FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master Thesis in Informatics

Adding C++ Support to MBEDDR

Zaur Molotnikov

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master Thesis in Informatics

Adding C++ Support to MBEDDR

C++ Unterstützung für MBEDDR

Author:	Zaur Molotnikov
Supervisor:	Dr. Bernhard Schätz
Advisor:	Dr. Daniel Ratiu
Date:	September 15, 2013

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. September 2013 Zaur Molotnikov

Acknowledgments

If someone contributed to the thesis... might be good to thank them here.

Abstract

An abstracts abstracts the thesis!

Contents

Outline of the Thesis

Part I: Introduction and Theory

CHAPTER 1: INTRODUCTION

This chapter presents an overview of the thesis and it purpose. Furthermore, it will discuss the sense of life in a very general approach.

CHAPTER 2: THEORY

No thesis without theory.

Part II: The Real Work

CHAPTER 3: OVERVIEW

This chapter presents the requirements for the process.

Part I.

Introduction and Theory

1. Introduction

Here I introduce the main concepts and existing work, relevant to this Master Thesis.

1.1. Projectional Editing

This section compares the traditional approach to build textual editors for the program code with the projectional approach, bringing up motivation for the least.

1.1.1. Traditional Approach

Traditionally programming languages are used in a textual form in text files, forming programs. However the textual nature is not typical for the structure of programs themselves, being rather low-level representation of code. Parsers are used to construct so called abstract syntax trees (ASTs) from the textual program representation. ASTs are structures in memory, usually graph-alike, reminding sometimes control flow graph, where nodes are different statements and edges are the ways control passes from one statement to the next one.

For the developer, using an editor, the degree to which the editor can support the development process is important. For this, the editor has to recognize the programming language construction and provide possible assistance. Among such assistance can be code formatting, syntax validation, source code transformations (including refactoring support), code analyses and verification, source code generation and others. Many of these operation rely indeed on the higher than text level notions related to program such as method, variable, statement. A good editor has to be aware of these higher level program structure.

Nowadays most of the editors work with text, and to provide assistance to programmer integrate with parser/compiler front-end for the programming language. This way to extract the program structure during editing is not perfect for several reasons. First of all, the program being edited as text is not syntactically correct at every moment, being incomplete, for example. Under this circumstances the parsing front-end can not be successfully invoked and returns error messages which are usually false-positive. Secondly, after minor editing of the code, usually the whole text file has to be processed again. Such compiler calls are usually computationally expensive, they slow down, sometimes significantly, the performance of the developer machine. Various techniques exist to speed it up, including partial and pre- compilation, but the problem is still relevant to large extent.

Moreover, the textual nature of the code complicates certain operations additionally. As an example, we can take a refactoring to rename a method. Every usage of the method, being renamed, has to be found and changed. To implement it correctly an editor must take into account various possible name collisions, as well as presume a compilable state of the program to even start the refactoring.

Not to mention the parsing problem itself. Parsing a program in a complex language like C++ is a difficult problem, it involves the need to resolve correctly scoping and typing, templates and related issues, work with pre-processor directives incorporated in the code. In this regard different compilers treat C++ in a different way, creating dialects, which may represent obstacles for the code to be purely cross-platform.

Listing 1.1: Closing several blocks

```
class MyClass {
  void doSomething() {
    while(true) {
      try {
        // ...
        }
      catch(Exception e) {
      }
    }
  }
};
```

The textual representation of program code, involves the need in formatting and preserving syntax. These both tasks, indeed, have nothing to do with program functionality, and addtionally load the developer, reducing productivity. As an example, here I can mention the need to close several blocks ending at the same point correctly, indenting the closing brace symmetrically to opening one, and putting in the exact amount of braces. The Listing 1.1 demonstrates it in the last few lines.

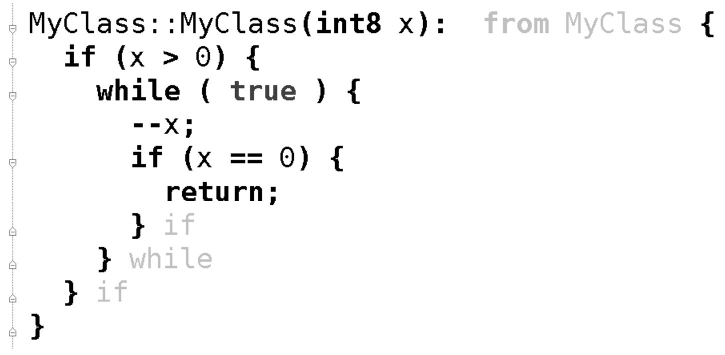
1.1.2. Projectional Approach

Another approach which can be taken in the editor for a language is called projectional approach. Projectional editors do not work with a low-level textual representation of a program, but rather with a higher level concept, ASTs.

Working with ASTs directly has several advantages over the conventional text editing. Firstly, all syntax errors are no longer possible, as there is no syntax. Secondly, there is no need to format the code anyhow, since it is only typical for textual code. Thirdly, all features, which in textual approach require parsing, can be implemented without parser, because AST is always known to the editor.

Projectional editors have to display somehow the AST to the developer, in order for him/her to work with it. Such visualization of an AST is called "projection", giving a name to the editor class.

The model of code is stored as AST in the projectional editor. As in the Model-View-Controller pattern ([1]) the view for the model can be implemented separately. Thus the code may look in a many different ways to the user. For example, the AST can be visualized as a graph, similar to control flow graph. This visualization, however, is not always advantageous being sometimes not compact and complicated to overview.



```
MyClass::MyClass(int8 x):  from MyClass {
  if (x > 0) {
    while ( true ) {
      --x;
      if (x == 0) {
        return;
      } if
    } while
  } if
}
```

Figure 1.1.: Example projection of an AST, "source code" view.

One of the well-accepted way to visualize AST is buy visualizing its textual representation, as if it would was written as a code in the programming language, see Figure 1.1. The statements in the projectional are only selected as whole. There is now way to just select the "while" word for cut or copy, without selecting the condition and the block belonging to the statement. This behavior represents the position of the condition and while-body in the AST as children of the while statement. The statement can be selected all, only with children.

Additionally, each block delimiters are just a part of block visualization. They are organized in a proper way automatically, and there is no way to delete or confuse them, as well as type them initially. Each closing brace is marked with the parent statement name (not a problem to implement such behavior in viewing AST), enhancing navigation through the AST's projection.

As one can see the projection to text in a projectional editor looks almost the same, as the conventional textual editor. This can cause some confusion for the developer at first, as attempts to edit this textual visualization as a real text will fail. Eventually, however, advantages of such visualization overwhelm the disadvantages. Among the benefits of the textual projection over text are quicker code construction after short learning, better way to select code fragments, since not individual characters or lines, but rather AST nodes or groups of nodes are selected, plus all the advantages, the projectional editing brings by itself, as discussed above.

1.2. C++ in Projection

The goal of this Master Thesis is to research different aspects connected with implementation of the C++ programming language in a projectional editor. As a tool base for this Jetbrains MPS has been selected together with the based on it mbeddr project. This section overviews the technologies.

1.2.1. Jetbrains MPS

Jetbrains MPS stands for Jetbrains Meta Programming System. In this IDE-like software it is possible to develop domain specific languages. The approach taken in the Jetbrains MPS is rather unique. One defines a language in it not through the canonical grammar approach, but instead through defining so called concepts, and relationships between them similar to those in ER diagrams. A logical part of a language can be made a concept. Related to C++, class, method declaration, field declaration can be represented as a concept.

Each concept can have children, which should be assigned with a role and cardinality. For example, the class concept can have children of method concept, with a role called "methods", and cardinality 0..n.

Concepts can form hierarchies as classes normally do in object oriented programming languages. For this each concept should have a base concept (inheritance), and can implement several interface concepts (interface implementation). Concepts can be abstract, for the use purely only in inheritance to create other non-abstract concepts with a common parent.

Each concept is described by several views on it. Among such views there are editor, behavior, constraints, type system, generator and few more views.

The editor view is designed to give a look for a concept, and a way to input it. This is where the projection of an AST node is defined. As editors are mostly defined to look like text, a program in the C++ programming languageimplemented in MPS looks almost like a regular C++ code.

The behavior view, can be used to define certain behavioral methods for a concept. A concept is represented there similar to a java class, and it is possible to define the methods in a Java-like language.

The constraints view can be used, to limit in a desirable way relationships the concept can have to other concepts.

The type system view is used mainly to define a type for each instance of a concept. The type is used later (for example, in the constraints), to determine suitability of the concept instance for a place, where it is used.

The generator view, as well as the textgen view, is used to define the way, the concept instance is going to be transformed when generation of the AST is invoked by the MPS user. In general, the generation is needed in the end of the programming, to obtain the source code from the AST in a textual representation. This textual representation is needed because all existing tools (like compilers) expect text code nowadays. It is worth mentioning that, in theory, one could implement a compiler, which works directly from the AST in the editor, eliminating thus the need in generators to text.

The representation of a new language created in concepts and views to them, present a seamless approach to creating a new language with a projectional editor. Thus Jetbrains MPS is a good suitable technology for the practical implementation of the Master Thesis goal.

1.2.2. mbeddr Project

The mbeddr project represent mainly an implementation of the C programming language in the Jetbrains MPS environment. Having embedded systems and software for them as a main focus, mbeddr provides certain language extensions to empower the programmer in the mentioned domain.

Being a different language the C++ programming language shares a lot of commonality with C. As Jetbrains MPS allows to some extent incremental language construction, as the basis for the C++ programming language implementation in Jetbrains MPS the mbeddr project was considered to be suitable.

The use of mbeddr however, could not be purely incremental, and required some changes to the mbeddr itself. The changes were introduced however, in a way to make mbeddr simply more extensible, without creating a dedicated branch working as a basis for the C++ programming language only.

Part II.

Differences between C and C++

2. C and C++

2.1. Structure Itself

1. Differences between C and C++
    a) Reference Type and Boolean Type
    b) Modules Support
    c) Object Oriented Approach Support

Appendix

A. Detailed Descriptions

Here come the details that are not supposed to be in the regular text.

Bibliography

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.