IMPLEMENTING THE LINQ QUERY LANGUAGE INTO THE C++ PROGRAMMING LANGUAGE USING A PREPROCESSOR

Jakub Judas, Miroslav Virius
FJFI ČVUT

ABSTRACT:
LINQ is a query language similar to SQL that enables to retrieve data form any object with some standard properties. It was presented by Microsoft as a part of the Visual Basic and C# programming languages for the .NET Framework. This article presents the implementation of the LINQ query language in the standard C++. It is built as a preprocessor that translates the queries into standard C++ language. We present the basics of the LINQ for C++ grammar and the main ideas of the preprocessor.

KEYWORDS:
LINQ, C++, Query language, Compiler

INTRODUCTION

Language integrated query, or LINQ, is a querying language that can be used for any objects in the source code. It was originally introduced as a part of the .NET framework from Microsoft, in the C# and Visual Basic programming languages. It enables the programmer to access objects as well as other data sources using queries similar to the SQL language.
Our aim is to design and to develop the preprocessor that will process the LINQ queries in C++; as far as we are aware, there is no implementation of such a tool publicly available.

THE LINQ SYNTAX

To implement LINQ into C++ using a preprocessor, we'll have to look at two different problems. The first one is the implementation of the language itself. The other one is parsing existing C++ code sufficiently to be able to perform necessary changes to the code.

Syntax description

The syntax of the query language we will be implementing is as follows:

```
from [identifier] in [source collection]
let [expression]
where [boolean expression]
orderby [[expression](ascending/descending)], [optionally
repeat]
select [expression]
group [expression] by [expression] into [expression]
```

where **from** determines the data source to be used, **let** defines new temporary variables to be used further in the query, **where** sets the restriction filters, **orderby** determines the ordering of the resulting data, **select** declares what variables will be returned, and **group by** groups the returned data into a group of containers rather than in one container.

EXISTING IMPLEMENTATIONS

The most complete existing implementation of the language is found in the original .NET based languages.

According to the specification from Microsoft, the queries are directly translated into invocations of methods **Where**, **Select**, **SelectMany**, **OrderBy**, **OrderByDescending**, **ThenBy**, **ThenByDescending**, and **GroupBy** etc**.** These methods are invoked according to the following scheme:

```
Data.Where(...).Select(...).OrderBy(...)
```

By methods of reverse engineering, we further determined that all code except for keywords gets moved into new functions, which are then passed as parameters to the methods above.

For example, for the query

```
var dotaz=from x in pole
       where x%4==0
       select x/2;
```

the compiler creates two new functions, one of which returns `x%4==0`, and the other returns `x/2;` Then, the methods **Where** and **Select** are called with these new functions as parameters.

The keyword **let** is handled by creating a new anonymous type and a select performed on the source data.

OUR IMPLEMENTATION

Because the original implementation in C# is very elegant, we will be closely following it in our implementation in C++, however taking into account the specifics of the programming language.

Because we will be implementing the language for the Vector container, instead of using a series of methods line the .NET implementation does, we will be using functions instead, because the Vector container does not contain such methods and the C++ language does not support the extension methods; and of course, introducing new containers would confuse the programmer.

Instead of the pipelined method calls, as it it in the C#, the LINQ query will be compiled to a series of nested calls, like this: OrderBy(Select(Where(Data,...),...),...)

We will also implement the LINQ query through an additional preprocessor instead of directly integrating it into the compiler, because this would limit us to the use of one single compiler – probably g++ – and it would make the project unsustainable in the long term because of the fact that changes would have to be made for every new version of the compiler.

To avoid having to do work the original C++ preprocessor already does, such as includes, we will place our additional preprocessing between the original preprocessor and the compiler. Thus, the compilation will proceed this way:

1) Standard preprocessing
2) Additional preprocessing for LINQ queries
3) Compilation
4) Linking

The additional preprocessor will need to be able to process C++ to a certain degree, but will not have to parse the code completely, which is a very complicated task.

The preprocessor needs to insert newly made functions, as mentioned previously, into the code so that they are accessible globally, and so that they are above the place where the LINQ query is performed – this requirement is imposed by the way the C++ compiler works. Because of this, we need the preprocessor to know where in the code the query is performed. This can be in a function, in a method in a class, etc.

In the C++ language, all functions, classes etc. are enclosed in pairs of braces. Thus all our preprocessor will have to perform to identify the location is to separate the code into blocks and sub blocks determined by these braces, and further tokenize the content of these blocks by the occurrence of the semicolon character, which signals the end of a line. Furthermore we will have to take into account strings and comments, which may contain keywords specific for the LINQ language, that are not to be compiled.

Using this concept, the functions performing the query will be rather simple. For example, the function **Where** from this implementation looks as follows:

```
template<typename INPT>
vector<INPT> linq_where(vector<INPT> inp,bool (*cond)(INPT))
{
    vector<INPT> out;
    typename vector<INPT>::iterator it;
    for (it = inp.begin(); it!=inp.end(); ++it)
    {
        if(cond(*it))
        {
            out.push_back(*it);
        }
    }
    return out;
}
```

Compiling the LINQ code

The LINQ queries can be found in the code by the occurrence of the keyword **from.** This will disable the programmer to use **from** for any other purpose, but that is a natural disadvantage of implementing new functions to the language.

We can use the strictly set structure of the query and separate the code by the keywords. The condition for where will occur between where and select, etc.
The compiler itself can then be built as a finite state machine. A state diagram for our current implementation is shown in Figure 1.

Example:

```
vector<double> dotaz=from int x in pole
where x%4==0
select x/2;
```

will be compiled to:

```
double linq_select_1(int x)
{
     return x/2;
}
bool linq_where_1(int data)
{
```

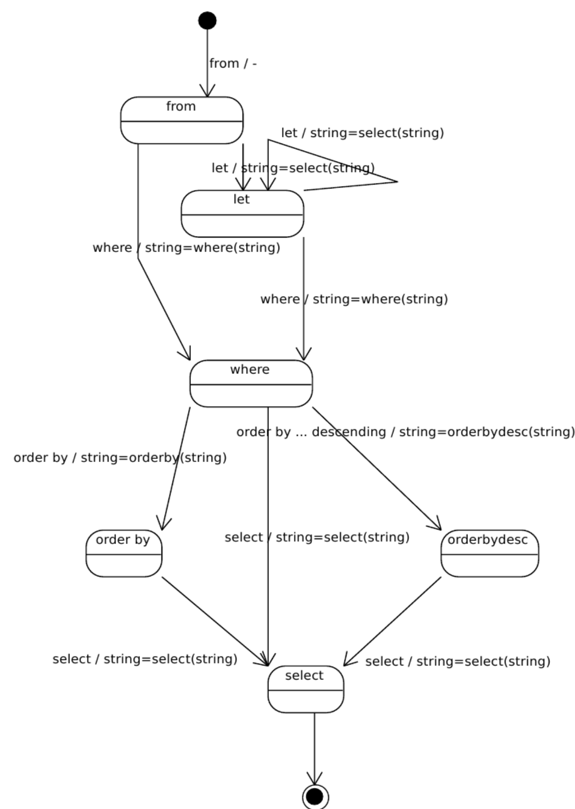

Fig. 1 - state diagram of the compiler FSM

```
     return x%4==0;
}...
vector<double>  dotaz=linq_select(linq_where( pole
  ,linq_Where_1),linq_Select_1);
```

CURRENT STATE AND OUTLOOK

We successfully implemented a limited part of the LINQ language preprocessor for the C++, namely: keywords **from**, **where**, **select** and **orderby**.

Our next aim is to implement the keywords **let** and **groupby**, as well as **join**, and additional features, as the keywords **descending**, and the functions for the aggregations. All of these keywords can be implemented in similar way we implemented the finished part.

REFERENCES

1. *C# Version 4.0 Specification*. Microsoft Corporation 2010.
   Available at <http:// download.microsoft.com/download/> [Cit. 5.4.2011]
2. *International Standard ISO/IEC 14882:2003. Programming Languages --- C++.* Genève: ISO 2003.