np. nieaktywny kod i same dyrektywy preprocesora).

Dyrektywa `#line` jest obsługiwana ponownie pod tym względem, że analizator składni odczytuje i interpretuje dyrektywę. Zebrane w ten sposób informacje są używane podczas wywoływania `GetMappedLineSpan()` w drzewie składni.

Pod względem semantycznym dyrektywa `#region` jest pusta. Jedyna rola analizatora składni sprowadza się do sprawdzenia, czy liczba dyrektyw `#region` odpowiada liczbie dyrektyw `#endregion`. Dyrektywy `#error` i `#warning` również są przetwarzane przez analizator składni, który generuje błędy i ostrzeżenia możliwe do przejrzenia po wywołaniu `GetDiagnostics()` w drzewie lub węźle.

Nadal użyteczne może być przeglądanie zawartości dyrektyw preprocesora w celu innym niż generowanie danych wyjściowych zestawu (np. dla podświetlania składni). Takie zadanie będzie znacznie łatwiejsze do przeprowadzenia za pomocą *strukturalnych drobiazgów*.

Strukturalne drobiazgi

Mamy dwa rodzaje drobiazgów:

Niestrukturalne drobiazgi

Komentarze, białe znaki i kod nieaktywny ze względu na kompilację warunkową.

Strukturalne drobiazgi

Dyrektywy preprocesora i dokumentacja XML.

Niestrukturalne drobiazgi są traktowane wyłącznie jako tekst, podczas gdy strukturalne mają również własną treść przetwarzaną na postać miniaturowego drzewa składni.

Właściwość `HasStructure` w `SyntaxTrivia` wskazuje, czy istnieje strukturalny drobiazg, a metoda `GetStructure()` zwraca węzeł główny dla miniaturowego drzewa składni:

```
var tree = CSharpSyntaxTree.ParseText (@"#define FOO");

// w programie LINQPad
tree.DumpSyntaxTree();   // strukturalne drobiazgi LINQPad wyświetla za pomocą komponentu Visualizer

SyntaxNode root = tree.GetRoot();

var trivia = root.DescendantTrivia().First();
Console.WriteLine (trivia.HasStructure);            // prawda
Console.WriteLine (trivia.GetStructure().Kind());   // DefineDirectiveTrivia
```

W przypadku dyrektyw preprocesora do strukturalnego drobiazgu można przejść bezpośrednio za pomocą wywołania `GetFirstDirective()` w `SyntaxNode`. Istnieje również właściwość `ContainsDirectives` wskazująca, czy obecny jest drobiazg preprocesora:

```
var tree = CSharpSyntaxTree.ParseText (@"#define FOO");
SyntaxNode root = tree.GetRoot();

Console.WriteLine (root.ContainsDirectives);          // prawda

// dyrektywa jest węzłem głównym strukturalnego drobiazgu
var directive = root.GetFirstDirective();
Console.WriteLine (directive.Kind());                 // DefineDirectiveTrivia
Console.WriteLine (directive.ToString());             // #define FOO

// jeżeli są dostępne kolejne dyrektywy, możemy je otrzymać w poniższy sposób
Console.WriteLine (directive.GetNextDirective());     // (null)
```

Po otrzymaniu węzła drobiazgu można rzutować go na konkretny typ i sprawdzić właściwości, podobnie jak w przypadku dowolnego innego węzła:

```
var hashDefine = (DefineDirectiveTriviaSyntax) root.GetFirstDirective();
Console.WriteLine (hashDefine.Name.Text);      // FOO
```

Wszystkie węzły, tokeny i drobiazgi mają właściwość `IsPartOfStructuredTrivia` wskazującą, czy dany obiekt należy do drzewa strukturalnego drobiazgu (np. element potomny obiektu drobiazgu).

Transformacja drzewa składni

Istnieje możliwość „modyfikacji" węzłów, tokenów i drobiazgów za pomocą metod o wymienionych poniżej prefiksach (większość z nich to rozszerzenia metod):

```
Add*
Insert*
Remove*
Replace*
With*
Without*
```

Ponieważ drzewo składni jest niemodyfikowalne, wszystkie wymienione powyżej metody zwracają nowy obiekt wraz z wprowadzonymi żądanymi modyfikacjami. Pierwotny obiekt pozostaje nienaruszony.

Obsługa zmian kodu źródłowego

Jeżeli tworzymy edytor kodu źródłowego np. C#, wówczas musimy uaktualniać drzewo składni na podstawie zmian wprowadzanych w kodzie źródłowym. Klasa `SyntaxTree` zawiera metodę o nazwie `WithChangedText()` przeznaczoną dokładnie do tego celu — częściowe ponowne przetwarzanie kodu źródłowego na podstawie zmian opisanych w egzemplarzu `SourceText` (z przestrzeni nazw `Microsoft.CodeAnalysis.Text`).

Aby utworzyć egzemplarz `SourceText`, należy użyć jego metody statycznej `From()`, przekazując jej kompletny kod źródłowy. Następnie na podstawie tych danych można przygotować drzewo składni:

```
SourceText sourceText = SourceText.From ("class Program {}");
var tree = CSharpSyntaxTree.ParseText (sourceText);
```

Alternatywne podejście polega na otrzymaniu egzemplarza `SourceText` na podstawie istniejącego drzewa za pomocą wywołania `GetText()`.

W tym momencie można już „uaktualnić" egzemplarz `sourceText` przez wywołanie metody `Replace()` lub `WithChanges()`. Na przykład pierwsze pięć znaków („class") można zastąpić słowem „struct", jak w poniższym fragmencie kodu:

```
var newSource = sourceText.Replace (0, 5, "struct");
```

Na koniec trzeba wywołać `WithChangedText()` w drzewie, aby je uaktualnić:

```
var newTree = tree.WithChangedText (newSource);
Console.WriteLine (newTree.ToString());          // struct Program {}
```

Tworzenie nowych węzłów, tokenów i drobiazgów za pomocą SyntaxFactory

Metody statyczne klasy `SyntaxFactory` pozwalają na programowe tworzenie węzłów, tokenów i drobiazgów, które następnie można wykorzystać do „transformacji" istniejących drzew składni lub przygotować zupełnie od początku nowe drzewa składni.

Najtrudniejszą częścią zadania jest ustalenie dokładnego rodzaju węzła i tokena do utworzenia. Rozwiązanie polega najpierw na przetworzeniu przykładowego fragmentu kodu i przeanalizowaniu wyniku w komponencie wizualizacji składni. Na przykład przyjmujemy założenie, że chcemy utworzyć węzeł składni dla następującego polecenia:

```
using System.Text;
```

Wizualizację drzewa składni można w programie LINQPad przeprowadzić w poniższy sposób:

```
CSharpSyntaxTree.ParseText ("using System.Text;").DumpSyntaxTree();
```

(Możemy przetworzyć `"using System.Text;"` bez błędów, ponieważ to polecenie tworzy prawidłowy program, choć oczywiście pusty pod względem funkcjonalnym. W przypadku większości innych fragmentów kodu konieczne jest opakowanie go definicją typu lub metodą, aby przetworzenie było możliwe).

W wyniku otrzymujemy przedstawioną poniżej strukturę, w której najbardziej interesuje nas drugi węzeł (`UsingDirective` i jego elementy potomne):

```
Kind                               Token Text
================================   ==========
CompilationUnit (node)
  UsingDirective (node)
    UsingKeyword (token)           using
      WhitespaceTrivia (trailing)
    QualifiedName (node)
      IdentifierName (node)
        IdentifierToken (token)    System
      DotToken (token)             .
      IdentifierName (node)
        IdentifierToken (token)    Text
    SemiColonToken (token)         ;
  EndOfFileToken (token)
```

Analizując te dane wyjściowe od wewnątrz, najpierw mamy dwa węzły `IdentifierName`, dla których elementem nadrzędnym jest `QualifiedName`. Kod tworzący ten węzeł przedstawia się następująco:

```
QualifiedNameSyntax qualifiedName = SyntaxFactory.QualifiedName (
  SyntaxFactory.IdentifierName ("System"),
  SyntaxFactory.IdentifierName ("Text"));
```

Wykorzystaliśmy przeciążoną metodę `QualifiedName()` akceptującą dwa identyfikatory. Ta przeciążona metoda automatycznie wstawia token kropki.

Wygenerowany token musimy teraz opakować w `UsingDirective`:

```
UsingDirectiveSyntax usingDirective =
  SyntaxFactory.UsingDirective (qualifiedName);
```

Ponieważ nie wskazaliśmy tokenów dla słowa kluczowego `using` oraz średnika na końcu polecenia, odpowiednie tokeny będą utworzone i dodane automatycznie. Jednak te automatycznie

tworzone tokeny nie obejmują białych znaków. Wprawdzie nie uniemożliwia to kompilacji, ale konwersja takiego drzewa na postać ciągu tekstowego skutkuje powstaniem syntaktycznie nieprawidłowego kodu:

```
Console.WriteLine (usingDirective.ToFullString());  // usingSystem.Text;
```

Rozwiązaniem problemu jest wywołanie `NormalizeWhitespace()` w węźle (lub dla jednego z jego elementów nadrzędnych). W ten sposób następuje automatyczne dodanie drobiazgów w postaci białych znaków (dla zapewnienia zarówno syntaktycznej poprawności, jak i czytelności). Aby zachować większą kontrolę, biały znak możemy dodać wyraźnie, jak w poniższym fragmencie kodu:

```
usingDirective = usingDirective.WithUsingKeyword (
  usingDirective.UsingKeyword.WithTrailingTrivia (
    SyntaxFactory.Whitespace (" ")));

Console.WriteLine (usingDirective.ToFullString());  // using System.Text;
```

W celu zachowania przejrzystości „zebraliśmy" istniejący w węźle element `UsingKeyword`, do którego dodaliśmy końcowe drobiazgi. Przy nieco większym wysiłku można by utworzyć odpowiadający mu token, co wymaga wywołania `SyntaxFactory.Token(SyntaxKind.UsingKeyword)`.

Ostatnim krokiem jest dodanie naszego węzła `UsingDirective` do istniejącego lub nowego drzewa składni (lub precyzyjniej: do węzła głównego drzewa). W tym pierwszym przypadku rzutujemy węzeł główny istniejącego drzewa na `CompilationUnitSyntax` i wywołujemy metodę `AddUsings()`. Następnie można utworzyć nowe drzewo na podstawie transformowanej jednostki kompilacji:

```
var existingTree = CSharpSyntaxTree.ParseText ("class Program {}");
var existingUnit = (CompilationUnitSyntax) existingTree.GetRoot();

var unitWithUsing = existingUnit.AddUsings (usingDirective);
var treeWithUsing = CSharpSyntaxTree.Create (
  unitWithUsing.NormalizeWhitespace());
```

Pamiętaj, że wszystkie fragmenty drzewa składni są niemodyfikowalne. Wywołanie `AddUsings()` zwraca zupełnie nowy węzeł, natomiast pierwotny pozostaje nietknięty. Zignorowanie wartości zwrotnej tego wywołania to pomyłka, którą wyjątkowo łatwo jest popełnić!

Wywołaliśmy `NormalizeWhitespace()` w jednostce kompilacji, aby wykonanie metody `ToString()` w drzewie dostarczyło syntaktycznie poprawny i czytelny kod. Alternatywne podejście polega na wyraźnym dodaniu do `usingDirective` drobiazgu w postaci nowego wiersza za pomocą poniższego polecenia:

```
.WithTrailingTrivia (SyntaxFactory.EndOfLine("\r\n\r\n"))
```

Przygotowanie jednostki kompilacji i drzewa składni zupełnie od początku jest podobnym procesem. Najłatwiejsze podejście polega na rozpoczęciu pracy z pustą jednostką kompilacji, a następnie wywołaniu w niej `AddUsings()`, jak to zrobiliśmy już wcześniej:

```
var unit = SyntaxFactory.CompilationUnit().AddUsings (usingDirective);
```

Dodanie definicji typu do jednostki kompilacji odbywa się przez utworzenie ich w podobny sposób, a następnie przez wywołanie `AddMembers()`:

```
// utworzenie prostej, pustej definicji klasy
unit = unit.AddMembers (SyntaxFactory.ClassDeclaration ("Program"));
```

Ostatnim krokiem jest utworzenie drzewa, jak w poniższym fragmencie kodu:

```
var tree = CSharpSyntaxTree.Create (unit.NormalizeWhitespace());
Console.WriteLine (tree.ToString());

// dane wyjściowe
using System.Text;
class Program{}
```

Klasa CSharpSyntaxRewriter

W przypadku bardziej skomplikowanych transformacji drzewa składni można wykorzystać podklasę klasy `CSharpSyntaxRewriter`.

Klasa `CSharpSyntaxRewriter` jest podobna do omówionej już `CSharpSyntaxWalker` (zob. sekcję „Klasa CSharpSyntaxWalker" nieco wcześniej w rozdziale), z wyjątkiem faktu, że każda metoda `Visit*()` akceptuje i zwraca węzeł składni. Dzięki zwróceniu danych innych niż przekazane istnieje możliwość „przepisania" drzewa składni.

Na przykład w przedstawionym poniżej fragmencie kodu zmieniamy zadeklarowane nazwy metod na zapisane dużymi literami:

```
class MyRewriter : CSharpSyntaxRewriter
{
  public override SyntaxNode VisitMethodDeclaration
    (MethodDeclarationSyntax node)
  {
    // "zastępujemy" identyfikator metody wersją zapisaną dużymi literami
    return node.WithIdentifier (
      SyntaxFactory.Identifier (
        node.Identifier.LeadingTrivia,          // zachowanie starego drobiazgu
        node.Identifier.Text.ToUpperInvariant(),
        node.Identifier.TrailingTrivia));       // zachowanie starego drobiazgu
  }
}
```

Oto przykładowy sposób użycia powyższej klasy:

```
var tree = CSharpSyntaxTree.ParseText (@"class Program
{
  static void Main() { Test(); }
  static void Test() {         }
}");

var rewriter = new MyRewriter();
var newRoot = rewriter.Visit (tree.GetRoot());
Console.WriteLine (newRoot.ToFullString());

// dane wyjściowe
class Program
{
  static void MAIN() { Test(); }
  static void TEST() {         }
}
```

Zwróć uwagę, że w przypadku wywołania `Test()` w metodzie głównej nie nastąpiła zmiana nazwy, ponieważ uwzględniliśmy jedynie *deklaracje* elementów składowych i zignorowaliśmy *wywołania*. Jednak w celu niezawodnej zmiany wywołań konieczne jest ustalenie, czy wywołania `Main()` i `Test()` odwołują się do typu `Program`, a nie innego. W takim przypadku samo drzewo składni będzie niewystarczające, potrzebny jest jeszcze *model semantyczny*.

Kompilacja i model semantyczny

Kompilacja składa się z drzew składni, odwołań i opcji kompilacji. Służy do dwóch wymienionych poniżej celów:

- umożliwienie kompilacji na postać biblioteki lub pliku wykonywalnego (faza *emisji*);
- udostępnienie *modelu semantycznego* dostarczającego informacje o symbolach (pobranych podczas *wiązania*).

Model semantyczny ma istotne znaczenie w trakcie implementacji funkcji takich jak zmiana nazwy symbolu lub zapewnienie możliwości uzupełniania kodu w edytorze.

Utworzenie kompilacji

Kiedy jesteśmy zainteresowani sprawdzeniem modelu semantycznego lub przeprowadzeniem pełnej kompilacji, wówczas pierwszym krokiem będzie utworzenie `CSharpCompilation` i przekazanie (prostej) nazwy zestawu, który ma zostać utworzony:

```
var compilation = CSharpCompilation.Create ("test");
```

Prosta nazwa zestawu jest ważna, nawet jeśli nie zamierzamy emitować zestawu, ponieważ stanowi fragment tożsamości typów wewnątrz kompilacji.

Domyślnie przyjmujemy założenie, że chcemy utworzyć bibliotekę. W pokazany poniżej sposób można wskazać inny rodzaj danych wyjściowych (plik wykonywalny Windows, plik wykonywalny wiersza poleceń itd.):

```
compilation = compilation.WithOptions (
  new CSharpCompilationOptions (OutputKind.ConsoleApplication));
```

Klasa `CSharpCompilationOptions` ma dużą liczbę opcjonalnych parametrów konstruktora służących do przekazywania opcji do kompilatora. Na przykład optymalizację kompilatora możemy włączyć w następujący sposób:

```
compilation = compilation.WithOptions (
  new CSharpCompilationOptions (OutputKind.ConsoleApplication,
    optimizationLevel:OptimizationLevel.Release));
```

Kolejnym krokiem jest dodanie drzew składni. Każde odpowiada „plikowi” dołączanemu w trakcie kompilacji:

```
var tree = CSharpSyntaxTree.ParseText (@"class Program
{
  static void Main() => System.Console.WriteLine (""Witaj"");
}");

compilation = compilation.AddSyntaxTrees (tree);
```

Na koniec musimy dodać odwołanie do zestawów .NET. Ponieważ trudno jest dokładnie określić, które zestawy będą potrzebne, najprościej jest dołączyć wszystkie. Poniższy kod zwraca wszystkie zestawy .NET (oraz wszystkie używane przez aplikację wywołującą):

```
string trustedAssemblies = (string)AppContext.GetData
                                        ("TRUSTED_PLATFORM_ASSEMBLIES");
string[] trustedAssemblyPaths = trustedAssemblies.Split(Path.PathSeparator);
```

Ten kod zwraca **zestawy systemu wykonawczego**, które są specyficzne dla bieżącej platformy i wersji .NET Core. Jeśli planujesz używać Roslyn do kompilacji bibliotek działających na różnych wersjach platformy .NET Core, to użyj **bibliotek referencyjnych**. Są one dostępne w pakiecie NuGet *Microsoft.NETCore.app.ref* (w przypadku .NET Core), *Microsoft.AspNetCore.App.ref* (w przypadku ASP.NET Core) oraz *Microsoft.WindowsDesktop.app.ref* (w przypadku Windows Forms/WPF).

Następnie możemy dodać odwołania do kompilacji w następujący sposób:

```
var references = trustedAssemblyPaths.Select
    (path => MetadataReference.CreateFromFile (path));
compilation = compilation.AddReferences (references);
```

Wywołanie `MetadataReference.CreateFromFile()` wczytuje zawartość zestawu do pamięci, ale nie za pomocą prostego mechanizmu refleksji. Zamiast tego wykorzystywany jest charakteryzujący się dużą wydajnością przenośny czytnik zestawów o nazwie *System.Reflection.Metadata*, który unika tworzenia obiektu `Assembly`. (Tworzenie tego obiektu może być powolne i blokować plik zestawu do zakończenia procesu).

Otrzymany na skutek wywołania `MetadataReference.CreateFromFile()` egzemplarz `PortableExecutableReference` zużywa dość dużą ilość pamięci, więc starannie wybieraj odwołania, które chcesz zachować. Ponadto jeśli przekonasz się, że ciągle tworzysz odwołania do tego samego zestawu, wówczas rozważ jego buforowanie (idealnie sprawdza się tutaj zestawu ze słabymi odwołaniami).

Wszystko to można zrobić w pojedynczym kroku przez wywołanie przeciążonej metody `CSharp` ↪`Compilation.Create()` pobierającej drzewa składni, odwołania i opcje. Ewentualnie możemy wszystko umieścić w pojedynczym wyrażeniu, jak w poniższym fragmencie kodu:

```
var compilation = CSharpCompilation.Create ("...")
  .WithOptions (...)
  .AddSyntaxTrees (...)
  .AddReferences (...);
```

Diagnostyka

Kompilacja może wygenerować błędy i ostrzeżenia, nawet jeśli drzewa składni są bezbłędne. Przyczynami mogą być: pominięcie operacji importu przestrzeni nazw, drobna pomyłka podczas odwoływania się do typu lub nazwy elementu składowego oraz nieprawidłowe ustalenie typu parametru. Dostęp do błędów i ostrzeżeń otrzymujemy po wywołaniu `GetDiagnostics()` w obiekcie kompilacji. Składnia błędów również zostanie dołączona.

Emisja zestawu

Utworzenie zestawu wynikowego jest proste i sprowadza się do wywołania `Emit()`:

```
EmitResult result = compilation.Emit (@"c:\temp\test.exe");
Console.WriteLine (result.Success);
```

Jeżeli wartością `result.Success` jest `false`, wówczas egzemplarz `EmitResult` ma także właściwość `Diagnostics` wskazującą błędy, które wystąpiły podczas emisji (obejmuje to również dane diagnostyczne z wcześniejszych etapów). Niepowodzenie wywołania `Emit()` na skutek błędu operacji wejścia-wyjścia spowoduje zgłoszenie wyjątku, a nie wygenerowanie kodów błędu.

W .NET 5 i .NET Core należy nadać rozszerzenie *.dll* nawet aplikacji konsolowej i Windows. Aby ją uruchomić, należy wywołać program *dotnet.exe* ze ścieżką do pliku *.dll*.

Metoda `Emit()` pozwala również na podanie ścieżki dostępu do plików *.pdb* (dla informacji debugowania) oraz dokumentacji XML.

Sprawdzanie modelu semantycznego

Wywołanie `GetSemanticModel()` w kompilacji zwraca *model semantyczny* dla drzewa składni:

```
var tree = CSharpSyntaxTree.ParseText (@"class Program
{
  static void Main() => System.Console.WriteLine (123);
}");

var references = ((string)AppContext.GetData("ZAUFANE_ZESTAWY_PLATFORMY"))
  .Split (Path.PathSeparator)
  .Select (path => MetadataReference.CreateFromFile (path));

var compilation = CSharpCompilation.Create ("test")
  .AddReferences (references)
  .AddSyntaxTrees (tree);

SemanticModel model = compilation.GetSemanticModel (tree);
```

(Podanie drzewa jest konieczne, ponieważ kompilacja może zawierać ich wiele).

Być może oczekujesz, że model semantyczny będzie podobny do drzewa składni, choć z większą liczbą właściwości i metod oraz ze znacznie dokładniejszą strukturą. Tak jednak nie jest, nie mamy również modelu DOM powiązanego z modelem semantycznym. Zamiast tego mamy dostęp do zestawu metod pozwalających na uzyskanie semantycznych informacji o określonym położeniu węzła w drzewie składni.

Oznacza to brak możliwości „przeglądania" modelu semantycznego, inaczej niż w przypadku drzewa składni. Zamiast tego mamy coś, co przypomina grę w zadawanie pytań, a prawdziwym wyzwaniem jest ustalenie odpowiedniego pytania do zadania. Dostępnych jest niemal 50 metod i rozszerzeń metod, ale w tym rozdziale omówimy tylko najczęściej używane, w szczególności te, które przedstawiają reguły związane z użyciem modelu semantycznego.

Kontynuujemy pracę z wcześniejszym przykładem. W poniższym fragmencie kodu prosimy o podanie informacji dla identyfikatora `WriteLine`:

```
var writeLineNode = tree.GetRoot().DescendantTokens().Single (
  t => t.Text == "WriteLine").Parent;

SymbolInfo symbolInfo = model.GetSymbolInfo (writeLineNode);
Console.WriteLine (symbolInfo.Symbol);    // System.Console.WriteLine(int)
```

Klasa `SymbolInfo` to opakowanie dla symboli i za chwilę do niej powrócimy. Jednak najpierw zajmiemy się samymi symbolami.

Symbole

W drzewie składni nazwy takie jak: `System`, `Console` i `WriteLine` są przetwarzane jako **identyfikatory** (węzeł `IdentifierNameSyntax`). Identyfikatory mają niewielkie znaczenie, a syntaktyczny analizator składni nie próbuje ich „zrozumieć", a jedynie stara się odróżnić je od słów kluczowych.

Model semantyczny ma możliwość przeprowadzenia transformacji identyfikatorów na **symbole**, które mają informacje o typie (dane wyjściowe fazy *wiązania*).

Wszystkie symbole implementują interfejs `ISymbol`, choć istnieją jeszcze bardziej szczegółowe interfejsy dla poszczególnych rodzajów symboli. W naszym przykładzie `System`, `Console` i `WriteLine` są mapowane na symbole następujących typów:

```
"System"      INamespaceSymbol
"Console"     INamedTypeSymbol
"WriteLine"   IMethodSymbol
```

Pewne typy symboli, np. `IMethodSymbol`, mają koncepcyjne odpowiedniki zdefiniowane w przestrzeni nazw `System.Reflection` (w wymienionym przypadku to `MethodInfo`), podczas gdy inne typy, np. `INamespaceSymbol`, już nie. Wynika to z istnienia w kompilatorze Roslyn systemu typów, który ma być przydatny kompilatorowi. Natomiast system typów w refleksji istnieje po to, aby zapewnić korzyści środowisku uruchomieniowemu CLR (po przetworzeniu kodu źródłowego).

Niemniej jednak praca z typami `ISymbol` jest pod wieloma względami podobna do użycia API Reflection omówionego w rozdziale 18. Poniżej rozbudowujemy nasz wcześniejszy przykład:

```
ISymbol symbol = model.GetSymbolInfo (writeLineNode).Symbol;

Console.WriteLine (symbol.Name);                  // WriteLine
Console.WriteLine (symbol.Kind);                  // Method
Console.WriteLine (symbol.IsStatic);              // True
Console.WriteLine (symbol.ContainingType.Name);   // Console
var method = (IMethodSymbol) symbol;
Console.WriteLine (method.ReturnType.ToString()); // void
```

Wygenerowane dane wyjściowe w ostatnim wierszu pokazują subtelną różnicę względem API Reflection. Zwróć uwagę na słowo „void" zapisane małymi literami, co jest nomenklaturą C# (refleksja jest niezależna od języka). Podobnie wywołanie metody `ToString()` w `INamedTypeSymbol` dla `System.Int32` zwraca wartość „int". Poniżej pokazano coś jeszcze, czego nie można zrobić za pomocą refleksji:

```
Console.WriteLine (symbol.Language);              // C#
```

W przypadku API drzew składni klasy węzłów składni są różne dla języków C# i VB (choć współdzielą ten sam abstrakcyjny typ bazowy `SyntaxNode`). Ma to sens, ponieważ wymienione języki charakteryzują się odmiennymi strukturami leksykalnymi. Z kolei `ISymbol` i jego interfejsy potomne są współdzielone przez C# i VB. Jednak ich wewnętrzne konkretne implementacje *są* charakterystyczne dla każdego języka, a dane wyjściowe generowane przez metody i właściwości odzwierciedlają różnice między językami.

Istnieje również możliwość sprawdzenia pochodzenia danego symbolu:

```
var location = symbol.Locations.First();
Console.WriteLine (location.Kind);                    // MetadataFile
```

Jeżeli symbol został zdefiniowany w naszym własnym kodzie źródłowym (np. w drzewie składni), wówczas właściwość `SourceTree` zwróci to drzewo, natomiast `SourceSpan` poda jego położenie:

```
Console.WriteLine (location.SourceTree == null);     // True
Console.WriteLine (location.SourceSpan);             // [0..0)
```

Typ częściowy może mieć wiele definicji. W takim przypadku będzie miał też więcej położeń.

Poniższe polecenie zwraca wszystkie przeciążenia wywołania `WriteLine()`:

```
symbol.ContainingType.GetMembers ("WriteLine").OfType<IMethodSymbol>()
```

Można także wywołać `ToDisplayParts()` w symbolu. Wartością zwrotną będzie kolekcja „części" tworzących pełną nazwę. W omawianym przykładzie `Symbol.Console.WriteLine(int)` składa się z czterech symboli rozdzielonych znakami przestankowymi.

Klasa SymbolInfo

Jeżeli opracowujemy funkcję uzupełniania kodu źródłowego w edytorze, wówczas musimy pobrać symbole dla kodu niekompletnego lub nieprawidłowego. Spójrz na poniższy przykład niekompletnego kodu:

```
System.Console.Writeline(
```

Ponieważ metoda `WriteLine()` jest przeciążona, nie ma możliwości dopasowania pojedynczego egzemplarza implementującego `ISymbol`. Zamiast tego chcemy wyświetlić użytkownikowi dostępne opcje. W tym celu metoda `GetSymbolInfo()` modelu semantycznego zwraca strukturę `ISymbolInfo` wraz z następującymi właściwościami:

```
ISymbol Symbol
ImmutableArray<ISymbol> CandidateSymbols
CandidateReason CandidateReason
```

Jeżeli wystąpi błąd lub niejasność, wartością właściwości `Symbol` będzie `null`, natomiast `Candida ↪teSymbols` zwróci kolekcję składającą się z najlepszych dopasowań. Właściwość `CandidateReason` zwraca typ wyliczeniowy zawierający informacje o przyczynie niepowodzenia.

W celu pobrania informacji o błędach i ostrzeżeniach dla sekcji kodu można również wywołać `GetDiagnostics()` w modelu semantycznym, wskazując `TextSpan`. Wywołanie metody `GetDiagnostics()` bez argumentów jest odpowiednikiem wywołania tej samej metody w obiekcie `CSharpCompilation`.

Dostępność symbolu

Interfejs `ISymbol` ma właściwość `DeclaredAccessibilitiy` określającą widoczność symbolu (publiczny, chroniony, wewnętrzny itd.). Jednak wartość tej właściwości jest niewystarczająca do ustalenia, czy dany symbol jest dostępny we wskazanym położeniu w kodzie źródłowym. Na przykład zmienne lokalne mają leksykalnie ograniczony zakres, a chronione elementy składowe klasy są dostępne w kodzie źródłowym w położeniach wewnątrz typu klasy lub jej pochodnych. Aby pomóc programiście w prawidłowym ustaleniu dostępności symbolu, klasa `SemanticModel` ma metodę `IsAccessible()`:

```
bool canAccess = model.IsAccessible (42, sprawdzanySymbol);
```

Ta metoda zwraca wartość `true`, jeśli `sprawdzanySymbol` może być dostępny dla położenia 42 w kodzie źródłowym.

Zadeklarowane symbole

Jeżeli wywołamy `GetSymbolInfo()` dla typu lub deklaracji elementu składowego, nie otrzymamy z powrotem żadnych symboli. Na przykład przyjmujemy założenie, że chcemy otrzymać symbol dla metody `Main()`:

```
var mainMethod = tree.GetRoot().DescendantTokens().Single (
  t => t.Text == "Main").Parent;

SymbolInfo symbolInfo = model.GetSymbolInfo (mainMethod);
Console.WriteLine (symbolInfo.Symbol == null);              // True
Console.WriteLine (symbolInfo.CandidateSymbols.Length);     // 0
```

Ma to zastosowanie nie tylko dla deklaracji typu i elementu składowego, ale również dowolnego węzła, w którym *wprowadzony* jest nowy symbol, a nie jedynie *używany* istniejący.

W celu pobrania symbolu należy wywołać metodę `GetDeclaredSymbol()`:

```
ISymbol symbol = model.GetDeclaredSymbol (mainMethod);
```

W przeciwieństwie do `GetSymbolInfo()`, działanie metody `GetDeclaredSymbol()` kończy się sukcesem lub niepowodzeniem. (Przyczyną niepowodzenia może być nieznalezienie prawidłowej deklaracji węzła).

Przechodzimy teraz do kolejnego przykładu. Przyjmujemy istnienie pokazanej poniżej metody `Main()`:

```
static void Main()
{
  int xyz = 123;
}
```

Ustalenie typu xyz może się odbywać w następujący sposób:

```
SyntaxNode variableDecl = tree.GetRoot().DescendantTokens().Single (
  t => t.Text == "xyz").Parent;

var local = (ILocalSymbol) model.GetDeclaredSymbol (variableDecl);
Console.WriteLine (local.Type.ToString());              // int
Console.WriteLine (local.Type.BaseType.ToString());     // System.ValueType
```

Klasa TypeInfo

Czasami potrzebne są informacje typu dotyczące wyrażenia lub literału, dla którego nie ma wyraźnego symbolu. Spójrz na poniższy fragment kodu:

```
var now = System.DateTime.Now;
System.Console.WriteLine (now - now);
```

W celu ustalenia typu `now - now` wywołujemy `GetTypeInfo()` w modelu semantycznym:

```
SyntaxNode binaryExpr = tree.GetRoot().DescendantTokens().Single (
  t => t.Text == "-").Parent;

TypeInfo typeInfo = model.GetTypeInfo (binaryExpr);
```

Klasa `TypeInfo` ma dwie właściwości: `Type` i `ConvertedType`. Druga z wymienionych wskazuje typ po przeprowadzeniu jakichkolwiek niejawnych konwersji:

```
Console.WriteLine (typeInfo.Type);              // System.TimeSpan
Console.WriteLine (typeInfo.ConvertedType);     // object
```

Ponieważ metoda `Console.WriteLine()` jest przeciążona i akceptuje typ `object`, ale nie `TypeSpan`, przeprowadzana jest niejawna konwersja na obiekt, na co wskazuje wartość właściwości `typeInfo.ConvertedType`.

Wyszukiwanie symboli

Potężną funkcją w modelu semantycznym jest możliwość wyszukania wszystkich symboli w określonym miejscu kodu źródłowego. Otrzymany wynik jest podstawą dla wartości wyświetlanych przez listę IntelliSense, gdy użytkownik oczekuje listy dostępnych symboli.

W celu pobrania listy należy wywołać metodę `LookupSymbols()` wraz z podanym konkretnym miejscem w kodzie źródłowym. Poniżej przedstawiono pełny przykład:

```
var tree = CSharpSyntaxTree.ParseText (@"class Program
{
  static void Main()
  {
    int x = 123, y = 234;
  }
}");

var references = ((string)AppContext.GetData ("ZAUFANE_ZESTAWY_PLATFORMY"))
  .Split (Path.PathSeparator)
  .Select (path => MetadataReference.CreateFromFile (path));

var compilation = CSharpCompilation.Create ("test")
  .AddReferences (references)
  .AddSyntaxTrees (tree);

SemanticModel model = compilation.GetSemanticModel (tree);

// rozpoczynamy wyszukiwanie symboli dostępnych, począwszy od szóstego wiersza
int index = tree.GetText().Lines[5].Start;

foreach (ISymbol symbol in model.LookupSymbols (index))
  Console.WriteLine (symbol.ToString());
```

Oto wynik działania powyższego kodu:

```
y
x
Program.Main()
object.ToString()
object.Equals(object)
object.Equals(object, object)
object.ReferenceEquals(object, object)
object.GetHashCode()
object.GetType()
object.~Object()
object.MemberwiseClone()
Program
Microsoft
System
Windows
```

(Po zaimportowaniu przestrzeni nazw `System` otrzymamy setki dodatkowych symboli dla typów zdefiniowanych w tej przestrzeni nazw).

Przykład zmiany nazwy symbolu

Aby zilustrować omówione dotąd funkcje, przygotujemy metodę odpowiedzialną za zmianę nazwy symbolu. Ta metoda będzie działać niezawodnie w większości przypadków. W szczególności:

- symbol może być typem, elementem składowym, zmienną lokalną, zakresem lub zmienną pętli;
- ustalenie symbolu może nastąpić na podstawie jego użycia lub deklaracji;
- w przypadku klasy lub struktury przeprowadzona będzie zmiana nazwy konstruktorów statycznych i egzemplarza;
- w przypadku klasy przeprowadzona będzie zmiana nazwy finalizatora (destruktora).

W celu zachowania przejrzystości pominiemy pewne operacje sprawdzenia, takie jak zagwarantowanie, że nowo podana nazwa nie jest już w użyciu, a także że symbol nie jest przypadkiem skrajnym, dla którego operacja zmiany nazwy zakończy się niepowodzeniem. Przedstawiona metoda analizuje tylko jedno drzewo składni i ma poniższą definicję:

```
public SyntaxTree RenameSymbol (SemanticModel model, SyntaxToken token,
                                string newName)
```

Oczywistym sposobem implementacji omówionej funkcjonalności jest utworzenie podklasy klasy `CSharpSyntaxRewriter`. Jednak znacznie bardziej eleganckim i elastyczniejszym podejściem jest przygotowanie metody `RenameSymbol()` w taki sposób, aby wywoływała działającą na niskim poziomie metodę zwracającą tekst, którego nazwa ma zostać zmieniona:

```
public IEnumerable<TextSpan> GetRenameSpans (SemanticModel model,
                                             SyntaxToken token)
```

W ten sposób edytor będzie mógł bezpośrednio wywołać `GetRenameSpans()` i zastosować jedynie wprowadzone zmiany (w ramach transakcji *Cofnij*). Unikamy tym samym utraty stanu edytora, która mogłaby skutkować zastąpieniem całego tekstu.

Metoda `RenameSymbol()` staje się względnie prostym opakowaniem dla `GetRenameSpans()`. Możemy wykorzystać metodę `WithChanges()` obiektu `SourceText` do zastosowania sekwencji zmian tekstu:

```
public SyntaxTree RenameSymbol (SemanticModel model, SyntaxToken token,
                                string newName)
{
  IEnumerable<TextSpan> renameSpans = GetRenameSpans (model, token);

  SourceText newSourceText = model.SyntaxTree.GetText().WithChanges (
    renameSpans.Select (span => new TextChange (span, newName))
               .OrderBy (tc => tc));

  return model.SyntaxTree.WithChangedText (newSourceText);
}
```

Metoda `WithChanges()` zgłasza wyjątek, jeśli zmiany nie będą przeprowadzone w kolejności. Dlatego też wywołujemy `OrderBy()`.

Kolejnym etapem jest przygotowanie metody `GetRenameSpans()`. Pierwszym krokiem jest wyszukanie symbolu odpowiadającego tokenowi, którego nazwa ma zostać zmieniona. Ponieważ ten token może być częścią deklaracji lub sposobu użycia, zaczynamy od wywołania `GetSymbolInfo()` i jeśli wynikiem jest `null`, wtedy wywołujemy `GetDeclaredSymbol()`:

```
public IEnumerable<TextSpan> GetRenameSpans (SemanticModel model,
                                             SyntaxToken token)
{
  var node = token.Parent;

  ISymbol symbol = model.GetSymbolInfo (node).Symbol
                ?? model.GetDeclaredSymbol (node);

  if (symbol == null) return null;    // brak symbolu, którego nazwa ma być zmieniona
```

Konieczne jest znalezienie definicji symbolu. W tym celu można wykorzystać właściwość `Locations` symbolu. Założenie o wielu miejscach zapewnia większą niezawodność rozwiązania w sytuacji użycia klas częściowych lub metod, choć aby to pierwsze było użyteczne, niezbędne będzie rozbudowanie przykładu do pracy z wieloma drzewami składni:

```
var definitions =
  from location in symbol.Locations
  where location.SourceTree == node.SyntaxTree
  select location.SourceSpan;
```

Następnym krokiem jest znalezienie potencjalnych sposobów użycia symbolu. Na przykład zaczynamy od wyszukiwania tokenów potomnych o nazwach dopasowanych do nazwy symbolu, ponieważ będzie to najszybszy sposób eliminacji większości tokenów. Później możemy wywołać `GetSymbolInfo()` w węźle nadrzędnym tokena i sprawdzić, czy dopasujemy symbol, którego nazwa ma zostać zmieniona:

```
var usages =
  from t in model.SyntaxTree.GetRoot().DescendantTokens()
  where t.Text == symbol.Name
  let s = model.GetSymbolInfo (t.Parent).Symbol
  where s == symbol
  select t.Span;
```

Operacje dotyczące wiązania, takie jak pobieranie informacji o symbolu, mają tendencje do wolniejszego wykonywania niż operacje wykorzystujące jedynie tekst bądź drzewa składni. Wynika to z faktu, że proces wiązania może wymagać wyszukiwania typów w zestawach, stosowania reguł ustalania typu oraz sprawdzania pod kątem rozszerzeń metod.

Jeżeli symbol jest czymś innym niż nazwany typ (zmienna lokalna, zmienna zakresu itd.), wówczas nasze zadanie się kończy i można zwrócić definicje plus sposoby użycia:

```
if (symbol.Kind != SymbolKind.NamedType)
  return definitions.Concat (usages);
```

Jeżeli symbol jest nazwanym typem, konieczne jest przeprowadzenie operacji zmiany nazwy jego konstruktorów i destruktora, jeśli będzie obecny. W tym celu sprawdzamy węzły potomne i wyszukujemy deklaracje typu, którego nazwa odpowiada tej przeznaczonej do zmiany. Wówczas otrzymujemy jego *zadeklarowany* symbol i po dopasowaniu go do symbolu przeznaczonego do zmiany odszukujemy metody konstruktora i destruktora tego typu oraz zwracamy ich identyfikatory:

```
var structors =
    from type in model.SyntaxTree.GetRoot().DescendantNodes()
                                          .OfType<TypeDeclarationSyntax>()
    where type.Identifier.Text == symbol.Name
    let declaredSymbol = model.GetDeclaredSymbol (type)
    where declaredSymbol == symbol
    from method in type.Members
    let constructor = method as ConstructorDeclarationSyntax
    let destructor = method as DestructorDeclarationSyntax
    where constructor != null || destructor != null
    let identifier = constructor?.Identifier ?? destructor.Identifier
    select identifier.Span;

  return definitions.Concat (usages).Concat (structors);
}
```

Poniżej przedstawiono pełny kod źródłowy wraz z przykładem jego użycia:

```
void Demo()
{
  var tree = CSharpSyntaxTree.ParseText (@"class Program
{
  static Program() {}
  public Program() {}

  static void Main()
  {
    Program p = new Program();
    p.Foo();
  }

  void void Foo() => Bar();
  void void Bar() => Foo();
}
");

  var references = ((string)AppContext.GetData
                          ("ZAUFANE_ZESTAWY_PLATFORMY "))
  .Split (Path.PathSeparator)
```

```
  .Select (path => MetadataReference.CreateFromFile (path));

  var compilation = CSharpCompilation.Create ("test")
    .AddReferences (references)
    .AddSyntaxTrees (tree);

  var model = compilation.GetSemanticModel (tree);

  var tokens = tree.GetRoot().DescendantTokens();

  // zmiana nazwy klasy Program na Program2
  SyntaxToken program = tokens.First (t => t.Text == "Program");
  Console.WriteLine (RenameSymbol (model, program, "Program2").ToString());

  // zmiana nazwy metody Foo na Foo2
  SyntaxToken foo = tokens.Last (t => t.Text == "Foo");
  Console.WriteLine (RenameSymbol (model, foo, "Foo2").ToString());

  // zmiana nazwy zmiennej lokalnej p na p2
  SyntaxToken p = tokens.Last (t => t.Text == "p");
  Console.WriteLine (RenameSymbol (model, p, "p2").ToString());
}

public SyntaxTree RenameSymbol (SemanticModel model, SyntaxToken token,
                                string newName)
{
  IEnumerable<TextSpan> renameSpans =
    GetRenameSpans (model, token).OrderBy (s => s);

  SourceText newSourceText = model.SyntaxTree.GetText().WithChanges (
    renameSpans.Select (s => new TextChange (s, newName)));

  return model.SyntaxTree.WithChangedText (newSourceText);
}

public IEnumerable<TextSpan> GetRenameSpans (SemanticModel model,
                                             SyntaxToken token)
{
  var node = token.Parent;

  ISymbol symbol =
    model.GetSymbolInfo (node).Symbol ??
    model.GetDeclaredSymbol (node);

  if (symbol == null) return null;    // brak symbolu, którego nazwa ma być zmieniona

  var definitions =
    from location in symbol.Locations
    where location.SourceTree == node.SyntaxTree
    select location.SourceSpan;

  var usages =
    from t in model.SyntaxTree.GetRoot().DescendantTokens ()
    where t.Text == symbol.Name
    let s = model.GetSymbolInfo (t.Parent).Symbol
    where s == symbol
    select t.Span;

  if (symbol.Kind != SymbolKind.NamedType)
```

```
    return definitions.Concat (usages);

  var structors =
    from type in model.SyntaxTree.GetRoot().DescendantNodes()
                                          .OfType<TypeDeclarationSyntax>()
    where type.Identifier.Text == symbol.Name
    let declaredSymbol = model.GetDeclaredSymbol (type)
    where declaredSymbol == symbol
    from method in type.Members
    let constructor = method as ConstructorDeclarationSyntax
    let destructor = method as DestructorDeclarationSyntax
    where constructor != null || destructor != null
    let identifier = constructor?.Identifier ?? destructor.Identifier
    select identifier.Span;

  return definitions.Concat (usages).Concat (structors);
}
```