LinqToWeb

Content

[Introduction 3](#_Toc254212240)

[Language 3](#_Toc254212241)

[Syntax 3](#_Toc254212242)

[Language Features 3](#_Toc254212243)

[Examples 3](#_Toc254212244)

[Framework 3](#_Toc254212245)

[LINQ Provider, LINQ Extension 4](#_Toc254212246)

[Compiler to C# 4](#_Toc254212247)

[Lexical analysis 4](#_Toc254212248)

[Parser 4](#_Toc254212249)

[Emitting the code 5](#_Toc254212250)

[Usage 5](#_Toc254212251)

[Parallelism 5](#_Toc254212252)

[Visual Studio 2010 Integration 5](#_Toc254212253)

[Conclusion 5](#_Toc254212254)

# Introduction

* What? Why? Where? Extraction on the “live” web is performed. An abstraction over more web resources is defined. Transparent for the developer.
* Strongly typed object model representing data on the web that was created for human readers.
* Integrating data from various sources in various formats into single object model.
* Used in any .NET program, simplifies development
* Thanks to the abstraction - it allows to work with data on web as with data in the memory. Downloading, extracting, modifying and data formatting over this abstraction, transparent for user.
* Project components – Language (name?) + Code generator (Compiler) + Framework + WebData + User’s program (project architecture picture)
* Model of the project components

# Language

* Used for …

## Syntax

* Language grammar, lexical elements, syntax, strongly typed
* Names starting with “\_” are not allowed, reserved for system
* Elements (simple types, classes, methods, blocks, datatype construct, assigning, foreach, datacontexts, method call, literals, variables declaration, variable use, type cast, adding elements to lists, operators, no IF, no RETURN)
* Comments – ignored, no semantic usage, maybe in the future will be copied to the C# code, to enable IntelliSense in the user’s program (comment classes, public properties, class members, constructors)
* Whitespaces ignored

## Language Features

* Foreach and dynamically declared and initialized variables
* DataContext, Current datacontext, datacontext chaining
* Examples (everywhere)
* Variables: Extraction-Object and Value-Type
* Reason: minimalistic, more declarative, able to transform extracted data into any unified form, looks like simple synchronous extraction algorithm
* Methods declaration – main method = no name, same signature allowed, same name = call them all, not defined order when calling methods with the same name within the single MethodCall
* Properties(public extracted data) – extraction-object arguments of main method
* Extraction arguments – value arguments main method

## Examples

* Various code examples

# Framework

* Organize this section into introduction + namespaces
* Framework architecture picture
* Introduction - How does it work? (Diagram – framework, application, compiled code), used by the code generated from the source (see language)
* ExtractionContext (public properties, methods, cache)
* Base objects – classes and lists (actions, parent)
* Extraction on request – when some data missing, transparent for user, diagram
* Getting data from objects, Actions list, DoNextAction method, parent action
* (optimization - invoking best next action (GetNextAction) based on MethodDecl analysis)
* ActionItem, Extraction method delegate (current data context, arguments)
* Calling actions, modifying and filtering arguments (extracting only requested data, replace object with the ghost that cannot obtain actions – does nothing, does not invoke method calls, see list)
* if method is called and some argument already processed this method (means it does not have this action in ActionList), remember him to do not add new actions within this method call to this variable and to child objects of this variable (it affects AddElement of chained list too)
* ExtractionList – ExtractionListEnumerator, LINQ integration
* ExtractionList, enumeration, synchronizing ActionList with the listContainer, if method is called and argument is a list and thius list is not enumerated now – replace it with its empty ghost (see modifying arguments)
* VariableList
* DataContext, dynamic creating (the current DataContext type is known only at runtime, dynamically using reflections get the proper method to create the new context on the current context – because every DataContext can contain different methods to create new DataContexts)
* ScopeStack
* StorageBase (knowledge base)
* Foreach methods, and dynamic variables
* LINQ example, WHERE, ORDER, GROUP, SELECT, … everything works
* How does the LINQ query work: list enumerated and only needed properties extracted during evaluation the Where condition.
* LINQ expression and ordering: The entire list has to be enumerated!!

## LINQ Provider, LINQ Extension

* ***(Future plan)***
* Analyze Where expression, use it as a hint what data are requested – in GetNextAction and in ExtractionListEnumerator.GetEnumerator yield return only items matching conditions

# Compiler to C#

* Introduction: From “Language” to single C# code file
* Compiler architecture picture

## Lexical analysis

* Used tool for generating lexer
* Tokens, ignoring comments and whitespaces

## Parser

* Used tool for generating parser, what grammar
* Abstract Syntax Tree, Node position (ExprPosition class)

## Emitting the code

* Element to what C# construct conversion
* CodeContext – declared local variables, classes, methods
* MethodCall adds new action for its arguments that are allowed
* (optimization: Analysis of MethodDecl, affected variables and call level)
* Adding elements to list in another object (a.list[] = x; // Convert to method call, list is special object, elements must be added only within methods that have this list as an argument – so user is enumerating him, otherwise only action can be added into the list … when enumerating list, actions are used to get items – see Framework/ExtractionList)
* Current datacontext
* DataContext creating
* Foreach methods, “dynamic” variables, method signature, where they are placed, using reflections

# Usage

* Write code, compile to C#, use in .NET project (ASP.NET, C# console, winform, library, …)
* Mono
* Diagram of typical usage

In the user’s program:

* Context creation
* Using public properties: data types, lists
* Lists allow enumerating, enumeration invokes extraction every time, but data from the cache are used. Depends on the framework, what is cached.
* Data types – allow reading their properties (or modifying), property containing some data (as a result of extraction or set by user’s program) remembers its value – so reading non-empty property does not invoke extraction.
* Examples.

# Parallelism

* More contexts can be created on different threads, using the same cache object the data are safely and properly shared but the code is running on separated threads. (More actions on different contexts are called at the same time, but the knowledge base is shared – the knowledge base is that slow thing)
* This solution keeps the actions processed in the same order every time
* Picture of parallelism architecture

# Visual Studio 2010 Integration

* ***(Future plan)***
* New file type
* Creates .sucker file and .cs read-only file, automatically references required assemblies
* .sucker file can be modified (in graphical environment or as a source code text)
* When a .sucker file is saved, it’s compiled into the corresponding .cs file
* .cs file is used in the user’s program

# Conclusion

* Only requested data extracted
* able to be updated automatically
* does not require long extraction processes
* define common abstraction over various web resources
* High performance, compiled