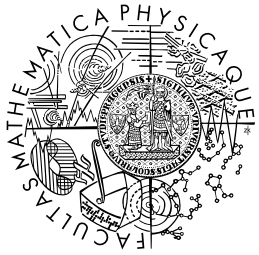

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Václav Čamra

FPVS: FreePascal Visual Studio Integration

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis:	Mgr. Pavel Ježek, Ph.D.
Study program:	Computer Science
Study branch:	Programming and Software Systems

Prague 2017

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: FPVS: FreePascal Visual Studio Integration

Author: Václav Čamra

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Pavel Ježek, Ph.D., Department of Distributed and Dependable Systems

Abstract: The Pascal programming language was designed and is still used to this day to teach procedural imperative programming. However, there are no modern high quality integrated development environments (IDEs) that could be used by students to write in the Pascal programming language.

In this thesis we attempt to fix this problem by creating a Visual Studio 2015 extension which would integrate Free Pascal programming language into Visual Studio. This extension adds a Free Pascal project into Visual Studio, allows for Free Pascal source code to be compiled, ran and debugged. Additionally, the extension adds syntax highlighting and code completion for subset of the Free Pascal language into Visual Studio.

Keywords: Visual Studio 2015, Free Pascal, Visual Studio extension

Název práce: FPVS: Integrace FreePascalu do Visual Studio

Autor: Václav Čamra

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí práce: Mgr. Pavel Ježek, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Programovací jazyk Pascal byl navržen a je i nadále používán pro výuku procedurálního imperativního programování. Neexistuje však žádné moderní kvalitní integrované vývojové prostředí (IDE), které by mohli studenti pro psaní v programovacím jazyce Pascal použít.

V této práci se pokoušíme tento problém napravit tak, že vytvoříme rozšíření pro Visual Studio 2015. Toto rozšíření přidává nový typ projektu – Free Pascal project – do Visual Studia, umožňuje zdrojový kód psaný v jazyce Free Pascal zkompilovat a výsledný program spustit a ladit. Naše rozšíření navíc zahrnuje zvýrazňování syntaxe a napovídání (code completion) pro podmnožinu programovacího jazyka Free Pascal.

Klíčová slova: Visual Studio 2015, Free Pascal, Rozšíření pro Visual Studio

I would like to thank my supervisor, Mgr. Pavel Ježek, Ph.D., for his patience and for the advice he has given me. Without his guidance this thesis would hardly reach the level of quality I believe it to have.

Contents

1. Introduction

This thesis aims to improve the quality of teaching of programming on university level. We are not attempting to find a universally applicable solution since we have based this thesis only on the experience of teachers of the undergraduate study program Computer Science on the Faculty of Mathematics and Physics of the Charles University in Prague. It is important to note that while few decades ago new students used to have some experience with programming when they entered the university already, nowadays vast majority of the newcomers have no experience at all. Because of that it is important to focus the first (winter) term to bring every student to roughly the same basic level and to teach them basic programming skills, e.g. dividing problems into subproblems, synthesis of the subsolutions into the overall solution, conversion of abstract solutions into ones that use basic programming constructs (variables, conditions, loops, expressions, etc.). It is highly undesirable to teach these skills only on an abstract level, therefore it is important to choose an existing programming language. It turns out that Pascal is still a good programming language in which to teach these basic skills to people who have never programmed before. Even though the students are very unlikely to use Pascal in their future careers we still prefer Pascal to other languages as it allows beginners to understand the basics of programming more easily while not making it any harder for experienced newcomers, as they have no issues learning in Pascal as well.

It is important to note that the amount of knowledge a programmer needs to possess nowadays is much greater than what they used to need a few decades ago. At the end of their studies, on top of the skills they already had to have (e.g. create a new algorithm, write it down in some programming language), programmers need to be capable of writing programs spanning hundreds of thousands of lines of code, to work in teams, to use modern frameworks, to write automatic tests, etc. In order to ease developers their job and allow them to focus on the more difficult tasks, it is important that developers have an Integrated Development Environments (IDEs) of high quality. However in the context where we want to teach students in Pascal we must note that Pascal has been developed many decades ago and thus the IDEs that are built for Pascal are designed to suit developers who used to write in Pascal around 20 years ago, and so there is a collision with the requirements of high quality IDE.

We would like to combine both of these views (the Pascal programming language and modern IDEs), because it is important for students to get used to modern IDEs as soon as possible so that they could then smoothly transition to learn more advanced programming concepts. This thesis shows in the next sections that there really is no modern IDE for Pascal, therefore the goal of this thesis is to propose an improvement.

1.1 Curriculum on our faculty

Before we examine the already existing IDEs we need to set criteria we will use to compare these IDEs. To set these criteria we will first introduce the context in which the teaching on our faculty takes place. As we have already stated,

the students on our faculty start by programming in the Pascal programming language, in which we teach only procedural imperative programming and which we do not use to teach any more advanced programming principles (e.g. object-oriented programming, functional programming). On the other hand the Pascal programming language is also used in course that describes the internal functioning of computers, where Pascal is used to write programs showcasing the various described principles. This course makes use of some other features of the Pascal language, for example pointer casting, pointer arithmetic and support for inline assembler.

In the second (summer) term the students on our faculty are taught the principles of object-oriented programming and event-driven GUI programming. It would be highly contra productive to teach these concepts in Pascal, because even though Pascal does support these features, they are implemented in an outdated manner. Instead, we choose the C# programming language as an example of a modern object oriented programming language that is used in real projects. It is important to note in the context of this thesis that for C# there do exist modern IDEs, e.g. Visual Studio, MonoDevelop that students write in and in which the students get used to working with such modern IDEs.

In the transition from winter to summer term students not only transition to new programming language, but also to new IDE, which tends to be quite difficult for students to cope with. Therefore we believe it would be good for students to use an IDE similar to Visual Studio already in the winter term, so that they could, at the start of summer term, focus only on the new programming language.

1.2 Existing IDEs for Pascal

In order to examine the already existing IDEs, we must set some criteria which will be used to compare the IDEs: It is important for students that the IDE has gradual learning curve, so that they can focus on the programming itself. Also, as we have stated in section 1.1, we would prefer the IDE to look similar to Visual Studio, in order for students to have an easier transition to summer term.

Borland Pascal IDE
Borland Pascal IDE (for screenshot see Figure 1.1) has been popular since its first release in 1983. It was very successful and was often used for commercial projects. The benefit Borland Pascal IDE brings to students is that it supports only the pure Borland Pascal programming language, without any additional features. However, since the project has been abandoned, with its last version, Borland Pascal 7, being released 25 years ago (in 1992), students are running into issues with compatibility. These issues stem from the fact that Borland Pascal IDE is only compatible with MS-DOS operating system. Since the most used operating system among students nowadays – Windows – doesn't include MS-DOS subsystem in 64-bit installations[1] students need to emulate MS-DOS environment (e.g. by using DOSBox) in order to run Borland Pascal. Since emulation is necessary, depending on the used emulation software students may not have copy-paste functionality between applications (between the IDE and native Windows applications). Similarly, sharing files between the emulated MS-

DOS and native Windows can be problematic.

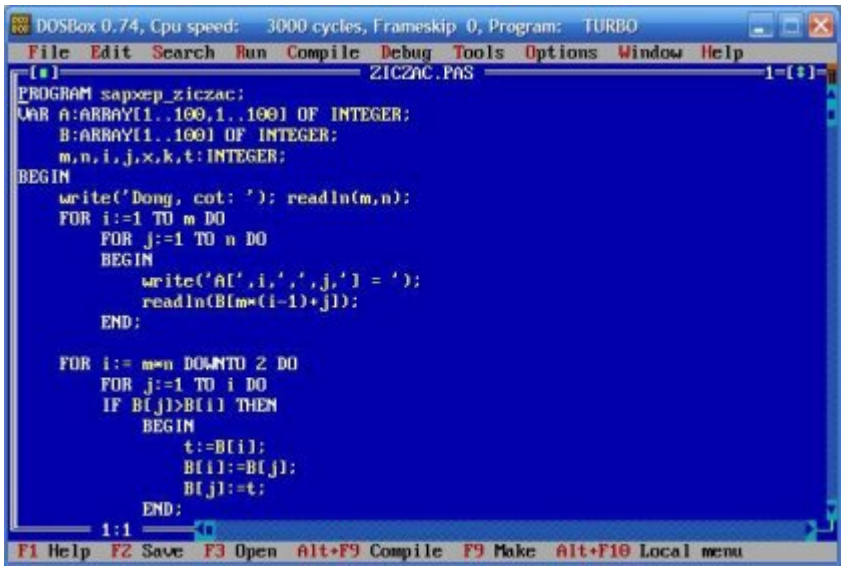

Figure 1.1: Borland Pascal IDE[1]

Feature wise Borland Pascal IDE doesn't offer much for students – only a text editor, a compiler support and a debugger. It lacks the features that programmers nowadays are often used to and which students could use as well, e.g. syntax highlighting and code completion.

Free Pascal IDE

Free Pascal team tried to address the compatibility issues Borland Pascal IDE has. Since the source code for both the Borland Pascal compiler and the Borland Pascal IDE is proprietary, they decided to create a new, open source compiler – Free Pascal Compiler – and IDE – Free Pascal IDE (for screenshot see Figure 1.2). The goal of the Free Pascal project is to have both the compiler and the IDE run under more operating systems (including Windows and Linux).

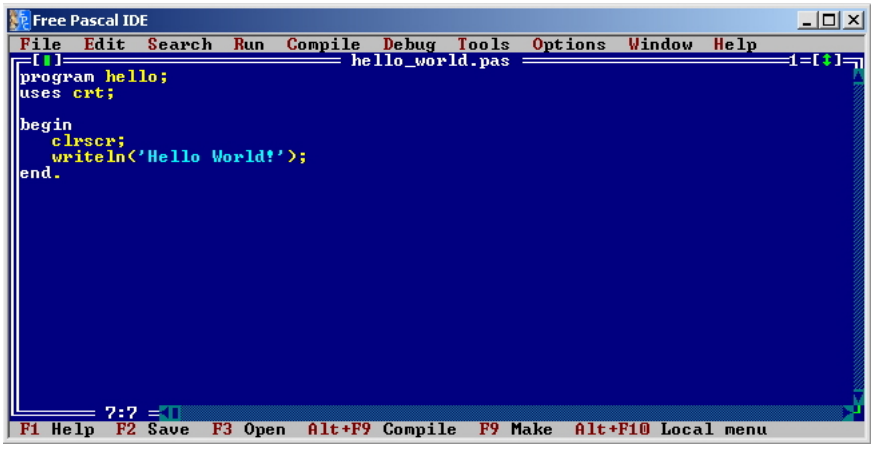

Figure 1.2: Free Pascal IDE

[1]Source: http://img.informer.com/screenshots/3409/3409919_1_4.png

Free Pascal has been first released in 1998. Since at that time there was no need to modify the design of Borland Pascal IDE, the Free Pascal IDE was designed to resemble it (as can be seen from the screenshots). Since the Free Pascal IDE does not need to run in emulated environment, many OS features are now enabled for students to use, namely inter-application copy-paste and accessing source code files from other SW than the IDE itself. On top of that Free Pascal IDE supports syntax highlighting. On the other hand the benefit we mentioned of the Borland Pascal IDE (the purity of the supported language) is not present in the Free Pascal IDE, because while the team was making a new compiler, they have added new features to the language as well, creating Free Pascal programming language.

Delphi

The Borland company also tried to address the issues Borland Pascal IDE had. They created a new, modern IDE – Delphi (for screenshot see Figure 1.3) – in 1995. Borland has, like Free Pascal team, also modified the Pascal language when rewriting the compiler for it, creating Object Pascal. Delphi is, like Borland Pascal IDE, no longer supported (last version Borland Delphi 2005 was released in 2004). Since Delphi was developed in time when it was popular to create GUI applications, Delphi is focused on the development of these applications. This leads to a lot of extra features that are unnecessary for students (e.g. GUI designer). The existence of these features alone would not be an issue, but because Delphi uses an extra window for every feature set and since these features are turned on by default, it clutters the screen and confuses the students.

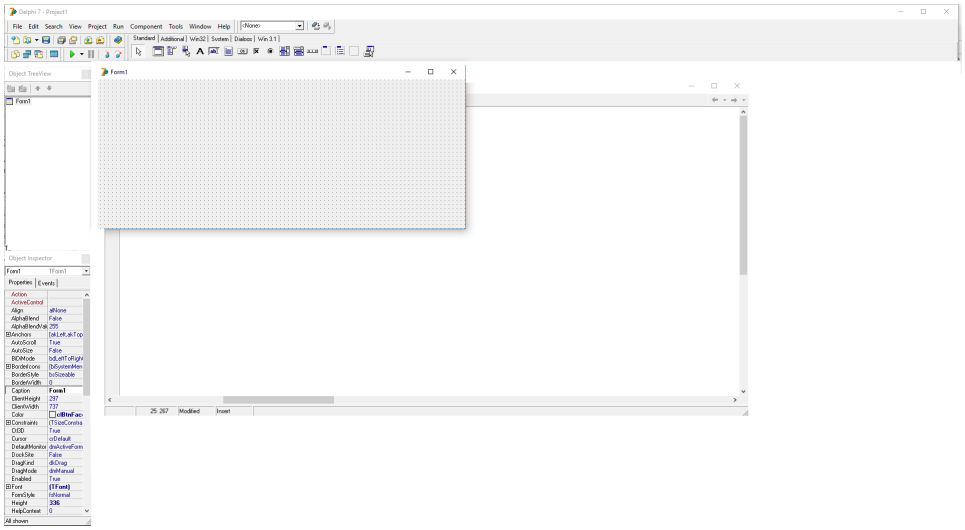

Figure 1.3: Delphi IDE on first start

Lazarus

Lazarus (for screenshot see Figure 1.4) is an IDE that tries to resemble Delphi. The main difference between Lazarus and Delphi is that while Delphi supports Object Pascal, Lazarus supports Free Pascal. Lazarus also shares the now outdated design paradigm of having an extra window for every feature. Since a lot of the extra features, like GUI designer, are turned on by default on startup, many

students are confused even after several weeks of experience with Lazarus.

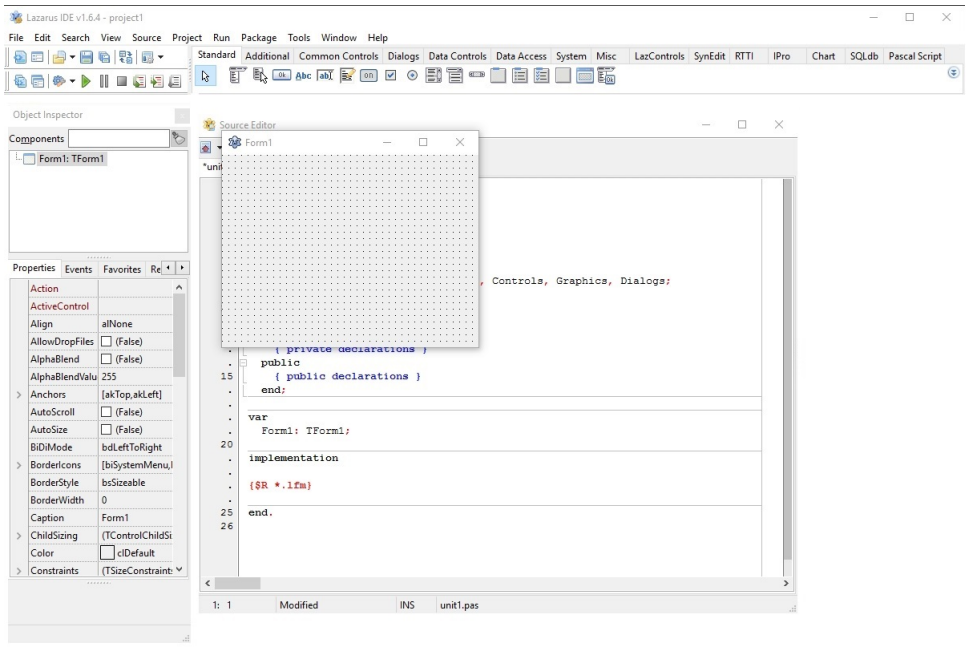

Figure 1.4: Lazarus IDE on first startup

As we can see in table 1.1, all of the mentioned IDEs offer compiler and debugger support. In order to be viable as a Pascal IDE, our suggested improvement will need to support at least that. A common feature in IDEs is syntax highlighting. We believe it to be very helpful for beginners and would like to include it as well. Another feature we would like to see in our extension is code completion. Since Visual Studio supports syntax highlighting and code completion for other languages, we believe it possible to integrate these for Pascal as well.

	Build support	Debugger support	Syntax highlighting	Code Completion	Supported language
Borland Pascal IDE	X	X			Borland Pascal
Free Pascal IDE	X	X	X		Free Pascal
Delphi	X	X	X	X	Delphi
Lazarus	X	X	X	X	Free Pascal
Visual Studio 2015	X	X	X	X	-

Table 1.1: Comparison of IDEs by selected features

Since there exist guides on how to extend projects [2] and how to extend language service (syntax highlighting and code completion) [3] in Visual Studio we believe that Visual Studio does support the addition of new programming

languages and since as we have stated before we would like the students to be able to program in Pascal in an IDE like Visual Studio, we believe it would be best to add the support for the Pascal programming language to Visual Studio. Since the newest version of Visual Studio at the time of writing of this thesis is Visual Studio 2015, we'll be adding the support to that version.

1.3 Requirements

We need to create an extension for Visual Studio 2015 that will add the support for the Pascal programming language. Because only the basics of programming are taught, we don't need to support unit or library files (only those starting with the PROGRAM keyword). This IDE will therefore only need to be able to compile a single file.

Next the students will need some debugging support as well, namely:

- Viewing values of global and local variables
- Stepping through the code
- Editing values of variables
- Viewing call stack

As we have mentioned in 1.1, Pascal is also used in a course which describes the internal functioning of computers. For that course we also want to support disassembler.

Additionally we would like this IDE to have syntax highlighting and to support code completion at least for identifiers from the same file. It would be much better though for students if we could also cover code completion for identifiers coming from the system unit (built-in types, procedures, functions) as well. It would also be helpful for students to offer code completion for keywords.

Since in our implementation we are likely to choose a variant of the Pascal language which will support more constructs than we need (e.g. object-oriented programming) we'll limit ourselves only to the procedural subset of the language. That includes:

- Basic statements (assignments, procedure and function calls)
- Conditions (if-then-else, switch-case)
- Loops (for, while, repeat-until)
- Expressions (arithmetic and logical operators)
- Procedures, functions
- Variables, constants
- Custom types (arrays, simple records, pointers to records)

Furthermore, we expect students to also want syntax highlighting for comments. Additionally, as we have mentioned in section 1.1, for the course where students learn about the internal functioning of computers, we will need to support in-line assembler and pointer arithmetic. Since the in-line assembler is not

used extensively by students, this support will only include the assembler code not being marked as error. No syntax highlighting or code completion for the assembler code will be supported.

Since the teachers on our faculty already have many example projects already created using mostly Free Pascal IDE and Lazarus IDE, it would be good if we could read the project files of these two IDEs. Since we will not need to modify the projects, this reading process can include a one-way project file conversion.

1.4 Goals

The goal of this thesis is to fully integrate a variant of the Pascal language which supports all constructs described in section 1.3 into Visual Studio 2015.

We want to add following features regarding Pascal programming language to Visual Studio, ranked by importance:

1. (Required) Automatic compilation support.
2. (Required) Execution support (running the programs).
3. (Required) Debugging support.
4. (Highly desired) Syntax highlighting, including errors.
5. (Desired) Code completion for built-in identifiers (i.e. `String`, `Writeln`, `Readln`).
6. (Desired) Code completion for custom identifiers.
7. (Desired) Code completion for keywords.

2. Background

In the **Analysis** and **Implementation** chapters we will be describing what design choices we had to make and the way the extension we created works. The implementation uses various systems, frameworks and technologies that the reader might not be familiar with. In order for the reader to fully understand some of the implications that stem from writing a Visual Studio extension or that stem from the design choices, some basic principles about the technologies will be explained in this chapter. If the reader understands what Common Object Model (COM), MSBuild and Managed Extension Framework (MEF) are and the basics of how they are used, he can safely skip this chapter.

2.1 Common Object Model

The Microsoft Common Object Model (COM) is a mechanism that allows the re-use of objects (components) independent of programming language (both the one they were written in and the one they are re-used in) and independent of the object's location (they can be present in the same process, in other processes, or even on another machine altogether).

COM strictly separates the interface and implementation. The interfaces are defined in a programming language neutral way (using the `Interface Definition Language`) and are identified by an `Interface Identifier` (IID), which is a `Globally Unique Identifier` (GUID) – a 128-bit number[1]. The implementation of these interfaces can be written in any programming language that has a compiler which emits COM-compatible binaries. The classes which provide the implementation are identified by their `Class Identifier`s (CLSIDs), which are also GUIDs. When a program wants to create an instance of a specific class, it provides the COM framework the CLSID of the class and COM will then load the specific library (usually registered in Windows registry), create an instance of the class and return a pointer to that instance.

It is important to note that COM interfaces do not support versioning – they are identified by their IID only. This means that any changes to the interface must be done by creating a new interface with its own IID. These interfaces can inherit and extend the original one or can be completely new ones. Since Visual Studio is built using COM and since Visual Studio has been in development for over 20 years (the first Visual Studio – Visual Studio 97 was released in 1997) there are many cases where the interfaces had to be updated, as can be seen on e.g. `IVsProject`, `IVsProject2`, `IVsProject3`, `IVsProject4` and `IVsProject5` interfaces, where each interface (except the first) extends the previous one.

All COM interfaces are usable from .NET by creating a .NET version of these interfaces and classes attributed with various .NET attributes (e.g. the `ComImport`[2] attribute). Programmers can then create and use these .NET classes like any other .NET classes. Microsoft has already created all the .NET variations of the COM interfaces of Visual Studio (e.g. the

[1] GUIDs are usually randomly generated, e.g. `d34a656c-8582-4e4d-a83c-5722ac31a139`

[2] Full name: `System.Runtime.InteropServices.ComImportAttribute`

`Microsoft.VisualStudio.Shell.Interop` assembly), which means that we as the programmers of the extension do not have to do this ourselves.

2.2 MSBuild

MSBuild is a .NET based build engine which is used by Visual Studio for C# projects, C++ projects and others. MSBuild uses XML-based script files, called project files. These project files contain project data (i.e. which files are part of the project, settings, etc.) and code that is called in order to perform an action on the project (e.g. compile it).

There are two kinds of data in MSBuild project files – `Properties` and `Items`. `Properties` are key-value pairs (e.g. `OutputAssembly=MyProgram.exe` – see Figure 2.1) that are mostly used to store project settings. `Items` are named sets (e.g. `Compile=[File1.cs; File2.cs]` – see Figure 2.1) which are mostly used to store sets of file paths relative to the project file.

There are two kinds of code in MSBuild project files – `Tasks` and `Targets`. `Tasks` are the smallest units of executable code. The execution logic of Tasks is written in .NET managed code (classes implementing the `ITask`[3] interface [4]) and they are imported into MSBuild by using the <`UsingTask`> element. Note that some tasks (like `Message` or `Csc` tasks) are imported into MSBuild project files by default. `Target`s are the pieces of code that are exposed to the user of MSBuild. `Target`s are defined in the project files (using XML elements) and use a sequence of `Task`s to perform their function. In the context of this thesis it is important to note that both custom `Tasks` and custom `Targets` can be created and added into the project files.

The project file shown in Figure 2.1 shows an example of `Target` and `Task` usage: The project file defines a `Compile` target, which uses the `Csc` (C# compile) task to compile the `File1.cs` and `File2.cs` files and generate the `MyProgram.exe` executable. This `Compile` target can be invoked by running the MSBuild command line tool with the `/target:Compile` argument.

[3]Full name: `Microsoft.Build.Framework.ITask`

```
<Project xmlns="http://schemas.microsoft.com/developer/
   msbuild/2003">
    <PropertyGroup>
        <OutputAssembly>MyProgram.exe</OutputAssembly>
    </PropertyGroup>

    <ItemGroup>
        <Compile>File1.cs</Compile>
        <Compile>File2.cs</Compile>
    </ItemGroup>

    <Target Name="Compile">
        <Csc Sources="@(Compile)"
        OutputAssembly="$(OutputAssembly)"/>
    </Target>
</Project>
```

Figure 2.1: Simple MSBuild project file example

2.3 Managed Extension Framework

When Visual Studio adopted WPF (in Visual Studio 2010 [5]) the text editor component of Visual Studio was fully rewritten in C#. The text editor is no longer using COM internally (although it is still a COM component from Visual Studio's point of view). Instead, the text editor uses Managed Extension Framework (MEF) to obtain the specific instances of classes implementing certain interfaces.

MEF is a dependency injection framework. Instead of creating instances of classes explicitly (using the `new` C# keyword and a name of the class) MEF allows to discover classes and create their instances implicitly. Each MEF component specifies both its dependencies (using the `Import`[4] attribute) and its capabilities (using the `Export`[5] attribute). When creating an instance of a class with specific capabilities, MEF uses reflection to find the appropriate class, create an instance of it (this works recursively for its dependencies) and return it. If a dependency cannot be fulfilled, MEF will simply not create the instance and return null.

Since the text editor usually uses MEF to create an `IEnumerable` of a specific interface (i.e. instances of all classes which `Export` that interface) and since MEF does not create instances of classes for which it cannot find all `Import`s, any bugs in our extension's code (e.g. a missing `Export` attribute) result in whole features not working (e.g. the entire syntax highlighting), which sometimes makes for challenging debug sessions.

[4]Full name: `System.ComponentModel.Composition.ImportAttribute`
[5]Full name: `System.ComponentModel.Composition.ExportAttribute`

3. Analysis

As we have stated in the **Introduction** chapter, we will be making a Visual Studio extension. Our first goal therefore will be to figure out what our extension needs to implement and how to connect it to Visual Studio itself. After that we will follow the goals stated in section 1.4 by their importance.

For the first goal – Automatic compilation support – we first need to decide on which version of the Pascal language and which compiler we will be using. Then, since Visual Studio triggers the compilation when a user clicks on the *Build project* button (which interacts only with projects), we will need to implement a custom project kind (similar to Visual Studio's C# project or C++ project). Then we will need to find out how to add our own code to be run when the *Build project* button is pressed. Likewise, to run the program (second goal) we will need to figure out how to connect our code to the *Start project without debugging* button. Then we will have to find out how to start the project with debugger attached so that Visual Studio would show the debugger data correctly to the user (stack trace, local/global variables, breakpoints, disassembly).

After these tasks are finished we will focus on syntax highlighting and code completion (called Intellisense in Visual Studio). To implement these we will need to be able to analyze the source code itself so that we can differentiate the various lexical tokens (for syntax highlighting) and are able to determine which tokens can be present at current text position (for code completion). Lastly we will need to import data on the built-in procedures, functions and types, so that we can offer code completion for those as well.

3.1 Pascal Language and Compiler

Before we can implement compilation support into Visual Studio, we have to first choose which version of the Pascal language and which compiler we will use. As we have stated in section 1.3, the Pascal language we choose must support pointer arithmetic. The original Wirth Pascal and even the Borland Pascal does not support pointer arithmetic directly, but allow for a work-around by casting the pointers to integers, then doing the arithmetic and then casting the integers back to pointer type. Since this process does not allow for clean examples of pointer arithmetic, we would prefer not to use Wirth or Borland Pascal. The newer Pascals – Object Pascal, Free Pascal, Delphi – all support pointer arithmetic and all the other constructs we require, making them, from our point of view, equivalent. For that reason we will choose a language based on how difficult it would be to implement compilation of source code written in that particular language into our extension. Therefore we need a language with a compiler that can be run under the Windows operating system. Additionally, we would prefer to use a compiler which can be legally acquired for as little cost as possible.

Object Pascal was created by Borland in 1986 and was replaced by Delphi in 1994. Because of the time of the replacement, Borland never created a Windows-compatible Object Pascal compiler, which means that the only native Object Pascal compiler can run only in MS-DOS environment. However, because Object Pascal can be compiled by the Free Pascal Compiler (with appropriate argument

on command line), there is no need to emulate MS-DOS environment in order to compile Object Pascal code. On the other hand, since Free Pascal Compiler (FPC) would be used and because of the equivalence (based on our requirements) of Object Pascal and Free Pascal, it will make more sense to actually use the Free Pascal language when using the Free Pascal Compiler. It is important to note that FPC can also compile Delphi. Since Delphi compiler must be bought and FPC is free and open-source, it would be a better choice to use FPC to compile Delphi instead. Since this would place us in the same situation as Object Pascal has, it again makes more sense to use the Free Pascal language instead. It is worth noting that with FPC, it is possible to extend our extension so that it supports Object Pascal and Delphi, simply by calling the FPC with the appropriate argument.

Since our extension will also include syntax highlighting and code completion, which will most probably require us to write syntax analysis (i.e. a compiler frontend) of the Pascal source code written by students, it would maybe be better to use a custom compiler with a custom version of the Pascal language. This language would contain only the constructs that we mentioned in section 1.3, which would allow us to limit which constructs that students can use on the compiler level. On the other hand, since a compiler backend is prone to have many hidden bugs, we would not only have to create this custom compiler, but also spend a lot of time maintaining it once our extension is released. Since the advantages of having a custom language do not outweigh the disadvantage of having to maintain a custom compiler, we could consider creating an interpreter instead. That would, however, be even more difficult, because the requirements (disassembler support, in-line assembler) imply that a compiler has to be used. Because of the difficulties a custom compiler would cause, we will rather use an existing compiler.

For the reasons explained in this section, the best choice for our extension is to use the Free Pascal language together with the Free Pascal compiler.

3.2 Project system

Since users can only use the Visual Studio's *Build Project* button with an open project, we will first need to add a new kind of project into Visual Studio – a Free Pascal project – which will handle the compilation of the students' source code. Before we do that, it is important to note the terminology used by Visual Studio:

- `Project` represents data (source code, images, libraries, settings, build scripts, etc.) provided by the user (student) of Visual Studio. Projects usually store their internal data (settings, which files are part of the project, etc.) in `Project file`s (for example .csproj files for C#). Note that the project file and build script can be the same (the mentioned .csproj project file is also an MSBuild build script).

- `Project template` is a template used to create new instances of projects. These templates usually consist of the default project file, default build script (so that users do not need to create their own every time) and a source code file. Project templates can be provided by both the creator of the extension and the user of Visual Studio.

- `Project system` represents the code which will be called by Visual Studio to act (build, remove an item, clean, etc.) on one specific project instance. The project system is provided by the creator of the extension.

It is obvious that we will have to implement a Project system and provide at least one project template. We will start by creating the project system.

From a programmer's perspective, a project system is a COM component (more about COM in section 2.1) implementing multiple COM interfaces (the more implemented interfaces, the more features of Visual Studio are enabled for that particular project). For example, to enable the *Build Project* button we need to implement the `IVsBuildableProjectCfg`[1] interface and provide it to Visual Studio. In order to create a project instance, however, more interfaces need to be implemented, for example `IVsProject`[2], in order to add new items into the project (even though we mentioned in section 1.3 that we will only be working with a single-file projects, we still need to be able to add that single file into the project, as projects start off empty), `IPersistentFileFormat`[3] to allow saving source code files, etc.

For our first prototype of the project system, instead of writing the implementations to all of these interfaces ourselves, we have decided to follow the *Creating a Basic Project System* guide [6]. This guide uses *Managed Package Framework for Projects* (MPF) – base source code which already implements all the aforementioned interfaces and many more. MPF is designed to allow other programmers to easily add new kinds of projects into Visual Studio. These projects use, by default, MSBuild (basics of MSBuild in section 2.2) based project files (similar to C#). Since MSBuild can be extended with custom tasks [4], having MSBuild based project files will not prevent us from adding Free Pascal compilation support later on. On the other hand, by not using the Free Pascal or Lazarus project files we have lost compatibility with Free Pascal IDE and Lazarus respectively, unless we write a tool that will be able to convert our MSBuild project files into Free Pascal or Lazarus ones. Note that we would still have to write a tool to make the conversion from Free Pascal/Lazarus project files into our new MSBuild ones (as mention in section 1.3. Nonetheless, since we do not require the compatibility between our extension and Free Pascal IDE or Lazarus and since it will be easier to write a conversion tool rather than to rewrite part of the MPF to use Free Pascal or Lazarus project files, we have decided to keep using MPF with MSBuild. Since MSBuild by default looks for build scripts in files ending with the `proj` suffix, we have decided to have our project files use the .fpproj extension (Free Pascal project), similar to C#'s .csproj.

It is important to note that the *Creating a Basic Project System* guide we followed also included adding a project template. The project system created by following this guide also supports replacing parts of the template's files with dynamically generated text. We have used this feature to set the program's name (the identifier after the `PROGRAM` keyword) based on the project's name.

[1]Full name: `Microsoft.VisualStudio.Shell.Interop.IVsBuildableProjectCfg`
[2]Full name: `Microsoft.VisualStudio.Shell.interop.IVsProject`
[3]Full name: `Microsoft.VisualStudio.Shell.Interop.IPersistentFileFormat`

3.3 Build

As we can see in figure 3.1 when the *Build project* button is pressed, Visual Studio calls MPF code, which in return calls MSBuild on our project file to perform the `Build` target. To run the compiler we will need to add the `Build` target into our project file template. Since there is no predefined task in MSBuild to run Free Pascal Compiler, we will also have to implement a task that does so.

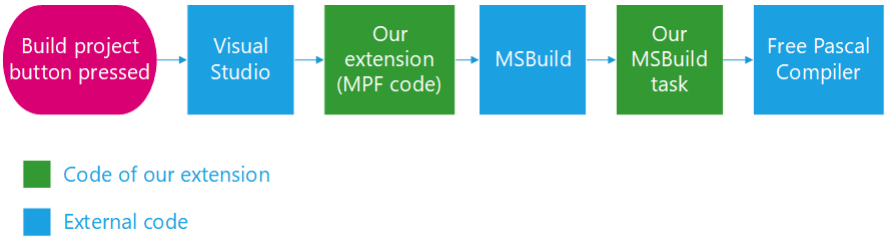

Figure 3.1: Build flow

It is important to note that we also need, in case of compilation errors, to connect the output of the build process to the Visual Studio's Error List (see figure 3.2), thus showing the students the warnings and errors produced by the compiler. This functionality must be implemented in both the MSBuild task (to parse the compiler's output and produce log messages) and in the Visual Studio extension (to add items into the Error List). While the functionality to convert log messages into Error List items is already contained in Managed Package Framework (MPF), the generated items do not support navigation, i.e. when user (student) double-clicks on an Error List item, we want to move the text cursor (caret) exactly to the line and column the warning/error reffers to.

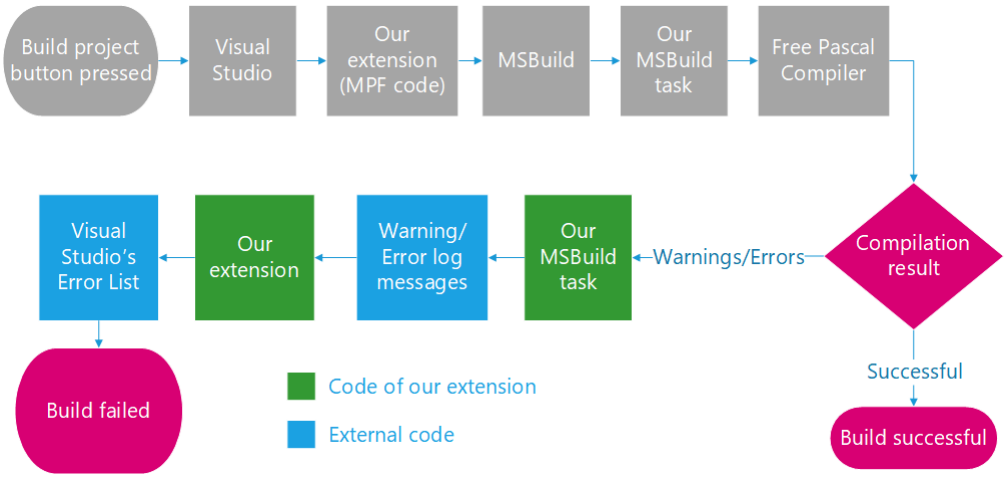

Figure 3.2: Build flow – warning and error messages

3.3.1 Build target

There are two ways to add the `Build` target into the project file. We can either add it into the project file itself (meaning there will be `Build` target in each project file) or we can create `.targets` file and reference it from the project file

(meaning there will be only a single file with the `Build` target). The former has the advantage of allowing users to modify the arguments as they see fit, on the other hand the latter allows for updates of our extension to also work on already existing projects. It is worth noting that the external targets file method is used by C#'s project files. Since we are aiming at students who we are not expecting to have the need to modify the project file we believe it to be better to use an external targets file.

However, having a separate targets file introduces another issue: portability of projects between different computers. The project files need to know the path (either absolute or relative) to the `.targets` file in order to import it, so we need to find a path that will be the same on every computer. We could create a constant path (e.g. `C:\Free Pascal MSBuild targets`) which we could use to resolve the location of the external targets file. It is important to note that MSBuild is multi-platform and also works on UNIX based operating systems, which use different format for paths. While we are not targeting UNIX based operating systems with our extension, we see no reason to make a design decision which would block the possibility of using our project file format on UNIX.

Instead, we will use a builtin MSBuild variable called `MSBuildExtensionsPath` (which is also used by C# projects), which is set by MSBuild to contain the path to folder where MSBuild extensions (targets and tasks) are stored.

3.3.2 Compile task

There are two ways we could implement the compilation task.

1. Use the already existing `Exec` task to run the Free Pascal compiler with all the required command line arguments. This would make it so that the entire compiler integration is written in MSBuild project file (or external targets file respectively).

2. Implement a custom task (a .NET class implementing the `ITask`[4] interface) which would call the compiler.

Since the `Exec` task does not allow us to access the output of the called program and since we need to access this output in order to parse the errors and warnings out of it, we will need to implement a custom task. This custom compile task will need to create the arguments for command line call to Free Pascal Compiler, run the compiler, parse its output and log any warnings or errors that the compiler emits.

Since this task needs to be accessible to the `Build` target in all project files, we must put the assembly containing this task to a well-known location. The best solution would be to keep it at the same place as the `.targets` file, i.e. the folder referenced by the `MSBuildExtensionPath` variable.

[4]Full name: `Microsoft.Build.Framework.ITask`

3.3.3 Error List

It is important to note that our task cannot put errors and warnings into the Error List straight away, because MSBuild is a separate tool and therefore has no reference to Visual Studio whatsoever. In order to gather the warnings and errors from it we need to supply MSBuild with a custom logger (a .NET class implementing the `ILogger`[5] interface). The Managed Package Framework (MPF) already does that with its own `IDEBuildLogger` class whenever it calls any MSBuild target. This logger contains the code necessary to create the GUI messages in Error List, so warning and error display works already.

A common feature of Error List is that upon double-clicking on the warning or error the text editor moves the cursor to where the warning/error originated from. Since `IDEBuildLogger` does not implement this feature, we will have to implement it ourselves.

3.4 Run without debugging

When the *Run without debugging* button is pressed, Visual Studio first runs the build process and then calls the `DebugLaunch` method (with an argument marking it as launch without debugger) on the `IVsDebuggableProjectCfg`[6] interface, which is implemented in Managed Package Framework (MPF) by the `ProjectConfig` class. In order to change the default behaviour we need to inherit from `ProjectConfig` and override the `DebugLaunch` method. We also need to inherit from the `ConfigProvider` class and override the `CreateProjectConfiguration` method and we will also need to override the `CreateConfigProvider` method on the project node class.

Since we are sure that the build process is successfully completed before the `DebugLaunch` method is called (Visual Studio always builds the project before running the program), we do not need to make any additional checks. We only need to create a separate process running the compiled executable. We have decided instead to run the executable via `cmd.exe` (Windows command line), which allows us to keep the console window active after the program has stopped executing (the `cmd.exe` will pause with the *Press any button to continue* message), which we believe the students will find helpful (since there will be no need to write empty `Readln()` statements at the end of the program).

3.5 Debugging

Visual Studio does not communicate with debuggers directly. Instead it uses an intermediate layer called the Debug Engine (see figure 3.3). Debug Engine is a COM component which handles all debugging related commands (e.g. setting breakpoints, reading/writing variable values) from Visual Studio and communicates these commands to the underlying debugger. Since Debug Engine is a COM interface, we can supply Visual Studio with our own. Because Debug Engine only

[5]Full name: `Microsoft.Build.Framework.ILogger`

[6]Full name: `Microsoft.VisualStudio.Shell.Interop.IVsDebuggableProjectCfg`

acts as intermediate layer between a debugger and Visual Studio, we will first decide on which debugger to use, then we can attempt to find/implement a Debug Engine which is capable of communicating with our chosen debugger.

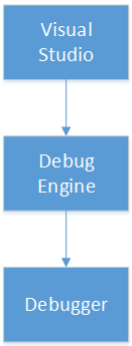

Figure 3.3: Layers used when debugging in Visual Studio

3.5.1 Underlying debugger and Debug Engine

It would be easiest for us to implement debugger support if we were to use Windows debugger (WinDbg), because Visual Studio contains by default a debug engine that uses Windows debugger (WinDbg). Windows debugger can, however, read only debug information written in the PDB (Program Database) format. Free Pascal Compiler does not support generating debug information in the PDB format (FPC can generate debug information in the DWARFv2, DWARFv3 and DWARFv4 formats) which means that we will need to either convert the DWARF debug information to PDB or find a debugger which would be able to use the DWARF debug symbols. There is a `cv2pdb` project [7] designed to convert CodeView and DWARF debug information generated by D compilers into PDB. However, since the project is scarcely updated and since we were unable to find any positive reviews, we have decided to try `cv2pdb` only after we have attempted to solve our problem with a DWARF compatible debugger.

Since both Lazarus and Free Pascal use GDB (GNU Debugger) and since Visual Studio uses GDB in order to debug Android applications [8], which means that there already exists a Debug engine capable of communicating with GDB, it would be best to use GDB as the underlying debugger.

The Debug engine used in debugging Android applications is called MIEngine (git repository at [9]). It is important to note that MIEngine is not distributed with the default installation of Visual Studio (only when Android SDK is installed as well) and it is also important to note that even the version that is included in the android SDK does not support local debugging and debugging console applications. For these reasons we would need to distribute a newer version of MIEngine (downloaded from its repository at[9]) along with our extension.

It is worth noting that GDB has direct support for the Pascal language, including Pascal expression evaluation, which means that by using GDB and MIEngine our extension will also be capable of evaluating any valid Pascal expression during debugging sessions.

Since implementing debugging using MIEngine and GDB will not be difficult and since MIEngine is often updated we will use MIEngine and GDB instead of trying to convert the debug information into PDB format by `cv2pdb`.

3.6 Syntax Highlighting

To make syntax highlighting work correctly under different themes, the Visual Studio's text editor does not highlight parts of the text directly. Instead, first the various parts of the source code – text spans in Visual Studio's terminology (substrings of the entire source code file) – are annotated with tags (e.g. to mark that given text span represents a keyword, a comment, or even an error) and then the text gets highlighted based on its tags and the current Visual Studio theme's settings. Note that a text span can be annotated with multiple tags, for example it can be marked as a keyword and as an error, making it (in the default theme of Visual Studio) a blue word with a red squiggle below it.

Since the coloring of the text is already implemented in Visual Studio, we will only need two things: A list of all the various tags we will want to support and a tagger – a class that will annotate (tag) the student's source code with the appropriate tags.

While Visual Studio supports the addition of new kinds of tags, we will reuse the ones which Visual Studio uses for C#, so that students will have an even easier time transitioning into C#. Reusing tags also ensures that various themes that students may install into their Visual Studio installation will have the desired effect on Free Pascal's syntax highlighting as well.

We have decided to reuse the following kinds of C# tags:

- `Keyword` tag for Free Pascal keywords
- `Comment` tag for all kinds of Free Pascal comments
- `Class name` tag for type identifiers (both basic types and records)
- `Identifier` tag for variables, procedures, functions, named constants (apart from enums), labels, units and program identifier.
- `Preprocessor keyword` tag for in-line assembler statements
- `Enum name` tag for enum values
- `String` tag for strings
- `Punctuation` tag for semicolons, colons, commas, dots, brackets, parentheses, roofs and double dot (..) tokens.
- `Operator` tag for relational operators (=, <>, <, <=, >, >=), assign operators (:=, +=, -=, *=, /=), arithmetic operators (+, -, *, /) and bit shift operators (<<, >>). Note that all operators that are made of alphabetical characters (e.g. `DIV`, `MOD`, `NOT`) are tagged as keywords.
- `Whitespace` tag for whitespace characters

The easiest way to tag the source code would be to analyze it only lexically (turn it into a stream of lexical tokens) and tag according to the kinds of the lexical tokens. This would, however, not be detailed enough (lexical analysis does not distinguish identifiers, e.g. type names, enum constants, variable names), so a more thorough syntax analysis will be required. It is important to note that

even though it is more difficult to implement and maintain the syntax analysis (compared to lexical analysis alone), it will have to be implemented for the code completion feature anyway. For that reason we will implement the tagger based on syntax analysis.

3.7 Code completion

To implement code completion (called Intellisense in Visual Studio) our extension has to provide a class implementing the `ICompletionSource`[7] interface. This class is responsible for generating possible completions once Intellisense session is initiated. In order to generate the correct completions, the `ICompletionSource` will need to know which syntax terminals of the Free Pascal language can be present at the current text position and, if one of the possible terminals is an identifier, which identifiers are visible at the current text position. It is important to note that since the goals we set in section 1.4 included code completion both for built-in identifiers and for custom identifiers, the `ICompletionSource` will need to analyze not only the source code file the student is currently editing (to obtain identifiers and decide visibility from the source code file), but it will also have to analyze the Free Pascal library binaries (to obtain information on built-in types, procedures, functions, etc.).

To determine which syntax terminal can be present at given position an explicit grammar of the supported subset of the Free Pascal language will have to be included in our extension. However, unlike grammars used in parsers, this grammar can use multiple identifier terminals (e.g. a procedure identifier and a variable identifiers can be two separate terminals).

To determine which identifiers are visible from the given position in the source code a data structure that can store the data on identifiers will be needed. This data structure must take into account identifier visibility (e.g. an identifier that has not yet been declared cannot be offered in code completion). To create this data structure a full syntax analysis will have to be performed on the source code. Since such an analysis will be useful for syntax highlighting as well, it will be easiest to create an abstract syntax tree to perform this analysis on.

3.7.1 Pascal library analysis

The Pascal library files – Units – are stored in `.ppu` files. Free Pascal is distributed with a console tool `PpuDump.exe`, which can create an XML output describing the variables, constants, types, procedures and functions exported by a particular unit file (e.g. `System` unit).

3.7.2 Data structures for code completion

To design a data structure for identifiers the visibility rules of the Pascal language must be taken into account:

- Only identifiers that have been declared in the text before the cursor can be used.

[7] Full name: `Microsoft.VisualStudio.Language.Intellisense.ICompletionSource`

- Upon leaving a block which declared new identifiers (i.e. the END at the end of a procedure or function) all the identifiers declared by that block are no longer accessible.
- Inner procedures/functions can use the identifiers used by the outer procedure/function.

By examining these rules we can discover that the visibility can be represented by an ordered tree, where root is the program itself, inner nodes are functions, procedures and records and leafs are variables, constants, labels, fields and types. The children are always ordered by the order in which they were declared in the source code. Each node can see its predecessors, its parents and its parent's predecessors, its grandparents and its grandparent's predecessors, etc. When a query on this data structure asks for available identifiers in the body of a subprogram (procedure/function), the available identifiers will be represented by the node representing said subprogram, its children and all the identifiers visible from the subprogram's node.

3.8 Source code analysis

As we have discussed in sections 3.6 and 3.7), our extension will need to perform syntax analysis on the source code so that it can answer the following questions:

- What kind of source code is in the given text span? (e.g. keyword, identifier, operator, comment)
- Which terminal of the Pascal language can be added at the caret's (text cursor) position?
- Which identifiers are visible (according to Pascal language specification) at the caret's position?

Since our extension would have to parse the source code up until the caret's position every time one of these questions would be asked and since that approach would be most probably highly inefficient and could cause performance issues, it will be better to parse the source code only once and create an abstract syntax tree (AST) instead. This tree will still need to be recreated (or at least modified) every time the source code changes.

3.8.1 Data structure

Initially, we attempted to create the Abstract Syntax Tree (AST) by extending Roslyn (C# and Visual Basic compiler platform) with custom Free Pascal syntax tree and all the various syntax nodes. This would make it easier for us to implement the AST, because we could reuse the base syntax node classes Roslyn uses (for example to have an efficient search algorithm for the tree structure). Additionally, since Roslyn is already integrated into Visual Studio, we would already have all the required callbacks registered in Visual Studio (for example when the source code changes). We have attempted to create a prototype of a data

structure extending Roslyn and found that it is impossible to extend the base classes, because they contain several abstract internal methods returning objects of internal types (thus making it impossible for us to implement these methods). A possible solution to this problem would be to fork the whole Roslyn, add the `InternalsVisibleTo` attribute to the Roslyn's source code and then redistribute the modified Roslyn together with our extension (the way it is done in Peachpie [10]). Since we also found out that we would have to implement many methods on the abstract syntax node which we would not need for our analysis (e.g. assembly references, public interface to programatically modify the data tree, etc.) and since we would not gain as much as we believed we would from extending Roslyn, we have instead decided to create a custom AST.

Before we start implementing our own AST, we need to take into considerations the requirements we will have on the data structure in order for us to be able to implement syntax highlighting and code completion.

- Every node of the tree represents a text span in the source code.
- Every character in the source code is represented by a node in the AST (including whitespace).
- The tree can be traversed both up and down (e.g. both children and parent can be accessed from each node).
- The tree should support persistency, i.e. ability to keep majority of the tree structure when reparsing only part of the source code. This property is important for performance reasons, as the extension will be reparsing the source code every time it changes (e.g. with every written character).
- The tree should be immutable, so that we can make analysis of the tree in other threads without any need for locking.

It is important to note that the requirements make it impossible to use a single tree in this data structure, because having immutable tree which allows traversal in both directions (i.e. it needs to have reference both for children and parent) and also allows for persistency (only some of the nodes are replaced when new tree is created) is impossible. Another issue rises with the combination of immutability, persistency and bijection between nodes and source code text spans. When the source code is modified (i.e. a whitespace is removed), all the nodes that come after the change would need to be replaced, because the start/end positions stored in the nodes would no longer be correct. In the end, we have decided to use a data structure heavily inspired by Roslyn.

Roslyn uses two separate trees – a *green* tree and a *red* tree (named after the colors of white board markers used when the team designed the structure). The green one (we have renamed it to *Persistent* tree) can only be traversed downwards, each node only knows its length and the terminal nodes (leafs) are aware of the string they represent. This tree is immutable and supports persistency. The *red* tree (we have renamed it to *On Demand* tree) is created lazily on demand from the *Persistent* tree. Whenever the source code changes, the *Persistent* tree is modified to conform to the change and when it is first needed, *On Demand* tree is created.

It is important to note that on top of the nonterminals and terminals of the Pascal grammar the data structure also needs to represent whitespace and comments (e.g. for syntax highlighting). We adapted the term *trivia* from Roslyn. Each node has reference to all the trivia that precedes and follows said node.

3.8.2 Parsing

Last thing we need is to convert the source code into the AST. For this purpose a lexical analyzer (lexer) and a syntax parser will be needed. The lexer will transform the source code into a stream of lexical tokens (terminals of the Free Pascal language and trivia – whitespace, comments, unknown tokens), then the syntax parser will use this stream of tokens to create the AST.

There are two general ways the lexer could be implemented – use a lexer generator, which would be supplied with a list of regexes and which tokens they are to be converted to, or a custom lexer. Since the lexer generator way will be easier to maintain than a custom lexer and since the performance of the automatically generated lexer is likely to be adequate, we have decided to use a lexer generator. However, if lexer proves to be a bottleneck, a custom lexer may have to be written instead.

There are many different lexer generators, but we have focused mainly on the ones with bigger communities around them – ANTLR[11] and Coco/R[12]. Since ANTLR seems to be richer in features and since it also seems to be updated more often, we have decided to use ANTLR as the lexer generator for our extension.

Syntax parser can also be generated either automatically (from a given grammar) or implemented manually. It is important to note that since the generated AST is to cover the entire source code, the syntax parser has to be able to recover from any error (so that syntax highlighting works for the entire source code, not only up until the first error). Since automatically generated syntax parsers are not capable of recovering from any syntax error, a manually written parser will be used instead. For this purpose a recursive-descent parser will be used.

4. Implementation

The Free Pascal Visual Studio integration implementation is contained in 2 solutions (see Figure 4.1): `MIDebugEngine` and `FreePascalVisualStudioIntegration`.

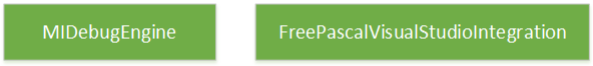

Figure 4.1: FPVS Solutions

The `MIDebugEngine` solution's goal is to create a `.vsix` (Visual Studio extension file) extension, which contains the modified `MIEngine` Debug engine. Since only very minor changes (described in section

The `FreePascalVisualStudioIntegration` solution's goal is to create a `.vsix` extension which adds the Free Pascal support (in the range described in section 1.3) into Visual Studio. Additionally, the `FreePascalVisualStudioIntegration` solution also creates a separate assembly containing the MSBuild `FpCompile` task and a `.targets` (MSBuild targets file) file.

4.1 MIDebugEngine solution

The `MIDebugEngine` solution is a clone of the `MIEngine` github repository (found at [9]). There were only two kinds of modifications we have done to the solution:

1. Replace the GUIDs which identified the Debug Engine (`EngineId` in the `EngineConstants.cs` and `Microsoft.MIDebugEngine.pkgdef` files) and the extension(`guidMIDebugPackagePkg` in the `MIDebugPackage.vsct` file). This change allows users to have both ours `MIDebugEngine` extension and the original `MIEngine` extension (used to debug Android applications) installed at the same time.

2. Since towards the end of our extension's development Visual Studio 2017 was released, and since having both Visual Studio 2015 and 2017 installed in parallel results in both Visual Studio's using the Development Tools 2017 (including `VsixInstaller.exe`, which installs our `.vsix` extensions), we had to modify the `MIDebugEngine` solution to emit vsix3-compatible extension (otherwise if users had both Visual Studio 2015 and 2017 installed, they would have difficulties installing our extension). This change consisted of installing the `Microsoft.VisualStudio.Sdk.BuildTools.14.0.` nuget package into the `MIDebugPackage` project and adding the `<VsixType>v3</VsixType>` property into the project's project file.

4.2 FreePascalVisualStudioIntegration solution

The `FreePascalVisualStudioIntegration` solution consists of the following projects (see figure 4.2):

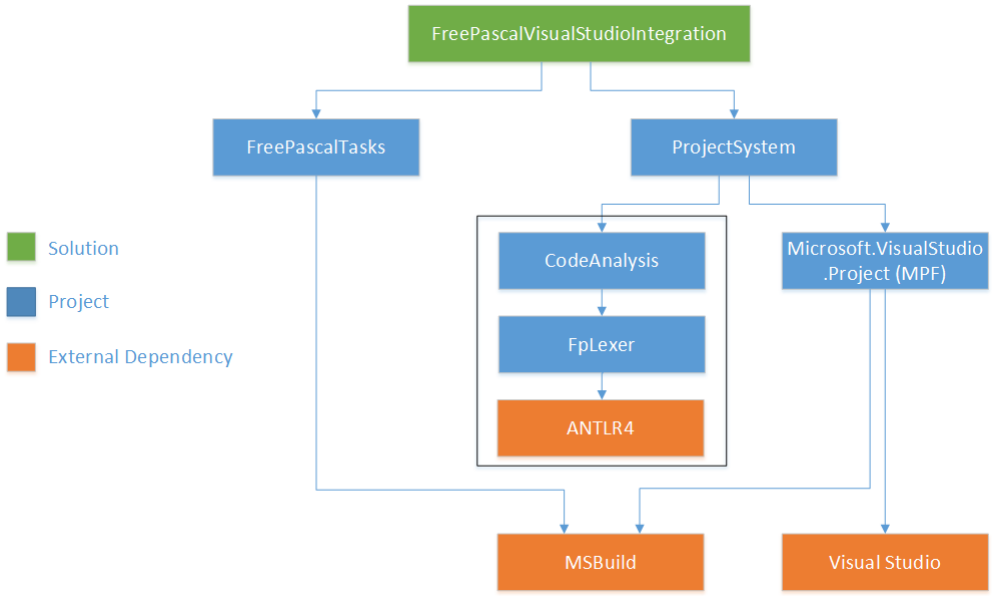

Figure 4.2: Solution and project hierarchy

`ProjectSystem` is the core project representing the Visual Studio extension. It contains the `FpPackage` package class and all the additional Visual Studio integration elements. It is responsible for adding every feature implemented by our extension into Visual Studio. As can be seen in figure 4.2 the `ProjectSystem` project is dependent on the `CodeAnalysis` project for syntax highlightning and code completion and `Microsoft.VisualStudio.Project` (Managed Package Framework – MPF) for project system, build and debugging functionality.

`Microsoft.VisualStudio.Project` is the Managed Package Framework (MPF) source code. It represents majority of the project system code.

`FreePascalTasks` contains `FreePascalCompile` task for MSBuild. It also contains the .targets file used by all Free Pascal projects. The `FreePascalTasks` project is independent of Visual Studio, which means that the targets and task implemented in this project can also be used on other platforms which can run MSBuild (e.g. Linux).

`FpLexer` is a lexer for the Free Pascal language. The project defines all the classes which represent lexical tokens (e.g. `IdentifierToken`). The project also defines an enum to represent all terminals and nonterminals of the grammar (`SyntaxKind` enum). The `FpLexer` project is independent of Visual Studio, which means that it can be reused in non-Visual Studio related projects. The `FpLexer` is only dependent on ANTLR4 lexer generator.

`CodeAnalysis` contains the definition of the subset of Free Pascal's grammar (according to section 1.3), Abstract Syntax Tree (AST), the `FpParser` class, which handles the transformation of the token stream generated by the `FpLexer` project to Abstract Syntax Tree, and the data structure for identifiers (for code completion). The `CodeAnalysis` project is only dependent on the `FpLexer`

project. There is no dependency on Visual Studio. This should allow the `CodeAnalysis` project to be distributed as a separate package (e.g. a nuget package) to be used as a library for Free Pascal source code analysis.

4.3 ProjectSystem project

The `ProjectSystem` project's goal is to create a .vsix (Visual Studio extension) package containing the Free Pascal support functionality. The project contains the central class – `FpPackage` class – of the extension, which implements the `IVsPackage` COM interface. This package registers majority of the features (except text editor based ones, i.e. Syntax highlightning and Code completion) our extension provides to Visual Studio. In addition, the `ProjectSystem` project exports text editor based functionality, i.e. Syntax highlightning and Code completion via Managed Extension Framework – MEF. Note that since these features are not tied to the Free Pascal projects, they are available in any kind of project (or even outside of them), as long as the opened files have the `.pas` extension.

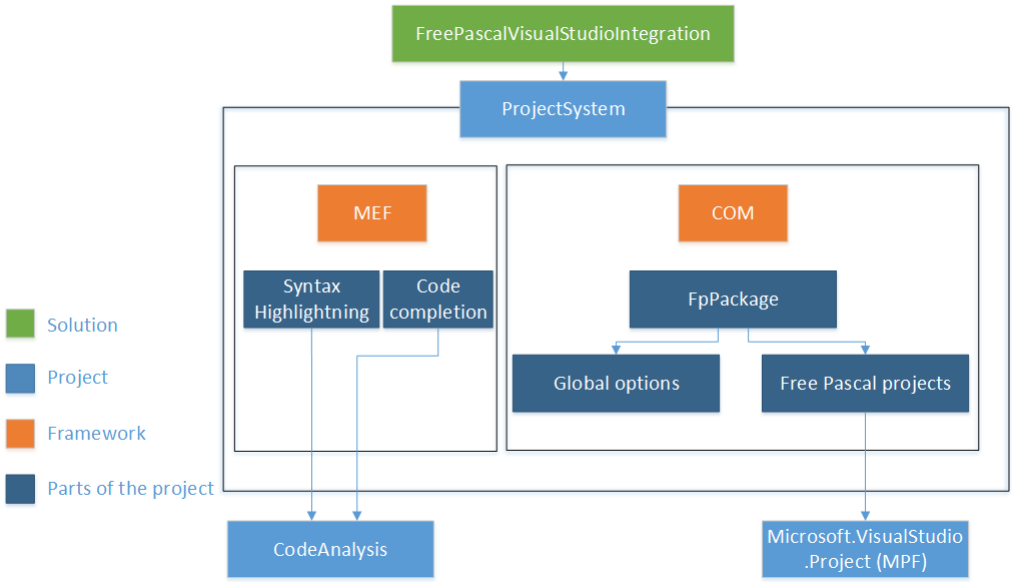

Figure 4.3: `ProjectSystem` project internals

4.3.1 Extension

The central class of the extension is the `FpPackage` class. Since the `FpPackage` inherits from MPF's (Managed Package Framework - the `Microsoft.VisualStudio.Project` project) `ProjectPackage` class, the implementation of `FpPackage` can be nearly empty. In order for Visual Studio to delay the instantiation of our package as much as possible (to boost Visual Studio's startup time), the features our extension provides are registered using C# attributes on the `FpPackage` class. The specific attributes will be explained in the chapters which describe the features the attributes are associated with.

Since the goal of the `ProjectSystem` project is to create a .vsix package, which on top of the compiled binaries also constains various metadata describing the extension (e.g. Product Name, License, Language, Release Notes, Product web

page) a `source.extension.vsixmanifest` file is used to provide these metadata to the extension. Among other things, this manifest file also specifies which versions of Visual Studio our extension is applicable to. Even though we have designed and tested the extension using Visual Studio 2015, we have also made it possible to install our extension on Visual Studio 2017 (which was released toward the end of our extension's development). In addition, the manifest also marks `MIEngine` as its dependency and embeds it into `ProjectSystem` project's resulting `.vsix` extension. This makes it so that only one `.vsix` has to be installed by the students.

4.3.2 Global properties

The extension uses the `app.config` file to store its project-independent settings (e.g. Standard library folder path). These settings are accessed and modified in the `GlobalOptionsGrid` class, which represents directly the GUI shown in Visual Studio's Options menu (Tools → Options → Free Pascal). This Option page is registered by attributing the `FpPackage` class with the `ProvideOptionPage`[1] attribute.

4.3.3 Free Pascal project

In this and the following subsections we will be describing the implementation of the project system (project representation, project creations, project settings and building, running and debugging of projects). As can be seen on figure 4.4, nearly all classes of the project system inherit from classes from Managed Package Framework (MPF). This means that in the implementation of these features majority of the code is already provided to us by MPF and we only need to override methods where we want the functionality to be different.

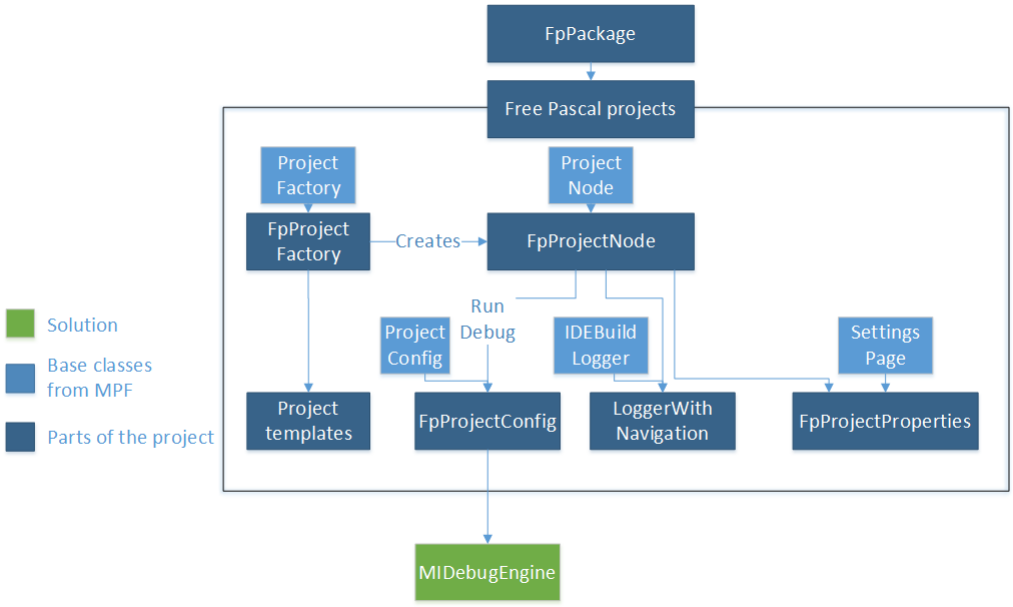

Figure 4.4: Free Pascal project main classes

[1]Full name: `Microsoft.VisualStudio.Shell.ProvideOptionPageAttribute`

The Free Pascal projects are represented by instances of the `FpProjectNode` class. Majority of the project's functionality comes from MPF (Managed Package Framework – the `Microsoft.VisualStudio.Project` project). The methods that are overriden in the `FpProjectNode` class will be described in the subsections which cover the features these methods are used for.

Since the MPF-implemented Free Pascal project supports the creation of new folders within the project structure, but not their removal (which could be slightly confusing to some users), a *Remove* button has been added to the Free Pascal project's (and its files and folders) context menu. This button is created in the `Commands.vsct` file and the code executed upon pressing the button is located in the overriden `ExecCommandThatDependsOnSelectedNodes` method of the `FpProjectNode` class.

4.3.4 New Free Pascal project creation

As can be seen on figure 4.5 the instances of `FpProjectNode` are created by the `FpProjectFactory` class. The `FpProjectFactory` class is responsible for both the creation of new projects and the loading of already existing ones. The `FpProjectFactory` is registered by attributing the `FpPackage` class with the `ProvideProjectFactory`[2] attribute. By registering the project factory we add the Free Pascal project into Visual Studio's *New Project* window. The physical project (the files themselves) are created from the template located in the `Templates/Projects/HelloWorldProject` subfolder of the `ProjectSystem` root folder. This path is registered by the `ProvideProjectFactory` attribute.

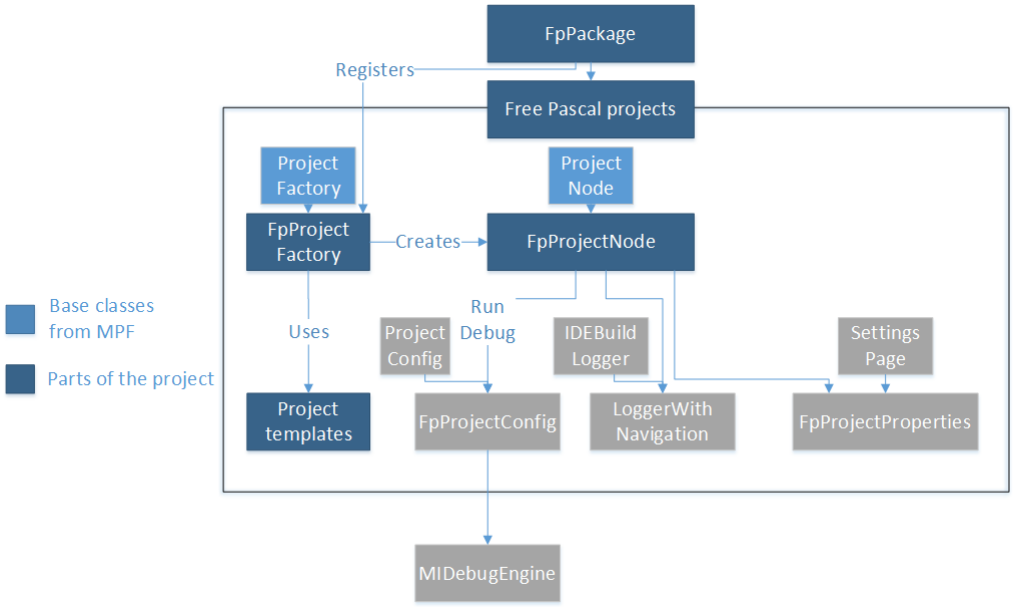

Figure 4.5: `ProjectSystem` classes involved in project creation

As can be seen on figure 4.6, when the new project is being created, first the `HelloWorldProject.fpproj` file is copied over (because the `.fpproj` extension is registered as the default Free Pascal project file extension by the

[2]Full name: `Microsoft.VisualStudio.Shell.ProvideProjectFactoryAttribute`

ProvideProjectFactory attribute) by the FpProjectFactory class into Visual Studio, creating an instance of the FpProjectNode class, and then the Program.pas file gets added by FpProjectFactory.

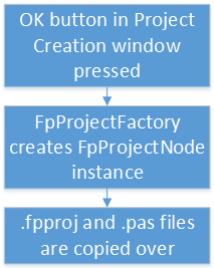

Figure 4.6: Project creation

When the file is being added the AddFileFromTemplate method of the FpProjectNode class is called. This method is responsible for the replacement of the $ProjectName$ token in the Program.pas file by the project's name. This replacement takes into account the lexical rules of Pascal identifiers, meaning that all digits before the first alphabetical character (or an underscore) of the project name and all non-alphanumerical characters (except for underscores) are ignored when creating the project name. Note that the implementation does not check for whether the result program's name will be a keyword, meaning that this process can create a source code file which will not compile.

Additionally, if the global setting of RenameSourceFileToMatchProjectName is set to True, the Program.pas file will be renamed to match the $ProjectName$ token (with the .pas extension). Additionally, since the .pas extension is recognized by the IsCodeFile method, the program.pas (or the renamed file) is added into the Compile Item Group in the project file (more about Item groups in section 2.2).

4.3.5 Adding new empty Pascal file

The ProjectSystem project includes a template to add a single empty source code file into an already existing project. This template is stored in the Templates/Items/EmptyFile folder and is registered by attributing the FpPackage class with the ProvideProjectItem[3] attribute. Note that this template has been added only for the purpose of re-adding the single source code file in case a student would remove his previous one as this extension assumes that there is only one source code file in one project (according to the requirements set in section 1.3).

4.3.6 Free Pascal building

When the *Build project* button is pressed, the ProjectNode class (from MPF – the Microsoft.VisualStudio.Project project) calls MSBuild to perform the Build target on the Free Pascal project file (see figure 4.7. This functionality is not overriden in the FpProjectNode class. The Build target calls the Free

[3]Full name: Microsoft.VisualStudio.Shell.ProvideProjectItemAttribute

Pascal Compiler and creates error and warning log messages according to what the compiler emits (the MSBuild part – grey area in figure 4.7 – of the compilation is explained in detail in section 4.5).

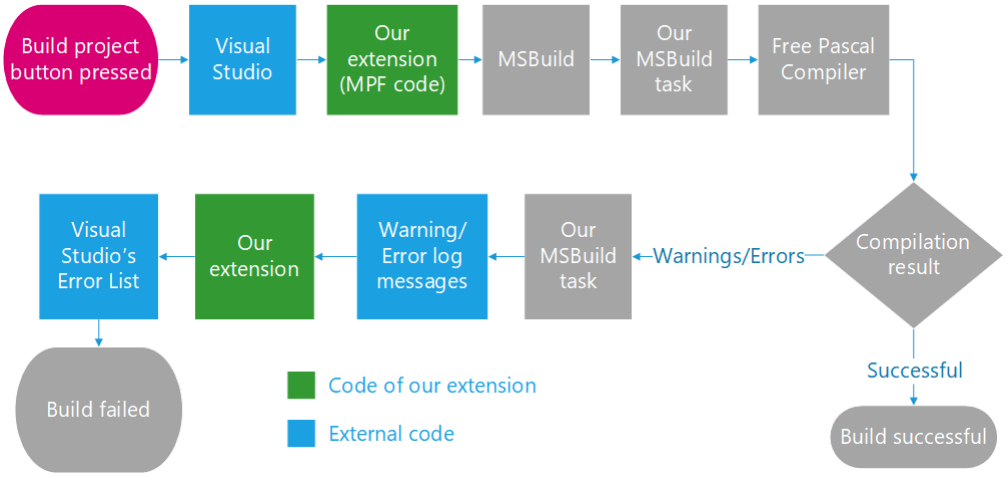

Figure 4.7: Build process within the `ProjectSystem` project

The warning and error log messages generated by MSBuild are captured by the `LoggerWithNavigation` class. The `LoggerWithNavigation` inherits from the `IDEBuildLogger` (from MPF) class, which contains the functionality of adding new warning and error messages into Visual Studio's Error List. The `LoggerWithNavigation` extends on that functionality by adding a callback to its `Navigate` method when the message is double clicked on. This callback is added in the `CreateTaskEvent` method. Note that the `CreateTaskEvent` method was not virtual in the original MPF. The addition of the virtual keyword to the `CreateTaskEvent` method is one of the few changes made into the MPF (`Microsoft.VisualStudio.Project`) project.

Likewise, the creation of the `IDEBuildLogger` was not virtual in the `SetOutputLogger` method of the `ProjectNode` class. A virtual `GetLogger` method was therefore added to the `ProjectNode` class in the MPF project. Since the `IDEBuildLogger` was instantiated only in one place in the code (the `SetOutputLogger` method), it has been rewritten to use the `GetLogger` method instead. Additionally, the `GetLogger` method is overriden in the `FpProjectNode` class to instantiate the `LoggerWithNavigation` class instead.

4.3.7 Running the program with and without a debugger

As can be seen in figure 4.8 when either of the *Start debugging* or the *Start without debugging* buttons is pressed, the `FpProjectConfig` class is called. The `FpProjectConfig` class then either runs the program by using `cmd.exe` or starts the debugging session using MI Debug Engine.

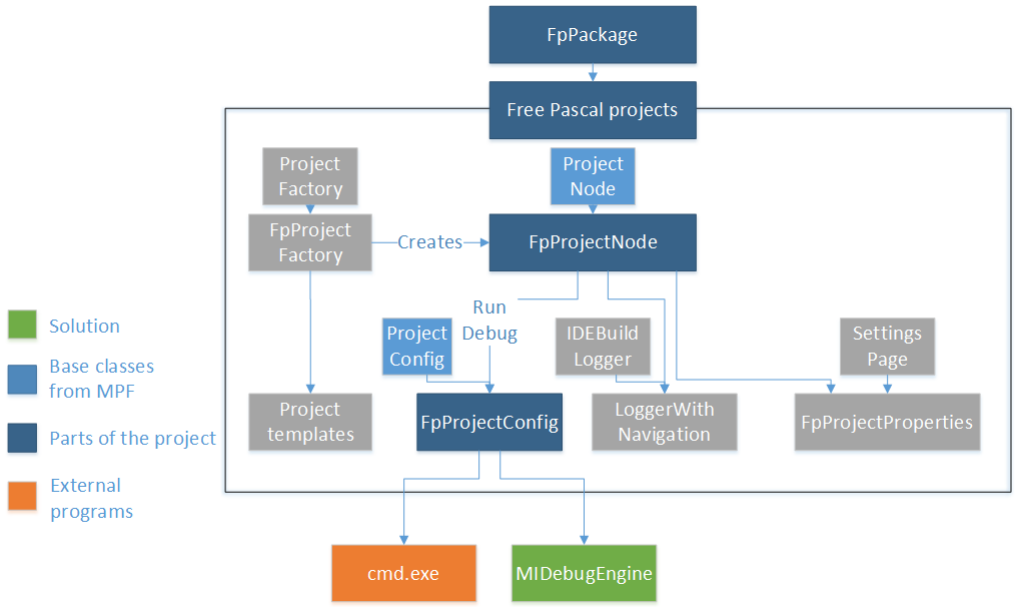

Figure 4.8: `ProjectSystem` classes involved in debugging

Regardless of whether the program is run with or without a debugger, the `DebugLaunch` method on the `FpProjectConfig` class is called. This method accepts a single `uint` argument, which represents a value in the `__VSDBGLAUNCHFLAGS` flag enum, which is used to determine whether to launch the program with or without a debugger. When the program is ran without debugger, it is ran by using `cmd.exe` and is continued by the `pause` command (so that the *Press any key to continue* message is shown before closing the console window). Otherwise, the static `LaunchDebugger`[4] method is used to run the MIEngine Debug engine, which will then handle all debugging communication with both the Visual Studio and the underlying debugger (GDB).

The Debug engine is configured by a XML string generated by the `GetMIOptions` method in the `FpProjectConfig` class. This XML format is defined by the `LaunchOptions.xsd` file (located in the `MIEngine/src/MICore` directory). This configuration, among other things, contains the `<SetupCommands>` element, which allows us to specify which GDB commands we want to run before the debugging begins. Currently it is used to set disassembly to intel syntax and to tell GDB that it is debugging Pascal. The latter setting allows for expression evaluation to work also for Pascal syntax.

The instances of the `FpProjectConfig` are created by the `FpConfigProvider` factory class, which is created by the overriden `CreateConfigProvider` method of the `FpProjectNode` class.

4.3.8 Project properties

There are two kinds of the project specific properties (accessible from a Free Pascal project's context menu): Configuration dependent (i.e. they may have different values for e.g. Debug or Release configuration) and configuration independent.

[4]Located in the `Microsoft.VisualStudio.Shell.VsShellUtilities` class

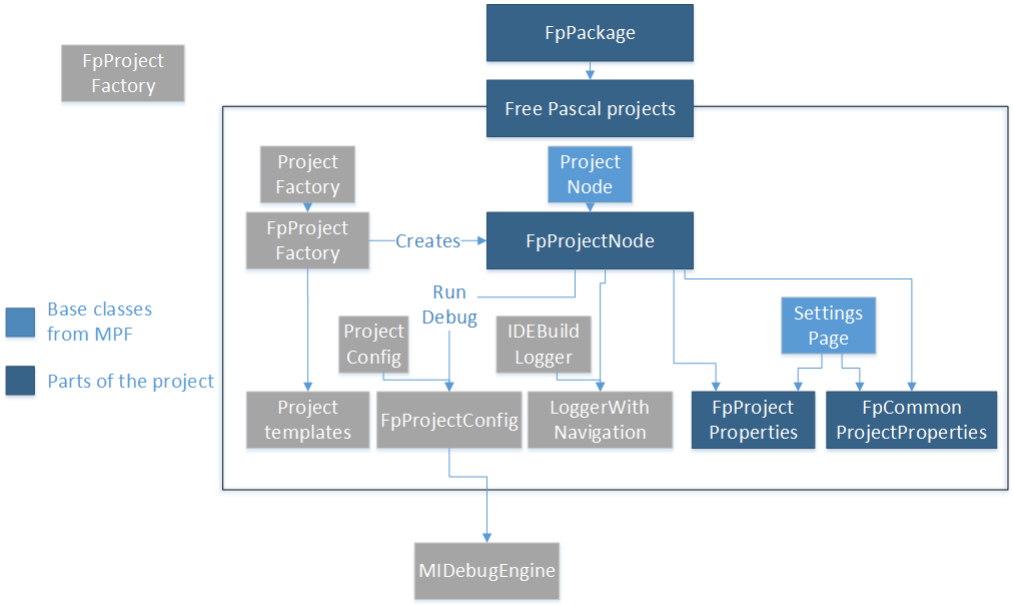

Figure 4.9: Free Pascal project properties

The configuration dependent properties are represented by the `FpProjectProperties` class (see figure 4.9). This class acts as an intermediate layer between the GUI and the project file. The `FpProjectProperties` class is registered to Visual Studio by attributing the `FpPackage` class with the `ProvideObject`[5] attribute. Additionally, the `GetConfigurationDependentPropertyPages` method of the `FpProjectNode` class has to be overriden to return the GUID of the property pages.

Similiarly, the extension is prepared to offer configuration independent properties (represented by the `FpCommonProjectProperties` class and returned by the `GetConfigurationIndependentPropertyPages` method). Note that even though this extension is currently not using any configuration independent properties, the `GetConfigurationIndependentPropertyPages` method must still be overriden to return an empty array (since the base implementation returns an array with a single GUID – a zero GUID), otherwise the project properties menu would be inaccessible due to Visual Studio not being able to found property pages with the zero GUID.

4.3.9 Managing abstract syntax trees

Since both syntax highlighting and code completion features require access to the Abstract syntax tree (AST – described in subsection 4.8.1) representing the current version of the source code, the `TreeUpdater` class was introduced (see figure 4.10) to hold an instance of the tree and to listen to any changes in the source code, so that the AST instance always represents the current version of the source code.

[5]Full name: `Microsoft.VisualStudio.Shell.ProvideObjectAttribute`

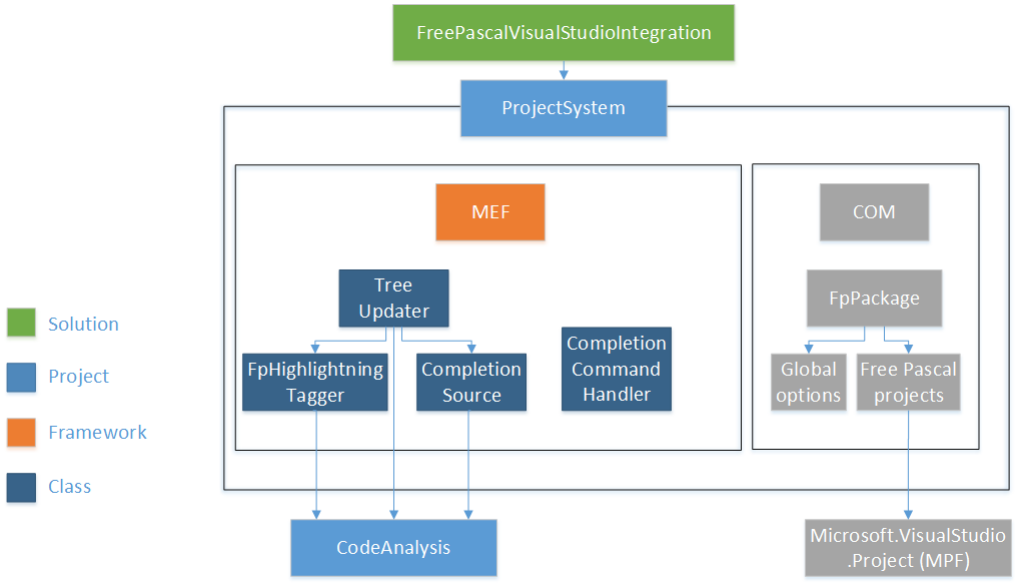

Figure 4.10: `ProjectSystem` project – MEF section

Whenever a tree is first requested for a given `ITextBuffer`[6] (an object representing the open source code document), the `TreeUpdater` creates an instance of the `FpSyntaxTree` class and it also registers a callback to the `ITextBuffer`'s `Changed` event. Since the current Parser implementation does not support reparsing of only part of the source code, the entire source code is reparsed instead, creating a new `FpSyntaxTree` instance, which then replaces the old one. Note that the `Changed` event includes a list of exact changes which happened (e.g. a character removed, a word inserted or replaced), which means that once the parser can parse only part of the source code, it should be easy to detect which part needs to be reparsed.

The `TreeUpdater` is exported via MEF(Managed Extension Framework, more info in section 2.3), which means that it is not directly instantiated by any other class within the `ProjectSystem` project. Instead, the classes which use the `TreeUpdater` class attribute their public `TreeUpdater` instance with the `Import` attribute.

4.3.10 Syntax highlighting

As described in section 3.6 Visual Studio uses tags to mark the source code before highlighting it. The tags are instances of classes implementing the `ITag`[7] interface. This extension uses two kinds of tags: the `ClassificationTag`[8] to mark pieces of source code as e.g. a keyword or an identifier and the `ErrorTag`[9] to add red squiggles under pieces of source code. Both kinds of tags are created by the `FpHighlightningTagger` class, which serves as the central class for the syntax highlighting feature (see figure 4.11).

[6]Full name: `Microsoft.VisualStudio.Text.ITextBuffer`
[7]Full name: `Microsoft.VisualStudio.Text.Tagging.ITag`
[8]Full name: `Microsoft.VisualStudio.Text.Tagging.ClassificationTag`
[9]Full name: `Microsoft.VisualStudio.Text.Tagging.ErrorTag`

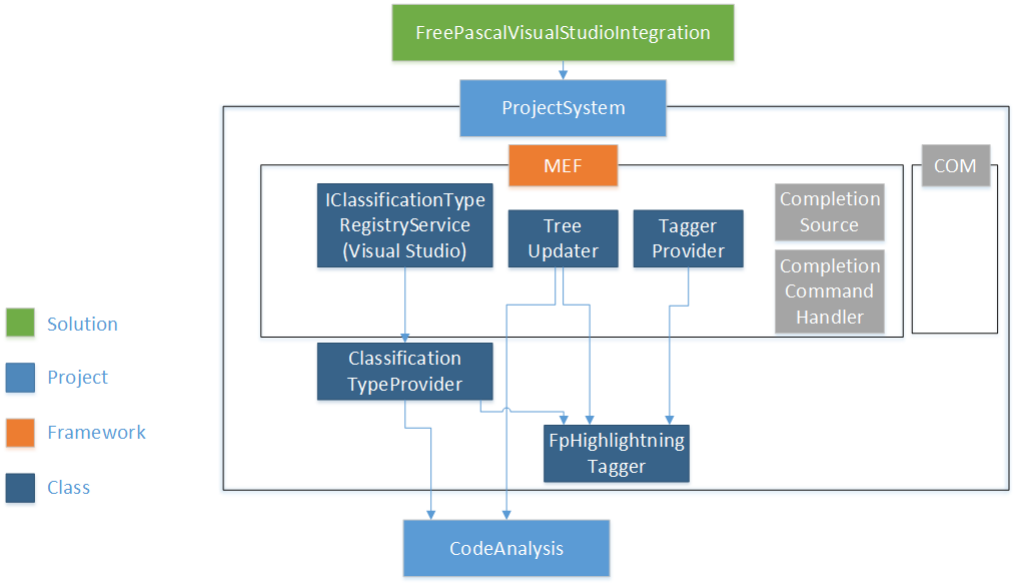

Figure 4.11: `ProjectSystem` project – Syntax highlighting classes

As can be seen on figure 4.12 when Visual Studio calls the `FpHighlightningTagger` to tag a text span (an interval) of source code, the `FpHighlightningTagger` class first finds the intersection between the interval and the AST (abstract syntax tree – described in subsection 4.8.1) provided by the `TreeUpdater` class. Then it uses the `ClassificationTypeProvider` class to classify the text spans represented by the leaf nodes in the aforementioned intersection (see figure 4.11).

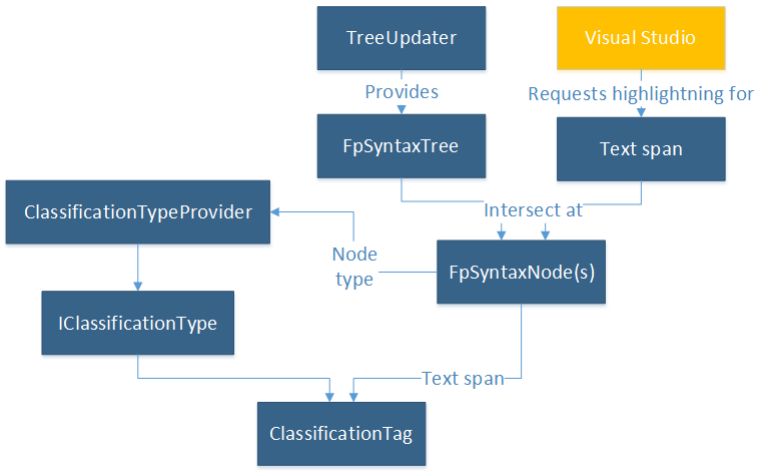

Figure 4.12: Syntax highlighting process

The `ClassificationTypeProvider` class uses the `IClassificationTypeRegistryService`[10] class provided by Visual Studio's text editor (respectively by MEF) to obtain the already existing instances of the `IClassificationType`[11] interface. To decide which instance of the

[10] Full name: `Microsoft.VisualStudio.Text.Classification.IClassificationTypeRegistryService`

[11] Full name: `Microsoft.VisualStudio.Text.Classification.IClassificationType`

`IClassificationType` interface to return, the `ClassificationTypeProvider` class uses the `SyntaxKind` enum (which contains values for all terminals and nonterminals of the supported subset of the Free Pascal language) together with the `SemanticModel` instance obtained from the AST (`SemanticModel` is described in subsection 4.8.4).

The `ErrorTag` version works by finding the intersection between the AST and the given interval, then by searching the intersection for any node of the `UnexpectedTrivia` class. Note that in the current implementation no `ErrorTag`s are created for missing keywords.

It is important to note that the `FpHighlightningTagger` is used in two different ways:

1. Visual Studio calls both the `GetTags` methods (for `ClassificationTag` and for `ErrorTag`) on the `FpHighlightningTagger` instance, giving them a collection of text spans (an ordered set of text intervals) to tag. This collection usually consists of only a single interval covering the line of text which was currently modified by the student (the only exception being when the source code file is first opened, then the entire file is given to the tagger to tag). This, however, does not update the highlightning properly when the student e.g. removes the end of a multiline comment (then all the other lines should get tagged as one huge comment).

2. The `FpHighlightningTagger` invokes the `TagsChanged` event which the Visual Studio is hooked up to, which forces Visual Studio to call the `GetTags` methods with the interval with which the `TagsChanged` event has been invoked.

Since only the multiline comments are causing issues with syntax highlighting when the tags are updated by whole lines (as described in 1.), the `FpHighlightningTagger` is keeping track of multiline comments separately and invokes the `TagsChanged` event when a multiline comment is created or removed.

4.3.11 Code completion

The code completion (Intellisense in Visual Studio) source code is based on the code written in the *Walkthrough: Displaying Statement Completion* guide [13]. The functionality is split into two classes: `CompletionCommandHandler` and `CompletionSource` (shown in figure 4.13).

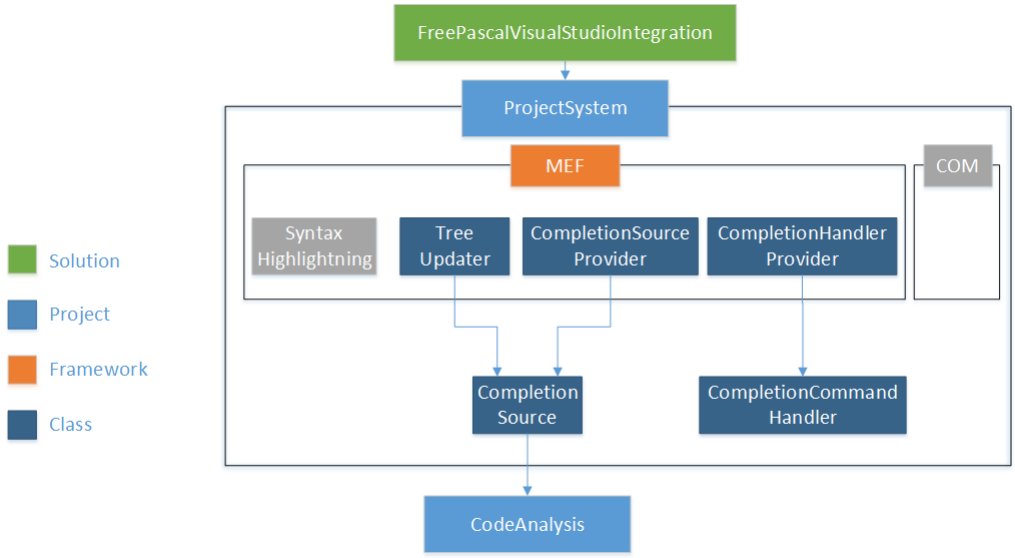

Figure 4.13: Syntax highlighting process

The `CompletionCommandHandler` class is responsible for the management of Intellisense sessions – the *Initiation*, to open a code completion window, the *Commitment*, to use a selected code completion option to modify the source code and close the Intellisense session, and the *Dismissal*, to close the Intellisense session without modifying the source code. The `CompletionCommandHandler` does this by integrating itself into the Visual Studio's text editor event handler pipeline, where it listens to commands (pressed keys on keyboard) and creates, commits and dismisses the Intellisense sessions accordingly.

It is important to note that the original implementation (from the *Walkthrough: Display Statement Completion* guide [13]) was not good enough, as it did not take into account that identifiers can contain underscores (underscores would not initiate the completion session, but they would commit an existing session), which we had to fix. In addition, when the `Enter` key was used to commit the session, a newline was emitted in the editor. Since this functionality is not present in the C# Intellisense, we have removed it. Lastly, we have added the possibility to initiate an Intellisense session by using the `CTRL + Space` key combination (same combination used in the C# Intellisense).

The `CompletionCommandHandler` instance is added into the text editor's pipeline by the `CompletionHandlerProvider` class, which is created by MEF. The `CompletionHandlerProvider` class is fully based on the *Walkthrough: Displaying Statement Completion* guide [13].

The `CompletionSource` class is responsible for the generation of correct code completions in its `AugmentCompletionSession` method. To accomplish that the `CompletionSource` first detects which kinds of syntax terminals can be used in the currently active Intellisense session (by using the `FpGrammar` class) and then it adds their string representations into the final `CompletionSet` instance. If one (or more) of the possible syntax terminals are identifiers, then the `SemanticModel` class (obtained from a `FpSyntaxTree` instance provided by `TreeUpdater` instance) is queried for the possible identifiers and these are inserted into the `CompletionSet` result as well.

The `CompletionSource` instance is created by the

`CompletionSourceProvider` class, which is created by MEF. The `CompletionSourceProvider` imports an `TreeUpdater` instance for the `CompletionSource`.

4.3.12 Project constants

There are two types of constants that are used in this project – string constants and GUIDs. The string constants are stored in the `FpConstants` class. The GUIDs are written in the `Commands.vsct` file. The .vsct (Visual Studio Command Table) file is an XML file which lists various GUI elements (e.g. menus, buttons, windows) that the extension adds to Visual Studio. These files are used so that Visual Studio can show these GUI elements without actually loading the extension (the extension is loaded only after one of these GUI elements is used, e.g. a button is pressed). It is important to note that these .vsct files are not used directly, but instead are first compiled into binaries, which are then in turn used by Visual Studio. During this compilation a C# source code file containing all the GUIDs defined in the .vsct file is also created. This source code file contains the `PackageGuids` class, which contains both the GUIDs and their string representations. This `PackageGuids` class is used across the project to reference the various GUIDs.

4.4 Microsoft.VisualStudio.Project project

`Microsoft.VisualStudio.Project` (see figure 4.14) is the source code of Managed Package Framework (MPF) and its classes are used by the `ProjectSystem` project as base classes. Judging from the vast amount of virtual and factory methods contained within MPF it seems that MPF classes are designed to be used primarily as base classes. We have used this fact to modify the MPF code as least as possible, since MPF itself seems to be heavily undocumented. However, since methods that create and use MSBuild logger were not virtual and we wanted to override their functionality (to provide our own logger), we had to refactor and virtualise some methods in MPF. These modifications were described in subsection4.3.6.

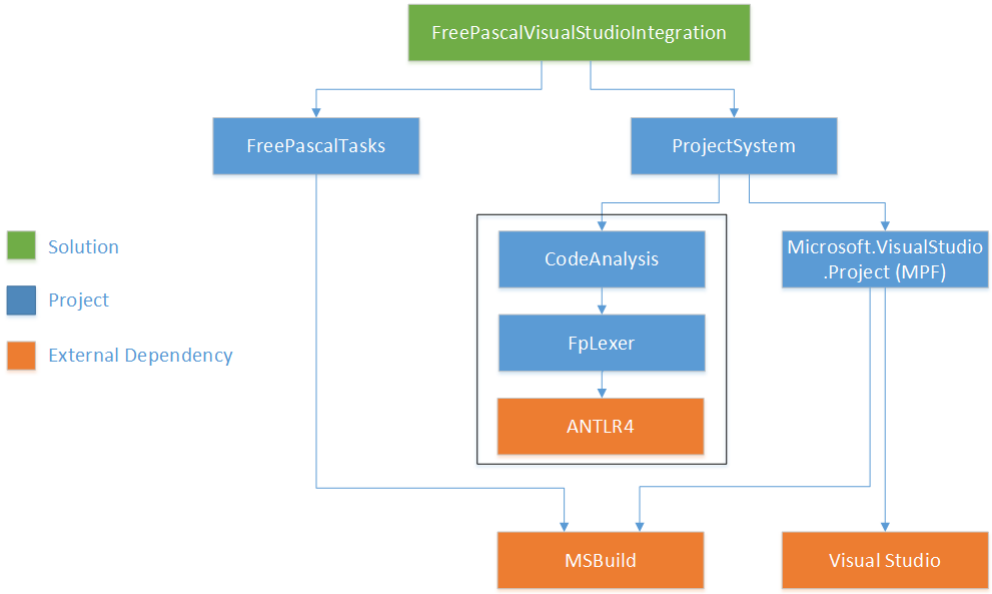

Figure 4.14: Solution and project hierarchy

4.5 FreePascalTasks project

The `FreePascalTasks` (see figure 4.14) project's goal is to create a `.targets` (MSBuild targets file, XML-based) file containing the `Clean`, `Rebuild` and `Build` MSBuild targets. These targets are ran when the student presses the *Clean project*, *Rebuild project* or *Build project* buttons respectively.

The `Clean` target will delete the folders defined by the `OutputPath` and `ObjectPath`) MSBuild variables. These variables are defined in the various Free Pascal project instances, which, unless changed by the student, define these variables as `bin` and `obj` respectively.

The `Rebuild` target uses the other two targets to do its job: it first runs the `Clean` target, then the `Build` target.

The `Build` target uses a custom MSBuild task in its implementation (see figure 4.15). By default, MSBuild does not know of any of our custom tasks (e.g. `FreePascalCompile` task). We must import these into MSBuild by adding the <`UsingTask`> element into the .targets file. This element contains a path to the assembly file where the task's code is contained and a full class name (including namespaces) of the class containing the task's code. The imported task will always be called by its class name, i.e. the `FreePascalCompile` task must have its code contained in the `FreePascalCompile` class. Since this poses an ambiguity issue for this thesis, we will strictly use `MSBuild`, `XML` and `Task` when referring to the .targets file (i.e. MSBuild properties, XML attributes and `FreePascalCompile` task) and `C#`, `class` and `instance`, when referring to the .NET concepts (i.e. C# properties, C# attributes and `FreePascalCompile` class/instance).

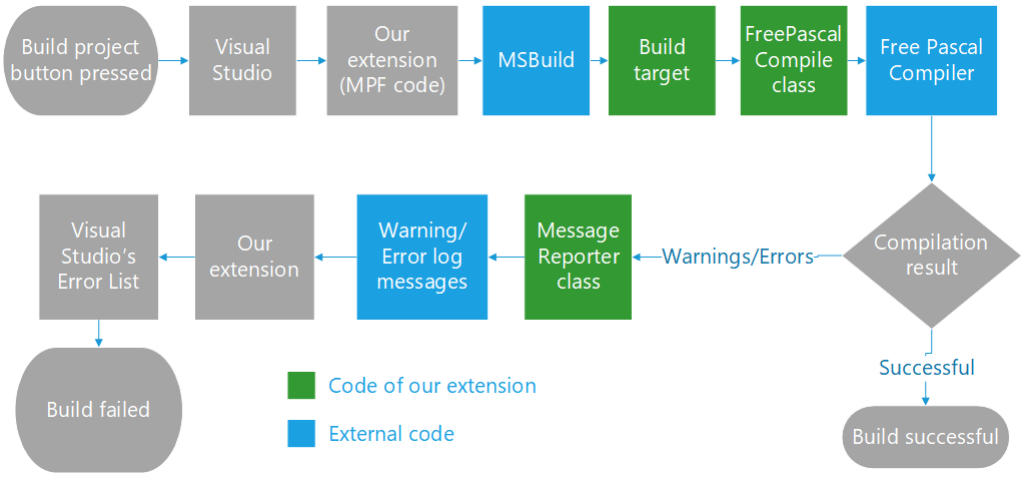

Figure 4.15: Build flow in `FreePascalTasks` project

The `Build` target redirects all its functionality to the `Fpc` target. The `Fpc` target first defines `Inputs` and `Outputs` sets. The `Inputs` set contains the source code file(s) and the `Outputs` set contains the .o (compiled object file) files and the .exe (compiled executable) file. MSBuild uses these sets to decide whether the source code file changed since the binaries were last compiled and if there has been no change, MSBuild skips the `Fpc` target. If MSBuild decides to run the `Fpc` target, first all the variables needed by the `FreePascalCompile` task are checked whether they have been set. If some of the variables are unset, the `Fpc` target fills them with default values. Then the `FreePascalCompile` task is called with all the various parameters.

Once the `FreePascalCompile` task is called in the .targets file, MSBuild first creates an instance of the `FreePascalCompile` class, then it uses reflection to fill public properties of the `FreePascalCompile` instance using the values of equally named XML attributes in the .targets file. Then MSBuild checks that all public properties of the `FreePascalCompile` instance attributed with the `Required`[12] C# attribute were filled in. If this check is successful, the `Execute` method on the `FreePascalCompile` instance is called.

The `Execute` method first uses the `CommandLineBuilder` class to create the command line arguments for the Free Pascal Compiler (fpc.exe). Then it runs the compiler with the arguments generated by the `CommandLineBuilder`. Lastly, it reads the output of the compiler and uses the `MessageReporter` class to generate log messages with warnings and errors, which are then used by the `ProjectSystem` project to generate entries to the Visual Studio's Error List.

The `CommandLineBuilder` class uses the `CmdBuilder` MSBuild helper class, which handles escaping and the addition of quotation marks (e.g. when the source code file name contains a space).

4.6 Code Analysis

The following two sections will describe the `CodeAnalysis` and `FpLexer` projects (see figure 4.16). Since it makes more sense to start with the `FpLexer` project,

[12]Full name: `Microsoft.Build.Framework.RequiredAttribute`

we will first describe the `FpLexer` project and then the `CodeAnalysis` one.

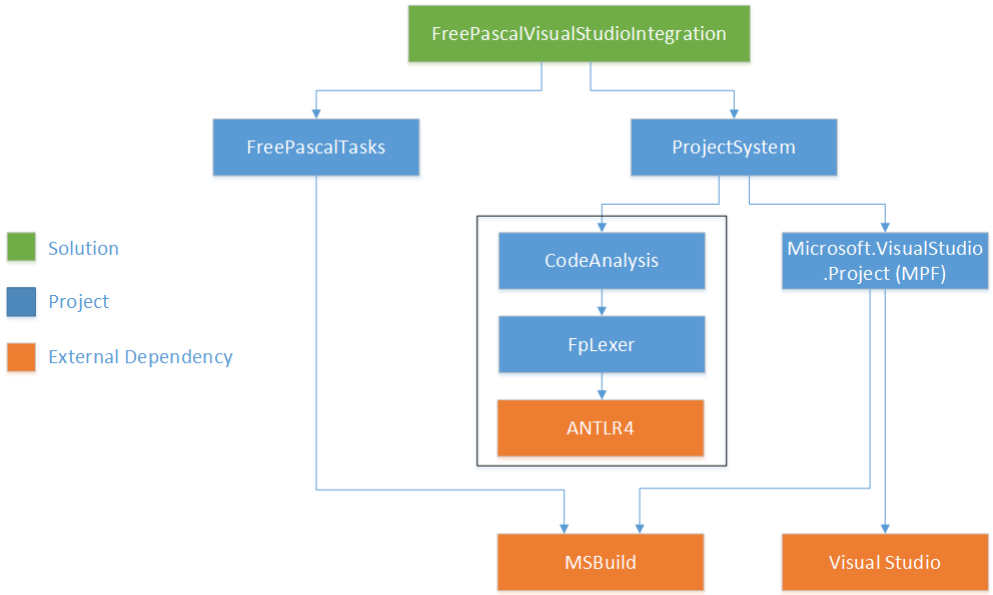

Figure 4.16: Solution and project hierarchy

4.7 FpLexer project

The `FpLexer` is responsible for the generation of `Tokens` out of `ISourceText`. The source text is parsed by the `FpTokenParser` class (more specifically the `GetTokens` method). Since the stream of `Tokens` also contains `TriviaTokens` (whitespaces, comments, etc.), a `TokenSequence` is also defined in the `FpLexer` project to allow for easier access to the `Token` stream (e.g. allows to access all continuous trivia at once, to peek next non trivia, etc.). In order to unambiguously identify the kind of a token, a `SyntaxKind` enum is defined in the `FpLexer` project as well.

4.7.1 Tokens

The `Token` class is an abstract class representing a single lexical Free Pascal token. Each token contains both the underlying string which the token represents and a value of the `SyntaxKind` enum.

As can be seen on figure 4.17, there are various descendants of the `Token` class. The `TriviaToken` abstract class is used to represent all tokens which have no meaning for the Free Pascal language. That includes whitespace (`WhitespaceToken` class), comments (`CommentToken` class) and characters which cannot start a valid token (`UnknownToken` class), e.g. the ‘ character.

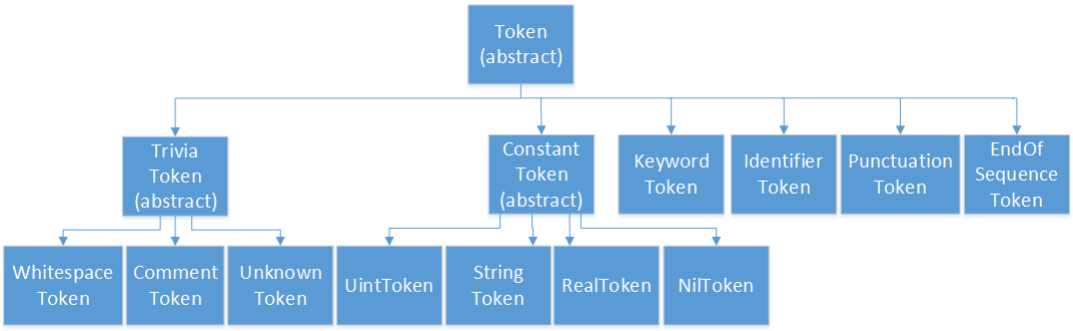

Figure 4.17: Token class hierarchy

The `ConstantToken` abstract class, which inherits directly from `Token` is used to represent integer (`UintToken` class), real (`RealToken` class), string (`StringToken` class) and pointer (`NilToken` class) constants found in the source code. These classes contain public properties which expose the .NET alternative (int, double, string) values to be used by further analysis.

The `KeywordToken` class is used to represent Free Pascal keywords. Note that the `KeywordToken` also stores the string value of the used keyword. This is done so that further analysis could see the exact case used in that specific part of source code (because Free Pascal is case-insensitive).

The `IdentifierToken` class is used to represent all identifiers. It exposes an additional property containing the upper case variant of the identifier. In case of future expansion of this extension, this property can be used to strip the underlying string of the `&` character (if a Free Pascal keyword is prefixed by the `&` character, it is considered an identifier, similiar to C#'s @ symbol).

The `PunctuationToken` class is used to represent Free Pascal punctuation and operator tokens.

Lastly, the `EndOfSequenceToken` is a special kind of token which has no meaning for the Free Pascal language. It is used to mark the end of the source code.

4.7.2 SyntaxKind

The `SyntaxKind` enum defines all terminals and nonterminals used by the subset of the Free Pascal grammar we use (as defined in section 1.3). In order to allow easier queries for whether a specific syntax kind is e.g. a keyword, the enum values are grouped together (e.g. keywords have underlying integer values between 50 and 199, inclusive, punctuation tokens have underlying integer values between 200 and 219, inclusive). These values are then used in the extension methods found in the `SyntaxKindExtensions` class (e.g. `IsTrivia` or `IsKeyword` methods).

4.7.3 Lexer

The `FpLexer` uses ANTLR4 to generate the tokens. The ANTLR4 lexer is configured by the `AntlrLexer.g4` file, which generates source code for C# `AntlrLexer` partial class. This class is then used by the `GetTokens` method of the `FpTokenParser` class to generate the various `Tokens`.

Since no class other than `FpTokenParser` uses the `AntlrLexer` class, replacement of the underlying lexer should not be difficult.

4.8 CodeAnalysis project

The `CodeAnalysis` project is responsible for the generation of abstract syntax trees and most of its analysis. The project contains:

1. `SyntaxTree`, `SyntaxNode` classes and their descendants.
2. Custom recursive-descent syntax parser.
3. Explicit grammar of the subset of the Free Pascal language (as defined in section 1.3).
4. Semantic model, i.e. identifier data for code completion.

4.8.1 Abstract syntax tree

The implementation of the abstract syntax tree consists of two separate tree structures (as described in subsection 3.7.2) – a `PersistentSyntaxTree` (Persistent one) and a `FpSyntaxTree` (OnDemand one). The trees mirror each other in structure (each node of the `FpSyntaxTree` contains a reference to its `PersistentSyntaxTree` counterpart), but differ in their internal data and purpose.

Both of the trees share the following properties:

- Immutability.
- Each kind of nonterminal has its own node class.
- Children nodes cannot be null.
- Nodes can be missing (i.e. those that are not present in the source code, but are required by the grammar, e.g. the PROGRAM keyword). These nodes are then marked with the `IsMissing` property set to true.

The `PersistentSyntaxTree` nodes only knows their length, their children, the symbol they represent in the grammar (as defined by the `SyntaxKind` enum in the `FpLexer` project) and the trivia which immediately precedes and follows the node. The `FpSyntaxTree` (`On Demand` tree) nodes also know their parent and the exact index in the source text they start on. The `On Demand` tree is instantiated only when it is required, and on top of that it also lazily builds the entire tree strucutre (The children of the `FpSyntaxTree` nodes are lazily instantiated using the `System.Lazy` class). The `FpSyntaxTree` nodes also have user-friendly API (compared to the `PersistentSyntaxTree`), which allows for the traversal of the tree by the specific children contained within the node (e.g. the `FpBlock` class has the `Declarations` and the `Body` public properties)

The conversion between the two kinds of nodes is done using the `FpSyntaxNodeFactory` class, which is called from the `ChildrenInitialisation` method of the `FpSyntaxNode` class.

The `FpSyntaxTree` structure allows for intersection queries (to find nodes that are present in given interval) by using the `ChildrenIn` method in the `SyntaxNode` class.

4.8.2 Parser

The parsing process is initiated by calling the static `Parse` method on the `FpSyntaxTree` class. This method will in turn call the static `Parse` method on the `PersistentSyntaxTree` class, which will use the `FpParser` class to create the tree.

The syntax parser logic is fully contained within the `FpParser` class. The `FpParser` is a recursive-descent parser that uses the tokens generated by the `FpTokenParser` class to create a tree of `PersistentSyntaxNode` (the `FpParser` does not create the instance of the `PersistentSyntaxTree` itself). The methods of the `FpParser` class use other `ParseXYZ` methods (e.g. the `ParseBlock` method uses `ParseDeclarationBlock` and `ParseCompoundStatement` methods) the `GetKeyword`, `GetPunctuation` and `GetIdentifier` helper methods to get the `PersistentSyntaxTerminal` leaf nodes of the tree. These helper methods are also responsible for error recovery, i.e. they generate `PersistentMissingTerminal` and `PersistentUnexpectedSyntaxTrivia` nodes. The creation of a `PersistentMissingTerminal` instance represents that a grammar production with a nonexistent terminal (token) occured (i.e. the `FpParser` added a virtual token into the source code, e.g. if a semicolon is skipped). The creation of a `PersistentUnexpectedSyntaxTrivia` instance represents that the token was re-classified as trivia (i.e. has the same status as whitespace or comments) and therefore has no meaning for the grammar.

To provide better code-readability all `ParseXYZ` methods are attributed with `Production` attributes. These attributes mark which method is responsible for which grammar productions and serve to ease synchronisation of the currently supported subset of the Free Pascal language between the `FpParser` and `FpGrammar` classes (this was particularly helpful during the development of the extension, when various parts of the Free Pascal grammar were being implemented).

4.8.3 Grammar

The Free Pascal grammar is represented by the `FpGrammar` class, which allows its users to get the lower-case string representation of all non-identifier terminals of the Free Pascal language from a `SyntaxKind` enum value (the `GetStringAlias` method). Additionally, the `FpGrammar` class contains the `GetPossibleKinds` method, which allows its users to get a set of terminals (represented by the `SyntaxKind` enum) which can be present in the source code at the start of the n-th child of a given nonterminal (the nonterminal and n are input arguments of the `GetPossibleKinds` method).

To support these two methods the `FpGrammar` class contains a `Dictionary` which maps the `SyntaxKind` non-identifier terminals to their lower-case string representation. Additionally, the `FpGrammar` class contains a set of `Production` instances representing the Free Pascal grammar. These instances are provided by the `FpProductionProvider` class, which uses the `ProducionAttribute` attributes used in the `FpParser` class to create the `Production` instances.

4.8.4 Semantic Model

The Semantic model is responsible for maintaining information on identifiers available in the student's Free Pascal source code. The identifiers are represented by the classes implementing the `ISymbol` interface. These include the `VariableSymbol`, `ConstantSymbol`, `ProgramSymbol`, `ProcedureSymbol`, `FunctionSymbol`, `UnitSymbol`, `ParameterSymbol` and the descendants of the `TypeSymbol` classes. The identifiers (`ISymbol`s) form a tree structure where the root is the `ProgramSymbol` and the descendants are the symbols which were declared within their parent's scope (in the student's Free Pascal source code).

The Semantic model is represented by the `SemanticModel` class. The `SemanticModel` class uses the `UnitManager` and `SymbolLoader` classes to create the `ISymbol` tree structure. The `UnitManager` class obtains data on identifiers declared by various units (mostly used to obtain data on the System unit) while the `SymbolLoader` class is responsible for the analysis of the `FpSyntaxTree` to obtain identifier data from the source code itself. Since the `SemanticModel` is tied to a specific instance of a `FpSyntaxTree`, which is an immutable class and which contains a reference to the `SemanticModel` instance, the `SemanticModel` is also designed to be immutable.

The `UnitManager` uses the `ppudump.exe` program distributed with Free Pascal compiler to analyse which symbols the given Free Pascal Unit file (.ppu) exports. Since the `UnitManager` class is used to analyse the Free Pascal standard library files (e.g. the System unit), which are not expected to be modified during the run of our extension, the `UnitManager` caches the exported symbols for unit files it has already analyzed In case a change would occur, the unit information will be reanalyzed

The `SymbolLoader` traverses the `FpSyntaxTree` tree to create the `ISymbol`s representing the identifiers located in the source code represented by the tree. Additionally, to obtain the values of constants (to display the constant values while offering the constant identifiers by code completion) the `ExpressionEvaluator` class is used. The `ExpressionEvaluator` can evaluate all constant expressions allowed in the Free Pascal grammar, including those that use other named constants.

5. User guide

`FPVS -- Free Pascal Visual Studio integration` is a Visual Studio 2015 extension designed to add support for the Free Pascal language into Visual Studio 2015. This user guide is designed for students who have never used Visual Studio 2015 before.

5.1 Installation

Before the installation of this extension make sure that you have a Visual Studio 2015 IDE (any version) and the Free Pascal Compiler installed. Note that while it is possible to install this extension on Visual Studio 2017 as well, it has not been tested and we can not guarantee that the extension will behave as intended.

To install the extension run the `install.bat` file (located in the `Install` directory on attached CD) with elevated rights (Run as Administrator).

5.2 Creating a Free Pascal project

To create a Free Pascal project first open the `File` menu, choose `New` and `Project` (see Figure 5.1).

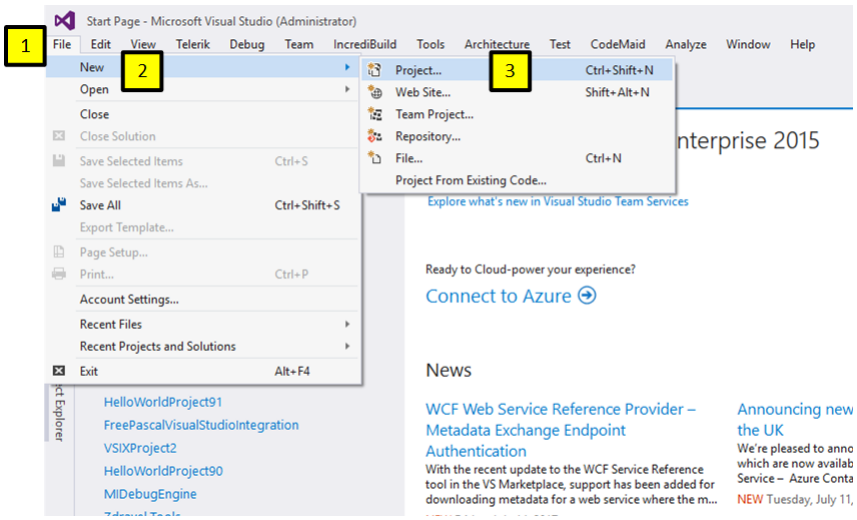

Figure 5.1: Open `New Project` window

On the left side of the `New Project` window select `Free pascal` templates (marked with number 1 on Figure 5.2), then choose the HelloWorldProject template (number 2 on Figure 5.2), pick a name for the project, modify the location where you want the project to be stored (number 3 on Figure 5.2) and click on the `Ok` button (number 4 on Figure 5.2).

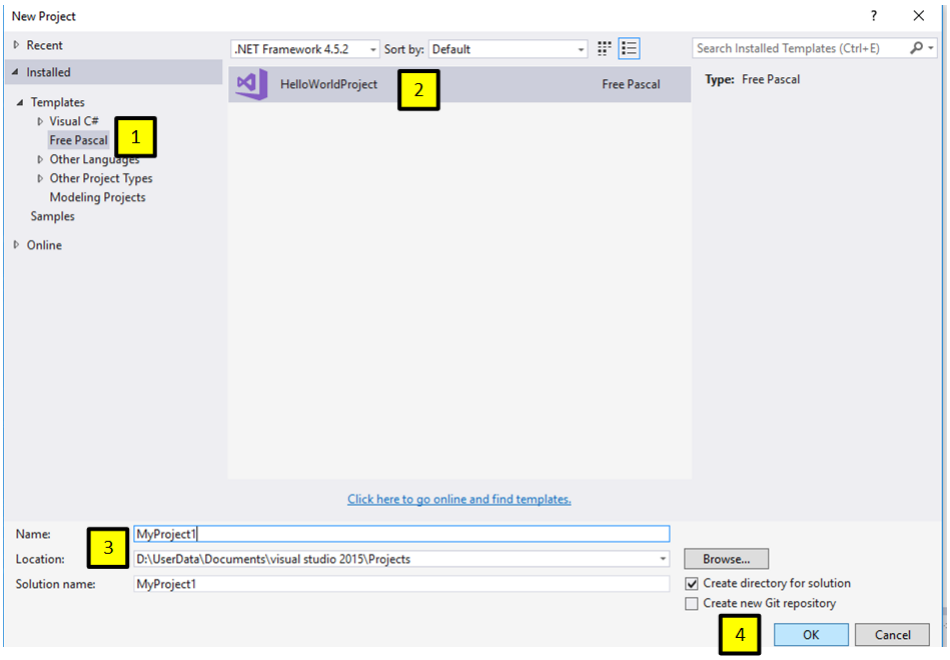

Figure 5.2: Creating project

Once the project is created, open the source code file via the Solution Explorer window (by default open on the right side of the screen) by double clicking on the `NameOfProject.pas` file (see Figure 5.3).

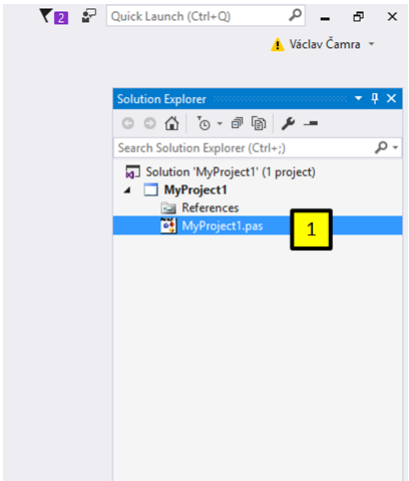

Figure 5.3: Opening Free Pascal code file from solution explorer

The result should look like figure 5.4.

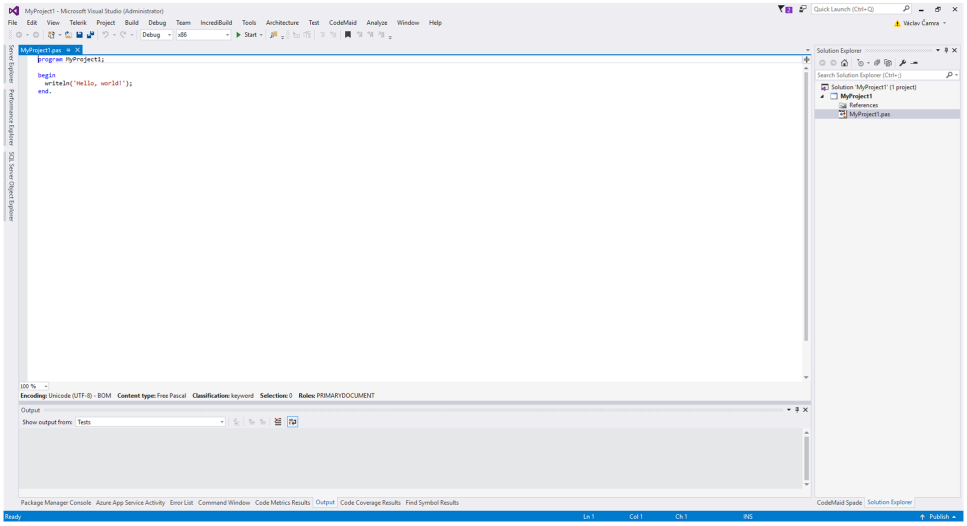

Figure 5.4: Opened Free Pascal code file from solution explorer

5.3 Saving and loading a Free Pascal project

The project can be saved by opening the `File` menu and clicking the `SaveAll` button (or by using the default shortcut Ctrl + Shift + S).

The project can be loaded by opening the `File` menu, opening the `Open` submenu and clicking on the `Project/Solution` button, then by selecting the `.sln` file in the root of the project folder.

5.4 Compiling a Free Pascal project

The project can be compiled by opening the `Build` menu and clicking the `Build Solution` button (or by using the default shortcut Ctrl + Shift + B). As can be seen on Figure 5.5 any warnings or errors regarding the compiled source code will be emitted in the Error List (by default at the bottom of the screen).

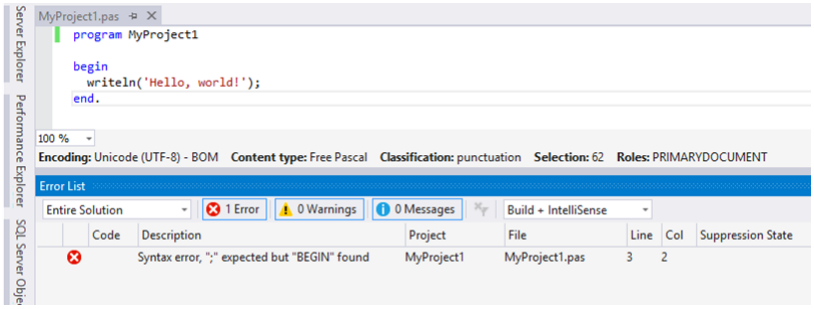

Figure 5.5: Error List

5.5 Debugging a Free Pascal project

The debugging session is started by pressing the *Start* button (alternatively you can open the `Debug` menu and click the *Start debugging* button) or by using the F5 default shortcut.

5.5.1 Stepping through the code

To step through the code we first need to create a breakpoint. Breakpoints mark a line of code before which we want the debugged program to pause. Breakpoints are set by moving the caret (text cursor) to the line at which we want the breakpoint to be made, then opening the `Debug` menu and clicking the *Toggle Breakpoint* button (F9 shortcut by default). The line with a breakpoint is marked by filled red circle to the left of it (see figure 5.6).

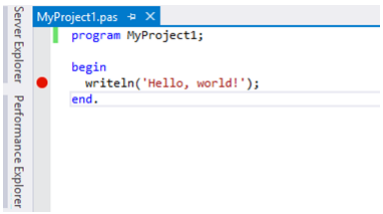

Figure 5.6: Breakpoint

Once the program pauses, you can step through the code by using the *Step Into* (F10), *Step Over* (F11) and *Step Out* (Shift + F11) buttons in the `Debug` menu.

5.5.2 Inspecting current state of the program

Once the program is paused, you can inspect the stack trace and current values of live variables.

The values of currently visible variables can be inspected by using the `Watch` windows (e.g. `Watch 1` window). You can put any variable name (even any valid Free Pascal expression) into the `Name` column, which will result in the variable (or expression) value and type being shown in the watch window (as shown in Figure 5.7). Note that the type of the variable (expression) may not match the type used directly in the source code (e.g. the `i` variable in Figure 5.7 was defined as `Integer`), because the compiler may internally use a different name for the type.

Note that if only the variable name is provided in the Name column, you can also modify the value of the variable by double-clicking on its Value and modifying it there.

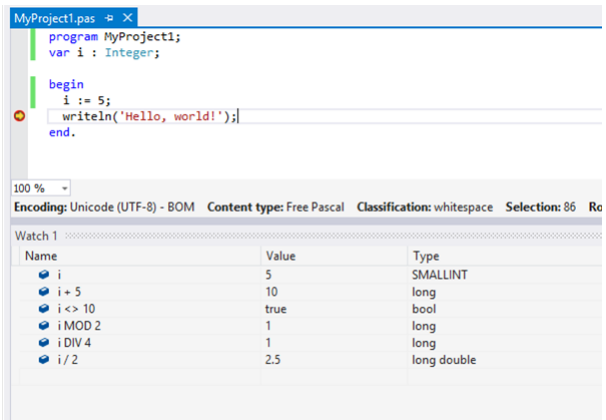

Figure 5.7: Watch List

The stack trace is visible in the `Call Stack` window. You can inspect the context of the caller functions be double-clicking on the lines they represent. This will change the variables that you can see and modify in the watch list.

5.5.3 Disassembler

The disassembly is accessible only when program is paused. To open the disassembly open the context menu (by clicking with right mouse button) anywhere within the text editor and press the *Go to disassembly* button. Example is shown on figure 5.8.

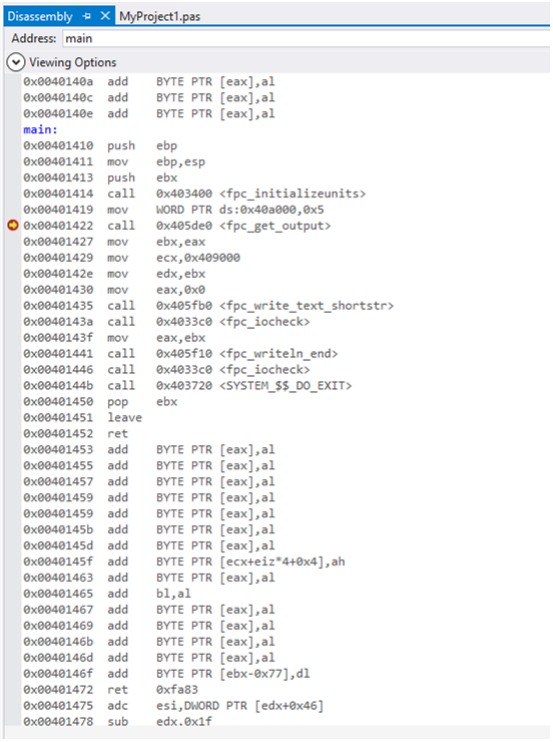

Figure 5.8: Disassembly

Note that the disassembly also supports breakpoints and stepping through the code one instruction at a time.

5.6 Options

The project specific properties can be accessed by opening the context menu of the project in the Solution Explorer window (by default on right side of the screen) and clicking the properties button. Then choose the `Configuration Properties` property page on the left side of the Property Pages window. There are multiple options you can configure in this window:

- Additional Library Paths – you can put project-specific paths to unit source code or binary files.
- Optimization Level – you can specify the optimization level to be used by the compiler, valid values are between 0 (lowest optimizations) and 3

(highest optimizations). If an invalid value is specified, then the compiler is set to optimization level 0.

- Output Path – you can specify where the executable will be placed after the compilation is successful. This path is relative to the project's root folder. By default Output Path is set to `bin\Configuration\`, for example `bin\Debug\`.
- Console arguments – you can specify command line arguments to be used when running or debugging the program from Visual Studio.

Conclusion

To conclude we will first verify that all goals as we have set them in section 1.4 were met by our extension:

1. (Required) Automatic compilation support – our extension allows the student's Free Pascal source code to be compiled by using the *Build project* button in Visual Studio. The source code is compiled by `fpc.exe` (Free Pascal compiler).

2. (Required) Execution support (running the programs) – the compiled program can be run by using the *Start without debugging* button in Visual Studio. This program is then run using `cmd.exe` (Windows command line).

3. (Required) Debugging support – the compiled program can be debugged by using the *Start debugging* button in Visual Studio. This debugging supports reading and writing of global and local variables, breakpoints, stepping through the code, viewing call stack and viewing disassembled code.

4. (Highly desired) Syntax highlighting, including errors – our extension supports syntax highlightning for the subset of the Free Pascal language as defined in section 1.3. The error highlighting (red squiggles by default) is displayed for pieces of source code which do not conform to the syntax of the subset of the Free Pascal language. The error highlighting, however, does not show that a required token (e.g. a semicolon) is missing.

5. (Desired) Code completion for built-in identifiers (i.e. `String`, `Writeln`, `Readln`) – All identifiers defined by the `System` unit (contains e.g. `Integer` type) are offered in the code completion system. In addition to offering the identifiers, the code completion also shows detailed information about the identifiers, for example the type of a variable identifier (e.g. `Integer`) or the base type of a named type (e.g. `array [1..10] of Integer`).

6. (Desired) Code completion for custom identifiers – All identifiers used by the subset of the Free Pascal language as defined in section 1.3 are included in the code completion system. Similar to the additional information offered for built-in identifiers the code completion also offers more information about custom identifiers.

7. (Desired) Code completion for keywords – all keywords that are used in the subset of the Free Pascal language as defined in section 1.3 are offered in code completion.

We have, however, not introduced a system to allow for loading of Free Pascal IDE or Lazarus IDE project files and converting them into our own.

Future work

While our extension offers all features we required in section 1.3, we believe it can still be enhanced to allow the students for better experience with our extension.

Following list contains possible features that could be added to this extension (in no particular order):

- Because of the nature of how strings work in Pascal (a record consisting of a length N, and an array of characters where only the first N are valid) the debugger does not allow for an easy way to see the text represented by a string. This could be improved by showing the string value next to the name of the variable, while still allowing the students to 'open' the string and see its internal structure (for learning purposes).
- Allow to set the Free Pascal Compiler path (which is stored in the MSBuild extension folder) from Visual Studio.
- Highlighting for paired parentheses, brackets and `BEGIN` and `END` keywords.
- Automatic formatting of the student's source code (similar to C#) to show students how code is usually formatted (while also helping them see the structure of their own source code better).
- Tooltips while writing subprogram (procedure/function) calls that would show the arguments the subprogram requires (similar to C#).
- Provide real-time Error List updates, either by running the compiler in the background or by adding Error List entries whenever the error squiggles are used to mark piece of the source code.
- Code snippets for e.g. procedure/function headers.

In addition, since Visual Studio 2017 was recently released (in March 2017 [14]) and even though our initial testing has shown that our extension works on Visual Studio 2017 as well, the extension was marked as not compatible with Visual Studio 2017. It would be helpful to modify our extension so that the warning message would no longer appear during installation in Visual Studio 2017.

Bibliography

[1] Microsoft. List of limitations in 64-bit Windows. Available at https://support.microsoft.com/en-us/help/282423/list-of-limitations-in-64-bit-windows.

[2] Microsoft. Extending projects. Available at https://msdn.microsoft.com/en-us/library/bb286970.aspx.

[3] Microsoft. Editor and language service extensions. Available at https://msdn.microsoft.com/en-us/library/dd885118.aspx.

[4] Microsoft. Task writing. Available at https://msdn.microsoft.com/en-us/library/t9883dzc.aspx.

[5] WPF in Visual Studio 2010 - Part 1 : Introduction. Available at http://blogs.msdn.com/b/visualstudio/archive/2010/02/16/wpf-in-visual-studio-2010-part-1.aspx.

[6] Microsoft. Creating a basic project system, part 1. Available at https://msdn.microsoft.com/en-us/library/cc512961.aspx.

[7] Rainer Schuetze. cv2pdb. Available at https://github.com/rainers/cv2pdb.

[8] Andrew B Hall. Debugging C++ code on Android with Visual Studio 2015. Available at https://blogs.msdn.microsoft.com/devops/2014/11/12/debugging-c-code-on-android-with-visual-studio-2015/.

[9] Microsoft. MIEngine git repository. Available at https://github.com/Microsoft/MIEngine.

[10] Peachpie. Available at http://www.peachpie.io/.

[11] ANTLR v4. Available at https://github.com/antlr/antlr4.

[12] Hanspeter Mössenböck, Markus Löberbauer, and Albrecht Wöß. The compiler generator Coco/R. Available at http://www.ssw.uni-linz.ac.at/Coco.

[13] Walkthrough: Displaying Statement Completion. Available at https://msdn.microsoft.com/en-us/library/ee372314.aspx.

[14] Join Us: Visual Studio 2017 Launch Event and 20th Anniversary. Available at https://blogs.msdn.microsoft.com/visualstudio/2017/02/09/visual-studio-2017-launch-event-and-20th-anniversary/.

Attachments

Content of the attached CD

- `src` folder containing the `FreePascalVisualStudioIntegration` and `MIDebugEngine` solutions. The solution folders contain `packages` folders, which contain all nuget libraries neccessary to compile the extension.
- `Install` folder containing the installation script, the `.vsix` extension and the MSBuild extension folder.
- `thesis.pdf` file containing this thesis.
- `README.txt` file containing information about the content of the CD and the author's email contact.