8
LINQ upiti

LINQ (Language Integrated Query – upit integrisan u jezik) jeste skup mogućnosti koje jezik C# i Framework pružaju za pisanje strukturiranih upita za pretraživanje lokalnih kolekcija objekata i udaljenih izvora podataka, na način koji ne narušava bezbednost tipova. LINQ je uveden u verziji C# 3.0 i Framework 3.5.

LINQ omogućava pretraživanje svake kolekcije koja implementira interfejs `IEnumerable<T>`, bez obzira na to da li je u pitanju niz vrednosti, lista ili XML DOM dokument, kao i udaljene izvore podataka, poput tabela na SQL Serveru. LINQ pruža prednosti i proveravanja ispravne upotrebe tipova u vreme prevođenja i dinamičkog sastavljanja upita.

Ovo poglavlje opisuje arhitekturu LINQ-a i osnove pisanja upita. Svi osnovni tipovi su definisani u imenskim prostorima `System.Linq` i `System.Linq.Expressions`.

Primeri u ovom i u naredna dva poglavlja ugrađeni su u interaktivnu alatku za izradu upita koja se zove LINQPad; možete je preuzeti na *http://www.linqpad.net*.

Uvod

Osnovne jedinice podataka s kojima LINQ radi jesu *sekvence* i *elementi*. Sekvenca je svaki objekat koji implementira interfejs `IEnumerable<T>` a element je svaka stavka sekvence. U primeru koji sledi, niz `names` je sekvenca, a `"Tom"`, `"Dick"` i `"Harry"` su njeni elementi:

```
string[] names = { "Tom", "Dick", "Harry" };
```

Ovakvu sekvencu zovemo *lokalna sekvenca* zato što predstavlja lokalnu kolekciju objekata u memoriji.

Operator za upit je metoda koja transformiše sekvencu. Tipičan operator za upit prihvata *ulaznu sekvencu* koju transformiše u *izlaznu sekvencu*. U klasi `Enumerable` u imenskom prostoru `System.Linq`, postoji oko 40 operatora za upite – a svi su implementirani u obliku statičkih proširenih metoda. To su *standardni operatori za upite*.

Upiti koji pretražuju lokalne sekvence zovu se lokalni upiti ili *LINQ-to-object* upiti.

LINQ podržava i sekvence koje se mogu dinamički popunjavati iz određenog udaljenog izvora podataka kao što SQL baza podataka. Te sekvence dodatno implementiraju interfejs `IQueryable<T>` i podržane su pomoću odgovarajućeg standardnog skupa operatora za upite u klasi `Queryable`. To ćemo detaljnije razmotriti u odeljku „Interpretirani upiti“, na strani 332 ovog poglavlja.

Upit je izraz koji, kada je nabrojen, transformiše sekvence pomoću operatora za upite. Najjednostavniji upit se sastoji od jedne ulazne sekvence i jednog operatora. Na primer, operator `Where` možemo primeniti na jednostavan niz da bismo iz njega izdvojili elemente koji sadrže barem četiri znaka, na sledeći način:

```
string[] names = { "Tom", "Dick", "Harry" };
IEnumerable<string> filteredNames = System.Linq.Enumerable.Where
                                    (names, n => n.Length >= 4);
foreach (string n in filteredNames)
  Console.WriteLine (n);

Dick
Harry
```

Pošto su standardni operatori za upite implementirani u obliku proširenih metoda, metodu `Where` možemo pozvati direktno za objekat `names` – isto kao da je to metoda instance:

```
IEnumerable<string> filteredNames = names.Where (n => n.Length >= 4);
```

Da bi to moglo da se kompajlira, morate uvesti imenski prostor `System.Linq`. Evo kako izgleda kompletan primer:

```
using System;
using System.Collections.Generic;
using System.Linq;

class LinqDemo
{
  static void Main()
  {
    string[] names = { "Tom", "Dick", "Harry" };

    IEnumerable<string> filteredNames = names.Where (n => n.Length >= 4);
    foreach (string name in filteredNames) Console.WriteLine (name);
  }
}

Dick
Harry
```

Kôd bismo mogli još skratiti ako za promenljivu `filteredNames` zadamo implicitno određivanje tipa:

```
var filteredNames = names.Where (n => n.Length >= 4);
```

Međutim, time smo umanjili razumljivost koda, naročito izvan integrisanog razvojnog okruženja, gde nema kratkih opisa koji bi nam pomogli.

U ovom poglavlju, izbegavaćemo implicitno određivanje tipa rezultata upita, osim na mestima gde je to obavezno (kao što ćemo videti u nastavku poglavlja, u odeljku „Strategije za projekcije“, na strani 330), ili kad je tip upita nebitan za sam primer.

Većina operatora za upite prihvata lambda izraz kao argument. Lambda izraz olakšava razumevanje i formiranje upita. U našem primeru, lambda izraz je sledeći:

```
n => n.Length >= 4
```

Ulazni argument odgovara jednom ulaznom elementu. U ovom primeru, ulazni argument `n` predstavlja svako ime u nizu a tip mu je `string`. Operator `Where` zahteva da lambda izraz vraća vrednost tipa `bool`, koja ako je `true`, znači da taj element treba uključiti u izlaznu sekvencu. Potpis operatora `Where` izgleda ovako:

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func<TSource,bool> predicate)
```

Sledeći upit izdvaja sva imena koja sadrže slovo „a“:

```
IEnumerable<string> filteredNames = names.Where (n => n.Contains ("a"));

foreach (string name in filteredNames)
  Console.WriteLine (name);             // Harry
```

Dosad smo upite sastavljali pomoću proširenih metoda i lambda izraza. Kao što ćemo uskoro videti, ta strategija pruža velike mogućnosti kombinovanja jer omogućava ulančavanje operatora za upite. U ovoj knjizi, to zovemo *tečna sintaksa* (engl. *fluent syntax*).[1] C# podržava i drugi oblik sintakse za pisanje upita, nazvan sintaksa *izraza upita*. Ovako izgleda naš prethodni upit napisan u obliku izraza za upit:

```
IEnumerable<string> filteredNames = from n in names
                                    where n.Contains ("a")
                                    select n;
```

Tečna sintaksa i sintaksa za upite međusobno su komplementarne. U naredna dva odeljka detaljno razmatramo oba oblika.

Tečna sintaksa

Tečna sintaksa je fleksibilna i fundamentalna. U ovom odeljku opisujemo kako se operatori za upite mogu ulančavati da bi se formirali složeniji upiti – i tako pokazati zbog čega su proširene metode važne za taj postupak. Opisujemo i kako treba formulisati lambda izraze za operator upita i uvodimo više novih operatora za upite.

Ulančavanje operatora za upite

U prethodom odeljku prikazali smo dva jednostavna upita, svaki s po jednim operatorom. Da biste sastavili složeniji upit, izrazu treba da dodate druge operatore, čime formirate lanac. Ilustracije radi, upit koji sledi izdvaja sva imena koja sadrže slovo „a“, sortira ih po dužini, a rezultat zatim pretvara u velika slova:

```
using System;
using System.Collections.Generic;
using System.Linq;

class LinqDemo
{
  static void Main()
  {
    string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
    IEnumerable<string> query = names
```

1. Izraz potiče iz rada autora Erika Evansa i Martina Faulera na temu tečnih interfejsa.

```
        .Where   (n => n.Contains ("a"))
        .OrderBy (n => n.Length)
        .Select  (n => n.ToUpper());

    foreach (string name in query) Console.WriteLine (name);
  }
}

JAY
MARY
HARRY
```

Promenljiva `n`, u našem primeru, jeste privatna u svakom lambda izrazu. Identifikator `n` možemo upotrebiti višekratno iz istih razloga zbog kojih možemo višekratno upotrebiti identifikator `c` u sledećoj metodi:

```
void Test()
{
  foreach (char c in "string1") Console.Write (c);
  foreach (char c in "string2") Console.Write (c);
  foreach (char c in "string3") Console.Write (c);
}
```

`Where`, `OrderBy` i `Select` su standardni operatori za upite koji se prevode u proširene metode klase `Enumerable` (ako uvezete imenski prostor `System.Linq`).

Već smo uveli operator `Where`, koji formira filtriranu verziju ulazne sekvence. Operator `OrderBy` vraća sortiranu verziju svoje ulazne sekvence, a metoda `Select` vraća izlaznu sekvencu u kojoj je svaki ulazni element transformisan ili *projektovan* pomoću zadatog lambda izraza (`n.ToUpper()`, u ovom slučaju). Podaci teku lancem operatora sleva nadesno, tako da se podaci prvo filtriraju, zatim sortiraju, a onda projektuju.

Operator upita nikad ne menja samu ulaznu sekvencu, nego uvek vraća novu sekvencu. To je u skladu s modelom *funkcionalnog programiranja*, kojim je LINQ inspirisan.

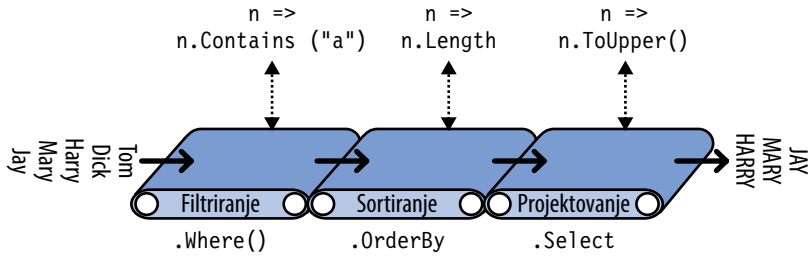
Ovako izgledaju potpisi pomenutih proširenih metoda (potpis metode `OrderBy` je neznatno pojednostavljen):

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func<TSource,bool> predicate)

public static IEnumerable<TSource> OrderBy<TSource,TKey>
  (this IEnumerable<TSource> source, Func<TSource,TKey> keySelector)

public static IEnumerable<TResult> Select<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
```

Kada se operatori za upite ulančavaju kao u ovom primeru, izlazna sekvenca jednog operatora jeste ulazna sekvenca operatora koji mu sledi. Konačan rezultat liči na proizvodnu liniju s transportnim trakama, kao što ilustruje slika 8-1.

Identičan upit možemo zadati i u obliku *korak po korak*, kao što sledi:

```
// Da bi se ovo prevelo, morate uvesti imenski prostor System.Linq:

IEnumerable<string> filtered   = names   .Where   (n => n.Contains ("a"));
IEnumerable<string> sorted     = filtered.OrderBy (n => n.Length);
IEnumerable<string> finalQuery = sorted  .Select  (n => n.ToUpper());
```



Slika 8-1. Ulančavanje operatora upita.

Upit `finalQuery` je po sadržaju identičan upitu `query` koji smo ranije napisali. Osim toga, i svaki međukorak je ispravan upit koji možemo izvršiti:

```
foreach (string name in filtered)
  Console.Write (name + "|");        // Harry|Mary|Jay|

Console.WriteLine();
foreach (string name in sorted)
  Console.Write (name + "|");        // Jay|Mary|Harry|

Console.WriteLine();
foreach (string name in finalQuery)
  Console.Write (name + "|");        // JAY|MARY|HARRY|
```

Zbog čega su važne proširene metode

Umesto sintakse s proširenim metodama, operatore za upite možete pozivati i pomoću standardne sintakse za statičke metode. Na primer:

```
IEnumerable<string> filtered = Enumerable.Where (names,
                                                 n => n.Contains ("a"));
IEnumerable<string> sorted = Enumerable.OrderBy (filtered, n => n.Length);
IEnumerable<string> finalQuery = Enumerable.Select (sorted,
                                                    n => n.ToUpper());
```

To je, zapravo, oblik u koji kompajler prevodi pozive proširenim metodama. Međutim, izbegavanje proširenih metoda ima svoju cenu, ako pokušavate da napišete upit koji se sastoji od samo jedne naredbe, kao što smo to ranije uradili. Vratimo se na upit kombinovan u jednoj naredbi – prvo sa sintaksom pomoću proširene metode:

```
IEnumerable<string> query = names.Where   (n => n.Contains ("a"))
                                 .OrderBy (n => n.Length)
                                 .Select  (n => n.ToUpper());
```

Njegov prirodan linearni oblik odražava tok podataka sleva nadesno i postavlja lambda izraze neposredno pored odgovarajućih operatora u upitu (*infiksna* notacija). Bez proširenih metoda, upit više nije *tečan*:

```
IEnumerable<string> query =
  Enumerable.Select (
    Enumerable.OrderBy (
      Enumerable.Where (
        names, n => n.Contains ("a")
      ), n => n.Length
    ), n => n.ToUpper()
  );
```

Kombinovanje lambda izraza

U prethodnim primerima, operatoru `Where` prosledili smo sledeći lambda izraz:

```
n => n.Contains ("a")      // Ulazni tip=string, povratni tip=bool.
```

Lambda izraz koji prihvata ulaznu vrednost i vraća tip `bool` zove se *predikat*.

Svrha lambda izraza zavisi od konkretnog operatora za upite. Kada ga zadate uz operator `Where`, lambda izraz određuje da li element treba uključiti u izlaznu sekvencu. U slučaju operatora `OrderBy`, lambda izraz preslikava svaki element u ulaznoj sekvenci na njegovo mesto po ključu za sortiranje. Uz operator `Select`, lambda izraz određuje kako se svaki element ulazne sekvence transformiše pre nego što se umetne u izlaznu sekvencu.

Lambda izraz pridružen operatoru za upite uvek deluje na pojedinačne elemente ulazne sekvence – a ne na sekvencu kao celinu.

Operator za upite izračunava vrednost lambda izraza na zahtev – najčešće jedanput po elementu ulazne sekvence. Lambda izrazi omogućavaju da operatorima za upite prosleđujete vlastitu logiku. To čini operatore za upite vrlo raznovrsnim – kao i jednostavnim „ispod haube“. Ovako izgleda kompletna implementacija metode `Enumerable.Where`, bez koda za obradu izuzetaka:

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func<TSource,bool> predicate)
{
  foreach (TSource element in source)
    if (predicate (element))
      yield return element;

}
```

Lambda izrazi i potpisi delegata Func

Standardni operatori za upite koriste generičke delegate `Func`. `Func` je porodica generičkih delegata opšte namene u imenskom prostoru `System` koji su definisani za sledeću svrhu:

Argumenti tipa u delegatu `Func` zadaju se istim redosledom kao i u lambda izrazima.

Stoga `Func<TSource,bool>` odgovara lambda izrazu `TSource=>bool`: onom koji prihvata argument tipa `TSource` i vraća vrednost tipa `bool`.

Slično tome, `Func<TSource,TResult>` odgovara lambda izrazu `TSource=>TResult`.

`Func` delegati su nabrojani u odeljku „Lambda izrazi“, na strani 135 poglavlja 4.

Lambda izrazi i tip elemenata

Standardni operatori za upite koriste sledeća generička imena tipova:

Generički tip	Značenje
`TSource`	Tip elemenata ulazne sekvence
`TResult`	Tip elemenata izlazne sekvence – različit od `Tsource`
`TKey`	Tip elemenata za *ključ* po kojem se elementi sortiraju, grupišu ili spajaju

Tip `TSource` određuje ulazna sekvenca. Tipovi `TResult` i `Tkey` se najčešće *izvode iz lambda izraza*.

Na primer, pogledajte potpis operatora `Select`:

```
public static IEnumerable<TResult> Select<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
```

Delegat `Func<TSource,TResult>` odgovara lambda izrazu `TSource=>TResult`: onom koji preslikava jedan *ulazni element* u jedan *izlazni element*. Pošto su `TSource` i `TResult` različiti tipovi, lambda izraz može da menja tip svakog elementa. Osim toga, lambda izraz *određuje tip izlazne sekvence*. Naredni upit pomoću operatora `Select` transformiše ulazne elemente tipa `string` u izlazne elemente celobrojnog tipa:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<int> query = names.Select (n => n.Length);

foreach (int length in query)
  Console.Write (length + "|");    // 3|4|5|4|3|
```

Kompajler *izvodi* tip `TResult` iz povratne vrednosti lambda izraza. U ovom slučaju, za `TResult` je izveden tip `int`.

Operator za upite `Where` je jednostavniji i ne zahteva izvođenje tipa izlazne sekvence, pošto su ulazni i izlazni elementi istog tipa. To je logično jer taj operator samo filtrira elemente; on ih ne *transformiše*:

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func<TSource,bool> predicate)
```

I najzad, razmotrite potpis operatora `OrderBy`:

```
// Malo pojednostavljen:
public static IEnumerable<TSource> OrderBy<TSource,TKey>
  (this IEnumerable<TSource> source, Func<TSource,TKey> keySelector)
```

`Func<TSource,TKey>` preslikava jedan ulazni element u *ključ za sortiranje*. Tip `TKey` se izvodi iz lambda izraza nezavisno od tipova ulaznih i izlaznih elemenata. Na primer, listu imena mogli smo da sortiramo po dužini imena (ključ tipa `int`) ili alfabetskim redosledom (ključ tipa `string`):

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<string> sortedByLength, sortedAlphabetically;
sortedByLength       = names.OrderBy (n => n.Length);   // ključ tipa int

sortedAlphabetically = names.OrderBy (n => n);          // ključ tipa string
```

Operatore za upite u klasi `Enumerable` možete pozivati i pomoću tradicionalnih delegata koji referenciraju metode umesto lambda izraza. Takvo rešenje je dobro za pojednostavljivanje određenih vrsta lokalnih upita – naročito u slučaju LINQ u XML – što je opisano u poglavlju 10. Međutim, nije upotrebljivo za sekvence koje implementiraju interfejs `IQueryable<T>` (na primer, kada pretražujete bazu podataka), zato što su operatorima definisani u interfejsu `Queryable` potrebni lambda izrazi da bi formirali stabla izraza. To ćemo razmotriti u nastavku poglavlja, u odeljku „Interpretirani upiti“, na strani 332.

Prirodni redosled

U LINQ-u je značajan redosled elemenata ulazne sekvence. Neki operatori za upite, kao što su `Take`, `Skip` i `Reverse`, uzimaju u obzir taj redosled.

Operator `Take` izdvaja prvih `x` elemenata ulazne sekvence, a ostale odbacuje:

```
int[] numbers  = { 10, 9, 8, 7, 6 };
IEnumerable<int> firstThree = numbers.Take (3);     // { 10, 9, 8 }
```

Operator `Skip` preskače prvih `x` elemenata i vraća sve preostale:

```
IEnumerable<int> lastTwo    = numbers.Skip (3);     // { 7, 6 }
```

`Reverse` radi tačno ono što biste očekivali:

```
IEnumerable<int> reversed   = numbers.Reverse();    // { 6, 7, 8, 9, 10 }
```

U lokalnim upitima (LINQ-to-objects), operatori kao što su `Where` i `Select` zadržavaju izvorni redosled elemenata u ulaznoj sekvenci (kao što to čine i drugi operatori za upite, osim onih koji izričito menjaju redosled).

Ostali operatori

Ne vraćaju svi operatori za upite izlaznu sekvencu. Operatori nad *elementima* izdvajaju određeni element iz ulazne sekvence. Primeri toga su operatori `First`, `Last` i `ElementAt`:

```
int[] numbers    = { 10, 9, 8, 7, 6 };
int firstNumber  = numbers.First();                        // 10
int lastNumber   = numbers.Last();                         // 6
int secondNumber = numbers.ElementAt(1);                   // 9
int secondLowest = numbers.OrderBy(n=>n).Skip(1).First();  // 7
```

Operatori za agregaciju (grupni) vraćaju skalarnu vrednost, najčešće numeričkog tipa:

```
int count = numbers.Count();          // 5;
int min = numbers.Min();              // 6;
```

Kvantifikatori vraćaju vrednost tipa `bool`:

```
bool hasTheNumberNine = numbers.Contains (9);          // true
bool hasMoreThanZeroElements = numbers.Any();          // true
bool hasAnOddElement = numbers.Any (n => n % 2 != 0);  // true
```

Pošto ti operatori vraćaju po jedan element, s njihovim rezultatom najčešće ne možete dalje pozivati druge operatore za upite, osim kada je i sam rezultujući element kolekcija.

Neki operatori za upite prihvataju dve ulazne sekvence. Primeri toga su operator `Concat`, koji nadovezuje jednu sekvencu na drugu, i operator `Union`, koji radi to isto, ali uklanja duplikate:

```
int[] seq1 = { 1, 2, 3 };
int[] seq2 = { 3, 4, 5 };
IEnumerable<int> concat = seq1.Concat (seq2);    //  { 1, 2, 3, 3, 4, 5 }
IEnumerable<int> union  = seq1.Union (seq2);     //  { 1, 2, 3, 4, 5 }
```

Operatori za spajanje sekvenci takođe pripadaju ovoj kategoriji. U poglavlju 9 detaljno su objašnjeni svi operatori za upite.

Izrazi upita

C# ima sintaksnu prečicu za pisanje LINQ upita, što se zove *izraz upita*. Nasuprot raširenom verovanju, izraz upita nije način da se SQL ugradi u C# kôd. Zapravo, dizajn izraza upita bio je inspirisan prvenstveno konceptom *kombinovanja lista* (engl. *list comprehension*) iz jezika za funkcionalno programiranje kao što su LISP i Haskell, mada je i SQL imao kozmetički uticaj.

U ovoj knjizi, sintaksu za izraze upita nazivamo jednostavno „sintaksa za upite“.

U prethodnom odeljku, napisali smo upit s tečnom sintaksom koji izdvaja znakovne nizove koji sadrže slovo „a“, sortira ih po dužini i pretvara u velika slova. Ovako izgleda isti upit u obliku sintakse za upite:

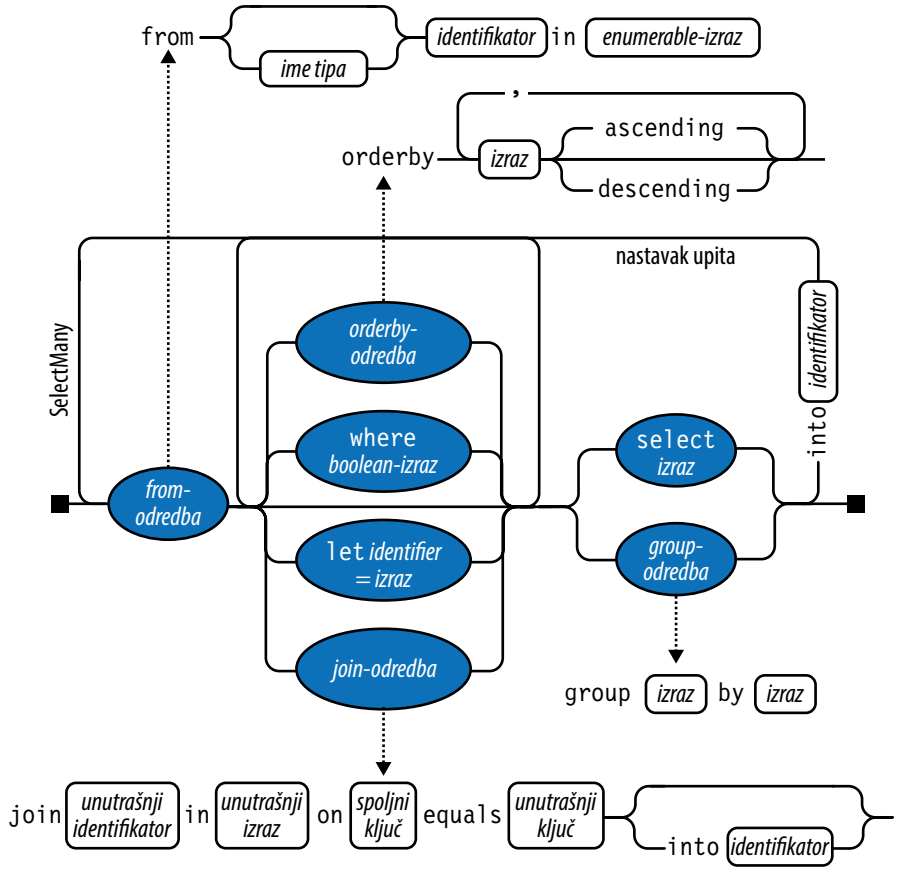
```
using System;
using System.Collections.Generic;
using System.Linq;

class LinqDemo
{
  static void Main()
  {
    string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

    IEnumerable<string> query =
      from    n in names
      where   n.Contains ("a")     // Filtrira elemente
      orderby n.Length             // Sortira elemente
      select  n.ToUpper();         // Pretvara svaki element (projekcija)

    foreach (string name in query) Console.WriteLine (name);
  }
}

JAY
MARY
HARRY
```

Izrazi upita uvek počinju odredbom `from` a završavaju se odredbom `select` ili `group`. Odredba `from` deklariše *promenljivu opsega* (u ovom slučaju, `n`), koju možete zamisliti kao način za pristupanje jednom po jednom elementu sekvence – otprilike kao `foreach`. Slika 8-2 ilustruje kompletnu sintaksu u obliku „šinskog“ dijagrama.

Da biste čitali taj dijagram, krenite od levog kraja a zatim nastavite po određenom „koloseku“ kao da ste voz. Na primer, iza obavezne odredbe `from`, možete opciono dodati odredbu `orderby`, `where`, `let` ili `join`. Iza toga, možete nastaviti do odredbe `select` ili `group`, ili se vratiti da biste dodali novu odredbu `from`, `orderby`, `where`, `let` ili `join`.

Kompajler obrađuje izraz upita tako što ga prevodi u tečnu sintaksu. To radi na sasvim mehanički način – vrlo slično načinu na koji naredbe `foreach` prevodi u pozive metoda `GetEnumerator` i `MoveNext`. To znači da sve što možete napisati u obliku izraza upita, možete napisati i u obliku tečne sintakse. Naš primer upita kompajler prevodi (prvo) u sledeći oblik:

```
IEnumerable<string> query = names.Where   (n => n.Contains ("a"))
                                 .OrderBy (n => n.Length)
                                 .Select  (n => n.ToUpper());
```



Slika 8-2. Sintaksa za upite.

Operatore `Where`, `OrderBy` i `Select` zatim razrešava prema istim pravilima koja bi važila kada bi upit bio napisan u tečnoj sintaksi. U ovom slučaju, oni se pretvaraju u pozive odgovarajućih proširenih metoda klase `Enumerable` jer je uvezen imenski prostor `System.Linq` i sekvenca `names` implementira interfejs `IEnumerable<string>`. Međutim, kada prevodi izraze upita, kompajler nije posebno sklon klasi `Enumerable`. Možete ga zamisliti kako u naredbu prvo mehanički umeće reči „Where“, „OrderBy“ i „Select“, a zatim prevodi rezultat kao da ste ta imena metoda vi napisali. To pruža veliku fleksibilnost u pogledu načina razrešavanja tih metoda. Na primer, upiti u baze podataka koje ćemo pisati u odeljcima koji slede, povezivaće se s proširenim metodama klase `Queryable`.

Ako iz našeg programa uklonimo direktivu `using System.Linq`, upit se neće kompajlirati zato što metode `Where`, `OrderBy` i `Select` neće imati s čim da se povežu. Izrazi upita se *ne mogu kompajlirati* dok ne uvezete `System.Linq` ili drugi imenski prostor u kojem se nalazi implementacija tih metoda upita.