TDT4510 Data and Information Management, Specialization Project
Fall 2008

Developing a SPARQL parser for .NET

Authors:
Ole Petter Bang and Tormod Fjeldskår

Supervisors:
Professor Svein Erik Bratsberg and
Fast representative Øystein Torbjørnsen

NTNU

Norwegian University of Science and Technology
Department of Computer and Information Science

June 3, 2009

Abstract

The Semantic Web is growing, both in size and popularity. At the core of the Semantic Web is the *Resource Description Framework* (RDF). RDF allows for encoding of information in web-based applications. The *SPARQL Protocol and RDF Query Language* is a SQL-like query language, recommended by W3C for querying RDF data.

A SPARQL parser front-end, called SharQL, has been created for the Microsoft .NET Framework using the *MPLex* and *MPPG* tools from Microsoft's *Managed Babel* package. The parser is capable of transforming textual SPARQL queries into abstract syntax trees (ASTs) representing the queries.

The parser has been tested using syntax tests for the SPARQL language released by the *Data Access Working Group* of W3C. The test suite contains categorized tests focusing on different aspects of the SPARQL grammar. All relevant tests pass using the SharQL parser, indicating the level of conformity with the SPARQL specification.

In order to utilize the results of the project, a back-end is needed in order to perform analysis, transformations and actual data queries against an RDF data store.

Preface

This report is the result of the course "TDT4510 Data and Information Management, Specialization Project" at the Norwegian University of Science and Technology (NTNU), during fall 2008. It is a response to the following task description:

> RDF is a schema for encoding information in web-based applications and SPARQL is a W3C standard query language similar to SQL that is used to query such RDF data. The task of the project is to build a parser for SPARQL that will generate an abstract syntax tree (AST). The project can use open-source components but the resulting code should be written in C# and not be limited by license terms blocking commercial use.

We would like to thank our supervisors, professor Svein Erik Bratsberg at NTNU and Øystein Torbjørnsen at FAST, for feedback and guidance during the course of this project.

Trondheim, December 2008

Ole Petter Bang   Tormod Fjeldskår

Contents

List of Tables

List of Figures

Chapter 1

Introduction

"The Semantic Web is a web of data." - W3C [3]

The *Semantic Web* attempts at a transition from the traditional web of application-centric data, to a web of interconnected data from various data sources. A corner stone of the Semantic Web is the *Resource Description Framework* (RDF). RDF allows for representation of metadata about web resources, or entities that can be identified by web resources. [4]

Describing web resources is one thing. To leverage the power of interconnected data, however, one needs a way of querying them. If e.g. a photo sharing application and a calendar sharing application both publish RDF data, it might be of interest to find all photos taken while the photographer was attending a certain photography conference.

SPARQL (SPARQL Protocol and RDF Query Language) is the language proposed by the *RDF Data Access Working Group* for querying RDF datasets. On January 15th 2008, the SPARQL specification reached the status of *W3C Recommendation*, the highest maturity level possible for W3C specifications. With SPARQL, queries like the aforementioned photography case are easy to express. [1]

Information Access Disruptions (iAD) is a constellation between FAST, two Norwegian enterprises and different research environments. Their goal is to develop the best search technology in the world. As part of this research, it is desirable to be able to query several different types of information from various data sources, including RDF.

The underlying search technology used by FAST is called *MARS*. The goal of this project is to create a parser for the SPARQL language, written in C# and running on Microsoft's .NET Framework. The SPARQL parser produced as part of this project should in turn be able to facilitate querying against RDF data sources using the .NET based MARS technology.

In Chapter 2, a brief introduction to RDF and SPARQL is given. Following this introduction, Chapter 3 presents the architectural decisions made while developing the SPARQL parser. This mainly concerns the choice of parser generator, but also explains briefly the class hierarchy facilitating AST creation. Chapter 4 describes the steps of creating the SPARQL parser, all the way from the setup of the development environment to the encapsulation of the resulting parser. Chapter 5 explains the methods used to test the resulting parser, to ensure that it behaves according to the specification. In Chapter 6, the results of the project are presented, both in terms of how the parser operates and how well it conforms with the specification. Finally, Chapter 7 concludes and summarizes the findings of the project and presents further work necessary for this parser to facilitate iAD research.

Chapter 2

RDF and SPARQL

RDF is a schema for encoding information in web-based applications and SPARQL is a W3C standard query language similar to SQL that is used to query RDF data. This chapter gives a short introduction to the nature of RDF data, followed by an introduction to SPARQL.

2.1 RDF Essentials

RDF (Resource Description Framework [4]) data represents information (particularly metadata) about resources in the World Wide Web. The concept of a "resource" is typically generalized, allowing for information not directly retrievable on the Web to be represented. Thus, it is entirely up to the producers and consumers of the RDF data to agree upon the semantics of the information, as long as it is represented according to the RDF schema. Different efforts have been made in order to standardize such semantics, including The Dublin Core Initiative [5].

RDF is intended for situations in which information needs to be processed by applications and exchanged without loss of meaning. Resources are identified using URIs[1], and described in terms of properties and their corresponding values. The properties are also identified using URIs, and their values are identified either by URIs or data typed literals. Altogether, resources and their properties are represented as *triples*, consisting of a subject (the resource), a predicate (the property), and an object (the property value). This enables RDF to represent simple statements about resources as a *graph* of nodes with arcs representing the resources, their properties and corresponding values.

[1]Actually, the term URI in the context of both RDF and SPARQL always refers to Internationalized Resource Identifiers (IRIs), a generalization of the Uniform Resource Identifier (URI), which may contain characters form the Universal Character Set (Unicode/ISO 10646).

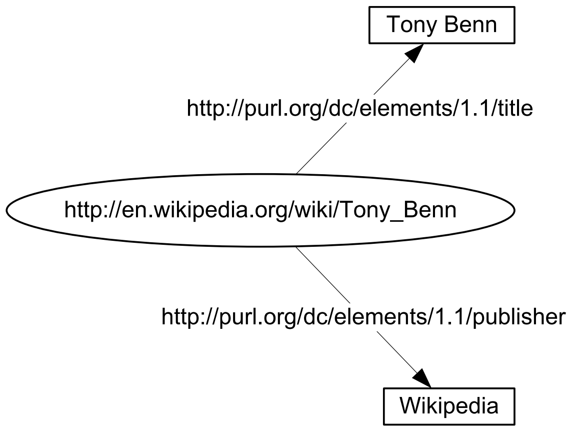

Tony Benn
http://purl.org/dc/elements/1.1/title
http://en.wikipedia.org/wiki/Tony_Benn
http://purl.org/dc/elements/1.1/publisher
Wikipedia

Figure 2.1: RDF sample (graph)

```
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:dc="http://purl.org/dc/elements/1.1/">
  <rdf:Description
      rdf:about="http://en.wikipedia.org/wiki/Tony_Benn">
    <dc:title>Tony Benn</dc:title>
    <dc:publisher>Wikipedia</dc:publisher>
  </rdf:Description>
</rdf:RDF>
```

Figure 2.2: RDF sample (XML)

Figure 2.1 illustrates a sample RDF graph. The resource in question is a Wikipedia web-page about the British politician Tony Benn, represented by the elliptic node labeled with the web-page URI. The standardized properties used to describe the resource are gathered from Dublin Core [5], each represented by an arc pointing at the corresponding rectangular property value node.

The XML-based representation of the very same information is given in Figure 2.2. A few, just as common, non-XML-based representations exist as well, like the Turtle [6] representation illustrated in Figure 2.3. The most obvious advantage non-XML-based representations like Turtle have over XML-based representations, is the strongly reduced level of verbosity. Also, both the Turtle and SPARQL languages are subsets of the Notation 3 [7] language, making the two syntactically very much alike.

```
@prefix dc: <http://purl.org/dc/elements/1.1/>.

<http://en.wikipedia.org/wiki/Tony_Benn>
  dc:title "Tony Benn";
  dc:publisher "Wikipedia".
```

Figure 2.3: RDF sample (Turtle)

2.2 Introduction to SPARQL

SPARQL (SPARQL Protocol and RDF Query Language [1]) is a query language designed to meet the use cases and requirements identified by the RDF Data Access Working Group [8].

SPARQL queries may consist of triple patterns, conjunctions, disjunctions and optional patterns. Most forms of queries contain a set of triple patterns, which are just like RDF triples except that each of the subject, predicate and object may be a variable. Such triple patterns match an RDF subgraph when an equivalent graph may be constructed from the triple pattern by substituting all variables with the corresponding terms from the subgraph.

Given the RDF data illustrated in Figure 2.2, the SPARQL query illustrated in Figure 2.4 matches the RDF subgraph describing the title property. This is done by substituting the *?title* variable with the property value "Tony Benn", which is also bound to the variable and then returned as illustrated in Table 2.1.

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT ?title
WHERE
{
  <http://en.wikipedia.org/wiki/Tony_Benn> dc:title ?title
}
```

Figure 2.4: SPARQL sample query

title
Tony Benn

Table 2.1: SPARQL sample query result set

Each solution gives one way in which the selected variables can be bound to an RDF subgraph so that the query pattern matches the data. When literals are matched, the data type is also considered. In addition to common data types, strings may be suffixed with a language tag, and custom data types may be used.

To further restrict the matches, *filters* can be used to restrict common data type values as well as to test different kinds of conditions using test functions. Restrictions on common data types include regular expressions on strings and arithmetic expressions for numeric values.

In an RDF triple the subject and object may be *blank nodes*, which are nodes without any identification other than a label to differentiate nodes from each other. Generally, blank nodes in a result set are not required to have the same labels as their corresponding blank nodes from the SPARQL query. The blank nodes simply act as non-distinguished variables in the query, not as references to specific blank nodes. Thus, an application writer should not expect a blank node in a query to refer to a particular blank node in the data.

Specific triple patterns may be marked as being optional, allowing for solutions to exist where the optional triple patterns does not match any RDF subgraph. Further, alternative triple patterns may be given, allowing for only one of several alternative graph patterns to match. If more than one of the alternatives match, all the possible pattern solutions are found. Also, several RDF graphs may be included and merged in a query, to allow query of several RDF stores.

The solution sequence of a SPARQL query may be influenced by a set of modifiers, most of which are familiar from languages like SQL. The modifiers available allow for ordering, projecting, selecting distinct solutions only, setting an offset defining the element position in the solution sequence from which to retrieve elements, as well as limiting the number of elements to be retrieved, starting from the beginning of the solution sequence unless an offset is stated explicitly. In addition a *REDUCED* modifier exists. The cardinality of any set of variable bindings in a reduced solution set is at least one and not more than the cardinality of the solution set with no *DISTINCT* or *REDUCED* modifiers.

SPARQL has four query forms. *SELECT* queries return all, or a subset of, the variables bound in a query pattern match. *CONSTRUCT* queries return RDF graphs constructed by substituting variables in a set of triple templates. An *ASK* query returns a boolean value indicating whether a specific query pattern matches or not. *DESCRIBE* queries return RDF graphs describing the resources found.

2.3 SPARQL Testimonials

Many companies have taken an interest in SPARQL. Following the press release announcing the publication of SPARQL [9], several early adopters responded with testimonials about W3C's new specification [10]. Among these were both smaller and larger companies. To give an impression of how SPARQL is received in the communities, some of the testimonials are quoted below.

Computas AS:

> *Computas AS is currently building systems for its customers where SPARQL is a fundamental core component. When conducting feasibility studies, we found that there are allready many high quality off-the-shelf components that puts the vision of the data web within reach, also for smaller enterprises.*
>
> *We are pleased to see SPARQL promoted to a W3C Recommendation, as it provides a stable platform for further work. We are allready experimenting with extensions to SPARQL, and will work with the W3C and its membership in the work that lies ahead.*
>
> **– Kjetil Kjernsmo, Senior Knowledge Engineer, Computas AS**

Hewlett-Packard:

> *Hewlett-Packard is pleased to support the SPARQL Recommendations.*
>
> *SPARQL is a key element for integrated information access across information silos and across business boundaries. HP customers can benefit from better information utilization by employing semantic web technologies.*
>
> *HP's Jena Semantic Web framework has a complete implementation of query language, protocol and result set processing. Jena is open-source, freely available, with a large and active developer community.*
>
> *HP is pleased to announce the first full release of SDB, a new SPARQL database system for Jena that leverages existing database installations to give enterprise-grade storage and query of RDF.*
>
> **– Jean-Luc Chatelain, CTO HP Software Information Management**

Oracle:

> *Oracle congratulates the W3C on achieving 'Recommendation' status for SPARQL. As an active participant in this working group, Oracle believes the standardization of SPARQL will play an instrumental role in achieving the vision of the Semantic Web. The community's work is intended to help organizations more effectively discover, automate, integrate and re-use data across various applications.*

Oracle Database 11g Semantic Store provides native support for efficient and scalable storage, bulk loading, inferencing, and graph-pattern based querying of semantic data represented using W3C's RDF, RDFS, and OWL languages. The Oracle Jena adaptor allows querying of semantic data stored in Oracle using the SPARQL query language while leveraging the performance and scalability of Oracle's Semantic Store.

– Don Deutsch, vice president Standards Strategy and Architecture, Oracle

Chapter 3

Architectural Decisions

This chapter presents the architectural decisions made. This mainly involves choosing a suitable parser generator, but the class hierarchy facilitating AST creation is also explained. Various alternatives are presented along with the rationale behind the final decisions.

3.1 Choosing a Parser Generator

The main architectural decision for this project was the choice of parser generator. A parser generator is a program that reads a language specification formatted using a certain syntax. Based on this specification, a parser is created. Different generators create different parser types. Some create top-down (recursive descent) *LL(k)* parsers, while others create bottom-up *LALR* parsers. LALR parsers can handle a wider range of grammars than LL(k) parsers [11].

Given that the resulting parser of this project is to be written in C#, the parser generator has to be able to generate C# source code. According to the SPARQL specification, *"The SPARQL grammar is LL(1) when the rules with uppercased names are used as terminals."* [1]. This section discusses some of the alternatives that were considered, and presents the rationale behind the choice.

3.1.1 ANTLR

ANTLR (<u>AN</u>other <u>T</u>ool for <u>L</u>anguage <u>R</u>ecognition) is an open source parser generator, written in Java. It is considered mature and has reached version 3.1 at the time of writing.

The parsers created by ANTLR are recursive descent LL(k) parsers which are less expressive than LR/LALR parsers, but the generated source code is more intuitive. Source code can be generated in several programming

	ANTLR	**Coco/R**	**MPLex/MPPG**
Rules definition	EBNF	Attributed EBNF	Yacc-like BNF
Parser type	LL(k)	LL(k)	LALR
AST generation	Automatic	Manual	Manual
Supported by	Community	Community	Microsoft
Visual Studio integration out-of-the-box	No	No	Yes (requires manual editing of project file)

Table 3.1: Summary of parser generators

languages, such as Java, C++ and C#. Rules are defined in a format similar to EBNF. [12]

3.1.2 Coco/R

Coco/R (Compiler compiler/Recursive descent) is similar to ANTLR in that is creates recursive descent LL(k) parsers. However, while ANTLR is written in Java with the possibility of generating source code in various languages, Coco/R comes in different versions supporting different languages. To generate C# code, the C# version of Coco/R is required. Rules are defined as an attributed EBNF grammar [13].

3.1.3 MPLex and MPPG

MPLex (Managed Package Lex) and *MPPG* (Managed Package Parser Generator) are provided by Microsoft as part of their Visual Studio SDK. The scanner generator *MPLex* and the parser generator *MPPG* are part of the Managed Babel package aimed at Visual Studio extension developers. [14]

MPLex and MPPG are closely related to the *Queensland University of Technology* open source projects *Gardens Point Scanner Generator* (GPLEX) [15] and *Gardens Point Parser Generator* (GPPG) [16]. The parsers generated by MPPG are C# only, bottom-up LALR parsers.

Being part of the Visual Studio SDK, integrating the actions of MPLex and MPPG into a Visual Studio project is relatively easy. This is an advantage over GPLEX and GPPG. Rules for MPLex and MPPG are specified in a format largely based on the syntax used by tools like Lex and Yacc [17].

3.1.4 Our Choice

The various properties of the different parser generators are listed in Table 3.1. In cooperation with the supervisors, the choice of parser generator ultimately fell on MPLex/MPPG. The LALR parsers created by MPPG are more expressive than the LL(k) parsers created by the two other alternatives [11]. This could be a future advantage if changes to the SPARQL language require such expressiveness. Even though ANTLR and Coco/R have large communities and good testimonials, the fact that MPLex and MPPG are provided and supported by Microsoft [14] was decisive. The ease of integration with the Visual Studio environment is an advantage as well.

3.2 Further Architectural Choices

As the actual code performing the lexical analysis and the parsing itself is generated using MPLex and MPPG, the remaining functionality subject to architectural decisions is the construction and representation of the abstract syntax tree (AST) representing the parsed queries.

If the parser was to be generated using Lex and Yacc [17], before the entry of object oriented languages, the abstract syntax tree would most likely have been constructed and represented using C variable pointers to *structs*, imitating objects. The C programming language has no knowledge of objects, making this approach the only viable option for implementations of notable size and complexity.

Inspired by its predecessors, the C# programming language still retains knowledge of structs. C# variables always represent structs by value; a struct may never be represented by reference, though it may be passed by reference between methods [18]. When reassigning a variable holding a C# struct, the entire content of the struct is copied to the memory location references by the variable. A variable referencing a class object would instead have changed the destination of the reference to where the class object already resides. Also, C# structs can only implement interfaces, they cannot inherit other classes or structs. Not being able to exploit the powerful concept of inheritance makes C# structs even less suitable for AST representation.

An AST is by nature based on references between tree nodes. Thus class objects are the only viable option for representing AST nodes. Using class objects, a suitable class hierarchy may be built, benefiting from important properties of object orientation like inheritance and shared interfaces in order to generalize the hierarchy.

For this project, the planned architecture involves using C# objects instantiated to represent the abstract syntax tree. Common base functionality should be implemented in a single base node, from which all other nodes inherit. This is further discussed in Section 4.2.

Chapter 4

Implementation

This chapter describes the steps of creating the SPARQL parser. First, the setup of the development environment is explained. Then the process of translating W3C's SPARQL specification to a format that MPLex and MPPG understand is described. Error handling, along with some necessary pre- and post-processing is then explained, as well as how the entire implementation is encapsulated in a facade class. Finally, a test client for visual inspection of resulting ASTs is presented.

4.1 Setup of the Development Environment

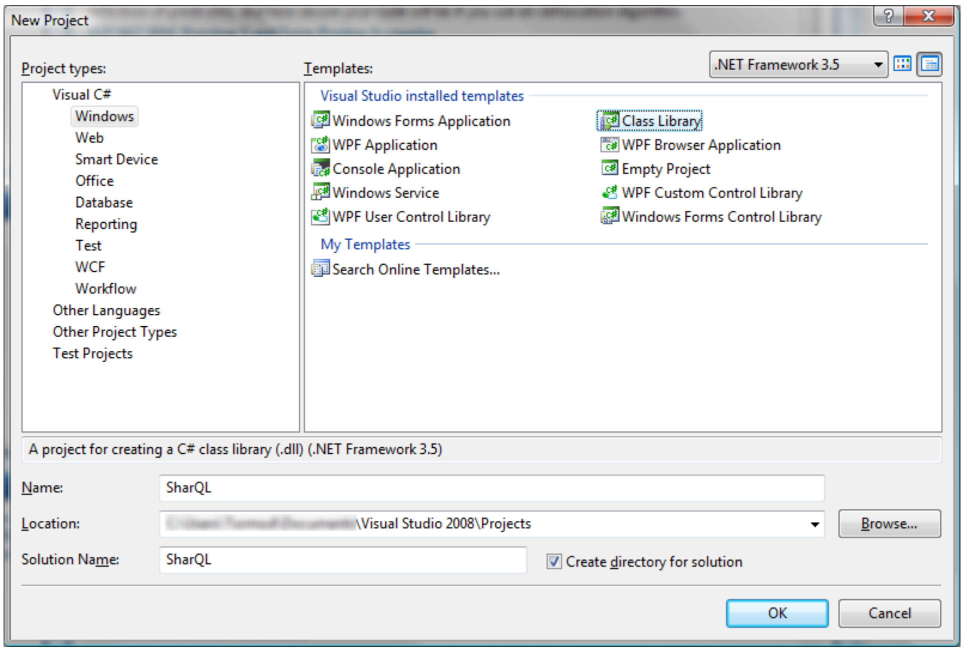
MPLex and MPPG are designed to easily integrate with Microsoft Visual Studio 2008. A Visual Studio solution can be configured to automatically generate source files for the scanner and parser, and then compile them along with the rest of the source files.

Before configuring the development environment, the following software had to be installed.

Microsoft Visual Studio 2008 The IDE used during the entire development, provided via MSDN Academic Alliance [19]. A 90-day trial version is available from `http://msdn.microsoft.com/en-us/vstudio/products/aa700831.aspx`.

Visual Studio 2008 SDK 1.1 Contains the tools MPLex and MPPG. Available from `http://www.microsoft.com/downloads/details.aspx?FamilyID=59ec6ec3-4273-48a3-ba25-dc925a45584d`.

The setup of a Visual Studio solution for autogeneration of the parser involves manual editing of the corresponding *project file*. A project file in Visual Studio is similar in semantics to a makefile in Unix development. It describes the various sub-actions performed as part of the main build action.



.NET Framework 3.5

A project for creating a C# class library (.dll) (.NET Framework 3.5)

Name: SharQL

Location: [illegible]\Visual Studio 2008\Projects

Solution Name: SharQL

[✓] Create directory for solution

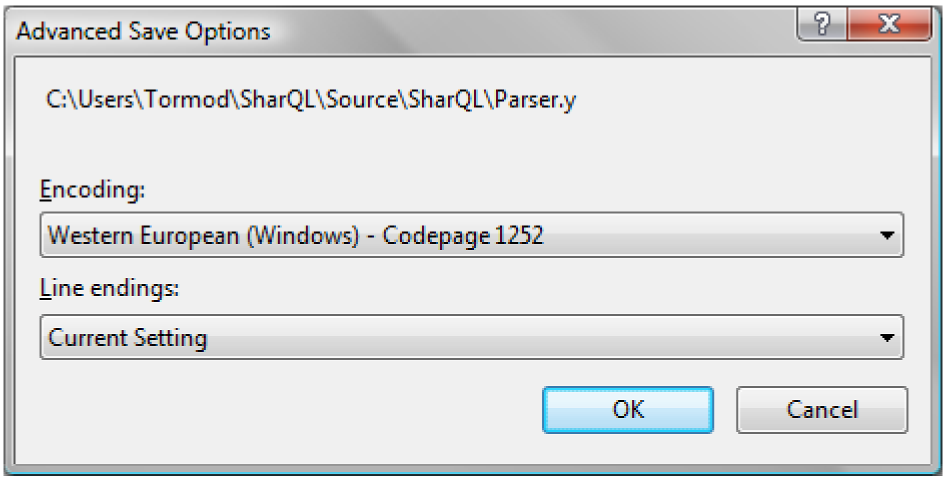
Figure 4.1: Creating the Visual Studio project

First, the Visual Studio solution with the parser project was created by clicking *File* → *New* → *Project...* and then choosing a C# Class Library, as shown in Figure 4.1.

The autogenerated file *Class1.cs* was deleted, and two new text files were added, called *Scanner.lex* and *Parser.y*. These two files were the ones that would later contain the SPARQL language specifications and dictate the generation of the parser. In order for these files to be handled properly by MPLex and MPPG, their encoding had to be set by clicking *File* → *Advanced Save Options...*. In the dialog that appeared, *Western European (Windows) - Codepage 1252* had to be set as the encoding as shown in Figure 4.2.

The scanner also requires an interface called *IErrorHandler* in order to communicate any irregularities in the input. Thus, an empty interface *IErrorHandler* was created and assigned public visibility. The solution layout now looked like shown in Figure 4.3.

At this point, the solution would build, but no parser would be generated. In order to set up MPLex and MPPG to process *Scanner.lex* and *Parser.y*, the project file had to be edited manually. To edit the project file, the project must first be unloaded. This was done by right-clicking the project root and selecting *Unload Project*. With the project unloaded, the project file can be edited by right-clicking the project root again and selecting *Edit [ProjectName].csproj* as shown in Figure 4.4.



Encoding: Western European (Windows) - Codepage 1252

Line endings: Current Setting

Figure 4.2: Choosing the right encoding

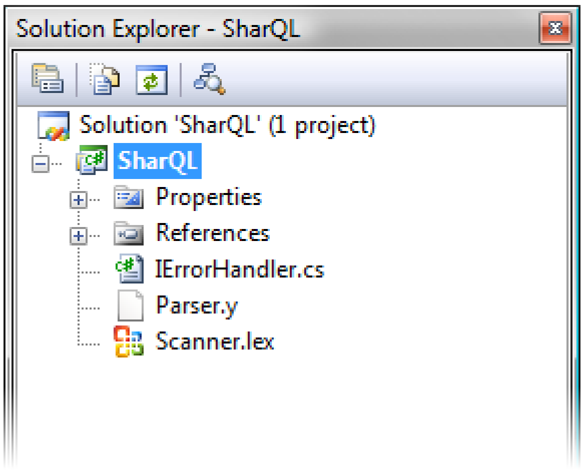


Figure 4.3: Visual Studio solution after adding initial items

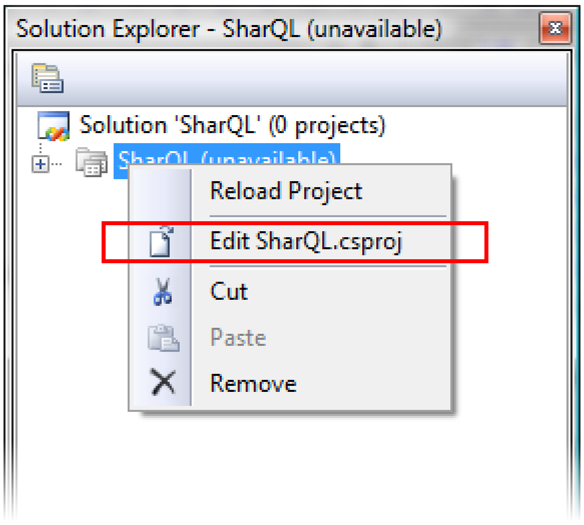
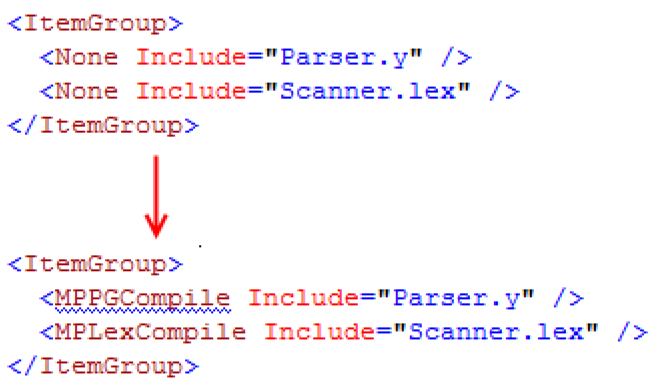

Solution Explorer - SharQL (unavailable)
Solution 'SharQL' (0 projects)
SharQL (unavailable)
Reload Project
Edit SharQL.csproj
Cut
Paste
Remove

Figure 4.4: Open the Visual Studio project file for editing

Towards the end of the project file, an ItemGroup element contained the scanner and parser specification files. Because both Parser.y and Scanner.lex was placed inside None-elements, no action would be applied to them. To let MPLex and MPPG handle these files, *Parser.y* had to be placed in an *MPPGCompile* element and *Scanner.lex* in an *MPLexCompile* element, as shown in Figure 4.5. When reloading the project by right-clicking the project root and selecting *Reload Project*, the changes made were reflected in the Properties explorer as shown in Figure 4.6; for *Parser.y*, the build action was set to *MPPGCompile* and for *Scanner.lex* it was set to *MPLexCompile*.

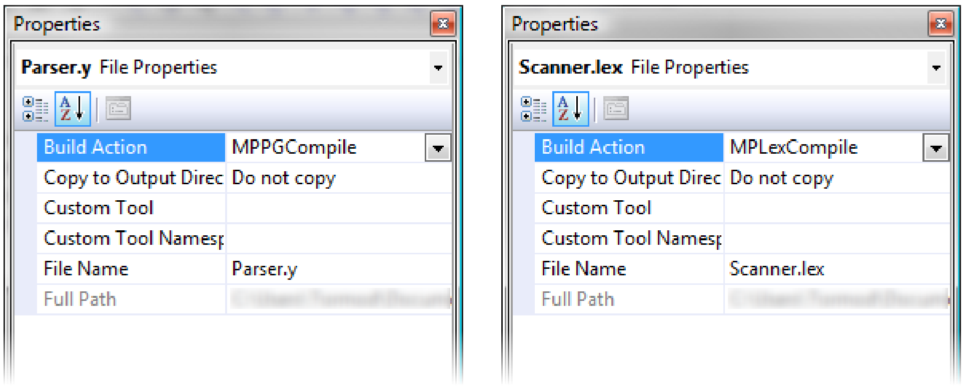
Also at this point, the solution would build, but no parser was generated. Setting the appropriate build actions was not sufficient for Visual Stu-



```
<ItemGroup>
  <None Include="Parser.y" />
  <None Include="Scanner.lex" />
</ItemGroup>
```

```
<ItemGroup>
  <MPPGCompile Include="Parser.y" />
  <MPLexCompile Include="Scanner.lex" />
</ItemGroup>
```

Figure 4.5: Actions for MPLex and MPPG


Properties

Parser.y File Properties

Build Action	MPPGCompile
Copy to Output Direc	Do not copy
Custom Tool	
Custom Tool Namesp	
File Name	Parser.y
Full Path	[redacted]

Properties

Scanner.lex File Properties

Build Action	MPLexCompile
Copy to Output Direc	Do not copy
Custom Tool	
Custom Tool Namesp	
File Name	Scanner.lex
Full Path	[redacted]

Figure 4.6: Build actions reflected in Properties explorer

```
%namespace SharQL

%%
```

Figure 4.7: An empty scanner declaration

dio to perform the actual parser generation. The assembly *Microsoft.VsSDK.Build.Tasks.dll* from the Visual Studio SDK was needed, and had to be copied into the project. A *lib* folder was created as part of the project. The assembly was added by right-clicking the new folder and selecting *Add* → *New Item...*. The file is usually located in the folder *%PROGRAMFILES%\MSBuild\Microsoft\VisualStudio\v9.0\VSSDK*.

The final step in setting up the integration of MPLex and MPPG with the Visual Studio solution was to hook up the project file to the newly added assembly. This was done by opening the project file for manual editing once again and pasting the XML from Appendix A immediately before the closing tag of the *Project* element.

Reloading the project and attempting to build the project at this point yielded an error message from MPLex: "*Parser error <syntax error, unexpected EOF>*". This was good news, as MPLex was obviously trying to parse the *Scanner.lex* file, but was unable to do so because the file was empty. Adding the code from Figure 4.7 to *Scanner.lex* fixed the error. At the same time, the code from Figure 4.8 was added to *Parser.y* to ensure a (quite useless) parser would be generated.

While this fixed the syntax error of *Scanner.lex*, three new errors arose, this time in the file *Scanner.cs*. These errors were all due to missing references. To resolve this issue, two more files had to be added to the *lib* folder of the project. *MPLex.exe* and *MPPG.exe* were copied from the *[SDK Root]\VisualStudioIntegration\Tools\Bin* folder. These are the tools

```
%namespace SharQL

%start Query

%%

Query :;

%%
```

Figure 4.8: An empty parser declaration

that generate the scanner and the parser, but they also contain some of the data types necessary for the scanner and parser to operate. With these files copied to the *lib* folder, a reference was added to each of them by right-clicking the *Reference* folder and selecting *Add Reference....* On the *Browse* tab, MPLex.exe and MPPG.exe was added as shown in Figure 4.9.

With these references resolved, the solution was now building properly. Source code for both a scanner and a parser was automatically generated, resulting in a .dll-file readily compiled to the output directory. The generated scanner and parser source code is temporarily stored in the *obj* folder during a build, and is left there for later inspection if desired.

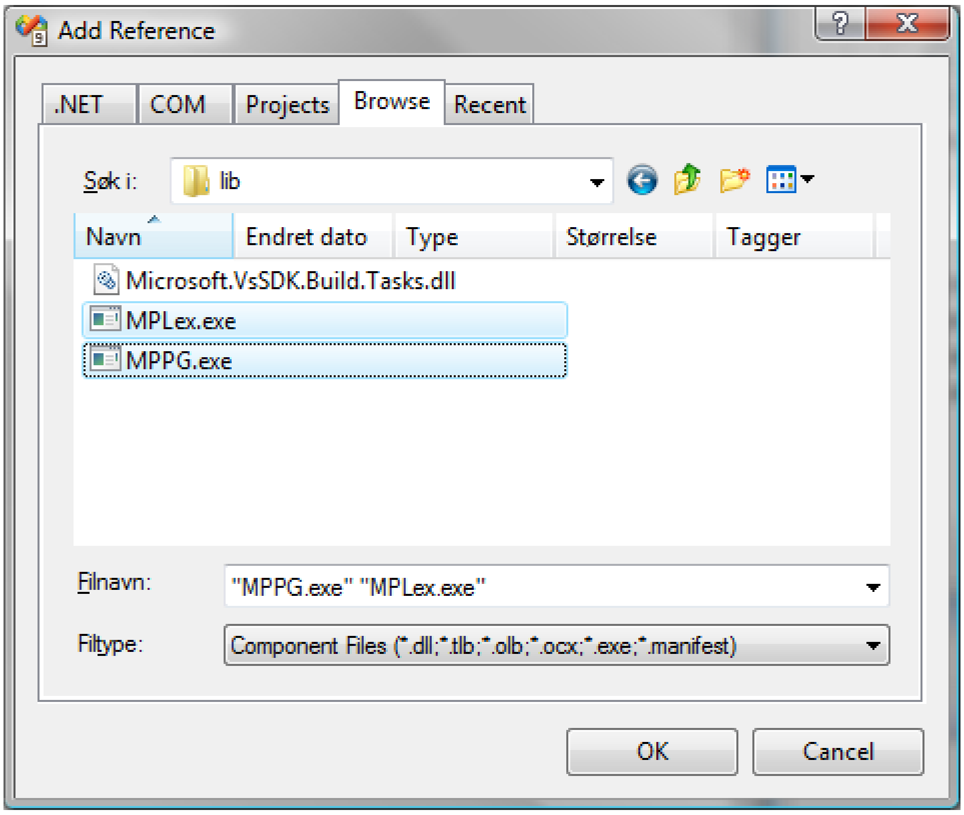
4.2 AST Preparations

MPLex and MPPG have no notion of an abstract syntax tree per se. In order for the resulting parser to produce an AST, node objects have to be created and structured as part of the semantic actions of the grammar productions. Node objects are created from node classes, and these classes constitute the AST class hierarchy. All the entities of this hierarchy reside in the *SharQL.Ast* namespace.

4.2.1 Designing the AST Class Hierarchy

At the root of the AST class hierarchy is a simple interface, *INode*, shown in Figure 4.10. This interface supports the construction of a tree structure, navigable in both directions through the *Parent* and *Children* properties. The *Accept* method is part of the *Visitor* interface, explained in Section 4.4.

Below the *INode* interface in the class hierarchy is the abstract class *NodeBase*, shown in Figure 4.11. The complete implementation of *NodeBase* is provided in Appendix D. The purpose of this class is to provide a basic implementation of the *INode* interface. All these implementations are virtual, allowing for subclasses to override them if necessary. Although this



Microsoft.VsSDK.Build.Tasks.dll
MPLex.exe
MPPG.exe

Filnavn: "MPPG.exe" "MPLex.exe"

Filtype: Component Files (*.dll;*.tlb;*.olb;*.ocx;*.exe;*.manifest)

Figure 4.9: Adding necessary references

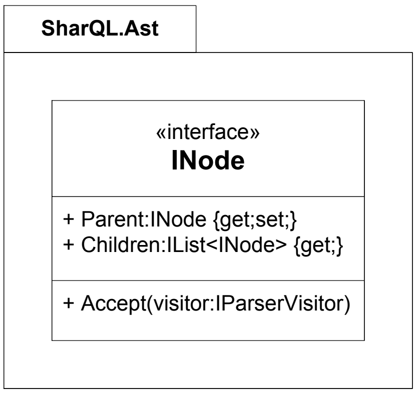

SharQL.Ast
«interface»
INode
+ Parent:INode {get;set;}
+ Children:IList<INode> {get;}
+ Accept(visitor:IParserVisitor)

Figure 4.10: The *INode* interface

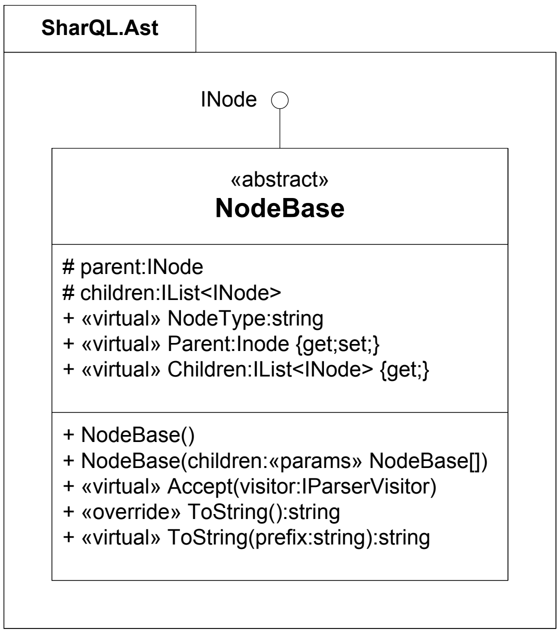

SharQL.Ast
INode
«abstract»
NodeBase
# parent:INode
# children:IList<INode>
+ «virtual» NodeType:string
+ «virtual» Parent:Inode {get;set;}
+ «virtual» Children:IList<INode> {get;}
+ NodeBase()
+ NodeBase(children:«params» NodeBase[])
+ «virtual» Accept(visitor:IParserVisitor)
+ «override» ToString():string
+ «virtual» ToString(prefix:string):string

Figure 4.11: The *NodeBase* abstract class

class has no abstract members, it is marked as abstract to prevent instantiation, and to allow for future abstract members.

The public *Parent* and *Children* properties expose the protected *parent* and *children* fields respectively. The *NodeType* read-only property is a unique identifier for each class, used as part of the Visitor pattern. The basic implementation uses reflection to retrieve the full name of the class. This property should be overridden by subclasses if performance is critical.

NodeBase provides two constructors. The default parameterless constructor makes sure the *children* field is initialized as an empty collection, while a second constructor allows for initializing the *children* field with initial *NodeBase* objects.

The *Children* property of *NodeBase* is exposed as an *IList<INode>* interface, and implemented as a custom *NodeCollection* class. This class is marked as *internal* and is thus not exposed by the API. A custom *IList* implementation was chosen over the standard ones to allow for customization.

The concrete AST classes all reside in the *SharQL.Ast.Nodes* namespace. Ultimately, they are all descendants of *NodeBase* and thus implement the *INode* interface.

```
%union { public NodeBase Value; }
```

Figure 4.12: This project's *%union* specification

```
Prologue : BaseDecl PrefixDeclList
{
  $$.Value = new PrologueNode($1.Value, $2.Value);
}
```

Figure 4.13: Building the AST via *ValueType*

4.2.2 Working With MPPG's *ValueType*

MPPG creates a struct called *ValueType* which is used for the implicit objects (`$$`, `$1`, `$2` etc.) available in the grammar productions. The *%union* construct is used in the parser specification file to specify which fields should be available in the *ValueType* struct. In this project, this *%union* construct is specified as shown in Figure 4.12, adding one single field: our AST node base type.

Without any further modifications, the AST can now be constructed as part of the semantic specification for all the productions, as shown in Figure 4.13.

Because our *ValueType* contains only one field, it would make the code more readable if the implicit objects of each production represented *NodeBase* objects rather than *ValueType* objects. Using the *operator overloading* feature of C#, it is possible to allow implicit casting between the *ValueType* and *NodeBase* types. This is done as shown in Figure 4.14. Note that *ValueType* is defined as a *partial struct*, making it trivial to extend.

With implicit casting enabled between *ValueType* and *NodeBase*, it is possible to assign a *NodeBase* object to a *ValueType* variable and vice versa.

```
public partial struct ValueType
{
  public static implicit operator NodeBase(ValueType t)
  {
    return t.Value;
  }
  public static implicit operator ValueType(NodeBase t)
  {
    return new ValueType() { Value = t };
  }
}
```

Figure 4.14: Implicit casting between *ValueType* and *NodeBase*

```
Prologue : BaseDecl PrefixDeclList
{
  $$ = new PrologueNode($1, $2);
}
```

Figure 4.15: Building the AST using implicit casting

```
list1         ::= list_item*
list2         ::= list_item+
list_item     ::= OPTIONAL_PART? REQUIRED_PART
```

Figure 4.16: EBNF sample rules

Thus, the production in Figure 4.13 could be rewritten as shown in Figure 4.15.

4.3 Specifying the Grammar

W3C specifies the SPARQL grammar using an *Extended Backus-Naur Form* (EBNF) based notation [20]. Since the MPLex and MPPG scanner and parser generators are two separate generators, the grammar must be split into a scanner part and a parser part. Further, the generators do not support specifying grammars directly using EBNF. Instead the grammar must be specified using specification languages closely resembling the BNF-based Lex and Yacc [17] specification languages.

These specification languages lack many of the extensions of EBNF, requiring several measures to be taken in the translation of the grammar.

4.3.1 EBNF and BNF

EBNF is an extended version of BNF, notably introducing the modifiers **?** (zero-or-one), ***** (zero-or-more) and **+** (one-or-more). Consider the rules in Figure 4.16. Two types of lists are specified, one that can be empty, and one that can not be empty. Further, list items have an optional part and a required part.

BNF lacks the aforementioned modifiers. Thus, to describe the same grammar using BNF, some translation is needed. Figure 4.17 shows how this is typically done (ϵ denoting the empty string). The ***** and **+** modifier from EBNF are achieved recursively in BNF, shown in the *list_tail* rule. The **?** modifier is achieved either by enumerating all legal combinations of required and optional elements, or by introducing a new non-terminal which may produce the optional part or the empty string.

```
list1         ::= list_tail
list2         ::= list_item list_tail
list_tail     ::= list_item list_tail | ϵ
list_item     ::= REQUIRED_PART | OPTIONAL_PART REQUIRED_PART

alternatively :
list_item     ::= optional_part REQUIRED_PART
optional_part ::= OPTIONAL_PART | ϵ
```

Figure 4.17: BNF translation from Figure 4.16

4.3.2 The Scanner Specification

The grammar tokens are defined in the scanner specification using regular expressions. Each token that should be returned to the parser also defines a belonging snippet of code for returning the corresponding token enumeration item recognized by the parser. Additionally, helper tokens used to build other tokens may be defined, to avoid repeating regular expressions.

Order of declaration

Token declaration order is of significance and must be taken into account. During token matching, the scanner will attempt to match the longest token possible. If there is a tie between two or more tokens, the one defined first is returned to the parser.

For the SPARQL grammar, however, the declaration order is not an issue. All tokens sharing a common prefix differ in length, thus a tie will never occur.

4.3.3 The Parser Specification

The grammar rules are declared in the parser specification. A rule defines one or more belonging productions, which are either non-empty defining a belonging snippet of code returning a token value or AST node to its superior production, or empty allowing for the production to be absent in the input string.

Grammar Constructs

The modifiers *, + and ? are absent in the BNF-based parser specification language, and must be realized as discussed in Section 4.3.1. Figure 4.18 shows a production from the SPARQL grammar specification containing a conditional grammar symbol, namely the *WHERE* keyword. In this specific case an additional production not specifying the *WHERE* keyword is

```
[13] WhereClause    ::=   'WHERE'? GroupGraphPattern

/* [13] */
WhereClause
: WHERE GroupGraphPattern
{
  $$ = new WhereClauseNode($2);
}
| GroupGraphPattern
{
  $$ = new WhereClauseNode($1);
}
;
```

Figure 4.18: SPARQL Grammar Conditional Symbol Sample

specified. The bracketed numbers correspond to the rule numbers in the W3C SPARQL grammar specification.

Declaration of lists of grammar symbols (one-or-more and zero-or-more instances) requires the introduction of a replacement rule, allowing for a recursive list of grammar symbols to be constructed during parsing. Figure 4.19 shows the realization of the *DataSetClause** zero-or-more instances list using the recursive list rule *DataSetClauseList*. The rule may result in an empty production (zero instances) or a production consisting of the non-terminals *DataSetClause* and *DataSetClauseList* (more instances).

Conflicts

MPPG creates shift-reduce parsers, consisting of a stack holding grammar symbols and an input buffer holding the rest of the input string to be parsed. The parser shifts input symbols onto the stack until it is ready to reduce a string of grammar symbols on the top of the stack into a superior grammar production. This process is repeated until an error is detected or until the stack contains the predefined grammar start symbol and the input is empty. [11]

Two types of conflicts may occur during shift-reduce parsing: shift/reduce conflicts and reduce/reduce conflicts. Shift/reduce conflicts occur when the parser cannot decide whether to shift input symbols onto the stack or to reduce grammar symbols on top of the stack. Reduce/reduce conflicts occur when the parser cannot decide which superior production to reduce the grammar symbols on top of the stack into. The former conflict typically occurs as a result of an ambiguous grammar, and is critical. The latter easily occurs when creating the parser specification from a grammar specification like the SPARQL grammar specification, because of the required rewriting from an EBNF-based grammar to a BNF-based gram-

```
[6] ConstructQuery    ::=   'CONSTRUCT' ConstructTemplate
                            DatasetClause* WhereClause
                            SolutionModifier

/* [6] */
ConstructQuery
: CONSTRUCT ConstructTemplate DatasetClauseList WhereClause
  SolutionModifier
{
  $$ = new ConstructQueryNode($2, $3, $4, $5);
}
;

DatasetClauseList
: DatasetClause DatasetClauseList
{
  $$ = new DatasetClauseListNode($1, $2);
}
| /* empty */
{
  $$ = null;
  Debug.WriteLine("Empty DataSetClauseList");
}
;
```

Figure 4.19: SPARQL Grammar List Symbol Sample

```
/* [2] */
Prologue
: BaseDecl
{
  $$ = new PrologueNode($1, null);
}
| BaseDecl PrefixDeclList
{
  $$ = new PrologueNode($1, $2);
}
| PrefixDeclList
{
  $$ = new PrologueNode(null, $1);
}
|
{ /* empty */ }
;

PrefixDeclList
: PrefixDecl PrefixDeclList
{
  $$ = new PrefixDeclListNode($1, $2);
}
|
{ /* empty */ }
;
```

Figure 4.20: SPARQL Grammar Reduce/Reduce Sample

mar.

Figure 4.20 addresses a reduce/reduce conflict encountered in translating the SPARQL grammar specification into parser grammar productions. In this specific case, both the *Prologue* and *PrefixDeclList* rules contain an empty production. In the context of a *Prologue* grammar rule, whenever the parser encounters an empty input string, it cannot decide whether to reduce the empty production to a *PrefixDeclList* rule or to a *Prologue* rule.

This reduce/reduce conflict may be prevented simply by removing the empty production belonging to the *Prologue* rule, since a *Prologue* rule may still produce an empty production via the *PrefixDeclList* rule.

4.4 The Visitor Pattern

The basic idea of the *Visitor* pattern is to separate the algorithm from the data structure. The Visitor pattern allows for defining new operations on the elements of an object structure, without changing the classes of the elements on which it operates. [21]

The *Visitor* pattern is commonly used to traverse and perform opera-

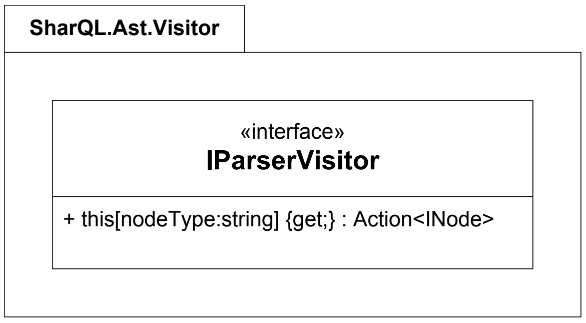


Figure 4.21: The *IParserVisitor* interface

```
public virtual void Accept(IParserVisitor visitor)
{
  foreach (INode child in children)
  {
    if (child != null)
      child.Accept(visitor);
  }
  visitor[NodeType](this);
}
```

Figure 4.22: Basic implementation of visitor pattern from *NodeBase*

tions on abstract syntax trees. Each node in the ASTs produced by SharQL has an *Accept* method which takes a single *IParserVisitor* object. The IParserVisitor interface is shown in Figure 4.21.

The only member of this interface is an indexer which takes a string identifier as input and returns an *Action<INode>* delegate. A delegate is essentially a type-safe method pointer, and the Action<INode> delegate may point to any method which accepts a single *INode* argument and returns *void*. The AST node that accepts a visitor will access this indexer, provide its *NodeType* string as the identifier and invoke the delegate it gets in return, providing a reference to itself. An implementation snippet of this is shown in Figure 4.22. Note that the visitor will traverse the tree in a depth-first manner; before being applied to a node, it is applied to any child nodes.

4.4.1 Visitor Example

Consider a sample AST node as shown in Figure 4.23. A visitor implementation that traverses the AST and performs a task on all instances of this class is shown in Figure 4.24.

```
public class SampleNode : NodeBase
{
  public static readonly string SampleNodeType
    = "SampleNodeIdentifier";

  public override string NodeType
  {
    get { return SampleNode.SampleNodeType; }
  }
}
```

Figure 4.23: Sample AST node

```
public class ParserVisitor : IParserVisitor
{
  public Action<INode> this[string nodeType]
  {
    get
    {
      if (nodeType == SampleNode.SampleNodeType)
        return VisitSampleNode;
      else
        return NoOp;
    }
  }

  public void VisitSampleNode(INode node)
  {
    SampleNode n = node as SampleNode;
    if (n != null)
    {
    // Do something
    }
  }

  public void NoOp(INode node)
  {
  // Ignore unrecognized node
  }
}
```

Figure 4.24: Sample visitor implementation

4.5 Error Handling

The purpose of error handling in the context of the SharQL parser, is identification and reporting of error conditions concerning the syntactical correctness of a SPARQL input query. The scanner basically looks for defined tokens in the input stream and passes them on to the parser, which in turn puts a set of tokens in the context of a defined production rule. On the occurrence of an undefined, missing or unexpected token, an error condition is present.

4.5.1 Traditional Parser Error Handling

In a traditional scanner-parser combination like Lex and Yacc [17], either the scanner immediately reports an error on the occurrence of an unexpected input stream character, or the character is passed on to the parser, possibly in the form of a special error token, which is in turn reported as unexpected by the parser. As the scanner only knows what characters to expect on a per-token basis, the informational value of a scanner error report is limited compared to a parser error report. The parser not only knows which unexpected token that was encountered, but reports back what it expected as well.

Combining token location information from the scanner with token context information from the parser would offer an improved informational value for the error reports, and is the approach taken for the SharQL parser. Achieving such error handling requires some further extensions to both the scanner and parser.

4.5.2 Adapting the Grammar Definition

Section 4.3 describes the specification of the scanner and parser grammars. To support error handling, the scanner grammar definition must be extended further.

After translating the W3C SPARQL grammar, undefined tokens encountered during scanning of the input stream were simply ignored, and were not reported to the parser. Figure 4.25 shows the additional token definition needed for handling identification of undefined tokens. The token definition simply matches any single character, and returns the character itself to the parser. It is important to place the token definition below all other regular token definitions in the scanner grammar definition. This is the only way to have all valid single-character token definitions remain unaffected, as described in Section 4.3.2.

```
.           { return (int)yytext[0]; }
```

Figure 4.25: Scanner grammar definition extension

```
%YYLTYPE CustomLexLocation
```

Figure 4.26: Specifying the Token Location Type

4.5.3 Tracking Token Locations

When the scanner matches tokens from the input stream, token location information is available for every token matched. Even undefined tokens will get matched thanks to the adaption of the grammar as described in Section 4.5.2. This means that location information will be available for each and every token, defined or undefined, passed on to the parser.

Location information is available through global variables in the context of the scanner. The information of interest is stored in the integer variables *yyline*, *yycol* and *yyleng*. These values represent the *line*, *column* and *length*, respectively, of the currently identified token.

A special variable *yylloc* also exists, that holds all the location information of the current token. When this variable is set, it may be retrieved by the parser for further use. The type of this variable is defined in the parser specification using the *%YYLTYPE* operator, as shown in Figure 4.26. The type used by SharQL for storing the token location information is *CustomLexLocation*, a custom class replacing the *LexLocation* class supplied with MPPG. The only reason a custom class is needed is due to a bug in the *LexLocation* class occuring when two instances are merged using the belonging *Merge* method. Our *CustomLexLocation* class simply corrects this bug, and otherwise acts just as the *LexLocation* class would, as shown in Figure 4.27. When the parser reduces tokens on the stack into a production, the *CustomLexLocation* objects for all the tokens involved are merged to represent the location information for the production as a whole.

The scanner specification is altered to call a *LoadYylval* method every time a token is identified. This method constructs an instance of the *CustomLexLocation* class, supplying it with the necessary location information available from the scanner, and assigns it to the *yylloc* variable, making it available to the parser, as shown in Figure 4.28.

4.5.4 Collecting Errors

When the parser encounters an unexpected token or when an expected token is missing, it calls the *yyerror* method on the scanner instance. An empty *yyerror* method is already declared, which is overridden in the scan-

```
public class CustomLexLocation : IMerge<CustomLexLocation>
{
  public int sLin; // Start line
  public int sCol; // Start column
  public int eLin; // End line
  public int eCol; // End column

  public CustomLexLocation()
  {
  }

  public CustomLexLocation(int sl, int sc, int el, int ec)
  {
    sLin=sl;
    sCol=sc;
    eLin=el;
    eCol=ec;
  }

  public CustomLexLocation Merge(CustomLexLocation last)
  {
    // This part is missing from the MPPG LexLocation class
    if (last == null)
    {
      return this;
    }

    return new CustomLexLocation(sLin, sCol, last.eLin,
      last.eCol);
  }
}
```

Figure 4.27: *CustomLexLocation* class replacing the *LexLocation* class.

```
  // Called each time a token has been returned.
  internal void LoadYylval()
  {
    // Collects token location information and makes it available
    // for the parser through the yylloc variable.
    yylloc = new CustomLexLocation(yyline, yycol, yyline, yycol
      + yyleng);
  }
```

Figure 4.28: Tracking Token Location

```
// Called by the parser on the occurrence of an unexpected or
// missing token.
public override void yyerror(string s, params object[] a)
{
  // Adds the reported error to the error handler.
  handler.AddError(s, new MPLEX.Parser.LexSpan(tokLin, tokCol,
    tokLin, tokECol, tokPos, tokEPos, new ScanBuffProxy(buffer)));
}
```

Figure 4.29: *yyerror* method.

ner specification. This method is used to collect the errors reported.

The generated scanner provides a property for supplying an error handler, which must implement an *IErrorHandler* interface. The interface itself is not defined, it is just referenced. Thus, an *IErrorHandler* interface has been defined, as decribed in Section 4.1.

MPPG supplies an *ErrorHandler* class providing basic error handling. For some reason, this class does not implement the *IErrorHandler* interface referenced by the scanner. In order to re-use the existing functionality in compliance with the scanner, a custom *SharQLErrorHandler* class inheriting the *ErrorHandler* class and implementing the *IErrorHandler* interface is used.

Figure 4.29 shows the *yyerror* method which uses the error handler to collect errors reported. The arguments to the *AddError* method includes an error message generated by the parser and an *MPLEX.Parser.LexSpan* object representing the location of the error as well as the input stream buffer in which the error token is located.

The errors collected are available through a collection property on the *Parser* facade class, as described in Section 4.7.

4.6 Handling Escape Sequences

The chapters A.2 and A.7 of the SPARQL specification[1] describes codepoint escape sequences and common escape sequences respectively. An escape sequence is merely a substitute representation of an actual character. In SPARQL, all escape sequences start with a backslash (\) followed by en escape specifier.

The processes of handling escape sequences are described in the next sections and the invocation of these processes are made by the *Parser* facade class described in Section 4.7.

4.6.1 Codepoint Escape Sequences

The Unicode standard specifies unique identifiers for an inconceivable amount of characters and symbols. These positive integer identifiers are

Escape	Codepoint
`\u`**`xxxx`**	A Unicode codepoint in the range [0,FFFF_{16}] corresponding to the hexadecimal value (**`xxxx`**)
`\U`**`xxxxxxxx`**	A Unicode codepoint in the range [0,$\mathrm{10FFFF}_{16}$] corresponding to the hexadecimal value (**`xxxxxxxx`**)

Table 4.1: Codepoint escape sequence formats supported by SPARQL [1]

```
<ab\u00E9xy> # 00E9₁₆ is Latin small e with acute - <abéxy>
\u03B1:a     # 03B1₁₆ is Greek small alpha - α:a
a\u003Ab     # 003A₁₆ is colon - a:b
```

Figure 4.30: Usage examples of codepoint escape sequences

commonly referred to as *Unicode codepoints* [22]. Anywhere in a SPARQL query, a codepoint escape sequence may occur as a substitute for the character represented by that codepoint. Such an escape sequence can take two forms, as shown in Table 4.1. Figure 4.30 shows examples of codepoint escape sequence usage. Note that this implementation only supports codepoints in the range [0,FFFF_{16}] in both forms, for the reasons explained in section 6.3.1.

Given that these codepoint escape sequences may occur anywhere in a SPARQL query, they should be processed and resolved before the query is being parsed. A preprocessor was developed to analyze a string input, replacing all occurrences of codepoint escape sequences with the actual character corresponding to the codepoint.

Two methods for resolving codepoint escape sequences were considered: regular expression (regex) transformation and manual analysis. While a regex transformation would result in less code, the regex pattern would be rather complex and result in poor performance compared to manual string analysis. Thus, the latter method was chosen. The algorithm used is roughly summarized in Figure 4.31.

The actual implementation was made in the method *ResolveCodepoint-EscapeSequences* of the class *SharQL.Utils.EscapeSequenceResolver*. *Argument-Exception*s are thrown if the codepoint is not recognized. Escape sequences that do not start with `\u` or `\U` are ignored and left for later processing, described next.

4.6.2 Common Escape Sequences

Common escape sequences are shorthand notations for the most common codepoint escape sequences and may only occur within string literals of a SPARQL query. Eight different common escape sequences are available, as

```
1. Create an empty string S.
2. For each character in input string I:
   2.1. If not a backslash, append character to S.
   2.2. If backslash, read next character:
        2.2.1 If lowercase 'u', read next 4 characters,
              resolve codepoint and append to S.
        2.2.2 If uppercase 'U', read next 8 characters,
              resolve codepoint and append to S.
        2.2.3 Else, append backslash and character to S
              for later processing.
3. Return S.
```

Figure 4.31: Algorithm for processing codepoint escape sequences

Escape	**Corresponding codepoint escape**
`\t`	`\u0009` (tab)
`\n`	`\u000A` (line feed)
`\r`	`\u000D` (carriage return)
`\b`	`\u0008` (backspace)
`\f`	`\u000C` (form feed)
`\"`	`\u0022` (quotation mark, double quote mark)
`\'`	`\u0027` (apostrophe-quote, single quote mark)
`\\`	`\u005c` (backslash)

Table 4.2: Common escape sequences supported by SPARQL [1]

shown in Table 4.2.

Given that these escape sequences may only occur within string literals, they are most easily processed after parsing is completed. An algorithm similar to the one from Figure 4.31 was implemented in the method *ResolveCommonEscapeSequences* of the class *SharQL.Utils.EscapeSequenceResolver*.

To traverse the resulting AST and resolve all common escape sequences in string literals, a visitor class was developed. The class *SharQL.Ast.Visitor.EscapeSequenceResolverVisitor* implements the interface *IParserVisitor* and applies the aforementioned method for resolving common escape sequences to all string literals. An extract of the class is shown in Figure 4.32.

4.7 The Parser Facade Class

The MPLex and MPPG tools generate a scanner and a parser, respectively. In order for the end-user to perform any parsing, a scanner and a parser have to be instantiated and the parser has to be aware of the existence of the scanner instance in order to read its output. The scanner, in turn, needs to know what input to tokenize.

```
public Action<INode> this[string nodeType]
{
  get
  {
    if (nodeType == new StringLiteralNode().NodeType)
      return resolveEscapeSequence;
    else
      return noOp;
  }
}

private void resolveEscapeSequence(INode node)
{
  StringLiteralNode n = node as StringLiteralNode;
  if (n != null)
  {
    string escapedString = n.Value;
    string resolvedString = Utils.EscapeSequenceResolver.
        ResolveCommonEscapeSequences(escapedString);
    n.Value = resolvedString;
  }
}

private void noOp(INode node)
{
  // Ignore
}
```

Figure 4.32: Extract from *SharQL.Ast.Visitor.EscapeSequenceResolverVisitor*

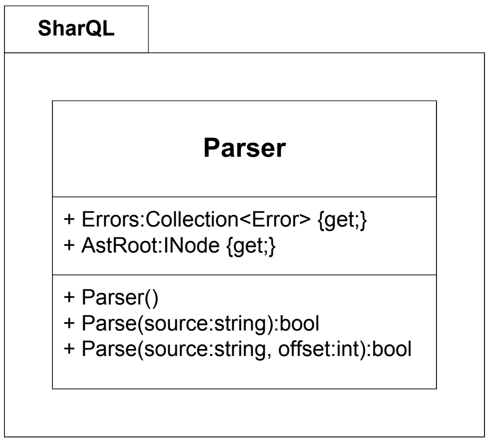

SharQL
Parser
+ Errors:Collection<Error> {get;}
+ AstRoot:INode {get;}
+ Parser()
+ Parse(source:string):bool
+ Parse(source:string, offset:int):bool

Figure 4.33: The *Parser* facade class

To encapsulate all this plumbing, a parser facade class, as shown in Figure 4.33, was developed. The intention of this class is to let the end-user instantiate it once, and make successive calls to one of its *Parse* methods without needing to have a notion of a scanner at all. When a parsing succeeds, the resulting AST is made available through the *AstRoot* property. Additionally, the parser facade reports any errors that may occur, through its *Errors* property.

Naming the facade class *Parser* for clarity was highly desirable. However, this crashed with the name that MPPG by default assigns its generated parser. Our solution to this was to modify the parser specification file by including a *%parsertype* specifier, thus renaming the resulting type of the parser class.

A UML sequence diagram showing parser facade class operation during parsing is shown in Figure 4.34.

4.8 Creating the SharQL Test Client

The intention of the SharQL Test Client is to have easy access to the SharQL parser library at any given time, and to be able to visualize the abstract syntax trees produced by the parser.

Given that SharQL is a .NET library, *Windows Forms* was chosen as platform for the test client. Visual Studio has great Windows Forms support, and creating graphical user interfaces is easily accomplished. So, a new Windows Forms project was created inside the SharQL solution, and the references required by SharQL where added, as shown in Figure 4.35.

The user interface was then designed as shown in Figure 4.36. The idea is that, once the query is parsed, the AST is displayed in the *TreeView* control to the left. When one of the nodes in the tree view is selected, details for

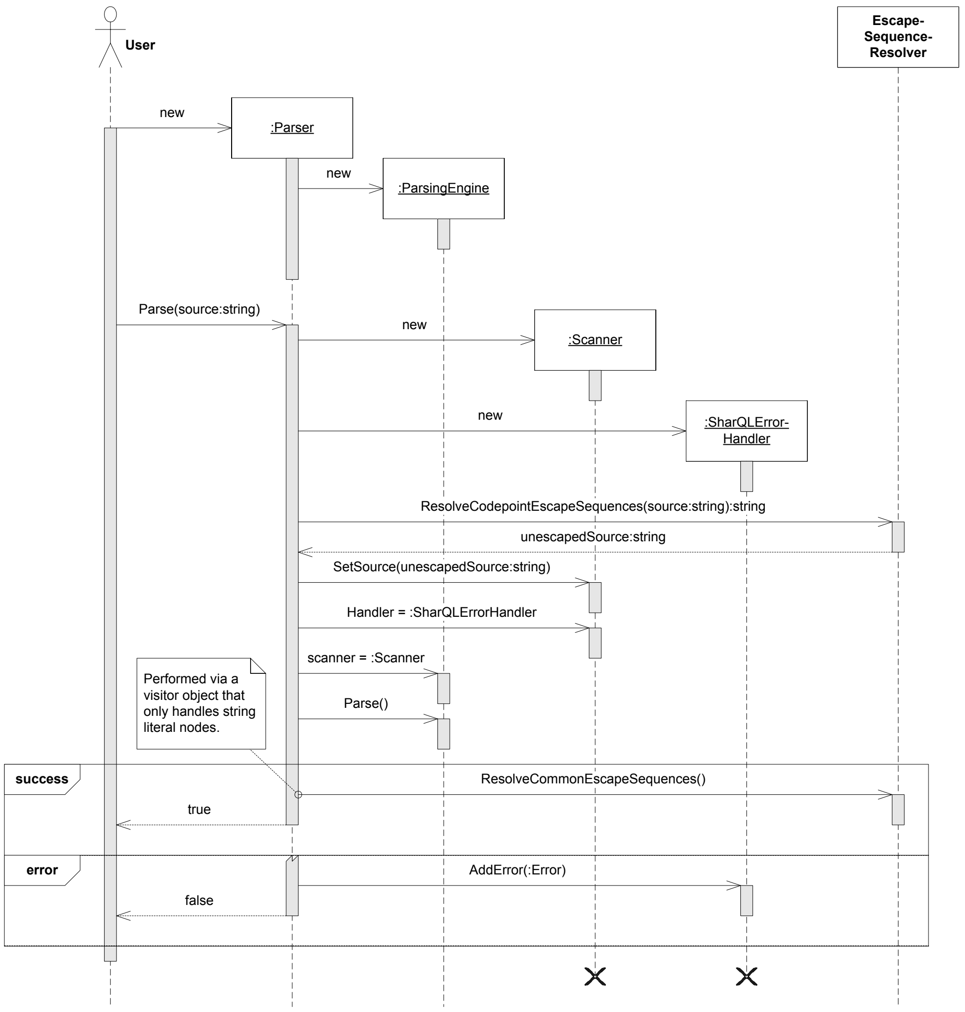

User
Escape-Sequence-Resolver
new
:Parser
new
:ParsingEngine
Parse(source:string)
new
:Scanner
new
:SharQLError-Handler
ResolveCodepointEscapeSequences(source:string):string
unescapedSource:string
SetSource(unescapedSource:string)
Handler = :SharQLErrorHandler
scanner = :Scanner
Parse()
Performed via a visitor object that only handles string literal nodes.
success
ResolveCommonEscapeSequences()
true
error
AddError(:Error)
false

Figure 4.34: UML sequence diagram showing parser facade class operation

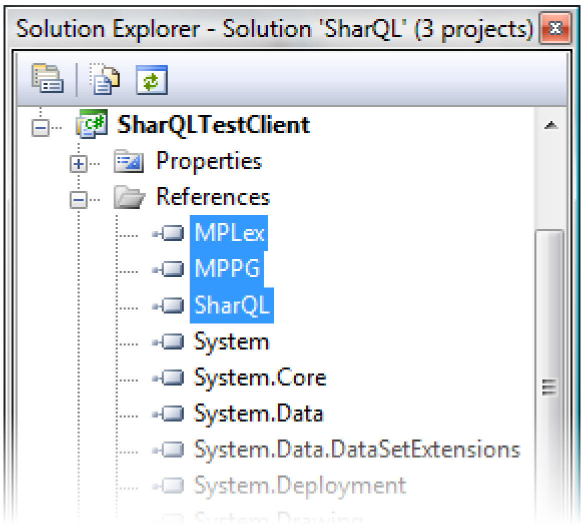


Figure 4.35: Necessary references for the test client

that node is displayed in the text box to the right. Any errors reported by the parser is shown in the lower pane.

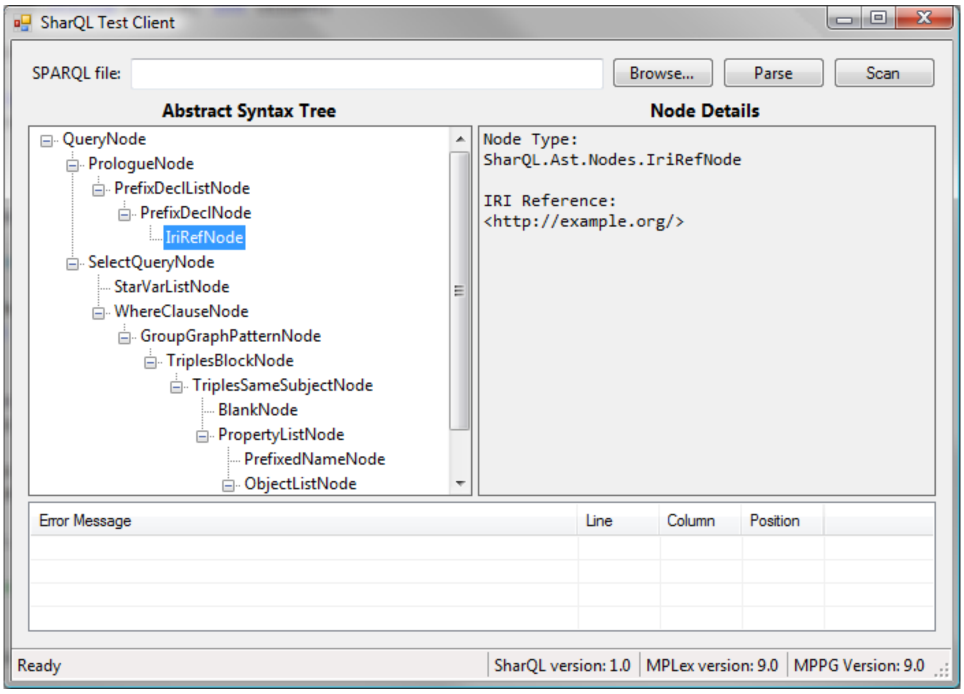
The test client can also perform a scan-only operation on the input file. By clicking the *Scan* button instead of the *Parse* button, the input is scanned and the stream of tokens produced by the scanner is displayed in a separate window.

Visualizing the abstract syntax tree is a matter of translating it to a tree of *TreeNode* objects. Once such a tree is created, it can be assigned to the *Nodes* property of the tree view in the test client.

Translating a node from the AST to a TreeNode object was done using the Visitor pattern explained in Section 4.4. Using this pattern, the translation was very simple. The *TreeBuilderVisitor* class first creates dictionary mapping INode objects to TreeNode objects. Remember that the visitor is traversing the tree depth-first. This means that when the visitor is applied to a node, it is safe to assume that the visitor has already been applied to all its child nodes.

Our visitor will not distinguish between different types of nodes. Thus, the same delegate is returned through the visitor indexer, regardless of the node type string provided. The implementation of the method represented by this delegate is shown in Figure 4.37. The TreeNode constructor accepts a textual caption as parameter. In this case, the class name is obtained from the NodeType property, assuming its default behavior of returning the full name of the class.

By assigning a reference to the actual AST node to the *Tag* property of the corresponding TreeNode object, detailed information about the node



```
QueryNode
  PrologueNode
    PrefixDeclListNode
      PrefixDeclNode
        IriRefNode
  SelectQueryNode
    StarVarListNode
    WhereClauseNode
      GroupGraphPatternNode
        TriplesBlockNode
          TriplesSameSubjectNode
            BlankNode
            PropertyListNode
              PrefixedNameNode
              ObjectListNode
```

```
Node Type:
SharQL.Ast.Nodes.IriRefNode

IRI Reference:
<http://example.org/>
```

Figure 4.36: The test client user interface

```
private void constructTreeNode(INode node)
{
  NodeBase nb = node as NodeBase;
  if (nb != null)
  {
    // TreeNodes maps INode objects to TreeNode objects
    TreeNodes[node] = new TreeNode(nb.NodeType.Split('.').Last());
    TreeNodes[node].Tag = node;
    foreach (INode child in nb.Children)
    {
      if (child != null && TreeNodes.ContainsKey(child))
      {
        TreeNodes[node].Nodes.Add(TreeNodes[child]);
      }
    }
  }
}
```

Figure 4.37: Code snippet from *TreeBuilderVisitor*

may be obtained later. Such information is needed when a node is selected. Then, a detailed description of the node is created and displayed in the right half of the user interface.

Chapter 5

Testing

The notion of testing is important in all software development projects. After all, without doing any kind of testing, one has no means of assessing whether the software works as it should. There are numerous ways of testing software. Black box testing is one way of performing tests, where the system as a whole is tested and its behavior compared to a reference of how it *should* behave. Another popular method is unit testing.

5.1 Unit Testing

Unit testing is a method of individually testing small parts, or units, of a larger system. Unit tests are usually based on a specification of how these units should behave. Often, the tests are even written before the work on implementing the unit has even begun. This is possible by defining interfaces that the units have to implement, and write tests against those interfaces.

It is important to write tests that cover as much as possible of the specification of how the unit should behave. After all, the intention of testing the behavior of the small units is to rule out subtle errors originating from a faulty part of a bigger system. By assuring that the individual units behave according to the specification, debugging integration errors in a later phase becomes more comprehensible.

Sometimes, the units are easily identified in the system specification. Otherwise, these units are usually identified and specified in the design phase. The SPARQL specification from W3C specifies the language, but does not consider the implementation of parsers. In this project, *MPLex* and *MPPG* generates a scanner and a parser, respectively. Thus, it is natural to consider the scanner as one unit and the parser as another unit. Testing the parser is a matter of feeding it with queries and to see that it parses those that are valid and refuses to parse queries that contain errors. Testing the scanner is a matter of verifying that it lexically separates queries into the

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

(...)

[TestClass]
public class MyUnitTests
{
  (...)
}
```

Figure 5.1: Creating a test class

```
[TestClass]
public void MyUnitTest1()
{
  MyClass myObj = new MyClass();
  myObj.DoSomething(1, 2, 3);
  Assert.IsTrue(myObj.SomeProperty == 5);
  Assert.IsFalse(myObj.AnotherProperty > 0);
}
```

Figure 5.2: Creating a test method

appropriate tokens in the correct order.

5.2 Automated Testing in Visual Studio 2008

Visual Studio 2008 has integrated support for automated testing. By adding a *Test Project* to a Visual Studio solution, one can write tests in one of several programming languages and benefit from automatic execution and reporting based on the results.

To create unit tests in a test project, start by creating a public class and tag it with the *TestClass* attribute, as shown in Figure 5.1. The using statement shown resolves the namespace where the TestClass attribute resides. When Visual Studio is instructed to run the tests of the test projects, it uses reflection to decide which of the classes that are test classes.

Within a test class, an arbitrary number of test methods can be defined. These are methods that perform specific tasks and assert that the states and output of the tested objects are as expected. Similar to test classes, test methods require a *TestMethod* attribute. Assertions are performed using one of several static methods of the *Assert* class. An example test method is shown in Figure 5.2. This example creates an object and performs a specific operation to alter its state. Successively, the test performs two assertions on its state, according to some specification.

The static *Assert* class provides test authors with several assertions meth-

Method name	**Description**
Are[Not]Equal	Verifies whether or not two objects are equal.
Are[Not]Same	Verifies whether or not two object variables refer to the same object.
Is[Not]InstanceOfType	Verifies whether or not an object is an instance of a specific type.
Is[Not]Null	Verifies whether or not an object is *null*.
IsFalse	Verifies that a condition is false.
IsTrue	Verifies that a condition is true.
Fail	Unconditionally fails the test.
Inconclusive	Indicates that an assertions can not be proved either true or false.

Table 5.1: Methods available in *Assert* class

```
SELECT *
WHERE { }
```

Figure 5.3: The *syntax-basic-01* test from W3C

ods. These methods are summarized in Table 5.1. Most of the assertion methods are overloaded in order to support various data types and/or to let test authors provide optional error messages for the test reports. Where the test names in the table contains a "[Not]" part, a pair of methods exist, one containing "Not" and one that does not.

Visual Studio offers several ways of organizing tests and the execution of tests. For small projects like this one, however, it is typically desirable to run all the tests. This is done by clicking *Test → Run → All Tests in Solution*.

5.3 W3C SPARQL Test Suite

The *Data Access Working Group* of W3C has released a set of test cases, including syntax tests for the SPARQL language [2]. All in all, the test suite contains 199 SPARQL queries which should either parse or fail to parse. These tests are listed in Appendix E. While such tests do not fully test all aspects of a SPARQL parser, they constitute a solid foundation.

The tests vary from simple select queries as shown in Figure 5.3, to more complex queries as the one shown in Figure 5.4. Invalid queries, as in Figure 5.5, are also provided. Together, these queries cover most of the syntax that a SPARQL parser could encounter.

The test queries do not specify how a resulting AST should look like. After all, the structure of an AST depends on the application. Thus, it is

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
CONSTRUCT { [] rdf:subject ?s ;
               rdf:predicate ?p ;
               rdf:object ?o }
WHERE {?s ?p ?o}
```

Figure 5.4: The *syntax-form-construct03* test from W3C

```
SELECT * WHERE {[] . }
```

Figure 5.5: The *syn-bad-bnode-dot* test from W3C

not possible to verify the correctness of the ASTs produced by our parser, based on these tests alone. The tests neither provide any means of testing the scanner isolatedly. Such tests have to be authored by manually identifying the token stream that should be produced. Once again, this is a natural constraint, given that the tokens chosen, like the AST, depends on the application.

5.3.1 Omission of Semantic Tests

Some of the syntax tests from the test suite actually test semantic matters. Table 5.2 lists the tests in question. All the tests are negative tests supposed to fail, but due to the current state of the parser, they all succeed. The reason why they should fail, is that the same blank node label is used across basic graph patterns, as shown in Figure 5.6. The parser is not supposed to care about semantics in this phase, and is for that reason handling the tests in questions as it should.

The fact that the tests listed in Table 5.2 are in fact semantic tests as well as syntactic tests, makes them unsuitable for testing the parser. For this reason, these tests have been omitted from the set of tests run when testing the parser. Of the total of 199 tests supplied, this leaves 188 tests that are still performed.

```
PREFIX : <http://xmlns.com/foaf/0.1/>

ASK { _:who :homepage ?homepage
      GRAPH ?g { ?someone :made ?homepage }
      _:who :schoolHomepage ?schoolPage }
```

Figure 5.6: The *syn-blabel-cross-graph-bad* test from W3C, semantically incorrect due to the use of the blank node *who* across basic graph patterns.

Test File	**Reason for failing**
syntax-sparql3\syn-blabel-cross-graph-bad.rq syntax-sparql3\syn-blabel-cross-optional-bad.rq syntax-sparql3\syn-blabel-cross-union-bad.rq	Blank node *who* reused across basic graph patterns
syntax-sparql4\syn-bad-34.rq syntax-sparql4\syn-bad-35.rq syntax-sparql4\syn-bad-36.rq syntax-sparql4\syn-bad-37.rq syntax-sparql4\syn-bad-38.rq syntax-sparql4\syn-bad-graph-breaks-BGP.rq syntax-sparql4\syn-bad-opt-breaks-BGP.rq syntax-sparql4\syn-bad-union-breaks-BGP.rq	Blank node *a* reused across basic graph patterns

Table 5.2: Failing Semantic Tests Omitted From the W3C SPARQL Test Suite [2].

5.4 Automating the Test Suite

Using the integrated support for automated testing in Visual Studio 2008, the W3C SPARQL Test Suite may be completely automated. The running of the test suite is essential in determining the parser's level of conformance with the SPARQL grammar specification, and because of the great importance of running such tests, this should be as easy to carry out as possible.

The goal of automating the test suite is to be able to run all the tests in a single operation, and to receive a report stating the result of each and every test. The test suite contains 188 syntax tests relevant for the parser, each of which must be implemented by a corresponding test method as shown in Figure 5.2. The tests only differ in which syntax test file to parse. Thus, a great amount common code was identified, extracted and generalized to build a small framework for automating this specific test suite.

Using reflection features available in the C# programming language, a method is able to determine the name of the caller method. Figure 5.7 shows the code snippet necessary for accessing the caller stack and fetching the second top-most stack frame to extract the caller method name. By naming each test method according to the corresponding test syntax file and having each test method call a common method, this latter method may determine which test syntax file to load and can carry out the test on its own.

Naming the test methods according to the file names of the test syntax files supplied in the W3C SPARQL Test Suite is advantageous since the corresponding test syntax file for a failing test can very quickly be determined. In addition, since the test syntax file may be derived from the test method's name, the code necessary for calling the common test method from each

```
StackFrame stackFrame = new StackFrame(1, true);
MethodBase methodBase = stackFrame.GetMethod();
string methodName = methodBase.Name;
```

Figure 5.7: C# Caller Method Name Extraction

```
[TestMethod]
public void sparql1__basic_01() { Assert.IsTrue(
  ParseQueryFile("syntax")); }
```

Figure 5.8: Test Method Sample

test method may be reduced to a minimum. Figure 5.8 shows an actual test method calling the common *ParseQueryFile* method. The test method name is actually a composition of the test set name followed by the specific test name. Thus, *sparql1__basic_01* corresponds to the *basic_01* syntax test in the *sparql1* test set.

With syntax tests mapped to test methods as described, the entire test suite containing the 188 syntax tests distributed over five tests sets, may be performed by a single operation.

5.5 Custom Tests

In addition to the tests in the W3C SPARQL Test Suite [2], a set of custom tests have been implemented to test other aspects of the parser. The tests supplied in the test suite test the parser as a whole without drilling down on a specific component. The purpose of the custom tests are to test components independently, allowing for more fine grained testing.

5.5.1 Scanner Tests

As the purpose of the scanner is to identify tokens in the input stream, a natural approach for testing the scanner separately is to assure that the correct tokens are identified in the correct order. The test suite tests are roughly split into categories, each of which focus on a certain aspect of the SPARQL grammar. For each such category a custom test has been implemented that feeds a specific test to the scanner and validates the tokens returned. Both token types and positions should be correct in order for the test to succeed.

Figure 5.9 shows the custom scanner test for the SPARQL query shown in Figure 5.3. The tokens returned should be *SELECT*, ***, *WHERE*, *{* and *}*, in that order.

```
[TestMethod]
public void Scanner_1_basic_01()
{
  Assert.IsTrue(assertScannerOutput(
    @"..\..\..\Tests\syntax-sparql1\syntax-basic-01.rq",
    new Tokens[]
    {
      Tokens.SELECT,
      (Tokens)((int)'*'),
      Tokens.WHERE,
      (Tokens)((int)'{'),
      (Tokens)((int)'}')
    })
  );
}
```

Figure 5.9: Custom Scanner Test

```
string escapedString = @"\u0061";
string expected = "a";
string actual;
actual = EscapeSequenceResolver.ResolveCodepointEscapeSequences(
  escapedString);
Assert.AreEqual(expected, actual);
```

Figure 5.10: Common escape sequence resolver test

5.5.2 Escape Tests

As described in Section 4.6, escape sequences are processed and resolved both ahead of and after the parsing. A set of custom tests have been implemented for testing this functionality separately as well. The escape sequence resolvers and the escape visitor are all tested separately.

Figure 5.10 shows an extract from the test for the common escape sequence resolver test

Chapter 6

Results

This chapter starts by illustrating, step by step, how the resulting SharQL parser handles a sample SPARQL query. Next, the results from running the SharQL parser against the *W3C SPARQL Test Suite* are presented. Finally, known nonconformities with the W3C SPARQL specification are explained.

6.1 A Simple Example

This section presents a complete example involving all essential steps in the parse process, beginning with the input SPARQL query and resulting in the AST generated by the parser.

Most implementation-specific details have been left out to as these are described in Chapter 4. The purpose of this example is to present a higher-level view of the parser and its usage.

6.1.1 Example Query

The example SPARQL query used is a rather simple query, as shown in Figure 6.1. The complexity of the query is reasonable in order to highlight the essential parsing steps at the desired level of detail.

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT ?title
WHERE
{
  <http://en.wikipedia.org/wiki/Tony_Benn> dc:title ?title
}
```

Figure 6.1: SPARQL example query, identical to the query presented in Figure 2.4

```
PREFIX
PNAME_NS
IRI_REF
SELECT
VAR1
'{'
IRI_REF
PNAME_LN
VAR1
'}'
```

Figure 6.2: SPARQL example query scanner generated token stream

6.1.2 The Scanner Token Generation

Given the example query in Figure 6.1 the scanner generates the token stream shown in Figure 6.2. The tokens are passed to the parser one by one in the order of appearance. This process is described in Section 4.3.2.

6.1.3 The AST generation

As the parser receives the tokens from the scanner, it collects tokens until a rule in the parser specification is matched by the tokens available. When a rule is matched, a corresponding AST class is instantiated to represent the rule. The AST class hierarchy is described in Section 4.2.

In this example the *PREFIX* token is the first one collected. As this does not match any rule by itself, the parser simply keeps hold of it while fetching the next token from the scanner.

The second token identified by the scanner is the *PNAME_NS* token, which has a corresponding rule in the parser specification, shown in Figure 6.3. When the rule is matched, a *PrefixDeclNode* object is created representing the token. The *sc.yytext* expression represents the textual representation of the token, being the namespace name "'dc"'.

The third token is the IRI_REF token. The rule matched for this token is shown in Figure 6.4. The corresponding object created for this token is an *IriRefNode* object.

Both the objects created replace their corresponding tokens in the parser collection of tokens received from the scanner, both represented as the matching rule from which they were constructed. At this point, the collection contains a *PREFIX* token, a *PrefixDeclNode* represented by the *Pname* rule and an *IriRefNode* represented by the *IriRef* rule. Those three elements match the *Prefix* rule, shown in Figure 6.5.

The *Prefix* rule is handed the elements in the parser collection and uses them to compose its corresponding object. The remaining tokens are handled in a similar manner, until the entire token stream has been processed

```
Pname
: PNAME_NS
{
  $$ = new PrefixDeclNode(sc.yytext, null);
}
;
```

Figure 6.3: *PNAME_NS* Token Parser Rule

```
IriRef
: IRI_REF
{
  $$ = new IriRefNode() { Value = sc.yytext };
}
;
```

Figure 6.4: *IRI_REF* Token Parser Rule

resulting in one *QueryNode* object representing the entire query. This node is the root of the generated AST, as shown in Figure 6.6.

The relationships between the entities involved in the parsing of a SPARQL query is shown in Figure 6.7.

6.2 Test Results

Testing has been an important tool in the development of the SharQL parser, both as a measure of correctness and as a bug tracking mechanism. The W3C SPARQL Test Suite [2] has been a great resource for validating the correctness of the parser as a whole through black box testing. The tests from W3C are listed in Appendix E. Custom tests have helped testing single components through unit testing.

```
/* [4] */
PrefixDecl
: PREFIX Pname IriRef
{
  PrefixDeclNode pn = (PrefixDeclNode)$2;
  $$ = new PrefixDeclNode(pn.Namespace, $3);
}
;
```

Figure 6.5: *Prefix* Parser Rule

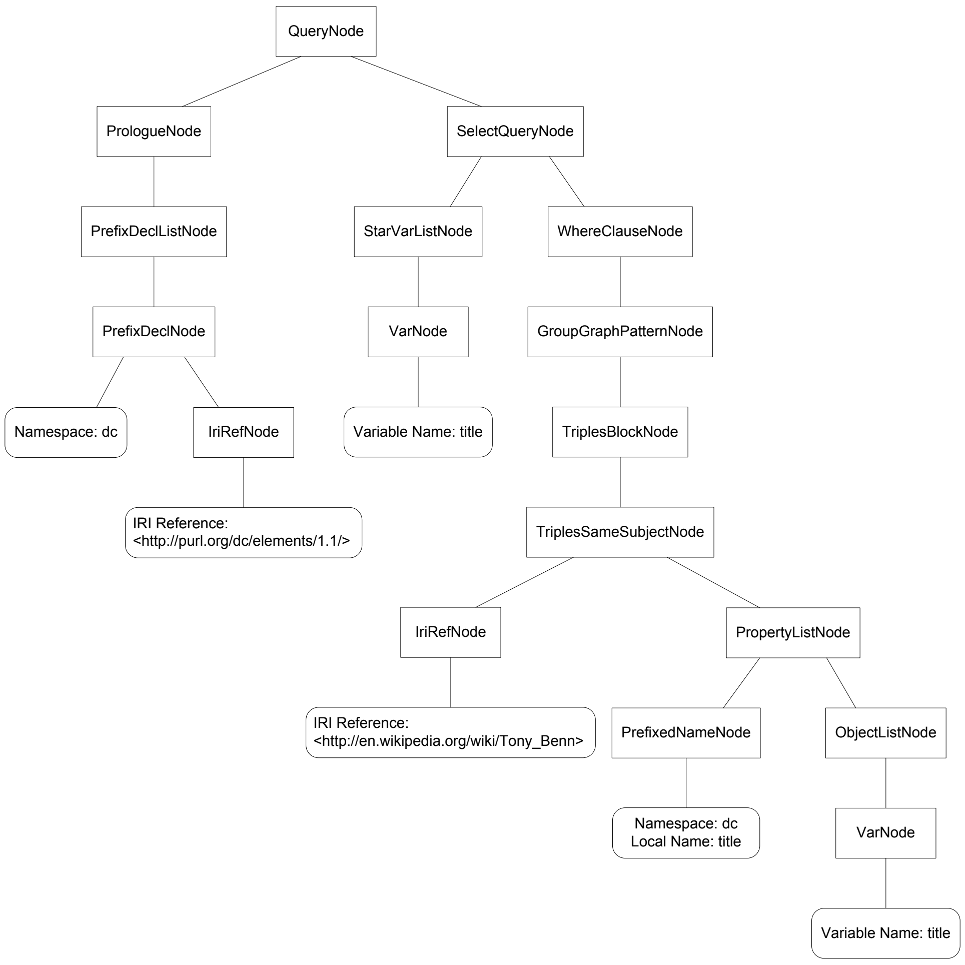


Figure 6.6: Example generated AST

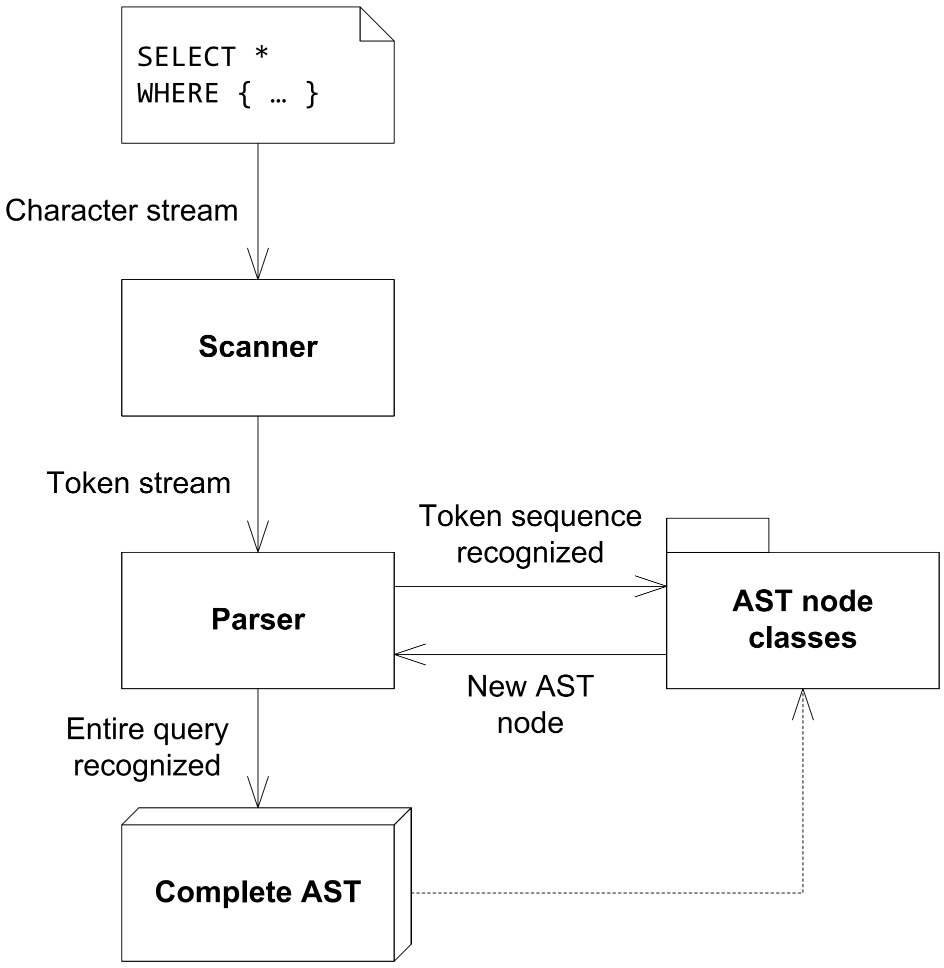
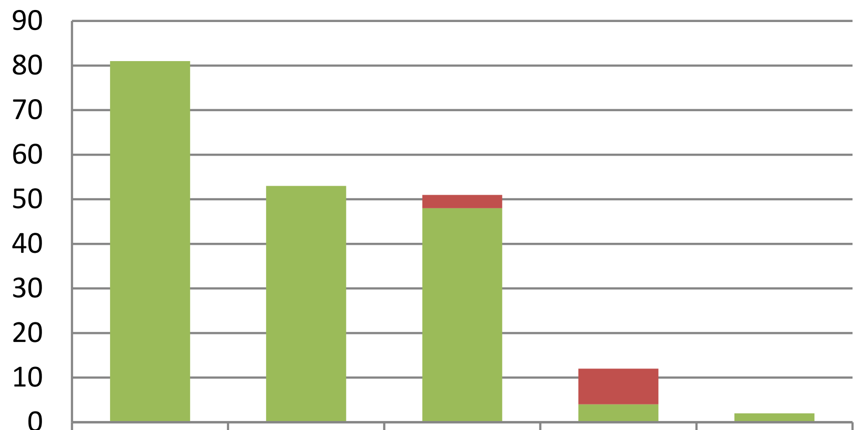


Figure 6.7: Relationships between the parser context entities



	Test Set 1	Test Set 2	Test Set 3	Test Set 4	Test Set 5
🟥 Failed	0	0	3	8	0
🟩 Succeeded	81	53	48	4	2

Figure 6.8: Test Suite Results

6.2.1 W3C SPARQL Test Suite

The tests described in Section 5.3 were run against our parser, resulting in a parse success or parse failure. Test queries that should parse, but did not, indicated errors in our parser. During development, quite a few errors were located by means of valid queries which did not parse successfully.

On the occurrence of a test failing, the test query in question has been examined and used as a basis for tracking the creation of the corresponding AST during parsing to help identifying the point of failure. In most cases this has lead to an error being identified and corrected.

A small amount of tests were identified as semantical tests as well as syntactical tests, despite being classified as syntax tests in the test suite. However, the parser does currently not operate on a semantical level, and the tests in question have thus been excluded, as discussed in Section 5.3.1.

The test suite is divided into five test sets, each containing a number of single test queries. Figure 6.8 shows the results of running all the tests supplied in the test suite, grouped by test set. The failing tests represent the semantical tests that have now been excluded from the test setup.

6.3 Specification Nonconformities

6.3.1 Unsupported Unicode Codepoints

A SPARQL query string is a Unicode character string [1]. As discussed in Section 4.6.1, the Unicode standard specifies unique identifiers (called codepoints) for all Unicode characters. The set of defined codepoints is divided into different *planes*. Each plane consists of 65,536 (2^{16}) codepoints. Plane 0 is the range [0000_{16},$FFFF_{16}$], Plane 1 is [$\mathbf{1}0000_{16}$,$\mathbf{1}FFFF_{16}$] etc. Plane 0 is called the *Basic Multilingual Plane* (BMP), while codepoints from the remaining planes are called *Supplementary Code Points*. [23]

The basic unit recognized by the MPLex generated scanner is the .NET *System.Char* type. This type represents a 16 bit integer value. Thus, a single *System.Char* object can only represent a Unicode character in the Basic Multilingual Plane. To represent Unicode characters from the remaining planes, two successive *System.Char* objects are required. This leads to undefined behavior by the MPLex generated scanner.

While a SPARQL query, according to W3C's specification, may contain characters from all Unicode planes, the SharQL parser developed in this project only has a defined behavior for queries containing characters from the Basic Multilingual Plane. While this violates the SPARQL specification, we consider it a minor issue, as the majority of common-use characters fit into the BMP [22].

6.3.2 Semantic Specifications

According to the SPARQL specification, Section A.6: *"The same blank node label may not be used in two separate basic graph patterns with a single query"* [1].

As discussed in Section 5.3.1, this requirement is considered a semantic rule and thus outside the scope of this project. The SharQL parser will not yield any errors when parsing SPARQL queries reusing the same blank node label in two separate basic graph patterns.

Chapter 7

Conclusion and Further Work

This chapter briefly discusses the course of the project in terms of key ingredients and decisions, resulting in the final outcome. Also, the planned further work is presented, placing the project in its superior context.

7.1 Conclusion

The project has been greatly influenced by the tools used, being Visual Studio and the Managed Babel package containing the MPLex and MPPG lexer and parser generator. Allowing for the lexer and parser generator to be integrated in the SharQL project itself has greatly simplified the parser generation by, in reality, making it completely transparent.

As the SPARQL grammar is defined on *Extended Backus-Naur Form* (EBNF), it had to be translated into the BNF-based specification languages used by MPLex and MPPG. This has introduced quite a few additional grammar rules to compensate for the lack of expressive power in the specification languages. Besides being the most time-consuming task in the project altogether, this translation has been the number one source of bugs during the development.

Conformance with the SPARQL grammar is essential, as well as keeping the implementation free of bugs in general. The W3C SPARQL Test Suite has been an important tool in the strive for correct behaviour. The suite contains nearly 200 tests relevant for the SharQL parser, testing different aspects of the grammar. However, despite being classified as syntactic tests, a rather small amount of the tests actually tested semantic matters as well. As such matters are not taken into account at this stage, the tests in question have been left out from the set of tests used to validate the conformity of the parser.

The outcome of the project is the SharQL parser. It runs on the Microsoft .NET framwork, and generates abstract syntax trees from SPARQL queries. The only nonconformity with the SPARQL specification is the lack of support for Unicode characters that exceed 16 bits in size. All relevant tests from the W3C SPARQL Test Suite pass without exceptions, indicating the level of conformity with the SPARQL specification.

7.2 Further Work

This project and its outcome directly prepares for a master thesis for the iAD Research Centre in spring 2009. The thesis will be part of the MARS project, and addresses the following two problems:

- Efficiently transforming RDF data to, and representing it in, the representation used in MARS engine.
- Generating an executable query plan from a SPARQL query for the MARS .NET engine.

RDF is verbose by nature, especially in its XML-based representation. Regardless of the representation used, triples still mainly consist of URIs, causing a great deal of redundancy. Also, RDF data represents a graph of nodes, which is quite different from typical relational data representations. These properties may call for a different approach in storing and indexing the data in order to allow for queries to be executed efficiently.

When a scheme for representing the RDF-data has been defined, the next step is to generate executable query plans. The outcome of this project is the SharQL parser currently generating abstract syntax trees from SPARQL queries. The ASTs serve as a basis for such query plans, and should be transformed from the current tree of nodes to a tree of query elements defined by the MARS .NET project.

References

[1] W3C. SPARQL query language for RDF. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/, Jan. 2008.

[2] W3C. DAWG Testcases. http://www.w3.org/2001/sw/DataAccess/tests/r2.

[3] W3C. W3C semantic web activity. http://www.w3.org/2001/sw/, July 2008.

[4] W3C. RDF primer. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/, Feb. 2004.

[5] DCMI-Libraries Working Group. Library application profile. http://dublincore.org/documents/2004/09/10/library-application-profile/, Sept. 2004.

[6] Turtle - Terse RDF Triple Language. http://www.w3.org/TeamSubmission/2008/SUBM-turtle-20080114/, Jan. 2008.

[7] Notation 3. http://www.w3.org/DesignIssues/Notation3.html, Mar. 2006.

[8] W3C. RDF Data Access Use Cases and Requirements. http://www.w3.org/TR/2005/WD-rdf-dawg-uc-20050325/, Mar. 2005.

[9] W3C. W3C Opens Data on the Web with SPARQL. http://www.w3.org/2007/12/sparql-pressrelease, January 2008.

[10] W3C. Testimonials for "W3C Opens Data on the Web with SPARQL" Press Release. http://www.w3.org/2007/12/sparql-testimonial, 2008.

[11] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, 2006.

[12] ANother Tool for Language Recognition (ANTLR). http://www.antlr.org/, Oct. 2009.

[13] Compiler compiler/Recursive descent (Coco/R). `http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/`, Oct. 2009.

[14] Microsoft. Managed babel. `http://msdn.microsoft.com/en-us/library/bb165037(VS.90).aspx`, Nov. 2007.

[15] Queensland University of Technology. The Gardens Point Scanner Generator (GPLEX). `http://plas.fit.qut.edu.au/gplex/`, Jan. 2007.

[16] Queensland University of Technology. The Gardens Point Parser Generator (GPPG). `http://plas.fit.qut.edu.au/gppg/`, Jan. 2007.

[17] The Lex & Yacc page. `http://dinosaur.compilertools.net/`.

[18] Microsoft. Structs (C#). `http://msdn.microsoft.com/en-us/library/saxz13w4(VS.85).aspx`, Nov. 2008.

[19] Microsoft. MSDN Academic Alliance. `http://msdn.microsoft.com/en-us/academic/`, Nov. 2008.

[20] W3C. Extensible Markup Language (XML) 1.1. `http://www.w3.org/TR/2004/REC-xml11-20040204/#sec-notation`, February 2004.

[21] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[22] The Unicode Consortium. About the Unicode Standard. `http://www.unicode.org/standard/standard.html`, Nov. 2008.

[23] The Unicode Consortium. Glossary of Unicode Terms. `http://unicode.org/glossary/`, Nov. 2008.

Glossary

AST Abstract Syntax Tree, a tree representation of the syntax of some source code.

BNF Backus-Naur Form, a metasyntax used to express context-free grammars.

C# One of the programming languages supported by Microsoft for creating applications that execute on the .NET Framework.

EBNF Extended Backus-Naur Form, an extension to BNF containing several shorthand notations.

Escape Sequence In the context of this project, an escape sequence is a way of representing Unicode characters by appending a character specifier to an escape character.

Grammar Production A grammar production specifies possible substitutions for a specific grammar symbol.

iAD Information Access Disruptions, a constellation between FAST, two Norwegian enterprises and different research environments with a goal to develop the best search technology in the world.

IRI Internationalized Resource Identifier, a generalization of the Uniform Resource Identifier (URI), allowing Unicode character rather than being restricted to a subset of ASCII characters.

LALR Parser Lookahead LR parser, produces the rightmost derivation, reading the input from left to right.

LL(k) Parser Produces the leftmost derivation, reading the input from left to right, using at most *k* tokens lookahead.

Managed Babel A package for integrating new languages into Microsoft's Visual Studio software. Contains tools that may also be used for general parsing purposes.

.NET Framework A software technology available from Microsoft which provides a library of pre-coded components and a virtual machine for managing application execution.

Parser A program that performs syntactical analysis on a sequence of tokens to determine the grammatical structure. A parser usually produces an Abstract Syntax Tree (AST) for further analysis.

RDF Resource Description Framework, a model for representing information about resources on the World Wide Web.

Scanner A program that reads a sequence of characters and produce a sequence of tokens which represent one or more characters. A scanner performs the first step (the lexical analysis) when parsing input in a given language.

Semantic Web An extension of the World Wide Web in which the semantics of information and services on the web is defined, making it possible for the web to understand and satisfy the requests of people and machines to use the web content.

SharQL The SPARQL parser created in this project.

SPARQL SPARQL Protocol And RDF Query Language, a query language for RDF data.

Token A categorized block of text, recognized by a scanner. A scanner outputs an array of tokens which may be interpreted by a parser.

Turtle A serialization format for RDF, resulting in a less verbose output than the equivalent XML serialization.

Unicode A specification assigning unique code points to symbols from most of the world's writing systems.

Unit Test A test that verifies that an individual unit of source code is working properly.

URI Uniform Resource Identifier, a compact string of characters used to identify or name a resource on the Internet.

Visitor Pattern A way of separating an algorithm from an object structure upon which it operates.

Visual Studio An integrated development environment (IDE) from Microsoft with support for developing .NET Framework applications.

W3C World Wide Web Consortium, the main international standards organization for the World Wide Web.

W3C SPARQL Test Suite A set of tests provided by SPARQL to help ensure that a SPARQL parser behaves according to the specification.

Appendix A

XML for Project File

```
<UsingTask TaskName="MPLex"
    AssemblyFile="lib\Microsoft.VsSDK.Build.Tasks.dll" />
<UsingTask TaskName="FindVsSDKInstallation"
    AssemblyFile="lib\Microsoft.VsSDK.Build.Tasks.dll" />
<!--Set the general properties for this installation of the SDK-->
<PropertyGroup>
  <VsSDKVersion>9.0</VsSDKVersion>
  <VSSDKTargetPlatformVersion>9.0</VSSDKTargetPlatformVersion>
  <VSSDKTargetPlatformRegRoot>
    Software\Microsoft\VisualStudio\$(VSSDKTargetPlatformVersion)
  </VSSDKTargetPlatformRegRoot>
</PropertyGroup>
<Target Name="FindSDKInstallation"
    Condition="'$(VsSDKInstall)'==''">
  <FindVsSDKInstallation SDKVersion="$(VsSDKVersion)">
    <Output TaskParameter="InstallationPath"
        PropertyName="VsSDKInstall" />
    <Output TaskParameter="IncludesPath"
        PropertyName="VsSDKIncludes" />
    <Output TaskParameter="ToolsPath"
        PropertyName="VsSDKToolsPath" />
  </FindVsSDKInstallation>
</Target>
<PropertyGroup>
  <TargetVSVersion Condition="'$(TargetVSVersion)'␣==␣''">
    $(VSSDKTargetPlatformVersion)
  </TargetVSVersion>
</PropertyGroup>
<!--
  ==========================================================
                Generate code from LEX files
  ==========================================================
-->
<PropertyGroup>
  <!--Make sure that the lexer runs before the C# compiler-->
  <CoreCompileDependsOn>
    $(CoreCompileDependsOn);GenerateCodeFromLex
  </CoreCompileDependsOn>
</PropertyGroup>
<Target Name="GenerateCodeFromLex"
    Condition="'$(BuildingProject)'!='false'"
    Inputs="@(MPLexCompile);$(LexFrameFile)"
    Outputs="@(MPLexCompile->'$(IntermediateOutputPath)%(FileName).cs')"
    DependsOnTargets="$(GenerateCodeFromLexDependsOn)">
  <MPLex InputFile="@(MPLexCompile)"
      OutputFile="@(MPLexCompile->'$(IntermediateOutputPath)%(FileName).cs')"
      CompressTables="$(CompressTables)" FrameFile="$(LexFrameFile)"
      SDKVersion="$(VsSDKVersion)">
    <Output TaskParameter="OutputFile" ItemName="Compile" />
    <Output TaskParameter="OutputFile" ItemName="FileWrites" />
  </MPLex>
</Target>
<!--
  ===========================================================
               Generate parser code from Y files
  ===========================================================
-->
<PropertyGroup>
  <GenerateParserCodeFromGrammarDependsOn>
    $(GenerateParserCodeFromGrammarDependsOn);FindSDKInstallation
  </GenerateParserCodeFromGrammarDependsOn>
  <CoreCompileDependsOn>
    $(CoreCompileDependsOn);GenerateParserCodeFromGrammar
  </CoreCompileDependsOn>
</PropertyGroup>
<Target Name="GenerateParserCodeFromGrammar"
    Condition="'$(BuildingProject)'!='false'"
    Inputs="@(MPPGCompile)"
    Outputs="@(MPPGCompile->'$(IntermediateOutputPath)%(FileName).cs')"
    DependsOnTargets="$(GenerateParserCodeFromGrammarDependsOn)">
  <!--Check if there are .lex files in the project because in
      this case mppg should generate the base classes
      used by the code generated by MPLex-->
  <CreateProperty Value="-mplex" Condition="'@(MPLexCompile)'␣!=␣''">
    <Output TaskParameter="ValueSetByTask"
        PropertyName="__GenerateForMPLex" />
  </CreateProperty>
  <!--Run the command line tool that generates the cs files.-->
  <!--Exec Command attribute should not span several lines in actual
      project file-->
  <Exec Command="&quot;$(VsSDKToolsPath)\MPPG.exe&quot;
$(__GenerateForMPLex)␣@(MPPGCompile->'&quot;%(Identity)&quot;')␣&gt;
@(MPPGCompile->'&quot;$(IntermediateOutputPath)%(FileName).cs&quot;')" />
  <!--Add the generated files to the list of the files to compile.-->
  <CreateItem
      Include="@(MPPGCompile->'$(IntermediateOutputPath)%(FileName).cs')">
    <Output TaskParameter="Include" ItemName="Compile" />
    <Output TaskParameter="Include" ItemName="FileWrites" />
  </CreateItem>
</Target>
```

Appendix B

Scanner Specification

```
%using SharQL.Utils;

/* Defines the generated scanner namespace. */
%namespace SharQL

/* Enables unicode characters. */
%option unicode

%{

  // Called each time a token has been returned.
  internal void LoadYylval()
  {
    // Collects token location information and makes it available for the
    // parser through the yylloc variable.
    yylloc = new CustomLexLocation(yyline, yycol, yyline, yycol + yyleng);
  }

  // Called by the parser on the occurence of an unexpected or missing
  // token.
  public override void yyerror(string s, params object[] a)
  {
    // Adds the reported error to the error handler.
    handler.AddError(s, new MPLEX.Parser.LexSpan(tokLin, tokCol, tokLin,
        tokECol, tokPos, tokEPos, new ScanBuffProxy(buffer)));
  }

  // Proxy class for the MPLEX.Lexer.ScanBuff class to allow for the
  // MPLEX.Parser.LexSpan class to be used with the generated
  // SharQL.ScanBuff class.
  public class ScanBuffProxy : MPLEX.Lexer.ScanBuff
  {
    private ScanBuff buffer;

    public override int Pos
    {
      get { return buffer.Pos; }
      set { buffer.Pos = value; }
    }

    public override int Read()
    {
      return buffer.Read();
    }

    public override int Peek()
    {
      return buffer.Peek();
    }

    public override int ReadPos
    {
      get { return buffer.ReadPos; }
    }

    public override string GetString(int b, int e)
    {
      return buffer.GetString(b, e);
    }

    public ScanBuffProxy(ScanBuff buffer)
    {
      this.buffer = buffer;
    }
  }

%}

/*
 * SPARQL Query Language for RDF
 * W3C Recommendation 15 January 2008
 *
 * http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/
 *
 */

iri_ref               <([^<>"{}|^`\\\u0000-\u0020])*>
langtag               @[a-zA-Z]+(\-[a-zA-Z0-9]+)*
exponent              [eE][\+\-]?[0-9]+
integer               [0-9]+
integer_positive      \+{integer}
integer_negative      \-{integer}
decimal               [0-9]+\.[0-9]*|\.[0-9]+
decimal_positive      \+{decimal}
decimal_negative      \-{decimal}
double                [0-9]+\.[0-9]*{exponent}|
    \.([0-9])+{exponent}|([0-9])+{exponent}
double_positive       \+{double}
double_negative       \-{double}
echar                 \\[tbnrf\\"']

/* Range 10000-EFFFF is left out, as MPLEX currently only supports 16-bit
    unicode characters. */
pn_chars_base         [A-Z]|[a-z]|[\u00C0-\u00D6]|[\u00D8-\u00F6]|
    [\u00F8-\u02FF]|[\u0370-\u037D]|[\u037F-\u1FFF]|
    [\u200C-\u200D]|[\u2070-\u218F]|[\u2C00-\u2FEF]|
    [\u3001-\uD7FF]|[\uF900-\uFDCF]|[\uFDF0-\uFFFD]

pn_chars_u            {pn_chars_base}|_
pn_chars              {pn_chars_u}|\-|[0-9]|\u00B7|
    [\u0300-\u036F]|[\u203F-\u2040]
pn_prefix             {pn_chars_base}(({pn_chars}|\.)*{pn_chars})?
pn_local              ({pn_chars_u}|[0-9]) (({pn_chars}|\.)*{pn_chars})?
pname_ns              {pn_prefix}?:
pname_ln              {pname_ns}{pn_local}
```

```
blank_node_label      _:{pn_local}
varname               ({pn_chars_u}|[0-9])({pn_chars_u}|[0-9]|
    \u00B7|[\u0300-\u036F]|[\u203F-\u2040])*
var1                  \?{varname}
var2                  \${varname}
string_literal1       '(([^\u0027\u005C\u000A\u000D])|{echar})*'
string_literal2       \"(([^\u0022\u005C\u000A\u000D])|{echar})*\"
string_literal_long1  '''(('|'')?([^'\\]|{echar}))*'''
string_literal_long2  \"\"\"((\"|\"\")?([^\"\\]|{echar}))*\"\"\"
ws                    \u0020|\u0009|\u000D|\u000A
nil                   \({ws}*\)
anon                  \[{ws}*\]
eq                    "="
less                  "<"
greater               ">"
or                    "||"
and                   "&&"
not                   "!="
lessorequal           "<="
greaterorequal        ">="
hats                  "^^"
comment               #[^\n]*

/*
 * A.8 Grammar
 *
 * (...) Keywords are matched in a case-insensitive manner with the
 * exception of the keyword 'a'.
 */
base                  [Bb][Aa][Ss][Ee]
prefix                [Pp][Rr][Ee][Ff][Ii][Xx]
select                [Ss][Ee][Ll][Ee][Cc][Tt]
distinct              [Dd][Ii][Ss][Tt][Ii][Nn][Cc][Tt]
reduced               [Rr][Ee][Dd][Uu][Cc][Ee][Dd]
construct             [Cc][Oo][Nn][Ss][Tt][Rr][Uu][Cc][Tt]
describe              [Dd][Ee][Ss][Cc][Rr][Ii][Bb][Ee]
ask                   [Aa][Ss][Kk]
from                  [Ff][Rr][Oo][Mm]
named                 [Nn][Aa][Mm][Ee][Dd]
where                 [Ww][Hh][Ee][Rr][Ee]
orderby               [Oo][Rr][Dd][Ee][R]\u0020[Bb][Yy]
asc                   [Aa][Ss][Cc]
desc                  [Dd][Ee][Ss][Cc]
limit                 [Ll][Ii][Mm][Ii][Tt]
offset                [Oo][Ff][Ff][Ss][Ee][Tt]
optional              [Oo][Pp][Tt][Ii][Oo][Nn][Aa][Ll]
graph                 [Gg][Rr][Aa][Pp][Hh]
filter                [Ff][Ii][Ll][Tt][Ee][Rr]
union                 [Uu][Nn][Ii][Oo][Nn]
str                   [Ss][Tt][Rr]
lang                  [Ll][Aa][Nn][Gg]
langmatches           [Ll][Aa][Nn][Gg][Mm][Aa][Tt][Cc][Hh][Ee][Ss]
datatype              [Dd][Aa][Tt][Aa][Tt][Yy][Pp][Ee]
bound                 [Bb][Oo][Uu][Nn][Dd]
sameterm              [Ss][Aa][Mm][Ee][Tt][Ee][Rr][Mm]
isiri                 [Ii][Ss][Ii][Rr][Ii]
isuri                 [Ii][Ss][Uu][Rr][Ii]
isblank               [Ii][Ss][Bb][Ll][Aa][Nn][Kk]
isliteral             [Ii][Ss][Ll][Ii][Tt][Ee][Rr][Aa][Ll]
regex                 [Rr][Ee][Gg][Ee][Xx]
true                  [Tt][Rr][Uu][Ee]
false                 [Ff][Aa][Ll][Ss][Ee]
```

```

%%

{base}                  {return (int)Tokens.BASE;}
{prefix}                {return (int)Tokens.PREFIX;}
{select}                {return (int)Tokens.SELECT;}
{distinct}              {return (int)Tokens.DISTINCT;}
{reduced}               {return (int)Tokens.REDUCED;}
{construct}             {return (int)Tokens.CONSTRUCT;}
{describe}              {return (int)Tokens.DESCRIBE;}
{ask}                   {return (int)Tokens.ASK;}
{from}                  {return (int)Tokens.FROM;}
{named}                 {return (int)Tokens.NAMED;}
{where}                 {return (int)Tokens.WHERE;}
{orderby}               {return (int)Tokens.ORDERBY;}
{asc}                   {return (int)Tokens.ASC;}
{desc}                  {return (int)Tokens.DESC;}
{limit}                 {return (int)Tokens.LIMIT;}
{offset}                {return (int)Tokens.OFFSET;}
{optional}              {return (int)Tokens.OPTIONAL;}
{graph}                 {return (int)Tokens.GRAPH;}
{filter}                {return (int)Tokens.FILTER;}
{union}                 {return (int)Tokens.UNION;}
{eq}                    {return (int)'=';}
{less}                  {return (int)'<';}
{greater}               {return (int)'>';}
{or}                    {return (int)Tokens.OR;}
{and}                   {return (int)Tokens.AND;}
{not}                   {return (int)Tokens.NOT;}
{lessorequal}           {return (int)Tokens.LESSOREQUAL;}
{greaterorequal}        {return (int)Tokens.GREATEROREQUAL;}
{str}                   {return (int)Tokens.STR;}
{lang}                  {return (int)Tokens.LANG;}
{langmatches}           {return (int)Tokens.LANGMATCHES;}
{datatype}              {return (int)Tokens.DATATYPE;}
{bound}                 {return (int)Tokens.BOUND;}
{sameterm}              {return (int)Tokens.SAMETERM;}
{isiri}                 {return (int)Tokens.ISIRI;}
{isuri}                 {return (int)Tokens.ISURI;}
{isblank}               {return (int)Tokens.ISBLANK;}
{isliteral}             {return (int)Tokens.ISLITERAL;}
{regex}                 {return (int)Tokens.REGEX;}
{true}                  {return (int)Tokens.TRUE;}
{false}                 {return (int)Tokens.FALSE;}
{hats}                  {return (int)Tokens.HATS;}
{iri_ref}               {return (int)Tokens.IRI_REF;}
{langtag}               {return (int)Tokens.LANGTAG;}
{exponent}              {return (int)Tokens.EXPONENT;}
{integer}               {return (int)Tokens.INTEGER;}
{integer_positive}      {return (int)Tokens.INTEGER_POSITIVE;}
{integer_negative}      {return (int)Tokens.INTEGER_NEGATIVE;}
{decimal}               {return (int)Tokens.DECIMAL;}
{decimal_positive}      {return (int)Tokens.DECIMAL_POSITIVE;}
{decimal_negative}      {return (int)Tokens.DECIMAL_NEGATIVE;}
{double}                {return (int)Tokens.DOUBLE;}
{double_positive}       {return (int)Tokens.DOUBLE_POSITIVE;}
{double_negative}       {return (int)Tokens.DOUBLE_NEGATIVE;}
{echar}                 {return (int)Tokens.ECHAR;}
{pname_ns}              {return (int)Tokens.PNAME_NS;}
{pname_ln}              {return (int)Tokens.PNAME_LN;}
{blank_node_label}      {return (int)Tokens.BLANK_NODE_LABEL;}
{var1}                  {return (int)Tokens.VAR1;}
```

```
{var2}                 {return (int)Tokens.VAR2;}
{string_literal1}      {return (int)Tokens.STRING_LITERAL1;}
{string_literal2}      {return (int)Tokens.STRING_LITERAL2;}
{string_literal_long1}{return (int)Tokens.STRING_LITERAL_LONG1;}
{string_literal_long2}{return (int)Tokens.STRING_LITERAL_LONG2;}
\*                     {return (int)'*';}
\{                     {return (int)'{';}
\}                     {return (int)'}';}
\(                     {return (int)'(';}
\)                     {return (int)')';}
,                      {return (int)',';}
;                      {return (int)';';}
\.                     {return (int)'.';}
a                      {return (int)'a';}
\[                     {return (int)'[';}
\]                     {return (int)']';}
\+                     {return (int)'+';}
\-                     {return (int)'-';}
\!                     {return (int)'!';}
\/                     {return (int)'/';}
{ws}                   {/*IGNORE*/}
{nil}                  {return (int)Tokens.NIL;}
{anon}                 {return (int)Tokens.ANON;}
{comment}              {/*IGNORE*/}

/*
 * Match for any single character not matching any other token definition.
 * (Makes parser aware of undefined tokens.)
 */
.                      {return (int)yytext[0];}

/* Special-case match for end of file. */
<<EOF>>                {return (int)Tokens.EOF;}

%{
  // Called each time a token has been returned.
  LoadYylval();
%}

%%
```

Appendix C

Parser Specification

```
%using SharQL.Ast.Nodes
%using SharQL.Ast
%using SharQL.Utils
%using System.Globalization
%using System.Diagnostics

/* Defines the generated parser namespace. */
%namespace SharQL

/* Makes the generated parser class a partial class. */
%partial

/* Defines the generated parser class name. */
%parsertype ParsingEngine

/* Defines the token location object type name. */
%YYLTYPE CustomLexLocation

%{

// Provides access to the generated Scanner class instance.
private Scanner sc { get { return scanner as Scanner; } }

%}

/*
 * SPARQL Query Language for RDF
 * W3C Recommendation 15 January 2008
 *
 * http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/
 *
 */

%start Query

%token IRI_REF PNAME_NS PNAME_LN BLANK_NODE_LABEL VAR1 VAR2 LANGTAG
%token INTEGER DECIMAL DOUBLE INTEGER_POSITIVE DECIMAL_POSITIVE
%token DOUBLE_POSITIVE INTEGER_NEGATIVE DECIMAL_NEGATIVE
%token DOUBLE_NEGATIVE
%token EXPONENT STRING_LITERAL1 STRING_LITERAL2 STRING_LITERAL_LONG1
%token STRING_LITERAL_LONG2 ECHAR NIL ANON
%token BASE PREFIX SELECT DISTINCT REDUCED CONSTRUCT DESCRIBE ASK FROM
%token NAMED WHERE ORDERBY ASC DESC LIMIT OFFSET OPTIONAL
```

```
%token GRAPH FILTER
%token UNION OR AND NOT LESSOREQUAL GREATEROREQUAL STR LANG LANGMATCHES
%token DATATYPE BOUND SAMETERM ISIRI ISURI ISBLANK
%token ISLITERAL REGEX TRUE
%token FALSE HATS

%union { public NodeBase Value; }

%%

/* Bracketed numbers refer to SPARQL grammar rule numbers.  */

/* [1] */
Query
: Prologue SelectQuery
{
  this.AstRoot = new QueryNode($1, $2);
}
| Prologue ConstructQuery
{
  this.AstRoot = new QueryNode($1, $2);
}
| Prologue DescribeQuery
{
  this.AstRoot = new QueryNode($1, $2);
}
| Prologue AskQuery
{
  this.AstRoot = new QueryNode($1, $2);
}
;

/* [2] */
Prologue
/* REDUCE/REDUCE
: BaseDecl
{
  $$ = new PrologueNode($1, null);
} */
: BaseDecl PrefixDeclList
{
  $$ = new PrologueNode($1, $2);
}
| PrefixDeclList
{
  $$ = new PrologueNode(null, $1);
}
/*
| REDUCE/REDUCE
{
  $$ = null;
  Debug.WriteLine("No Prologue");
}*/
;

PrefixDeclList
: PrefixDecl PrefixDeclList
{
  $$ = new PrefixDeclListNode($1, $2);
}
| /* empty */
{
  $$ = null;
  Debug.WriteLine("End of PrefixDeclList");
}
;

/* [3] */
BaseDecl
: BASE IriRef
{
  $$ = new PrefixDeclNode($2);
}
;

/* [4] */
PrefixDecl
: PREFIX Pname IriRef
{
  PrefixDeclNode pn = (PrefixDeclNode)$2;
  $$ = new PrefixDeclNode(pn.Namespace, $3);
}
;

Pname
: PNAME_NS
{
  $$ = new PrefixDeclNode(sc.yytext, null);
}
;

IriRef
: IRI_REF
{
  $$ = new IriRefNode() { Value = sc.yytext };
}
;

/* [5] */
SelectQuery
: SELECT DistinctReducedModifier StarVarList DatasetClauseList
    WhereClause SolutionModifier
{
  $$ = new SelectQueryNode($2, $3, $4, $5, $6);
}
;

DistinctReducedModifier
: DISTINCT
{
  $$ = new DistinctReducedModifierNode()
  {
    Modifier = DistinctReducedModifierNode.Modifiers.Distinct
  };
}
| REDUCED
{
  $$ = new DistinctReducedModifierNode()
  {
    Modifier = DistinctReducedModifierNode.Modifiers.Reduced
  };
}
| /* empty */
{
  $$ = null;
  Debug.WriteLine("No DistinctReducedModified");
}
;

StarVarList
: '*'
{
  $$ = new StarVarListNode();
}
| Var VarList
{
  $$ = new StarVarListNode($1, $2);
}
;

VarList
: Var VarList
{
  $$ = new VarListNode($1, $2);
}
| /* empty */
{
  $$ = null;
  Debug.WriteLine("Empty VarList");
}
;

DatasetClauseList
: DatasetClause DatasetClauseList
{
  $$ = new DatasetClauseListNode($1, $2);
}
| /* empty */
{
  $$ = null;
  Debug.WriteLine("Empty DataSetClauseList");
}
;

/* [6] */
ConstructQuery
: CONSTRUCT ConstructTemplate DatasetClauseList WhereClause
    SolutionModifier
{
  $$ = new ConstructQueryNode($2, $3, $4, $5);
}
;

/* [7] */
DescribeQuery
: DESCRIBE StarVarOrIRIrefList DatasetClauseList ConditionalWhereClause
    SolutionModifier
{
  $$ = new DescribeQueryNode($2, $3, $4, $5);
}
;

StarVarOrIRIrefList
: '*'
{
  $$ = new StarVarOrIRIrefListNode();
```

```
}
| VarOrIRIref VarOrIRIrefList
{
  $$ = new StarVarOrIRIrefListNode($1, $2);
}
;

VarOrIRIrefList
: VarOrIRIref VarOrIRIrefList
{
  $$ = new VarOrIRIrefListNode($1, $2);
}
| /* empty */
{
  $$ = null;
  Debug.WriteLine("Empty VarOrIRIrefList");
}
;

ConditionalWhereClause
: WhereClause
{
  $$ = new ConditionalWhereClauseNode($1);
}
| /* empty */
{
  $$ = null;
  Debug.WriteLine("Empty ConditionalWhereClause");
}
;

/* [8] */
AskQuery
: ASK DatasetClauseList WhereClause
{
  $$ = new AskQueryNode($2, $3);
}
;

/* [9] */
DatasetClause
: FROM DefaultOrNamedGraphClause
{
  $$ = new DatasetClauseNode($2);
}
;

DefaultOrNamedGraphClause
: DefaultGraphClause
{
  $$ = new DefaultOrNamedGraphClauseNode($1)
  {
    Type = DefaultOrNamedGraphClauseNode.Types.Default
  };
}
| NamedGraphClause
{
  $$ = new DefaultOrNamedGraphClauseNode($1)
  {
    Type = DefaultOrNamedGraphClauseNode.Types.Named
  };
}
```

```
;

/* [10] */
DefaultGraphClause
: SourceSelector
{
  $$ = new DefaultGraphClauseNode($1);
}
;

/* [11] */
NamedGraphClause
: NAMED SourceSelector
{
  $$ = new NamedGraphClauseNode($1);
}
;

/* [12] */
SourceSelector
: IRIref
{
  $$ = new SourceSelectorNode($1);
}
;

/* [13] */
WhereClause
: WHERE GroupGraphPattern
{
  $$ = new WhereClauseNode($2);
}
| GroupGraphPattern
{
  $$ = new WhereClauseNode($1);
}
;

/* [14] */
SolutionModifier
: OrderClause
{
  $$ = new SolutionModifierNode($1, null);
}
| LimitOffsetClauses
{
  $$ = new SolutionModifierNode(null, $1);
}
| OrderClause LimitOffsetClauses
{
  $$ = new SolutionModifierNode($1, $2);
}
| /* empty */
{
  $$ = null;
  Debug.WriteLine("Empty SolutionModifier");
}
;

/* [15] */
LimitOffsetClauses
: LimitClause
```

```
{
  $$ = new LimitOffsetClausesNode($1, null);
}
| LimitClause OffsetClause
{
  $$ = new LimitOffsetClausesNode($1, $2);
}
| OffsetClause
{
  $$ = new LimitOffsetClausesNode(null, $1);
}
| OffsetClause LimitClause
{
  $$ = new LimitOffsetClausesNode($2, $1);
}
;

/* [16] */
OrderClause
: ORDERBY OrderCondition OrderConditionList
{
  $$ = new OrderClauseNode($2, $3);
}
;

OrderConditionList
: OrderCondition OrderConditionList
{
  $$ = new OrderConditionListNode($1, $2);
}
| /* empty */
{
  $$ = null;
  Debug.WriteLine("Empty OrderConditionList");
}
;

/* [17] */
OrderCondition
: AscOrDesc BrackettedExpression
{
  $$ = new OrderConditionNode($1, $2, null, null);
}
| Constraint
{
  $$ = new OrderConditionNode(null, null, $1, null);
}
| Var
{
  $$ = new OrderConditionNode(null, null, null, $1);
}
;

AscOrDesc
: ASC
{
  $$ = new AscOrDescNode()
  {
    Order = AscOrDescNode.Orders.Ascending
  };
}
| DESC
```

```
{
  $$ = new AscOrDescNode()
  {
    Order = AscOrDescNode.Orders.Descending
  };
}
;

/* [18] */
LimitClause
: LIMIT INTEGER
{
  $$ = new LimitClauseNode()
  {
    Limit = Convert.ToInt32(sc.yytext)
  };
}
;

/* [19] */
OffsetClause
: OFFSET INTEGER
{
  $$ = new OffsetClauseNode()
  {
    Offset = Convert.ToInt32(sc.yytext)
  };
}
;

/* [20] */
GroupGraphPattern
: '{' ConditionalTriplesBlock GroupGraphPatternBlockList '}'
{
  $$ = new GroupGraphPatternNode($2, $3);
}
;

ConditionalTriplesBlock
: TriplesBlock
{
  $$ = $1;
}
| /* empty */
{
  $$ = null;
  Debug.WriteLine("No TriplesBlock");
}
;

GroupGraphPatternBlockList
: GroupGraphPatternBlock GroupGraphPatternBlockList
{
  $$ = new GroupGraphPatternBlockListNode($1, $2);
}
| /* empty */
{
  $$ = null;
  Debug.WriteLine("End of GroupGraphPatternBlockList");
}
;

```

```
GroupGraphPatternBlock
: GraphPatternNotTriples ConditionalDot ConditionalTriplesBlock
{
  $$ = new GroupGraphPatternBlockNode($1, null, $3);
}
| Filter ConditionalDot ConditionalTriplesBlock
{
  $$ = new GroupGraphPatternBlockNode(null, $1, $3);
}
;

ConditionalDot
: '.'
{
  Debug.WriteLine("Dot");
}
| /* empty */
{
  Debug.WriteLine("No Dot");
}
;

/* [21] */
TriplesBlock
: TriplesSameSubject ConditionalDottedTriplesBlock
{
  $$ = new TriplesBlockNode($1, $2);
}
;

ConditionalDottedTriplesBlock
: '.'
{
  $$ = null;
}
| '.' TriplesBlock
{
  $$ = $2;
}
| /* empty */
{
  $$ = null;
  Debug.WriteLine("No ConditionalDottedTriplesBlock");
}
;

/* [22] */
GraphPatternNotTriples
: OptionalGraphPattern
{
  $$ = $1;
}
| GroupOrUnionGraphPattern
{
  $$ = $1;
}
| GraphGraphPattern
{
  $$ = $1;
}
;

```

```
/* [23] */
OptionalGraphPattern
: OPTIONAL GroupGraphPattern
{
  $$ = new OptionalGraphPatternNode($2);
}
;

/* [24] */
GraphGraphPattern
: GRAPH VarOrIRIref GroupGraphPattern
{
  $$ = new GraphGraphPatternNode($1, $2);
}
;

/* [25] */
GroupOrUnionGraphPattern
: GroupGraphPattern GroupGraphPatternUnionList
{
  $$ = new GroupOrUnionGraphPatternNode($1, $2);
}
;

GroupGraphPatternUnionList
: UNION GroupGraphPattern GroupGraphPatternUnionList
{
  $$ = new GroupGraphPatternUnionListNode($2);
}
| /* empty */
{
  $$ = null;
  Debug.WriteLine("End of GroupGraphPatternUnionList");
}
;

/* [26] */
Filter
: FILTER Constraint
{
  $$ = new FilterNode($2);
}
;

/* [27] */
Constraint
: BrackettedExpression
{
  $$ = $1;
}
| BuiltInCall
{
  $$ = $1;
}
| FunctionCall
{
  $$ = $1;
}
;

/* [28] */
FunctionCall
```

```
: IRIref ArgList
{
  $$ = new FunctionCallNode($1, $2);
}
;

/* [29] */
ArgList
: NIL
{
  $$ = new ArgListNode();
}
| '(' Expression ExpressionList ')'
{
  $$ = new ArgListNode($2, $3);
}
;

ExpressionList
: ',' Expression ExpressionList
{
  $$ = new ExpressionListNode($2, $3);
}
| /* empty */
{
  $$ = null;
  Debug.WriteLine("End of ExpressionList");
}
;

/* [30] */
ConstructTemplate
: '{' ConditionalConstructTriples '}'
{
  $$ = $2;
}
;

ConditionalConstructTriples
: ConstructTriples
{
  $$ = $1;
}
| /* empty */
{
  $$ = null;
  Debug.WriteLine("No ConstructTriples");
}
;

/* [31]  */
ConstructTriples
: TriplesSameSubject ConditionalConstructDottedTriples
{
  $$ = new ConstructTriplesNode($1, $2);
}
;

ConditionalConstructDottedTriples
: '.'
{
  $$ = new ConditionalConstructDottedTriplesNode(null);
```

```
}
| '.' ConstructTriples
{
  $$ = new ConditionalConstructDottedTriplesNode($2);
}
| /* empty */
{
  $$ = null;
  Debug.WriteLine("No ConstructTriples");
}
;

/* [32] */
TriplesSameSubject
: VarOrTerm PropertyListNotEmpty
{
  $$ = new TriplesSameSubjectNode($1, null, $2);
}
| TriplesNode PropertyList
{
  $$ = new TriplesSameSubjectNode(null, $1, $2);
}
;

/* [33] */
PropertyListNotEmpty
/* REDUCE/REDUCE
: Verb ObjectList
{
  $$ = new PropertyListNode($1, $2, null);
}*/
: Verb ObjectList PropertyListTail
{
  $$ = new PropertyListNode($1, $2, $3);
}
;

PropertyListTail
: ';' PropertyListTail
{
  $$ = $2;
}
| ';' Verb ObjectList PropertyListTail
{
  $$ = new PropertyListNode($2, $3, $4);
}
| /* empty */
{
  $$ = null;
  Debug.WriteLine("End of PropertyList");
}
;

/* [34] */
PropertyList
: PropertyListNotEmpty
{
  $$ = $1;
}
| /* empty */
{
  $$ = null;
  Debug.WriteLine("End of PropertyList");
}
;

/* [35] */
ObjectList
/* REDUCE/REDUCE
: Object
{
  $$ = new ObjectListNode($1, null);
} */
: Object ObjectListTail
{
  $$ = new ObjectListNode($1, $2);
}
;

ObjectListTail
: ',' Object ObjectListTail
{
  $$ = new ObjectListNode($2, $3);
}
| /* empty */
{
  $$ = null;
  Debug.WriteLine("End of ObjectListTail");
}
;

/* [36] */
Object
: GraphNode
{
  $$ = $1;
}
;

/* [37] */
Verb
: VarOrIRIref
{
  $$ = $1;
}
| 'a'
{
  $$ = new IriRefNode()
  {
    Value = "http://www.w3.org/1999/02/22-rdf-syntax-ns#type"
  };
}
;

/* [38] */
TriplesNode
: Collection
{
  $$ = $1;
}
| BlankNodePropertyList
{
  $$ = $1;
}
```

```
;

/* [39] */
BlankNodePropertyList
: '[' PropertyListNotEmpty ']'
{
  $$ = new BlankNodePropertyListNode($2);
}
;

/* [40] */
Collection
: '(' GraphNode GraphNodeList ')'
{
  $$ = new CollectionNode(new GraphNodeListNode($2, $3));
}
;

GraphNodeList
: GraphNode GraphNodeList
{
  $$ = new GraphNodeListNode($1, $2);
}
| /* empty */
{
  $$ = null;
  Debug.WriteLine("End of GraphNodeList");
}
;

/* [41] */
GraphNode
: VarOrTerm
{
  $$ = $1;
}
| TriplesNode
{
  $$ = $1;
}
;

/* [42] */
VarOrTerm
: Var
{
  $$ = $1;
}
| GraphTerm
{
  $$ = $1;
}
;

/* [43] */
VarOrIRIref
: Var
{
  $$ = $1;
}
| IRIref
{
  $$ = $1;
}
;

/* [44] */
Var
: VAR1
{
  $$ = new VarNode(sc.yytext.Substring(1));
}
| VAR2
{
  $$ = new VarNode(sc.yytext.Substring(1));
}
;

/* [45] */
GraphTerm
: IRIref
{
  $$ = $1;
}
| RDFLiteral
{
  $$ = $1;
}
| NumericLiteral
{
  $$ = $1;
}
| BooleanLiteral
{
  $$ = $1;
}
| BlankNode
{
  $$ = $1;
}
| NIL
{
  $$ = null;
}
;

/* [46] */
Expression
: ConditionalOrExpression
{
  $$ = $1;
}
;

/* [47] */
ConditionalOrExpression
: ConditionalAndExpression ConditionalOrExpressionList
{
  $$ = new ConditionalOrExpressionNode($1, $2);
}
;

ConditionalOrExpressionList
: OR ConditionalAndExpression ConditionalOrExpressionList
```

```
{
  $$ = new ConditionalOrExpressionListNode($2, $3);
}
| /* empty */
{
  $$ = null;
  Debug.WriteLine("End of ConditionalOrExpressionList");
}
;

/* [48] */
ConditionalAndExpression
: ValueLogical ConditionalAndExpressionList
{
  $$ = new ConditionalAndExpressionNode($1, $2);
}
;

ConditionalAndExpressionList
: AND ValueLogical ConditionalAndExpressionList
{
  $$ = new ConditionalAndExpressionListNode($2, $3);
}
| /* empty */
{
  $$ = null;
  Debug.WriteLine("End of ConditionalAndExpressionList");
}
;

/* [49] */
ValueLogical
: RelationalExpression
{
  $$ = $1;
}
;

/* [50] */
RelationalExpression
: NumericExpression
{
  $$ = $1;
}
| NumericExpression '=' NumericExpression
{
  $$ = new RelationalExpressionNode(Constants.Operators.Equal, $1, $3);
}
| NumericExpression NOT NumericExpression
{
  $$ = new RelationalExpressionNode(Constants.Operators.Not, $1, $3);
}
| NumericExpression '<' NumericExpression
{
  $$ = new RelationalExpressionNode(Constants.Operators.Less, $1, $3);
}
| NumericExpression '>' NumericExpression
{
  $$ = new RelationalExpressionNode(Constants.Operators.Greater, $1, $3);
}
| NumericExpression LESSOREQUAL NumericExpression
{
  $$ = new RelationalExpressionNode(Constants.Operators.LessOrEqual, $1,
      $3);
}
| NumericExpression GREATEROREQUAL NumericExpression
{
  $$ = new RelationalExpressionNode(Constants.Operators.GreaterOrEqual,
      $1, $3);
}
;

/* [51] */
NumericExpression
: AdditiveExpression
{
  $$ = $1;
}
;

/* [52] */
AdditiveExpression
: MultiplicativeExpression AdditiveExpressionList
{
  $$ = new AdditiveExpressionNode($1, $2);
}
;

AdditiveExpressionList
: '+' MultiplicativeExpression AdditiveExpressionList
{
  $$ = new AdditiveExpressionListNode(Constants.Operators.Add, $1, $2);
}
| '-' MultiplicativeExpression AdditiveExpressionList
{
  $$ = new AdditiveExpressionListNode(Constants.Operators.Subtract, $1,
      $2);
}
| NumericLiteralPositive AdditiveExpressionList
{
  $$ = new AdditiveExpressionListNode(Constants.Operators.None, $1, $2);
}
| NumericLiteralNegative AdditiveExpressionList
{
  $$ = new AdditiveExpressionListNode(Constants.Operators.None, $1, $2);
}
| /* empty */
{
  $$ = null;
  Debug.WriteLine("End of AdditiveExpressionList");
}
;

/* [53] */
MultiplicativeExpression
: UnaryExpression UnaryExpressionList
{
  $$ = new MultiplicativeExpressionNode($1, $2);
}
;

UnaryExpressionList
: '*' UnaryExpression UnaryExpressionList
{
  $$ = new UnaryExpressionListNode(Constants.Operators.Multiply, $2, $3);
}
| '/' UnaryExpression UnaryExpressionList
{
  $$ = new UnaryExpressionListNode(Constants.Operators.Divide, $2, $3);
}
| /* empty */
{
  $$ = null;
  Debug.WriteLine("End of UnaryExpressionList");
}
;

/* [54] */
UnaryExpression
: '!' PrimaryExpression
{
  $$ = new UnaryExpressionNode(Constants.Operators.Not, $2);
}
| '+' PrimaryExpression
{
  $$ = new UnaryExpressionNode(Constants.Operators.Add, $2);
}
| '-' PrimaryExpression
{
  $$ = new UnaryExpressionNode(Constants.Operators.Subtract, $2);
}
| PrimaryExpression
{
  $$ = $1;
}
;

/* [55] */
PrimaryExpression
: BrackettedExpression
{
  $$ = $1;
}
| BuiltInCall
{
  $$ = $1;
}
| IRIrefOrFunction
{
  $$ = $1;
}
| RDFLiteral
{
  $$ = $1;
}
| NumericLiteral
{
  $$ = $1;
}
| BooleanLiteral
{
  $$ = $1;
}
| Var
{
  $$ = $1;
```

```
}
;

/* [56] */
BrackettedExpression
: '(' Expression ')'
{
  $$ = $2;
}
;

/* [57] */
BuiltInCall
: STR '(' Expression ')'
{
  $$ = new BuiltInCallNode(BuiltInCallNode.CallType.Str, $3);
}
| LANG '(' Expression ')'
{
  $$ = new BuiltInCallNode(BuiltInCallNode.CallType.Lang, $3);
}
| LANGMATCHES '(' Expression ',' Expression ')'
{
  $$ = new BuiltInCallNode(BuiltInCallNode.CallType.LangMatches, $3, $5);
}
| DATATYPE '(' Expression ')'
{
  $$ = new BuiltInCallNode(BuiltInCallNode.CallType.Datatype, $3);
}
| BOUND '(' Var ')'
{
  $$ = new BuiltInCallNode(BuiltInCallNode.CallType.Bound, $3);
}
| SAMETERM '(' Expression ',' Expression ')'
{
  $$ = new BuiltInCallNode(BuiltInCallNode.CallType.SameTerm, $3, $5);
}
| ISIRI '(' Expression ')'
{
  $$ = new BuiltInCallNode(BuiltInCallNode.CallType.IsIri, $3);
}
| ISURI '(' Expression ')'
{
  $$ = new BuiltInCallNode(BuiltInCallNode.CallType.IsUri, $3);
}
| ISBLANK '(' Expression ')'
{
  $$ = new BuiltInCallNode(BuiltInCallNode.CallType.IsBlank, $3);
}
| ISLITERAL '(' Expression ')'
{
  $$ = new BuiltInCallNode(BuiltInCallNode.CallType.IsLiteral, $3);
}
| RegexExpression
{
  $$ = $1;
}
;

/* [58] */
RegexExpression
: REGEX '(' Expression ',' Expression ')'
```

```
{
  $$ = new RegexExpressionNode($3, $5);
}
| REGEX '(' Expression ',' Expression ',' Expression ')'
{
  $$ = new RegexExpressionNode($3, $5, $7);
}
;

/* [59] */
IRIrefOrFunction
: IRIref
{
  $$ = $1;
}
| IRIref ArgList
{
  $$ = new FunctionCallNode($1, $2);
}
;

/* [60] */
RDFLiteral
: String
{
  $$ = new RdfLiteralNode($1, null);
}
| String LANGTAG
{
  $$ = new RdfLiteralNode($1, null) { LangTag = sc.yytext };
}
| String HATS IRIref
{
  $$ = new RdfLiteralNode($1, $3);
}
;

/* [61] */
NumericLiteral
: NumericLiteralUnsigned
{
  $$ = $1;
}
| NumericLiteralPositive
{
  $$ = $1;
}
| NumericLiteralNegative
{
  $$ = $1;
}
;

/* [62] */
NumericLiteralUnsigned
: INTEGER
{
  $$ = new NumericLiteralNode() { Type =
      NumericLiteralNode.LiteralType.Int,
    IntValue = Convert.ToInt32(sc.yytext) };
}
| DECIMAL
```

```
{
  $$ = new NumericLiteralNode() { Type =
      NumericLiteralNode.LiteralType.Decimal,
    DecimalValue = Convert.ToDecimal(sc.yytext,
        CultureInfo.InvariantCulture) };
}
| DOUBLE
{
  $$ = new NumericLiteralNode() { Type =
      NumericLiteralNode.LiteralType.Double,
    DoubleValue = Convert.ToDouble(sc.yytext,
        CultureInfo.InvariantCulture) };
}
;

/* [63] */
NumericLiteralPositive
: INTEGER_POSITIVE
{
  $$ = new NumericLiteralNode() { Type =
      NumericLiteralNode.LiteralType.Int,
    IntValue = Convert.ToInt32(sc.yytext) };
}
| DECIMAL_POSITIVE
{
  $$ = new NumericLiteralNode() { Type =
      NumericLiteralNode.LiteralType.Decimal,
    DecimalValue = Convert.ToDecimal(sc.yytext,
        CultureInfo.InvariantCulture) };
}
| DOUBLE_POSITIVE
{
  $$ = new NumericLiteralNode() { Type =
      NumericLiteralNode.LiteralType.Double,
    DoubleValue = Convert.ToDouble(sc.yytext,
        CultureInfo.InvariantCulture) };
}
;

/* [64] */
NumericLiteralNegative
: INTEGER_NEGATIVE
{
  $$ = new NumericLiteralNode() { Type =
      NumericLiteralNode.LiteralType.Int,
    IntValue = Convert.ToInt32(sc.yytext) };
}
| DECIMAL_NEGATIVE
{
  $$ = new NumericLiteralNode() { Type =
      NumericLiteralNode.LiteralType.Decimal,
    DecimalValue = Convert.ToDecimal(sc.yytext,
        CultureInfo.InvariantCulture) };
}
| DOUBLE_NEGATIVE
{
  $$ = new NumericLiteralNode() { Type =
      NumericLiteralNode.LiteralType.Double,
    DoubleValue = Convert.ToDouble(sc.yytext,
        CultureInfo.InvariantCulture) };
}
;
```

```

/* [65] */
BooleanLiteral
: TRUE
{
  $$ = new BooleanLiteralNode() { Value = true };
}
| FALSE
{
  $$ = new BooleanLiteralNode() { Value = false };
}
;

/* [66] */
String
: STRING_LITERAL1
{
  string str = sc.yytext;
  $$ = new StringLiteralNode() { Value = str.Substring(1, str.Length - 2)
      };
}
| STRING_LITERAL2
{
  string str = sc.yytext;
  $$ = new StringLiteralNode() { Value = str.Substring(1, str.Length - 2)
      };
}
| STRING_LITERAL_LONG1
{
  string str = sc.yytext;
  $$ = new StringLiteralNode() { Value = str.Substring(3, str.Length - 6)
      };
}
| STRING_LITERAL_LONG2
{
  string str = sc.yytext;
  $$ = new StringLiteralNode() { Value = str.Substring(3, str.Length - 6)
      };
}
;

/* [67] */
IRIref
: IRI_REF
{
  $$ = new IriRefNode() { Value = sc.yytext };
}
| PrefixedName
{
  $$ = $1;
}
;

/* [68] */
PrefixedName
: PNAME_LN
{
  string str = sc.yytext;
  $$ = new PrefixedNameNode() { Pname_Ns = str.Split(':')[0], Pn_Local =
      str.Substring(str.IndexOf(':') + 1) };
}
| PNAME_NS
```

```
{
  $$ = new PrefixedNameNode() { Pname_Ns = sc.yytext.Split(':')[0] };
}
;

/* [69] */
BlankNode
: BLANK_NODE_LABEL
{
  $$ = new BlankNode(){ Value = sc.yytext };
}
| ANON
{
  $$ = new BlankNode();
}
;

%%
```

Appendix D

NodeBase Class

```
using System.Collections.Generic;
using SharQL.Ast.Visitor;
using System.Diagnostics;

namespace SharQL.Ast
{
  /// <summary>
  /// The base implementation of the <see cref="INode"/> interface.
  /// </summary>
  public abstract class NodeBase : INode
  {
    #region Protected Fields

    /// <summary>The parent of the node.</summary>
    protected INode parent;
    /// <summary>The children of the node.</summary>
    protected IList<INode> children;

    #endregion

    /// <summary>
    /// Gets a string uniquely identifying the type of the node.
    /// </summary>
    /// <remarks>Obtains the type name by reflection. This can
    /// be slow. For better performance, override and return a
    /// constant.</remarks>
    public virtual string NodeType
    {
      get { return this.GetType().FullName; }
    }

    /// <summary>
    /// Default constructor.
    /// </summary>
    public NodeBase()
    {
      children = new NodeCollection();
    }

    /// <summary>
    /// Constructs the object and adds a set of children.
    /// </summary>
    /// <param name="children">The children to add</param>
    public NodeBase(params NodeBase[] children)
      : this()
    {
      foreach (NodeBase node in children)
      {
        if(node != null) node.Parent = this;
        this.children.Add(node);
      }
    }

    /// <summary>
    /// Gets or sets the parent of the node. Should be set to
    /// <c>null</c> if node is root.
    /// </summary>
    public virtual INode Parent
    {
      get { return parent; }
      set { parent = value; }
    }

    /// <summary>
    /// Gets the list of children of the node.
    /// </summary>
    public virtual IList<INode> Children
    {
      get { return children; }
    }

    /// <summary>
    /// Accepts a visitor. The visitor pattern can be used for
    /// e.g. optimizing the AST.
    /// </summary>
    /// <param name="visitor">The visitor</param>
    public virtual void Accept(IParserVisitor visitor)
    {
      foreach (INode child in children)
      {
        if (child != null)
          child.Accept(visitor);
      }
      visitor[NodeType](this);
    }

    /// <summary>
    /// Returns a string representation of the node.
    /// </summary>
    /// <returns>A string representation of the node</returns>
    public override string ToString()
    {
      return NodeType;
    }

    /// <summary>
    /// Returns a prefixed string representation of the node.
    /// </summary>
    /// <param name="prefix">The string prefix to which the string
    /// representation should be appended.</param>
    /// <returns>A prefixed string representation of the node</returns>
    public virtual string ToString(string prefix)
    {
      return prefix + ToString();
    }
```

```
        }
    }
```

Appendix E

The W3C SPARQL Test Suite

The tests included in the W3C SPARQL Test Suite are listed in the following tables.

syntax-sparql1	
Category	**Tests included**
Basic	`syntax-basic-01`...`syntax-basic-06`
Blank Nodes	`syntax-bnodes-01`...`syntax-bnodes-05`
Expressions	`syntax-expr-01`...`syntax-expr-05`
Forms	`syntax-forms-01, syntax-forms-02`
Limit/Offset	`syntax-limit-offset-01`... `syntax-limit-offset-04`
Lists	`syntax-lists-01`...`syntax-lists-05`
Literals	`syntax-lit-01`...`syntax-lit-20`
Ordering	`syntax-order-01`...`syntax-order-07`
Patterns	`syntax-pat-01`...`syntax-pat-04`
Naming	`syntax-qname-01`...`syntax-qname-08`
Structural	`syntax-struct-01`...`syntax-struct-14`
Union	`syntax-union-01, syntax-union-01`

Table E.1: Tests included in the W3C Test Suite part 1

syntax-sparql2	
Category	**Tests included**
Blank Nodes	`syntax-bnode-01`...`syntax-bnode-03`
Dataset	`syntax-dataset-01`...`syntax-dataset-04`
Escapes	`syntax-esc-01`...`syntax-esc-05`
Forms	`syntax-form-ask-02`, `syntax-form-construct01`... `syntax-form-construct06`, `syntax-form-describe01`, `syntax-form-describe02`, `syntax-form-select01`, `syntax-form-select02`
Function	`syntax-function-01`...`syntax-function-04`
General	`syntax-general-01`...`syntax-general-14`
Graphs	`syntax-graph-01`...`syntax-graph-05`
Keywords	`syntax-keywords-01`...`syntax-keywords-03`
Lists	`syntax-lists-01`...`syntax-lists-05`

Table E.2: Tests included in the W3C Test Suite part 2

syntax-sparql3	
Category	**Tests included**
Positive Tests	`syn-01`...`syn-08`, `syn-blabel-cross-filter`
Negative Tests	`syn-bad-01`...`syn-bad-31`, `syn-bad-bnode-dot`, `syn-bad-bnodes-missing-pvalues-01`, `syn-bad-bnodes-missing-pvalues-02`, `syn-bad-empty-optional-01`, `syn-bad-empty-optional-02`, `syn-bad-filter-missing-parens`, `syn-bad-lone-list`, `syn-bad-lone-node`, `syn-blabel-cross-graph-bad`, `syn-blabel-cross-optional-bad`, `syn-blabel-cross-union-bad`

Table E.3: Tests included in the W3C Test Suite part 3

syntax-sparql4	
Category	**Tests included**
Positive Tests	`syn-09`...`syn-11`, `syn-leading-digits-in-prefixed-names`
Negative Tests	`syn-bad-34`...`syn-bad-38`, `syn-bad-GRAPH-breaks-BGP`, `syn-bad-OPT-breaks-BGP`, `syn-bad-UNION-breaks-BGP`

Table E.4: Tests included in the W3C Test Suite part 4

syntax-sparql5	
Category	**Tests included**
Reduced	`syntax-reduced-01`, `syntax-reduced-01`

Table E.5: Tests included in the W3C Test Suite part 5

Appendix F

Source Code and Report

On the enclosed CD-ROM, the complete source code for the SharQL parser is included, as well as a digital copy of this report.