Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Computer science department

Semester 6 Project

2017-18

IntelliProlog

Owen McGill

Client:	HEIA-FR
Supervisor:	Frédéric Bapst

HEIA-FR
HTA-FR

Fribourg, Mars 2018

Hes·so
Haute Ecole Spécialisée
de Suisse occidentale
Fachhochschule Westschweiz

Table of Contents

1 IntelliProlog Introduction

This chapter will discuss the global structure of this report concerning the development of an IntelliJ plugin for the programming language Prolog, specifically the GNU Prolog [1] implementation developed by Daniel Diaz [2].

1.1 Introduction

This report will be structured as a tutorial for developing a custom language plugin for the IntelliJ platform.

1.1.1 Motivation

The motivation behind this project is that in the Logical Programming course at the Applied Sciences University of Fribourg, students learn to program in GNU Prolog.

More and more students are using IntelliJ or other IDEs developed by JetBrains for their development needs and for the moment there does not exist a good way to write and run GNU Prolog code in IntelliJ.

A previous project developed a plugin for the Eclipse IDE, but since that IDE is used less and less nowadays, the decision was taken to start a new project to develop a plugin for IntelliJ and possibly other variants of IDEs developed by JetBrains.

1.2 Structure of project

The structure of the project will be as follows:

- Tutorial: How to create a custom language plugin
- Differences between the JetBrains IDEs

- Comparison of the different Prolog implementations
- Quick overview of different licenses for software

1.2.1 Tutorial: How to create a custom language plugin

The tutorial is the main part of this report, it will go over the main aspects of the development of a custom language plugin for the IntelliJ IDEA platform.

The features we'll be implementing are the following:

1. Launching the current file in the GNU Prolog interpreter
2. Parser and Lexer for GNU Prolog
3. Highlighting including syntax and errors
4. Code commenting and folding

Before we get to implementing the features, we will go over the steps for setting up a project needed to develop our IntelliJ Platform plugin.

1.2.2 Differences between the JetBrain IDEs

During the process of the project we discovered that there are differences between the different JetBrains IDEs, mainly that the main IntelliJ IDE and the "smaller" IDEs, eg. Goland, PyCharm, WebStorm, etc.

We will go through some of these differences and the impact these have on the development of a JetBrains IDE plugin.

1.2.3 Comparison of the different Prolog implementations

In this section of the report we will compare a small sample of the different Prolog implementations that are available. We will have a quick look at ISO Prolog and Edinburgh Prolog, the interfacing between Prolog and other programming languages and some other features more specific to some of the implementations.

1.2.4 Quick overview of different licenses for software

In this section of the report we will do a quick overview of a couple of different licenses for software, more specifically Open Source Software (OSS) licenses, the licenses that we will discuss are the MIT license and Apache 2.0 license.

2 Project Setup

This chapter goes over the necessary setup needed for developing a plugin for the IntelliJ platform. We will also go over the different plugins that will be used during the development of our Prolog plugin.

This chapter is divided into 3 main sections, these being:

1. Introduction
2. SDK setup
3. Project initialisation

2.1 Introduction

In this section we will go over the requirements needed for the development of IntelliJ plugins.

The most basic requirement needed to develop a plugin for the JetBrains platform is Java, the programming language that is used to develop and run all IntelliJ IDEs.

The other requirements are the following:

- IntelliJ IDEA Ultimate or Community Edition
- IntelliJ IDEA Community Edition source code
- Plugin DevKit plugin
- Grammar-Kit plugin
- PsiViewer plugin

2.1.1 IntelliJ IDEA Ultimate or Community Edition

IntelliJ IDEA is needed for the development of our plugin, and two versions exist:

- Ultimate Edition is a commercial closed-source IDE developed by JetBrains and costs €149 the 1st year, but has a 30 day trial. Students can get a free license with their student email address. The Ultimate Edition allows the use of any plugin developed for a JetBrains IDE as well as including other advanced features.
- Community Edition is an Apache 2.0 licensed open-sourced IDE developed by JetBrains. The Community Edition can only use plugins developed specifically for it.

The two different editions can be downloaded from the IntelliJ IDEA website [3].

2.1.2 IntelliJ IDEA Community Edition source code

The IntelliJ IDEA Community Edition source code is not essential but very useful during the development of plugins for the JetBrains IDEs since it allows easier debugging and inspection of the source code that is the basis of the plugin.

The source code can be checked out from the IntelliJ IDEA Community Edition repository on GitHub [4], using either IntelliJ IDEA or from the command line.

2.1.3 Plugin Devkit plugin

The Plugin DevKit is an IntelliJ plugin developed by JetBrains, and adds support for developing IntelliJ plugins using the IntelliJ IDEA build system. It also adds an easy way of building and launching our plugin in a separate instance of IntelliJ IDEA.

This plugin is bundled with all IntelliJ IDEA IDEs, it just needs to be enabled in the IDE settings.

2.1.4 Grammar-Kit plugin

The Grammar-Kit plugin is developed by Greg Shrago and recommended by JetBrains, it helps write and visualize BNF grammar definitions as well JFlex lexer definitions. It is also capable of generating parser/ElementTypes/PSI classes and running the JFlex generator. Both of those capabilities are essential for developing a custom language plugin.

This plugin can be installed through the plugin interface in IntelliJ IDEA.

2.1.5 PsiViewer plugin

The PsiViewer plugin is a Program Structure Interface (PSI) tree viewer, developed by a group of people and recommended by JetBrains. It allows us to see the tree of ElementTypes defined in our Parser and Lexer defined with the Grammar-Kit plugin.

This plugin can be installed through the plugin interface in IntelliJ IDEA.

2.2 SDK setup

The setup of the IntelliJ Platform SDK and common JDK is a very easy step in the setup process for the development of IntelliJ plugins but probably one of the most important ones.

We will first setup the common JDK, followed by the IntelliJ Platform SDK and IntelliJ Community Edition source files.

2.2.1 Common JDK

The setup of the common JDK is accomplished through the Project Structure dialog that can be reached through the File menu in an open project or the Configure menu on the IntelliJ IDEA start page, illustrated in figures 2.1 and 2.2.

In the Project Structure dialog, select the SDK item on the left side, followed by clicking the + sign and selecting JDK, as illustrated in the figure 2.3.

We then select the JDK source folder we wish to setup, in our case we selected the Java 8 JDK.

2.2.2 IntelliJ Platform SDK

The setup of the IntelliJ Platform SDK is done in the same window as the common JDK, and the same + sign but selecting IntelliJ Platform SDK instead, as illustrated in the figure 2.4.

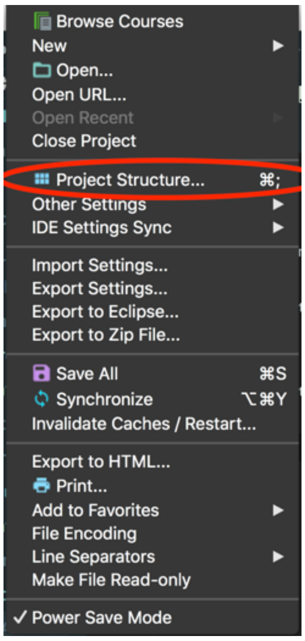


Figure 2.1: Project Structure in File menu

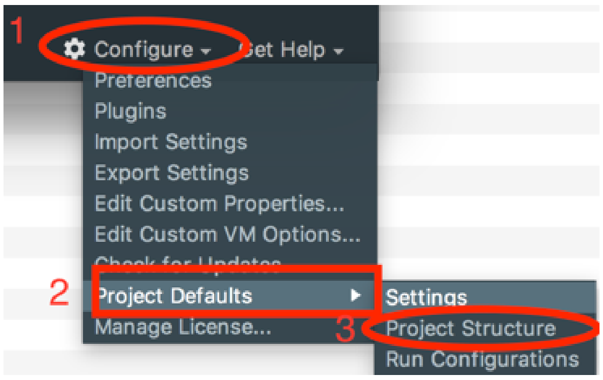


Figure 2.2: Project Structure in Configure menu on start page

We then select the directory containing the installation of IntelliJ IDEA, IntelliJ should suggest it by default, after that select the previously configured common JDK, illustrated in figure 2.5.

2.2.3 IntelliJ Community Edition source code

After configuring the common JDK and IntelliJ Platform SDK, we can setup the IntelliJ source code. This is done by changing to the `Sourcepath` tab while the selection is on the IntelliJ Platform SDK, and then clicking the + symbol and selecting the root directory where you checked out the IntelliJ source code from GitHub, illustrated in figure 2.6.

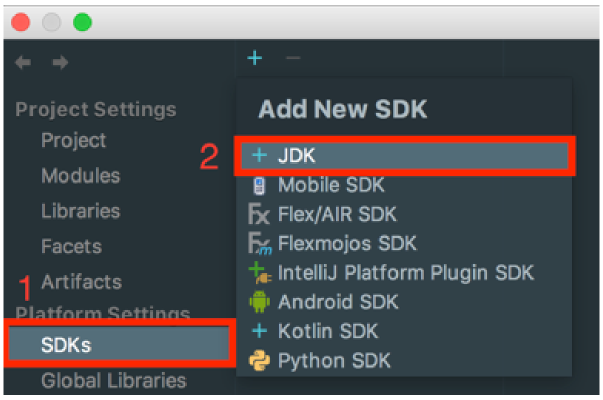


Figure 2.3: Setup common JDK in Project Structure

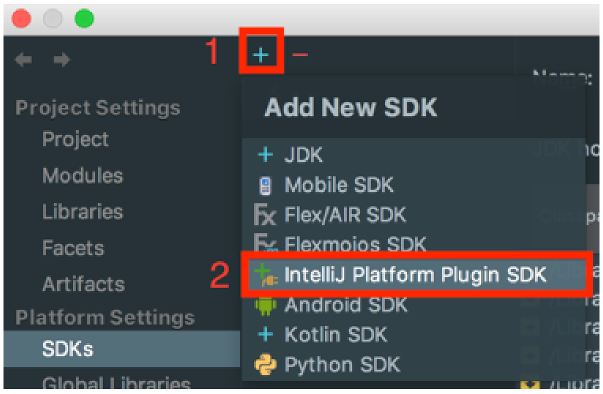
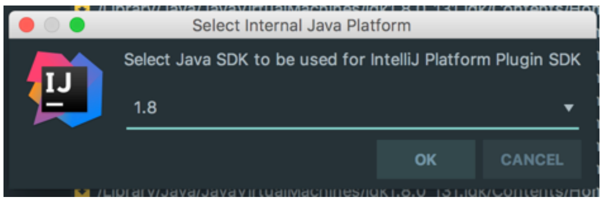


Figure 2.4: Setup IntelliJ Platform SDK



Select Java SDK to be used for IntelliJ Platform Plugin SDK

1.8

Figure 2.5: Select common JDK for the IntelliJ Platform SDK

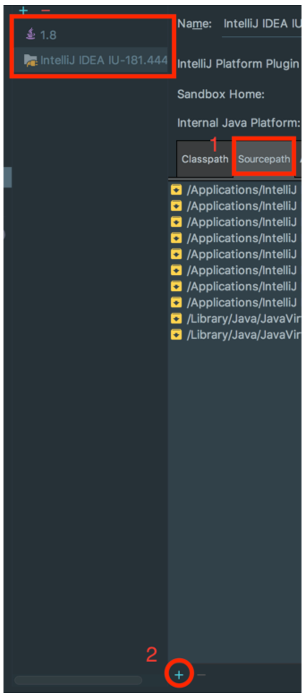


Figure 2.6: Configuring sourcepath of IntelliJ Community

2.3 Project Creation

Creating the plugin project is as simple as creating any project in IntelliJ IDEA with the only difference being we select IntelliJ Platform Plugin as the project type.

The steps are :

- Select `New project`, illustrated in figure 2.7
- Select `IntelliJ Platform Plugin`, illustrated in figure 2.8
- Define a name and location for the project, illustrated in figure 2.9

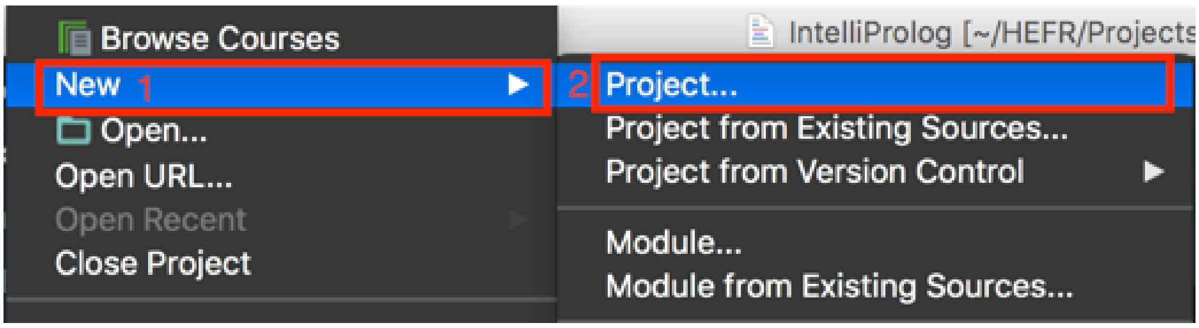
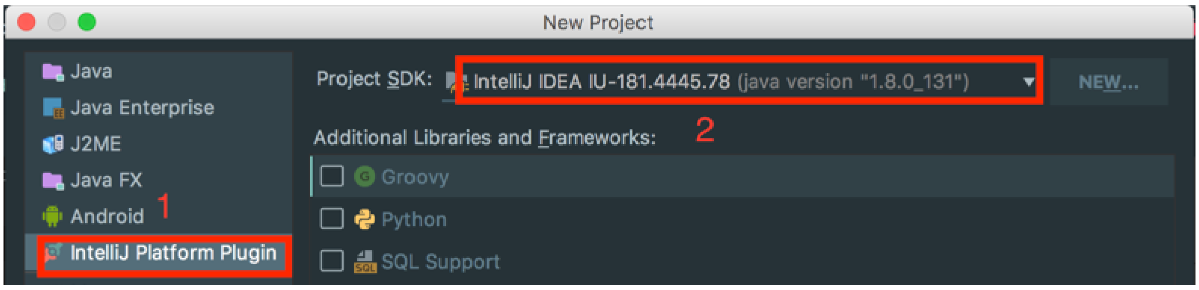

Browse Courses
New 1
Open...
Open URL...
Open Recent
Close Project
IntelliProlog [~/HEFR/Projects
2 Project...
Project from Existing Sources...
Project from Version Control
Module...
Module from Existing Sources...

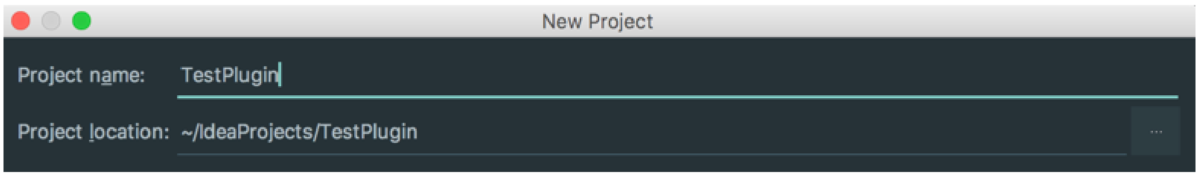
Figure 2.7: New project menu entry



Project SDK: IntelliJ IDEA IU-181.4445.78 (java version "1.8.0_131")

Additional Libraries and Frameworks:

Figure 2.8: Select IntelliJ Platform Plugin project type

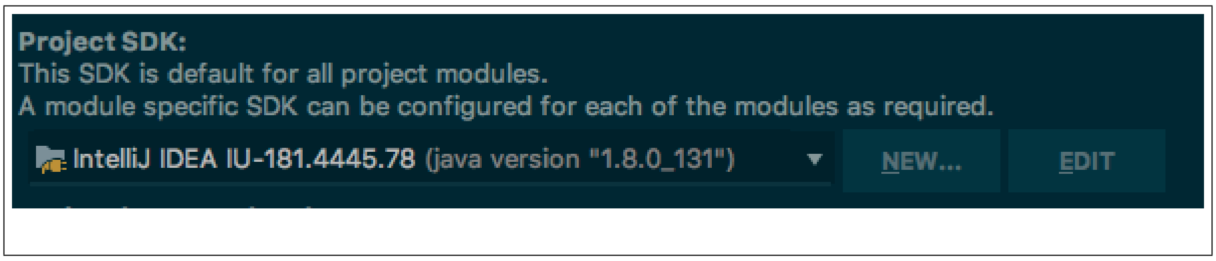


Project name: TestPlugin

Project location: ~/IdeaProjects/TestPlugin

Figure 2.9: Define project name and project location

After the project is created we need to define the project SDK, illustrated in figure 2.10, this is accomplished in the Project Structure window.



Project SDK:
This SDK is default for all project modules.
A module specific SDK can be configured for each of the modules as required.
IntelliJ IDEA IU-181.4445.78 (java version "1.8.0_131")

Figure 2.10: Define project SDK

3 File Launcher

The file launcher functionality is the main part of this project, the reason being that the students for which this plugin was created for, used to have to write their source code in a text editor or an IDE and then having to switch over to a terminal or the graphical interface of GNU-Prolog on Windows to run their code.

This situation was not very enjoyable though for the students and was also not very productive. So to remediate to this situation the project was centered around the ability to be able to launch the desired file directly from IntelliJ IDEA, without having to open a separate window or change to a differenct window.

In this chapter we will go over the necessary parts necessary to enable this functionality in a plugin for IntelliJ IDEs.

The sections are:

- Introduction: Setup the basics for recognising files of the desired language, in our case Prolog
- SDK: Creating an SDK to represent the execution of our source code
- Module: Creating the implementation of modules to allow separatiion of settings between different environments
- Actions: The implementation of launching the file using actions to launch a file

3.1 Introduction

The basics for recognising files of the programming language that is targeted by the plugin we are developing is creating classes that represent the language, these classes are:

- A class extending the Language class provided by Jetbrains
- A class extending the LanguageFileType class provided by Jetbrains

- A class extending the FileTypeFactory class provided by Jetbrains

If we want to provide icons for our plugin we can also create another class grouping all the icons we desire to be present in our plugin.

3.1.1 Language subclass

The `Language` subclass is used to define a language for our plugin and is used in a lot of situations during the development of plugins for the Jetbrains IDEs.

For our project which is targeting the GNU-Prolog language, we will generalise to Prolog so we will call the class `PrologLanguage`. The `Language` subclass is a singleton and therefore has a private constructor and a static instance field as seen in listing 1.

```java
package ch.eif.intelliprolog;

import com.intellij.lang.Language;

public class PrologLanguage extends Language {
    public static final PrologLanguage INSTANCE = new PrologLanguage();

    private PrologLanguage() {
        super("Prolog");
    }
}
```

Listing 1: PrologLanguage class

3.1.2 LanguageFileType subclass

The `LanguageFileType` subclass is used to identify and describe a file that belongs to our language, for these reasons it has a couple of methods that define basic information for our file type.

The information that we need to define is:

- The name to call our file type
- A description of our file type
- The default extension for our file type
- The icon to use for our file type, we will get to icons in a bit

There are a couple other methods that you can override, but these are not explicitly necessary for our plugin, these can be found by looking at the source of `LanguageFileType` [5].

The example from our plugin is depicted in listing 2.

```java
package ch.eif.intelliprolog;

import com.intellij.openapi.fileTypes.LanguageFileType;
import org.jetbrains.annotations.NotNull;

import javax.swing.*;

public class PrologFileType extends LanguageFileType {
    public static final PrologFileType INSTANCE = new PrologFileType();

    private PrologFileType() {
        super(PrologLanguage.INSTANCE);
    }

    @NotNull
    @Override
    public String getName() {
        return "Prolog file";
    }

    @NotNull
    @Override
    public String getDescription() {
        return "Prolog language file";
    }

    @NotNull
    @Override
    public String getDefaultExtension() {
        return "pl";
    }

    @NotNull
    @Override
    public Icon getIcon() {
        return PrologIcons.FILE;
    }
}
```

Listing 2: PrologFileType class

3.1.3 FileTypeFactory subclass

The `FileTypeFactory` subclass is used to register our new FileType with the IntelliJ IDE, which provides a platform extension point named `com.intellij.fileTypeFactory`.

The only thing we need to do in the `FileTypeFactory` is override the method named `void createFileTypes([@NotNull] FileTypeConsumer consumer)` which calls the `consume` method on the `FileTypeConsumer`and pass it the instance of our FileType.

The example from our plugin is depicted in listing 3.

```
package ch.eif.intelliprolog;

import com.intellij.openapi.fileTypes.FileTypeConsumer;
import com.intellij.openapi.fileTypes.FileTypeFactory;
import org.jetbrains.annotations.NotNull;

public class PrologFileTypeFactory extends FileTypeFactory {

    @Override
    public void createFileTypes(@NotNull FileTypeConsumer consumer) {
        consumer.consume(PrologFileType.INSTANCE);
    }
}
```

Listing 3: PrologFileTypeFactory class

The registration of our FileTypeFactory is done in the `plugin.xml` file found in the `resources/META-INF` folder of our project.

The line depicted in listing 4 needs to be inserted in the `extensions` part of the file:

```
implementationClass="ch.eif.intelliprolog.editor.PrologSyntaxHighlighterFactory"/>
```

Listing 4: FileTypeFactory registration

3.1.4 Icons class

We can create a class that is entirely responsible for loading and keeping references to the icons we wish to use in our plugin. This will be a standard Java class that only has static fields, each loading and keeping a reference to a particular icon.

An example for our plugin is depicted in listing 5.

This class can then be used in all our other classes that need references to icons.

```java
package ch.eif.intelliprolog;

import com.intellij.openapi.util.IconLoader;

import javax.swing.*;

public class PrologIcons {
    public static final Icon FILE =
     ↪  IconLoader.getIcon("/ch/eif/intelliprolog/icons/logo.png");
}
```

Listing 5: PrologIcons class

3.2 SDK

Our plugin uses an IntelliJ IDEA specific feature which is the idea of SDKs, in the basic version of IntelliJ IDEA the most common SDK that will be defined is the Java SDK also called the JDK. The reason why this is an IntelliJ IDEA specific feature is explained in the section explaining the differences between the different types of JetBrains IDEs.

An SDK lets us define an API library that is used to build a project and in the case of multi-module projects lets us define an SDK for each module. This functionality is very useful for projects where the frontend and backend is in the same project but they are written in different languages, an example being a backend using Java with Spring [6] and the frontend Typescript [7] / Javascript [8] with AngularJS [9] , for more information about the idea of SDKs in IntelliJ visit their help page about SDKs [10].

3.2.1 Custom SDK

In our project we created our own custom SDK, the reason being it was one of the easiest ways to provide persistance between launches of the IDE and the configuration is in a common place for regular IntelliJ users.

To implement a custom SDK, we need to extend the `SdkType` abstract class provided by JetBrains. This class has a couple of abstract methods that have to be implemented, these being:

- `string suggestHomePath()`
- `boolean isValidSdkHome(String path)`
- `String suggestSdkName(String currentSdkName, String sdkHome)`

- `void saveAdditionalData(SdkAdditionalData additionalData, Element additional)`
- `AdditionalDataConfigurable createAdditionalDataConfigurable(SdkModel sdkModel, SdkModificator sdkModificator)`
- `String getPresentableName()`

There are also some other methods that can be overriden, for a full list check the source code for the `SdkType` class [11]. In our plugin we override a couple of these.

- `FileChooserDescriptor getHomeChooserDescriptor()`
- `String getVersionString(String sdkHome)`
- `Icon getIcon()`
- `Icon getIconForAddAction()`
- `boolean isRootTypeApplicable(OrderRootType type)`

And of course we also have some static fields as well as a constructor.

In depth explanations and examples from our plugin of these methods follow.

String suggestHomePath()

The `String suggestHomePath()` method is used for the file chooser when choosing the executable. If your plugin is aimed at being crossplatform, Windows, macOS or Linux, you will have to take this into account in this method and return different paths for each, the path to where the main executable is normally found.

The example from our plugin, written in Kotlin, is depicted in listing 6.

The `getLatestVersions(versions)`method call at line 167, is used to get the latest version of the executable if a directory was selected which contains multiple versions of the same executable, this method is not really necessary.

boolean isValidSdkHome(String path)

The `boolean isValidSdkHome(String path)` method is used to check that the path really points to an executable that we are looking for. This method uses some other methods, all of these are depicted in listing 7.

```
override fun suggestHomePath(): String? {
    val versions: List<File>
    if (SystemInfo.isLinux) {
        val versionsRoot = File("/usr/bin")
        if (!versionsRoot.isDirectory) {
            return null
        }
        versions = (versionsRoot.listFiles(object : FilenameFilter {
            override fun accept(dir: File, name: String): Boolean {
                return !File(dir, name).isDirectory && isProlog(name.toLowerCase())
            }
        })?.toList() ?: listOf())
    } else if (SystemInfo.isWindows) {
        var cDrive = "C:"
        val versionsRoot = File(cDrive, "GNU-Prolog")
        if (!versionsRoot.isDirectory)
            return cDrive
        versions = versionsRoot.listFiles()?.toList() ?: listOf()
    } else if (SystemInfo.isMac) {
        val macVersions = ArrayList<File>()
        val brewVersionsRoot = File("/usr/local/Cellar/gnu-prolog")
        if (brewVersionsRoot.isDirectory) {
            macVersions.addAll(brewVersionsRoot.listFiles()?.toList() ?: listOf())
        }
        versions = macVersions
    } else {
        return null
    }
    val latestVersion = getLatestVersion(versions)

    return latestVersion?.prologPath?.absolutePath
}
```

Listing 6: suggestHomePath method

String suggestSdkName(String currentSdkName, String sdkHome)

The `String suggestSdkName(String currentSdkName, String sdkHome)` method is used to get a string to display the information of the selected SDK, in our case that means appending the version of the executable to `GNU-Prolog` or returning `Unknown` if we do not have the version.

This method uses some other methods, all of these are depicted in listing 8.

```kotlin
override fun isValidSdkHome(path: String?): Boolean {
    return checkForProlog(path!!)
}

fun checkForProlog(path: String): Boolean {
    val file = File(path)
    if (file.isDirectory) {
        val children = file.listFiles(object : FileFilter {
            override fun accept(f: File): Boolean {
                if (f.isDirectory)
                    return false
                return f.name == "gprolog"
            }
        })
        return children.isNotEmpty()
    } else {
        return isProlog(file.name)
    }
}

fun isProlog(name: String): Boolean = name == "gprolog" || name == "gprolog.exe" ||
↪  name.matches("gprolog-[.0-9*]+".toRegex())
```

Listing 7: isValidSdkHome method

void saveAdditionalData(SdkAdditionalData additionalData, Element additional)

This method can be used to save additional data, data that is not the path to the SDK (main executable), examples are other executables for package managers or external formatters like `gofmt`, these can then be used during build time.

In our plugin the SDK does not use any additional data so we can simply override it and leave it empty.

AdditionalDataConfigurable createAdditionalDataConfigurable(SdkModel sdkModel, SdkModificator sdkModificator)

This method can be used to return an instance of a class implementing the `AdditionalDataConfigurable` interface, which enables us to modify the `SdkAdditionalData` information through a form, that we would need to create ourselves.

```
override fun suggestSdkName(currentSdkName: String?, sdkHome: String?): String {
    val suggestedName: String
    if (currentSdkName != null && currentSdkName.isNotEmpty()) {
        suggestedName = currentSdkName
    } else {
        val versionString = getVersionString(sdkHome)
        if (versionString != null) {
            suggestedName = "GNU-Prolog$versionString"
        } else {
            suggestedName = "Unknown"
        }
    }
    return suggestedName
}

override fun getVersionString(sdkHome: String?): String? {
    if (sdkHome == null) {
        return null
    }
    val versionString: String? = getPrologVersion(File(sdkHome))
    if (versionString != null && versionString.isEmpty()) {
        return null
    }

    return versionString
}

fun getPrologVersion(prologPath: File): String? {
    if (prologPath.isDirectory) {
        return null
    }
    return "1.4.4"
}
```

Listing 8: suggestSdkName method

String getPresentableName()

The `String getPresentableName()` method is similar to `suggestSdkName(String currentSdkName, String sdkHome)` but returns a more generic name that is used in the list of available SDKs that can be defined, the method is depicted in listing 9.

FileChooserDescriptor getHomeChooserDescriptor()

The `FileChooserDescriptor getHomeChooserDescriptor()` method defines the file chooser dialog that is shown when we click on the `New` button. We need to return a `FileChooserDescriptor`, in the file chooser descriptor we can set some defaults for the

```
override fun getPresentableName(): String {
    return "GNU-Prolog"
}
```

Listing 9: getPresentableName method

dialog, display hidden files, and also what constitutes a valid file or directory, most often it uses the `boolean isValidSdkHome(String path)` method.

This method does not need to be overriden as the `SdkType` base class already provides an implementation, but it can be useful to override it if we have some very specific verifications we want to do, as depicted in listing 10.

String getVersionString(String sdkHome)

The `String getVersionString(String sdkHome)` method is used to retrieve the version of the executable. This is not absolutely necessary but becomes useful when your plugin supports different versions of the same executable, it can in some places replace a separate version manager like `nvm` [12], if you want to test your project on different versions.

In our plugin we simply return `1.4.4`, as depicted in listing 11, since that is at the moment of writing the only version of GNU-Prolog used but in other cases we can use a Regex to extract the version number from the parent directory or even the executable depending on the language.

Icon getIcon()

This method simply returns the icon we wish to display next to our SDK, in our case we simply get the icon from our `PrologIcons` class.

PrologSdkType(), constructor

In the constructor of our custom SDK we simply call the parent class constructor with the name of our SDK, which in our case is `GPROLOG`.

Static fields

In our class we define only a single static field, an instance field.

3.2.2 Registering the custom SDK

The `SdkType` needs to be added to the extensions section of the `plugin.xml` file, the element that needs to be added is depicted in listing 12.

```kotlin
override fun getHomeChooserDescriptor(): FileChooserDescriptor {
    val isWindows = SystemInfo.isWindows
    return object : FileChooserDescriptor(true, false, false, false, false, false) {
        @Throws(Exception::class)
        override fun validateSelectedFiles(files: Array<VirtualFile>?) {
            if (files!!.size != 0) {
                if (!isValidSdkHome(files[0].path)) {
                    throw Exception("Not valid gprolog " + files[0].name)
                }
            }
        }

        override fun isFileVisible(file: VirtualFile, showHiddenFiles: Boolean): Boolean {
            if (!file.isDirectory) {
                if (!file.name.toLowerCase().startsWith("gprolog")) {
                    return false
                }
                if (isWindows) {
                    val path = file.path
                    var looksExecutable = false
                    for (ext in WINDOWS_EXECUTABLE_SUFFIXES) {
                        if (path.endsWith(ext)) {
                            looksExecutable = true
                            break
                        }
                    }
                    return looksExecutable && super.isFileVisible(file, showHiddenFiles)
                }
            }
            return super.isFileVisible(file, showHiddenFiles)
        }
    }.withTitle("Select GProlog executable").withShowHiddenFiles(SystemInfo.isUnix)
}
```

Listing 10: getHomeChooserDescriptor method

```
    override fun getVersionString(sdkHome: String?): String? {
        if (sdkHome == null) {
            return null
        }
        val versionString: String? = getPrologVersion(File(sdkHome))
        if (versionString != null && versionString.isEmpty()) {
            return null
        }

        return versionString
    }

    override fun createAdditionalDataConfigurable(sdkModel: SdkModel, sdkModificator:
     ↪  SdkModificator): AdditionalDataConfigurable? {
        return null
    }

    override fun saveAdditionalData(additionalData: SdkAdditionalData, additional:
     ↪  Element) {
    }

    override fun loadAdditionalData(additional: Element?): SdkAdditionalData? {
        return null
    }

    override fun getPresentableName(): String {
        return "GNU-Prolog"
    }

    override fun getIcon(): Icon {
        return GPROLOG_ICON
    }

    override fun getIconForAddAction(): Icon {
        return icon
    }

    override fun setupSdkPaths(sdk: Sdk) {
    }

    override fun isRootTypeApplicable(type: OrderRootType): Boolean {
        return true
    }
}
```

Listing 11: getVersionString method

```
<annotator language="Prolog"
 ↪  implementationClass="ch.eif.intelliprolog.editor.PrologAnnotator"/>
```

Listing 12: SdkType registration

3.3 Module

A module in IntelliJ IDEA is a discrete unit encompassing different functionalities as well as source files, run configurations, etc. Each module is independent of the the others contained within the same project and can even have completely different SDKs defined.

IntelliJ IDEA has some common module types available, mostly for the Java Programming Language, a short non-exhaustive list of them is:

- Java
- Java Enterprise
- Spring
- IntelliJ Platform Plugin

Since we are creating a plugin for a language that has no direct connection to Java we will need to implement our own module type.

The basic building blocks for a module type are:

- Extending the `ModuleType` class provided by JetBrains
- Extending the `ModuleBuilder` class provided by JetBrains

3.3.1 ModuleType

Our module is very simple for our plugin mainly because we do not need to define a lot for the GNU-Prolog language and for that reason we will only implement the abstract methods from the `ModuleType` class, these methods are as follows:

- `T createModuleBuilder()`
- `String getName()`
- `String getDescription()`
- `Icon getNodeIcon(boolean isOpened)`

We will also override the `Icon getIcon()` method. We also have a static instance field in this class.

T createModuleBuilder()

This method creates our `ModuleBuilder` instance and since we are creating our own module builder, the return type will be `PrologModuleBuilder`. The only thing we do in this method is return a new `PrologModuleBuilder` instance, as depicted in listing 13.

```kotlin
override fun createModuleBuilder(): PrologModuleBuilder {
    return PrologModuleBuilder()
}
```

Listing 13: createModuleBuilder method

String getName() and String getDescription()

The `getName()` method simply returns the name of our module, in our case that is `Prolog Module`, as depicted in listing 14.

```kotlin
override fun getName(): String {
    return "Prolog Module"
}
```

Listing 14: getName method

The `getDescription()` method simply returns the description of our module, in our case that is `Prolog Module`, as depicted in listing 15.

```kotlin
override fun getDescription(): String {
    return "Prolog Module"
}
```

Listing 15: getDescription method

Icon getIcon() and Icon getNodeIcon(boolean isOpened)

The `getIcon()` method returns our desired icon for the module, in our case it comes from our `PrologIcons` class, as depicted in listing 16.

The `getNodeIcon(boolean isOpened)` method returns our desired icon for the module, in our case it comes from our `PrologIcons` class, as depicted in listing 17.

```
override fun getIcon(): Icon {
    return PrologIcons.FILE
}
```

Listing 16: getIcon method

```
override fun getNodeIcon(isOpened: Boolean): Icon {
    return PrologIcons.FILE
}
```

Listing 17: getNodeIcon method

3.3.2 ModuleBuilder

The module builder is used for the project wizard when we create a new project with our module type, since we are developing a plugin for `GNU Prolog` which does not have a very complex system surrounding it we will implement a very simple module builder based on the Java module builder. This prevents us from having to create everything ourselves. The `JavaModuleBuilder` provides wizard steps to specify the root directory and name our project.

The methods that we will implement or override are the following:

- `String getBuilderId()`
- `ModuleWizardStep modifySettingsStep(SettingsStep settingsStep)`
- `String getGroupName()`
- `String getPresentableName()`
- `ModuleWizardStep[] createWizardSteps(WizardContext wizardContext, ModulesProvider modulesProvider)`
- `abstract ModuleType getModuleType()`
- `abstract void setupRootModel(ModifiableRootModel modifiableRootModel)`
- `boolean isSuitableSdktype(SdkTypeId sdktype)`

In depth explanations and examples taken from our plugin follow.

String getBuidlerid()

This method is used to specify a unique name/id for our module builder so as not to clash with any other module builders defined through plugins added to IntelliJ, as depicted in listing 18.

```
override fun getBuilderId() = "intelliprolog.module.builder"
```

Listing 18: getBuilderId method

ModuleWizardStep modifySettingsStep(SettingsStep settingsStep)

This method is used to modify a settings step when the context changes, in particluar after changing from the first step to the second step, at least as far as I understand it.

In our case we are using the the standard Java `modifySettingsStep(SettingsStep settingsStep, ModuleBuilder moduleBuilder)`, as depicted in listing 19.

```
override fun modifySettingsStep(settingsStep: SettingsStep): ModuleWizardStep? =
        StdModuleTypes.JAVA!!.modifySettingsStep(settingsStep, this)
```

Listing 19: modifySettingsStep method

String getGroupName() and String getPresentableName()

This method returns the name of the group we want our module to appear in during the project wizard. We are simply going to call the group `Prolog`, as depicted in listing 20.

```
override fun getGroupName(): String? = "Prolog"
```

Listing 20: getGroupName method

This method returns the name to be displayed for the module type to appear during the project wizard. We are simply going to return `Prolog`, as depicted in listing 21.

ModuleWizardStep[] createWizardSteps(WizardContext wizardContext, ModulesProvider modulesProvider)

This method returns all the `ModuleWizardSteps` to be displayed for this module, it is in this method that we can create and return our own custom steps for our module.

```kotlin
override fun getPresentableName(): String? = "Prolog"
```

Listing 21: getPresentableName method

In our case we simply return the default module steps, as depicted in listing 22, since that is all we need for our plugin.

```kotlin
override fun createWizardSteps(wizardContext: WizardContext, modulesProvider:
↪   ModulesProvider): Array<ModuleWizardStep> =
        moduleType.createWizardSteps(wizardContext, this, modulesProvider)
```

Listing 22: createWizardSteps method

ModuleType getModuleType()

This method simply returns our custom module type, `PrologModuleType`, more specifically our static instance field, as depicted in listing 23.

```kotlin
override fun getModuleType(): PrologModuleType {
    return PrologModuleType.INSTANCE
}
```

Listing 23: getModuleType method

void setupRootModel(ModifiableRootModel modifiableRootModel)

This method is probably one of the most important in the `ModuleBuilder` class in my opinion, since it allows us to define the directory structure of the module in question as well as files if needed, as depicted in listing 24

boolean isSuitableSdktype(SdkTypeId sdktype)

This method is used to check if the SDK is suitable for the module. In our own case we simply check if the SDK passed in as argument is an instance of our custom SdkType, as depicted in listing 25.

3.3.3 Registering the module type

The `ModuleType` needs to be added to the extensions section of the `plugin.xml` file, the element that needs to be added is depicted in listing 26.

```
override fun setupRootModel(rootModel: ModifiableRootModel?) {
    if (myJdk != null) {
        rootModel!!.sdk = myJdk
    } else {
        rootModel!!.inheritSdk()
    }

    val contentEntry = doAddContentEntry(rootModel)
    if (contentEntry != null) {
        val srcPath = contentEntryPath!! + File.separator + "src"
        File(srcPath).mkdirs()
        val sourceRoot =
         ↪  LocalFileSystem.getInstance()!!.refreshAndFindFileByPath(FileUtil.toSystemIndependentNa
        if (sourceRoot != null) {
            contentEntry.addSourceFolder(sourceRoot, false, "")
        }
    }

}
```

Listing 24: setupRootModel method

```
override fun isSuitableSdkType(sdkType: SdkTypeId?): Boolean {
    return sdkType is PrologSdkType
}
```

Listing 25: isSuitableSdktype method

```
<lang.syntaxHighlighterFactory language="Prolog"
```

Listing 26: ModuleType registration

3.4 Action

The action system in IDEA allows plugins to add their own actions in menus and toolbars. An example of an action included in IDEA is the `File | Open File...` .

For our plugin we will define actions to launch the current file in the GNU-Prolog interpreter, this interpreter will run in the IDEA console window.

In this section we will go over the basics of an action as well as the actions we will implement in our plugin.

3.4.1 Basic workings of an action

An action is defined using a class that extends the abstract class `AnAction` provided by JetBrains.

The main method in the `AnAction` class is `void actionPerformed(AnActionEvent e)`, the other commonly used method is `void update(AnActioNEvent e)`.

For the complete list of methods available in the `AnAction` class, checkout the source code `AnAction.java` [13].

The actions are then registered in the `plugin.xml` file under the `actions` section.

void actionPerformed(AnActionEvent e)

This method is called when the action in the menu or toolbar is clicked. It is in this method that your business logic for the action is launched.

void update(AnActionEvent e)

This method is called everytime that the view of the action in the menu or toolbar is updated, in the case of the toolbar this is called twice a second, so it needs to be fast and should not perform any file system actions. It is mainly used to update the presentation of the action, changing the text of the action or disabling the action completely if the action would not be able to be executed.

Registering actions

To register actions in the plugin we have to add appropriate elements to the `plugin.xml` file, in the actions section.

The parent element is `action` with an id uniquely identifying the action, the class of the action and the text to display. Within the `action` element we can add other elements, these allow to define where the action should be located as well as mouse or keyboard shortcuts. More details are available in the Actions section [14] of the IntelliJ Platform SDK DevGuide.

3.4.2 Plugin actions

The actions we are going to implement in our plugin are the following:

- `LoadPrologFileInConsoleAction`
- `LoadPrologFileInConsoleWithTraceAction`
- `RunPrologConsoleAction`

These actions will be available in the `Run` menu as well as the right-click context-menu of the file editor.

Before looking at our actions we will take a look at the methods and classes needed by our actions. We will call these our REPL helpers.

REPL Helpers

Our REPL helpers consist of 5 classes:

- `PrologREPLUtils`
- `PrologConsole`
- `PrologConsoleProcessHandler`
- `PrologConsoleRunner`

These classes are inspired by the classes from the Haskell IDEA plugin [15], more specifically the repl [16] part of it.

PrologREPLUtils The `PrologREPLUtils` class contains some static functions that we use in the other classes and to avoid repeating ourselves we have put them in here. We will not be going over them in detail since they are quite self-explanatory.

The functions are the following:

- `PrologConsoleProcessHandler findRunningPrologConsole(Project project)`, function used to check if we already have a running instance of a PrologConsole and if we do return it so that we do not launch multiple instances.
- `Module getModule(AnActionEvent e)`, function to get the current module from an action event.
- `Module getModule(Project project)`, function to get the current module from a project.
- `String getActionFile(AnActionEvent e)`, function that gets the path to the file we are currently viewing in our editor.

As well as an inner class `PrologConsoleMatcher`, which is used in the `findRunningPrologConsole` function to only get Prolog Consoles.

PrologConsole This class represents a Prolog version of a basic IDEA `LanguageConsole`. We only extend the `LanguageConsoleImpl` class [17] provided by JetBrains.

This class sets up the view of the console window and actions that are available in the console window. If you would want to customise the appearance of the console window, it would be in this class that one would do it.

PrologConsoleProcessHandler The `PrologConsoleProcessHandler` extends from the `ColoredProcessHandler` [18] class provided by JetBrains, this class provides a process handler that supports ANSI coloring. The process handler is what controls our console process, starting and killing it, and integrates with the actions associated with our console view.

PrologConsoleRunner The `PrologConsoleRunner` is the class responsible for setting up our Prolog console, it extends the `AbstractConsoleRunnerWithHistory` abstract class with a type parameter of `PrologConsole`.

The `AbstractConsoleRunnerWithHistory` class provides the basic functionality for running consoles. It also launches an external process with line input and history handling.

We implement the abstract methods provided by the abstract class, these methods are:

- `T createConsoleView()`, returns an instance of our `PrologConsole`.
- `Process createProcess() throws ExecutionException`, creates a CommandLine and returns the process attached to the command line.
- `OSProcessHandler createProcessHandler(final Process process)`, returns an instance of our `PrologConsoleProcessHandler`.
- `ProcessBackedConsoleExecuteActionHandler createExecuteActionHandler()`, creates a new instance of a `ConsoleHistoryController` and returns a new `ProcessBackedConsoleExecuteActionHandler`.

We also create two static methods:

- `void run(Module module, String sourceFilePath, boolean withTrace)`, this method creates an instance of `PrologConsoleRunner`, tries to initialise and run our console runner, depicted in listing 27.
- `GeneralCommandLine createCommandLine(Module module, String workingDir, String sourceFilePath, boolean withTrace) throws CantRunException`, this method creates our commandline. We check that we have a PrologSDK configured, we then create a `GeneralCommandLine` [19] and pass it our `gprolog` commandline parameters [20], which are:
  - `--entry-goal trace`, if we are launching the console with a file in trace mode.
  - `--consult-file <path to file>`, if we are launching the console with a file.

  If we are on a Windows system, we also have to set an environment variable [21] to make sure that `gprolog` is launched in text mode, the environment variable is `LINEDIT gui=no`. All of this is depicted in listing 28

LoadPrologFileInConsoleAction

The `LoadPrologFileFileInConsoleAction` action launches our file that is currently open in the editor within the `gprolog` REPL that is running inside a IDEA console window.

The action implements the two methods listed earlier, `actionPerformed` and `update`.

```java
public static void run(@NotNull Module module, String sourceFilePath, boolean
 ↪  withTrace) {
    String srcRoot =
     ↪  ModuleRootManager.getInstance(module).getContentRoots()[0].getPath();
    String path = srcRoot + File.separator + "src";
    PrologConsoleRunner runner = new PrologConsoleRunner(module, INTERPRETER_TITLE,
     ↪  path, sourceFilePath, withTrace);
    try {
        runner.initAndRun();
        runner.getProcessHandler();
    } catch (ExecutionException e) {
        ExecutionHelper.showErrors(module.getProject(), Arrays.<Exception>asList(e),
         ↪  INTERPRETER_TITLE, null);
    }
}
```

Listing 27: run method

void update(AnActionEvent e) In this method we check if a file is available to be run and if yes we set it to be visible with an appropriate text, depicted in listing 29.

void actionPerformed(AnActionEvent e) In this method we first check if an editor and project are available, then retrieve the path to the file we wish to run in the REPL, the whole method is depicted in listing 30.

The next step is making sure that the current state of the file is saved to the filesystem, using the methods `commitAllDocuments()` and `saveAllDocuments()` from `PsiDocumentManager` and `FileDocumentManager` respectively. This is necessary because IntelliJ IDEA uses a Virtual File System, that encapsulates most of the activities necessary for working with files. The reason they do this is so that IDEA can add some extra features to the files, like tracking modifications and abstracting the underlying implementation of the file system of the operating system. More information is available in the IntelliJ Platform SDK DevGuide documentation [22]

After we are sure that the file has been correctly written to disk, we can run our console using the `run` method of `PrologConsoleRunner`, giving it the reference to our project, the path to the file and if we want to run it with trace turned on, which in this case we don't so we pass it `false`.

LoadPrologFileInConsoleWithTraceAction

This action class is identical to `LoadPrologFileInConsoleAction` with one exception, we pass in `true` when we run our console runner.

```java
private static GeneralCommandLine createCommandLine(Module module, String workingDir,
 ↪  @Nullable String sourceFilePath, boolean withTrace) throws CantRunException {
    Sdk sdk = ProjectRootManager.getInstance(module.getProject()).getProjectSdk();
    VirtualFile homePath;
    if (sdk == null || !(sdk.getSdkType() instanceof PrologSdkType) ||
     ↪  sdk.getHomePath() == null) {
        throw new CantRunException("Invalid SDK Home path set. Please set your SDK
         ↪  path correctly.");
    } else {
        homePath = sdk.getHomeDirectory();
    }
    GeneralCommandLine line = new GeneralCommandLine();
    if (SystemInfo.isWindows) {
        line.withEnvironment("LINEDIT", "gui=no");
    }
    line.setExePath(new File(homePath.getPath()).getAbsolutePath());
    line.withWorkDirectory(workingDir);

    if (sourceFilePath != null) {
        final ParametersList list = line.getParametersList();
        if (withTrace) {
            list.addParametersString("--entry-goal " + "trace");
        }
        list.addParametersString("--consult-file " + sourceFilePath);
    }

    return line;
}
```

Listing 28: createCommandLine method

RunPrologConsoleAction

This action class is very similar to the previous actions but in this action we only check if a module is available in the `update` method and in the `actionPerformed` method we simply run our console runner, with a `null` as the path and `false` for the trace argument.

3.4.3 Registering our actions

The actions need to be added to the actions section of the `plugin.xml` file, the elements that need to be added are depicted in listing 31.

```java
    public void update(AnActionEvent e) {
        Presentation presentation = e.getPresentation();
        String filePath = PrologREPLUtils.getActionFile(e);
        if (filePath == null) {
            presentation.setVisible(false);
        } else {
            File file = new File(filePath);
            presentation.setVisible(true);
            presentation.setText(String.format("Load \"%s\" in GNU Prolog
             ↪   Interpreter", file.getName()));
        }
    }
}
```

Listing 29: update method

```java
public void actionPerformed(AnActionEvent e) {
    Editor editor = e.getData(CommonDataKeys.EDITOR);
    if (editor == null) {
        return;
    }

    Project project = editor.getProject();
    if (project == null) {
        return;
    }

    String filePath = PrologREPLUtils.getActionFile(e);

    PsiDocumentManager.getInstance(project).commitAllDocuments();
    FileDocumentManager.getInstance().saveAllDocuments();

    PrologConsoleRunner.run(PrologREPLUtils.getModule(project), filePath, false);
}
```

Listing 30: actionPerformed method

```
<actions>
    <!-- Add your actions here -->
    <action id="ch.eif.intelliprolog.repl.actions.LoadPrologFileInConsoleAction"
            class="ch.eif.intelliprolog.repl.actions.LoadPrologFileInConsoleAction"
            text="Load/Reload Current File in REPL...">
        <keyboard-shortcut keymap="$default" first-keystroke="ctrl shift L"/>
        <add-to-group group-id="EditorPopupMenu" anchor="last"/>
        <add-to-group group-id="RunMenu" anchor="first"/>
    </action>
    <action
     ↪  id="ch.eif.intelliprolog.repl.actions.LoadPrologFileInConsoleWithTraceAction"

             ↪  class="ch.eif.intelliprolog.repl.actions.LoadPrologFileInConsoleWithTraceAction"
            text="Load/Reload Current File in REPL with trace...">
        <keyboard-shortcut keymap="$default" first-keystroke="ctrl shift T"/>
        <add-to-group group-id="EditorPopupMenu" anchor="last"/>
        <add-to-group group-id="RunMenu" anchor="first"/>
    </action>
    <action id="ch.eif.intelliprolog.repl.actions.RunPrologConsoleAction"
            class="ch.eif.intelliprolog.repl.actions.RunPrologConsoleAction"
            text="Run Prolog REPL...">
        <keyboard-shortcut keymap="$default" first-keystroke="ctrl shift R"/>
        <add-to-group group-id="EditorPopupMenu" anchor="last"/>
        <add-to-group group-id="RunMenu" anchor="first"/>
    </action>
</actions>
```

Listing 31: Plugin actions registration

4 Parser and Lexer

The parser and lexer are key to implementing the next features since they will allow us and IntelliJ IDEA to understand and breakdown our files.

4.1 Introduction

Parsers and lexers are a fundamental part of any programming language compiler since they decompose a source file into its component parts and then produces an Abstract Syntax Tree. The Abstract Syntax Tree defines the structure of the program and can then be used to understand what it is supposed to do, it is for this reason that parsers and lexers are fundamental when creating a programming language.

In JetBrains IDEs, parsers and lexers are used so that the IDE can understand the source file it is editing and provide features that programmers have become very used to and sometimes even rely on.

A non-exhaustive list of some of these features is:

- Syntax Highlighting
- Code completion, suggesting variables or methods, etc.
- Finding usages
- Refactoring

These features are only possible since the IDE understands our source code, or more precisely the structure of it, and therefore knows what each character and word represents in the specific language the parser and lexer were designed for.

In this section we are going to look at how to create a parser and lexer for the IntelliJ Platform.

4.1.1 Program Structure Interface (PSI)

The `Program Structure Interface` is the layer provided by the IntelliJ Platform and is used for parsing files and representing the structure of these files.

The PSI is composed of multiple PSI elements in a tree hierarchy, a PSI tree, just like a Abstract Syntax Tree, it is this PSI tree and its elements that enable some of the most useful features listed above.

More information about the `Program Structure Interface` [23] is available in the IntelliJ Platform SDK DevGuide [24]

4.2 Parser

A parser for the IntelliJ Platform is composed of three elements, these elements being:

- A token type, represented by a class that extends `IElementType` [25], used for the lexer which we will see soon but it is easier to define it now.
- An element type, also represented by a class extending `IElementType`, used for the parser.
- A grammar in the Backus-Naur Form [26].

4.2.1 Token and element type

To create our token and element types we simply need to create two classes extending `IElementType` and JetBrains recommends to put both of these in a `psi` package within your project.

The token type class, depicted in listing 32, has a constructor with a string argument that calls the superclass constructor with the string parameter and the instance of our custom language that we created at the beginning of this tutorial. We also override the `toString` method and return the string returned by the super class appended to `PrologTokenType`, this mainly helps when debugging the parser and lexer.

The element type class, depicted in listing 33 is very similar to the token type class, it has the same constructor and that normally would be all but in our plugin we added two methods to help identify the element when doing the syntax highlighting. These methods are:

```
public class PrologTokenType extends IElementType {
    public PrologTokenType(@NotNull String debugName) {
        super(debugName, PrologLanguage.INSTANCE);
    }

    @Override
    public String toString() {
        return "PrologTokenType." + super.toString();
    }
}
```

Listing 32: PrologTokenType class

- `boolean isParenthesis(IElementType elementType)`, checks if the element passed as argument is a parenthesis, either a left parenthesis or right parenthesis.
- `boolean isBracket(IElementType elementType)`, checks if the element passed as argument is a bracket, either a left bracket or right bracket.

These two methods are helper functions, and therefore not necessary in your own plugin.

```
public class PrologElementType extends IElementType {
    public PrologElementType(@NotNull String debugName) {
        super(debugName, PrologLanguage.INSTANCE);
    }

    public static boolean isParenthesis(IElementType elementType) {
        return elementType.equals(PrologTypes.LPAREN) ||
         ↪   elementType.equals(PrologTypes.RPAREN);
    }

    public static boolean isBrackets(IElementType elementType) {
        return elementType.equals(PrologTypes.LBRACKET) ||
         ↪   elementType.equals(PrologTypes.RBRACKET);
    }
}
```

Listing 33: PrologElementType class

4.2.2 Grammar

We are now getting to the meat of the parser, defining our context-free grammar for our language.

A context-free grammar [27], for people who haven't attended a computation theory course at university, is a type of formal grammar, a way of describing a language that has a set of strict rules.

In our case we write the context-free grammar using the Backus-Naur Form, which is a formal way of describing a context-free grammar.

Grammar-kit parser options

To generate our parser definition for the IntelliJ Platform we are going to use the Grammar-Kit plugin [28] we installed during the setup part of our project.

So that Grammar-Kit can produce the parser definition correctly, we need to define a couple of parts not directly related to the grammar definition. These elements are:

- The name of the parser class, in our plugin this is `PrologParser`.
- The class the parser class extends, for the IntelliJ Platform this is `ASTWrapperPsiElement` [29].
- The PSI class prefix, in our plugin this is `Prolog`.
- The PSI implementation class suffix, which is `Impl`.
- The package where the generated PSI elements should be stored, for our plugin this is `ch.eif.intelliprolog.psi`.
- The package where the generated implementation of the PSI elements should be stored, for our plugin this is `ch.eif.intelliprolog.psi.impl`.
- The element type holder class, an interface containing references to all the types in our parser as well as a factory for creating these elements when they are encountered during the parsing process, in our own plugin this is `ch.eif.intelliprolog.psi.PrologTypes`.
- The element type class, the class we created earlier.
- The token type class, the class we created earlier.

The definition of these values are written between braces at the top of our grammar definition file, the definition from our plugin is depicted in listing 34.

Grammar-kit grammar definition

The grammar definition for our plugin is taken from the logtalk3 intellij plugin [30] and adapted for our plugin.

```
{
  parserClass="ch.eif.intelliprolog.PrologParser"

  extends="com.intellij.extapi.psi.ASTWrapperPsiElement"

  psiClassPrefix="Prolog"
  psiImplClassSuffix="Impl"
  psiPackage="ch.eif.intelliprolog.psi"
  psiImplPackage="ch.eif.intelliprolog.psi.impl"

  elementTypeHolderClass="ch.eif.intelliprolog.psi.PrologTypes"
  elementTypeClass="ch.eif.intelliprolog.psi.PrologElementType"
  tokenTypeClass="ch.eif.intelliprolog.psi.PrologTokenType"
}
```

Listing 34: Grammar-Kit parser options

The reason we used the grammar definition from the logtalk plugin is because logtalk as is discussed in the prolog implementations comparison, logtalks syntax is a superset of GNU-Prolog and creating a correct and efficient grammar definition is a long and difficult task that can take a lot of time.

We did try to write our own grammar definition based on a Prolog BNF grammar [31] found on GitHub. The problem with this BNF grammar is that it uses left recursion, which is not fully supported in Grammar-Kit, so we fell back on the logtalk grammar definition.

The logtalk grammar definition most likely could be improved, but decided we would not try to during this project and leave it for future improvements.

In this grammar definition we do not use some of the more advanced features of Grammar-Kit, if you want more information about these features visit the GitHub repository [28]-GH, we will not cover these more advanced features since they are outside the scope of this project.

An extract of the final grammar definition for our Prolog plugin, can be found in listing 35.

Generation and testing of grammar

After we have created our grammar definition we need to generate the parser with PSI classes, thanks to the Grammar-Kit plugin this is very easy. To generate the PSI classes we just have to select `Generate Parser Code` from the context menu of our BNF file. After the generation do not forget to mark the `gen` folder as `Generated Sources Root` using the using the context menu of the folder in question.

```
{
  parserClass="ch.eif.intelliprolog.PrologParser"

  extends="com.intellij.extapi.psi.ASTWrapperPsiElement"

  psiClassPrefix="Prolog"
  psiImplClassSuffix="Impl"
  psiPackage="ch.eif.intelliprolog.psi"
  psiImplPackage="ch.eif.intelliprolog.psi.impl"

  elementTypeHolderClass="ch.eif.intelliprolog.psi.PrologTypes"
  elementTypeClass="ch.eif.intelliprolog.psi.PrologElementType"
  tokenTypeClass="ch.eif.intelliprolog.psi.PrologTokenType"
}

prologFile ::= item_*

private item_ ::= (sentence|COMMENT|CRLF)

sentence ::= (operation|compound|atom) DOT

term ::= (operation|basic_term) //basic_term instead of term to avoid left recursion

basic_term ::=
 ↪  (parenthesized_block|braced_block|list|number|variable|STRING|compound|atom)
```

Listing 35: Extract from the BNF grammar definition

If we want to test our grammar definition before generating it, this is also possible using the Grammar-Kit plugin, just click on `Live Preview` in the context menu of the BNF file.

4.3 Lexer

The lexer defines how the contents of a source file for our language is tokenised, broken into individual tokens, used by the parser.

Their have been various ways to define lexers in the past, using programs like `Flex` [32] or `Lex` [33], which was written by the former CEO of Google Eric Schmidt. IntelliJ recommends to use `JFlex` [34], which we will.

The steps for creating a lexer for the IntelliJ Platform are:

- Define a lexer using the `JFlex` lexer definition syntax and generate the lexer class from it.

- Create a `FlexAdapter`, which will help interpret the lexer class for our plugin.
- Define a base PSI element representing the file being lexed and parsed.
- Define the parser definition, we did not do this in the previous part since we need the lexer to fully implement it.
- Register the parser definition with the plugin.

4.3.1 Lexer definition

A `JFlex` lexer definition has 3 main elements in the file, separated into 3 section delimited between them using `%%` has separator, these sections are:

- User code, depicted in listing 36, everything in this section is copied verbatim to the resulting lexer class, we normally include `package` and `import` statements.
- Options and macros, depicted in listing 37, in this section we define options for the generated lexer class, like the name of the class, which class it extends. We also define macros used later in the rules section, these macros behave like C preprocessor macros. The final part of this section is defining the possible lexical states available.
- Rules and actions, depicted in listing 38, this section is the main part of the lexer. We define how the scanner matches the input using regular expressions and lexical states, more detailed information is available in the `JFlex` Manual [35].

```
package ch.eif.intelliprolog;

import com.intellij.lexer.FlexLexer;
import com.intellij.psi.tree.IElementType;
import ch.eif.intelliprolog.psi.PrologTypes;
import com.intellij.psi.TokenType;

%%
```

Listing 36: User code in Flex file

As with the parser definition we are also using the definition from the logtalk3 intellij plugin [30], this for the same reasons detailed for the parser.

```
%%
 // OPTIONS
%class PrologLexer
%implements FlexLexer
%unicode
%function advance
%type IElementType
%eof{   return;
%eof}

ATOM_CHAR = [:jletterdigit:]
UNQUOTED_ATOM = [:lowercase:] {ATOM_CHAR}*

ANONYMOUS_VARIABLE = "_" {ATOM_CHAR}*
NAMED_VARIABLE = [:uppercase:] {ATOM_CHAR}*

STYLE_COMMENT = ("%!"|"%%")[^\r\n]*
END_OF_LINE_COMMENT = ("%")[^\r\n]*
BLOCK_COMMENT = "/*" [^*] ~"*/" | "/*" "*"+ "/"
DOC_COMMENT = "/**" {DOC_COMMENT_CONTENT} "*"+ "/"
DOC_COMMENT_CONTENT = ( [^*] | \*+ [^/*] )*

 // LEXICAL STATES
%state SENTENCE, PARENTHESIZED_SYMBOLS, SINGLE_QUOTE_STRING, DOUBLE_QUOTE_STRING,
 ↪  CHAR_CODE

%%
```

Listing 37: Options, macros and lexical states in Flex file

Generate lexer class

Generating the lexer class is similar to generating the parser class, we just have to click on `Run JFlex Generator` in the context menu for the flexe definition file.

If it is the first time that the generator is run, IDEA will offer to download the JFlex generator as well as a skeleton file detailing how it should be run. This should be saved in the project root directory.

4.3.2 FlexAdapter

The `FlexAdapter` class will be called `PrologLexerAdapter`, visible in listing 39, for our plugin and extends the `FlexAdapter` class [36] provided by JetBrains.

The only thing we need to implement in this class is a constructor that calls the super class constructor with our lexer class.

```
<YYINITIAL, SENTENCE, PARENTHESIZED_SYMBOLS> {

   {COMMENT}                                         { return PrologTypes.COMMENT; }

   {NON_PRINTABLE}                                   { return TokenType.WHITE_SPACE; }

}
```

Listing 38: Extract from rules and actions part of Flex file

```java
package ch.eif.intelliprolog;

import com.intellij.lexer.FlexAdapter;

public class PrologLexerAdapter extends FlexAdapter {
    public PrologLexerAdapter() {
        super(new ch.eif.intelliprolog.PrologLexer(null));
    }
}
```

Listing 39: PrologLexerAdapter class

4.3.3 Root PSI file

The root PSI file class represents a file that belongs to our language and is used as the root element in the PSI tree generated by our parser and lexer after going through our source file.

The class that we will call `PrologFile`, depicted in listing 40, extends the `PsiFileBase` class [37], and has a constructor calling the super class constructor with the `FileViewProvider` argument and the instance of our `PrologLanguage` class.

We also override the `FileType getFileType()` method returning the instance of our `PrologFileType` and override the `toString()` method returning `Prolog File`.

4.3.4 Parser Definition

The parser definition class, called `PrologParserDefinition` visible in listing 41 implements the `ParserDefinition` [38], and defines the implementation of a custom language parser.

This class is where we define which parser, lexer, PSI file base and elements to be used in our parser, to accomplish this we need to implement the methods defined in `ParserDefinition`.

```java
package ch.eif.intelliprolog.psi;

import ch.eif.intelliprolog.PrologFileType;
import ch.eif.intelliprolog.PrologLanguage;
import com.intellij.extapi.psi.PsiFileBase;
import com.intellij.openapi.fileTypes.FileType;
import com.intellij.psi.FileViewProvider;
import org.jetbrains.annotations.NotNull;

public class PrologFile extends PsiFileBase {
    public PrologFile(@NotNull FileViewProvider viewProvider) {
        super(viewProvider, PrologLanguage.INSTANCE);
    }

    @NotNull
    @Override
    public FileType getFileType() {
        return PrologFileType.INSTANCE;
    }

    @Override
    public String toString() {
        return "Prolog File";
    }

}
```

Listing 40: PrologFile class

These methods are:

- `Lexer createLexer(Project project)`, returns the lexer we want to use for lexing our files, we return the `PrologLexerAdapter`.
- `PsiParser createParser(Project project)`, returns the parser we want to use for parsing our files, we return the generated `PrologParser`.
- `IFileElementType getFileNodeType()`, returns the element type for a file in the specified language, we return an `IFileElementType` with the instance of `PrologLanguage`.
- `TokenSet getCommentTokens()`, returns a set of token types representing comments in our language, we return a `TokenSet` created from the comment type of our parser.
- `TokenSet getStringLiteralElements()`, returns a set of token types representing string literals in our language, we return a `TokenSet` created from the string type of our parser.

- `PsiElement createElement(ASTNode node)`, returns the appropriate `PsiElement` for a given `ASTNode`, we use the factory generated from our parser.
- `PsiFile createFile(FileViewProvider viewProvider)`, returns the root PSI element of the virtual file, we return a `PrologFile` instance.
- `SpaceRequirements spaceExistanceTypeBetweenTokens(ASTNode left, ASTNode right)`, checks if the two tokens need a space between them, we return `SpaceRequirements.MAY`

4.3.5 Register the parser definition

The `ParserDefinition` needs to be added to the extensions section of the `plugin.xml` file, the element that needs to be added is visible in listing 42.

```
public class PrologParserDefinition implements ParserDefinition {
    public static final TokenSet WHITE_SPACES =
     ↪  TokenSet.create(TokenType.WHITE_SPACE);
    private static final TokenSet COMMENTS = TokenSet.create(PrologTypes.COMMENT);
    private static final TokenSet STRINGS = TokenSet.create(PrologTypes.STRING);

    private static final IFileElementType FILE = new
     ↪  IFileElementType(PrologLanguage.INSTANCE);

    @NotNull
    @Override
    public Lexer createLexer(Project project) {
        return new PrologLexerAdapter();
    }

    @Override
    public PsiParser createParser(Project project) {
        return new PrologParser();
    }

    @Override
    public IFileElementType getFileNodeType() {
        return FILE;
    }

    @NotNull
    @Override
    public TokenSet getCommentTokens() {
        return COMMENTS;
    }

    @NotNull
    @Override
    public TokenSet getStringLiteralElements() {
        return STRINGS;
    }

    @NotNull
    @Override
    public PsiElement createElement(ASTNode node) {
        return PrologTypes.Factory.createElement(node);
    }

    @Override
    public PsiFile createFile(FileViewProvider viewProvider) {
        return new PrologFile(viewProvider);
    }

    @Override
    public SpaceRequirements spaceExistanceTypeBetweenTokens(ASTNode left, ASTNode
     ↪  right) {
        return SpaceRequirements.MAY;
    }
}
```

Listing 41: PrologParserDefinition

```
<lang.parserDefinition language="Prolog"
 ↪   implementationClass="ch.eif.intelliprolog.PrologParserDefinition"/>
```

Listing 42: ParserDefinition registration

5 Highlighting

Highlighting is used to visually distinguish distinct elements of source code to help programmers identify errors and get a better overall view of the source code they are reading or writing.

5.1 Introduction

Highlighting in IntelliJ is achieved through 3 parts, syntax highlighting, error highlighting and annotation.

Within IntelliJ IDEA highlighting is provided by the lexer, parser and the `Annotator` [39] interface. The quality of the highlighting is very dependent on the quality of the lexing and parsing, for this reason we need to ensure that our lexing and parsing is done correctly. Even if the the lexing and parsing are not perfect we can still have highlighting that is useful.

The implementation of the highlighting is inspired/based on the Logtalk IDEA plugin [30].

In this section we will see what is needed to perform syntax highlighting, basic error highlighting and possible ways to improve it, and finally how to add annotations to our source code.

5.2 Syntax highlighting

Syntax highlighting as briefly explained in the introduction is achieved with the lexer and the tokens that are generated when lexing the file. These tokens can be assigned a `TextAttributesKey` class [40], which contains the information of how the token should be coloured and represented.

To implement syntax highlighting we need to first define a `SyntaxHighlighter` and a `SyntaxHighlighterFactory` we can then register the `SyntaxHighlighterFactory` with our plugin.

To provide users a way to define their own colours for the syntax highlighting, we need to implement the `ColorSettingsPage` interface [41] and register it with our plugin.

5.2.1 PrologSyntaxHighlighter

To provide the `TextAttributesKey` classes we use a class that implements the `SyntaxHighlighter` interface [42], we will use the `SyntaxHighlighterBase` class [43] as the base class for our own implementation.

Our `PrologSyntaxHighlighter` class will extend `SyntaxHighlighterBase` as described above, we will implement the two methods defined by `SyntaxHighlighter`, these are:

- `Lexer getHighlightingLexer()`, this method just returns the lexer used for highlighting the file, we return our `PrologLexerAdapter`.
- `TextAttributesKey[] getTokenHighlights(IElementType tokenType)`, this method does the heavy lifting for the syntax highlighting, we have a long chain of `if/else if` that compares the type of the token and returns the `TextAttributesKey` array for the token type.

We also have a second `getTokenHighlights` method but it takes a `PsiElement` as argument and will be used by the annotator we will implement further on in this section.

TextAttributesKey

IntelliJ provides a set of default `TextAttributesKey` values in the `DefaultLanguageHighlighterColor` class [44].

To create a `TextAttributesKey`, we need to call the static `TextAttributesKey createTextAttributesKey(String externalName, TextAttributes defaultAttributes)` method and give it a unique identifier and `TextAttributes` arguments.

TextAttributesKey array

The reason we return an array of `TextAttributesKey` classes, is that we can layer them, we can have one `TextAttributesKey` for the color, another for the size and a final one for the font weight.

In our plugin at the moment we always return a single element array. In our `PrologSyntaxHighlighter`, we define constants for the `TextAttributesKey` classes, an example of the `TextAttributesKey` for integers can be seen in listing 43 and the corresponding array in listing 44.

```
private static final TextAttributesKey INTEGER =
        createTextAttributesKey("INTEGER_TERM",
         ↪  DefaultLanguageHighlighterColors.NUMBER);
```

Listing 43: ‘TextAttributesKey‘ for integers

```
private static final TextAttributesKey[] INTEGER_KEYS = new
 ↪  TextAttributesKey[]{INTEGER};
```

Listing 44: ‘TextAttributesKey‘ array for integers

5.2.2 PrologSyntaxHighlighterFactory

The `SyntaxHighlighterFactory` abstract class [45] provides the means of registering our `SyntaxHighlighter` class with our plugin.

To use it we simply extend it and implement its abstract method `SyntaxHighlighter getSyntaxHighlighter([@Nullable] Project project, [@Nullable] VirtualFile virtualFile)`, this method returns an instance of our `PrologSyntaxHighlighter`.

Registering the syntax highlighter

The `SyntaxHighlighterFactory` needs to be added to the extensions section of the `plugin.xml` file, the element that needs to be added is visible in listing 45.

```
<lang.syntaxHighlighterFactory language="Prolog"
```

Listing 45: SyntaxHighlighterFactory registration

5.2.3 ColorSettingsPage

The `ColorSettingsPage` allows the users of the plugin to change the colours associated with the elements of the file.

To implement the `ColorSettingsPage` interface we create a `PrologColorSettingsPage` class implementing the methods defined in it, these methods are:

- `Icon getIcon()`, this method is used to define what `Icon` should be displayed, we return the icon defined in our `PrologIcons` file.
- `SyntaxHighlighter getHighlighter()`, this method is used to return the `SyntaxHighlighter` we created above.
- `String getDemoText()`, this method is used to define a string of text to demonstrate the changes applied to the colours of the elements. Preferably this text should contain all the elements that are being highlighted. An example is visible in listing 46.
- `Map<String,TextAttributesKey> getAdditionalHighlightingTagToDescriptorMap()`, this method is used if we define some additional elements not highlighted by the syntax highlighter in the demo text. In our plugin we do not use this so we return `null`.
- `AttributesDescriptor[] getAttributeDescriptors()`, this method returns an array containing all the `AttributesDescriptor` instances, these contain a string with the name of the element as well as a reference to the corresponding `TextAttributesKey`. An example of this can be found in the listings 47 and 48.
- `ColorDescriptor[] getColorDescriptors()`, this method returns an array of `ColorDescriptor` for defining fore and background colours, in our plugin we return `ColorDescriptor.EMPTY_ARRAY`.
- `String getDisplayName()`, this method returns the name to display in the settings, we return `Prolog`.

```java
public String getDemoText() {
    return "factorial(0, 1).\n" +
            "factorial(N, F) :-\n" +
            "         N>0,\n" +
            "         N1 is N-1,\n" +
            "         factorial(N1, F1),\n" +
            "         F is F1 * N.\n" +
            "not(P) :- P, !, fail.\n" +
            "not(_).\n";
}
```

Listing 46: getDemoText example

Registering the color settings page

The `ColorSettingsPage` needs to be added to the extensions section of the `plugin.xml` file, the element that needs to be added is visible in listing 49.

```
private static final AttributesDescriptor[] DESCRIPTORS = new AttributesDescriptor[]{
        new AttributesDescriptor("Operator", PrologSyntaxHighlighter.OPERATOR),
        new AttributesDescriptor("Cut", PrologSyntaxHighlighter.CUT),
        new AttributesDescriptor("Quoted Atom", PrologSyntaxHighlighter.QUOTED_ATOM),
        new AttributesDescriptor("Unquoted Atom",
         ↪  PrologSyntaxHighlighter.UNQUOTED_ATOM),
        new AttributesDescriptor("Keyword Atom",
         ↪  PrologSyntaxHighlighter.KEYWORD_ATOM),
        new AttributesDescriptor("Anonymous Variable",
         ↪  PrologSyntaxHighlighter.ANONYMOUS_VARIABLE),
        new AttributesDescriptor("Named Variable",
         ↪  PrologSyntaxHighlighter.NAMED_VARIABLE),
        new AttributesDescriptor("Parenthesis", PrologSyntaxHighlighter.PARENTHESIS),
        new AttributesDescriptor("Brackets", PrologSyntaxHighlighter.BRACKETS),
        new AttributesDescriptor("Bad Character",
         ↪  PrologSyntaxHighlighter.BAD_CHARACTER),
        new AttributesDescriptor("Quoted Compound Name",
         ↪  PrologSyntaxHighlighter.QUOTED_COMPOUND_NAME),
        new AttributesDescriptor("Unquoted Compound Name",
         ↪  PrologSyntaxHighlighter.UNQUOTED_COMPOUND_NAME),
        new AttributesDescriptor("Keyword Compound Name",
         ↪  PrologSyntaxHighlighter.KEYWORD_COMPOUND_NAME)
};
```

Listing 47: AttributesDescriptor array example

```
public AttributesDescriptor[] getAttributeDescriptors() {
    return DESCRIPTORS;
}
```

Listing 48: getAttributeDescriptors example

```
<lang.foldingBuilder language="Prolog"
 ↪  implementationClass="ch.eif.intelliprolog.editor.PrologFoldingBuilder"/>
```

Listing 49: ColorSettingsPage registration

5.3 Error highlighting

Highlighting errors in IDEA is done by two different ways, these can be used at the same time.

The first way is highlighting lexer errors, these errors occur when the contents of the file do not follow the rules defined by the lexer.

The second way happens during parsing, these errors occur when a sequence of tokens does not conform to the rules defined in the grammar of the language. We will not go through how to implement this type of highlighting.

5.3.1 Highlighting lexer errors

The highlighting of lexer errors is achieved by specifying in the lexer definition a wildcard rule using the `dot` character, which is only matched when no other rule matches and returns a token of type `TokenType.BAD_CHARACTER`. In the syntax highlighter, when we encounter a `TokenType.BAD_CHARACTER` in the `getTokenHighlights` method, we return the `TextAttributesKey` from `HighlighterColors.BAD_CHARACTER`.

5.3.2 Highlighting parser errors

The highlighting of parser errors is achieved, as far as I understand after looking through the IntelliJ Platform SDK DevGuide [46], the source files and other plugins that have implemented this feature, during the parsing in the `ParserClass` by calling the `PsiBuilder.error()` [47] when a sequence of tokens does not conform to the grammar of the language.

5.4 Annotator

The [`Annotator` interface [39] in IntelliJ IDEA displays messages under PSI elements when we hover over them, giving us more information about what they are. We can also highlight a region of text as a warning or error and then optionally provide a fix for the error or suggest a better way of writing the code being highlighted.

In our plugin we will implement some very basic annotations to demonstrate the `Annotator` interface, we will add information annotations for:

- Binary operators
- Left operators
- Functor keywords
- Atom keywords

We will implement the `Annotator` interface in the `PrologAnnotator` and then register it with our plugin.

5.4.1 PrologAnnotator

The `PrologAnnotator` class implements the `Annotator` interface and the only method that is necessary to implement is the `void annotate(PsiElement element, AnnotationHolder holder)` method.

We will add 3 extra static methods, these are :

- `boolean shouldAnnotate(PsiElement element)`, this method checks if the `PsiElement` is one of the elements we whish to annotate, to accomplish this we use some static methods that we will define in the `PrologPsiUtil` class we will create after this.
- `void highlightTokens(PsiElement element, AnnotationHolder holder, PrologSyntaxHighlighter highlighter)`, this method gets a new `TextAttributesKey` array from the highlighter, creates the info annotation on the element with a message from the following method. The final part is setting the new `TextAttributesKey` values from the array. This method is depicted in listing 50.
- `String getMessage(PsiElement element)`, this method returns a string describing the element in question.

In the `annotate` method, depicted in the listing 51, from the `Annotator` interface we check if we should annotate the element and if `true`, call the `highlightTokens` method

5.4.2 Register annotator

The `Annotator` needs to be added to the extensions section of the `plugin.xml` file, the element that needs to be added is visible in listing 52.

```java
private static void highlightTokens(PsiElement element, AnnotationHolder holder,
 ↪  PrologSyntaxHighlighter highlighter) {
    TextAttributesKey[] keys = highlighter.getTokenHighlights(element);
    Annotation annotation = holder.createInfoAnnotation(element.getNode(),
     ↪  getMessage(element));
    for (TextAttributesKey key : keys) {
        TextAttributes attributes =
         ↪  EditorColorsManager.getInstance().getGlobalScheme().getAttributes(key);
        annotation.setEnforcedTextAttributes(attributes);
    }
}
```

Listing 50: highlightTokens method

```java
public void annotate(@NotNull PsiElement element, @NotNull AnnotationHolder holder) {
    if (shouldAnnotate(element)) {
        highlightTokens(element, holder, new PrologSyntaxHighlighter());
    }
}
```

Listing 51: annotate method

```xml
<annotator language="Prolog"
 ↪  implementationClass="ch.eif.intelliprolog.editor.PrologAnnotator"/>
```

Listing 52: ColorSettingsPage registration

6 Code handling

Code handling is a very big part of any IDE and Jetbrains IDEs are well known to have some of the best code handling.

6.1 Introduction

In this section we will cover some of the more basic code handling aspects, commenting and code folding.

These features are more visual than other code handling features but are still very useful since they allow programmers to reduce the visual clutter of the source code. Visual clutter can be very distracting when writing bigger and more complicated programs, especially if the files are also very big.

Other code handling features that we envisioned implementing but did not for different reasons, mainly lack of time, are:

- Refactoring
- Code completion
- Find Usages

The implementations of these features are taken from the Logtalk IDEA plugin [30].

Without further ado, let us start with commenting.

6.2 Commenting

The ability of commenting source code is one of the most important any programming language must provide. The reason why this is of such an importance is mainly for documenting the source code, but also for the debugging process, it allows us to keep parts of the program in the file and not be executed.

Being able to select a portion of code and quickly comment it in or out, using the IDE, allows the developer to be more productive.

Providing commenting capabilities in IDEA is very simple, we only need to provide the plugin an implementation of the `Commenter` interface [48] and register it with our plugin.

6.2.1 Commenter implementation

The `Commenter` interface defines 5 methods to be implemented, these are:

- `String getLineCommentPrefix()`, returns the line comment string for the language we are commenting, in Prolog this is `%`.
- `String getBlockCommentPrefix()`, returns the block comment prefix string for the language we are commenting, in Prolog this is `/*`.
- `String getBlockCommentSuffix()`, returns the block comment suffix string for the language we are commenting, in Prolog this is `*/`.
- `String getCommentedBlockCommentPrefix()`, returns the commented block comment prefix string for the language we are commenting, in Prolog this is `/**`.
- `String getCommentedBlockCommentSuffix()`, returns the commented block comment suffix string for the language we are commenting, in Prolog this is `*/`.

6.2.2 Register commenter

The `Commenter` needs to be added to the extensions section of the `plugin.xml` file, the element that needs to be added is visible in listing 53.

```
<lang.commenter language="Prolog"
↪   implementationClass="ch.eif.intelliprolog.editor.PrologCommenter"/>
```

Listing 53: Commenter registration

6.3 Folding

Folding allows a developer to hide parts of the source code, to diminish the visual clutter, and allow him or her to focus only on the parts important at the moment.

There are normally different elements that can be folded, most often these elements are:

- Elements surrounded by pairs of braces, brackets or parenthesis
- Whole functions, methods or classes
- Multiline/block comments

In this plugin we are going to provide the ability to fold:

- Comments
- Lists, which are surrounded by brackets
- Sentences/Rules, the Prolog equivalent of functions

To provide folding JetBrains provides like for most of the features we have implemented an interface, the `FoldingBuilder` interface [49], that we need to implement and register with our plugin.

6.3.1 FoldingBuilder

The `FoldingBuilder` interface defines 3 methods that need to be implemented, these are:

- `FoldingDescriptor[] buildFoldRegions(ASTNode node, Document document)`, builds an array containing all the nodes that can be folded together starting from the `node` argument, depicted in listing 54.
- `String getPlaceholderText(ASTNode node)`, returns the text used to display instead of the folded region, depicted in listing 55.
- `boolean isCollapsedByDefault(ASTNode node)`, sets if a node should start of being folded, an example would be if we wanted all comments to be folded and only become visible when we want to see them, depicted in listing 56.

```
@Override
public FoldingDescriptor[] buildFoldRegions(@NotNull ASTNode node, @NotNull Document
 ↪  document) {
    List<FoldingDescriptor> descriptors = new ArrayList<>();
    collectDescriptorsRecursively(node, document, descriptors);
    return descriptors.toArray(new FoldingDescriptor[descriptors.size()]);
}
```

Listing 54: buildFoldRegions method

```java
@Override
public String getPlaceholderText(@NotNull ASTNode node) {
    PsiElement psi = node.getPsi();
    if (isComment(node)) {
        return "/*...*/";
    } else if (isList(psi)) {
        return "[...]";
    }

    return collapseWhiteSpace(psi.getText()).substring(0, 10) + "...";
}

private String collapseWhiteSpace(String text) {
    return WHITES.matcher(text).replaceAll(" ");
}
```

Listing 55: getPlaceHolderText method

```java
@Override
public boolean isCollapsedByDefault(@NotNull ASTNode node) {
    return false;
}
```

Listing 56: isCollapsedByDefault method

6.3.2 Register FoldingBuilder

The `FoldingBuilder` needs to be added to the extensions section of the `plugin.xml` file, the element that needs to be added is visible in listing 57.

```xml
<lang.foldingBuilder language="Prolog"
 ↪  implementationClass="ch.eif.intelliprolog.editor.PrologFoldingBuilder"/>
```

Listing 57: FoldingBuilder registration

7 Differences between the JetBrains IDEs

JetBrains provides a large choice of different IDEs, their main IDE being IntelliJ IDEA, they also provide some other IDEs like PyCharm, WebStorm and GoLand.

7.1 Introduction

In this section we will compare JetBrains IDEs, the differences in their architecture and how this affects the development of plugins for them.

We will focus on IntelliJ IDEA [3] and PyCharm [50] since both of these provide Community editions and are therefore free and open source software, which means we can inspect their source code, available on GitHub, IntelliJ IDEA GitHub repo [4] and PyCharm GitHub repo [50]-GH.

The reason we are going through these differences is that during the process of this project, we saw in some of the forums and the Plugin Compatibility with IntelliJ Platform Products page [51] that not all JetBrains IDEs are created equal. All of them are based on the `IntelliJ Platform`. Some of the IDEs have their own extra modules that are not present in the others.

The modules included in all the IDEs are:

- `com.intellij.modules.platform`
- `com.intellij.modules.lang`
- `com.intellij.modules.vcs`
- `com.intellij.modules.xml`
- `com.intellij.modules.xdebugger`

So if we want to develop a plugin for all the JetBrains IDEs we should only use classes and interfaces from these modules.

Next we will take a more in detail look at IntelliJ IDEA and PyCharm.

7.2 IntelliJ IDEA

The IntelliJ IDEA IDEs, the Community and Ultimate editions provide the most functionality out of all the JetBrains IDEs since the IntelliJ Platform is the foundation of the IntelliJ IDEA Community edition, they also include the `com.intellij.modules.java` module. The Android Studio IDE also includes the `com.intellij.modules.java` module.

The Ultimate edition is based on the Community Edition and has extra proprietary modules from `com.intellij.modules.ultimate` and the database modules in `com.intellij.modules.database`.

So if we develop a plugin that uses classes or interface from these modules, we will be restricting our plugin to being able to be used in the IntelliJ IDEA, Ultimate edition if we use the `com.intellij.modules.ultimate` and `com.intellij.modules.database`, and Android Studio IDEs.

The Ultimate edition allows us to install nearly all the available plugins for IntelliJ IDEs, JetBrains provides plugins that add the features of their other IDEs.

7.3 PyCharm

The PyCharm IDE is based on the IntelliJ IDEA Community edition IDE with a Python overlay. It also has two versions, Community and Ultimate editions, like IntelliJ IDEA.

The PyCharm IDEs have the basic modules as defined in the introduction, as well as the database modules `com.intellij.modules.database` and python modules `com.intellij.modules.python`.

Another difference between PyCharm and the IntelliJ IDEA IDEs, is that the non-IDEA IDEs have a simplified project creation method (`Open Directory`).

8 Comparison of the different Prolog implementations

Prolog [52] is a general-purpose logic programming language, initially conceived in the 1970s by Alain Colmerauer [53].

8.1 Introduction

In this section we will compare 2 implementations and an extension for these implementations, GNU Prolog [1] the language we have focused on in this project, SWI-Prolog [54] and the Logtalk [55] extension that adds object-oriented elements to Prolog.

We will compare these implementations across the following features and tools:

- Compiled code
- Unicode support
- Object-orientated support
- Interfacing with other programming languages
- Availability of an interactive interpreter

But first Prolog is divided in two dialects ISO Prolog and Edinburgh Prolog, Edinburg Prolog being the standard upon which ISO Prolog is based. The main difference between them is mostly cosmetic and the built-in predicates for input and output.

8.2 GNU-Prolog

GNU Prolog is a Prolog implementation, constisting mainly of a compiler, developed by Daniel Diaz [2] an associate professor at the Panthéon Sorbonne University of Paris [56].

It conforms to ISO Prolog and adds a constraint solver over Finite Domains.

The features available in GNU Prolog are:

- Compile our source code to abstract machine code to run on a Warren Abstract Machine [57].
- Native interfacing with the C programming language, there also exists GNU Prolog for Java [58]
- Interactive interpreter, allows us to consult Prolog and run them interactively.

It does not provide support for Unicode encoding or object-orientated elements but can be added with Logtalk which we will see shortly.

8.3 SWI-Prolog

SWI-Prolog is a Prolog implementation, developed by Jan Wielemaker [59] a researcher at the Vrije Universiteit Amsterdam [60].

It conforms to ISO Prolog as well Edinburgh Prolog.

The features available in SWI-Prolog are:

- Compiles the source code to abstract machine code to run on a Warren Abstract Machine [57].
- Support for Unicode encoding.
- Native interfacing with C programming language and Java through the JPL package [61]
- Interactive interpreter, allows us to consult Prolog and run them interactively.

As with GNU Prolog, SWI-Prolog does not support object-orientated elements but can be added with Logtalk.

SWI-Prolog also has a built-in editor, PCeEmacs, and a lot of extra functionality can be added through its package system.

8.4 LogTalk

Logtalk [55] is a Prolog implementation with object-oriented features that extends other Prolog implementations, using their compiler as its back-end compiler. It was created by Paulo Moura.

It's features will be dependant on the Prolog implementation used as the back-end compiler.

9 Quick overview of the different licenses for software

Licenses are an integral part of any software project and because of this we thought we should cover some of these licenses and their differences.

9.1 Introduction

Understanding what a license means and the impact it will have on our software is very difficult to understand unless we were a lawyer. To remedy the problem of understanding these licenses there are a couple of websites that give quick summaries and checklists what a license allows, disallows and enforces. The websites are Choose a License [62] and tl;dr Legal [63].

For our project we have chosen to use a Apache 2.0 License, since the IntelliJ Platform SDK and most of the plugins developed for IntelliJ use this license.

Next we will do a quick overview of the Apache 2.0 License as well as the MIT License.

9.2 MIT License (Expat)

The MIT License [64] is another one of the most popular and used OSS licenses.

A quick summary available at Choose a license / MIT License [65], follows:

> A short and simple permissive license with conditions only requiring preservation of copyright and license notices. Licensed works, modifications, and larger works may be distributed under different terms and without source code.

9.3 Apache License 2.0 (Apache-2.0)

The Apache License 2.0 [66] is one of the most popular and used OSS licenses.

A quick summary available at Choose a license / Apache License 2.0 [67], follows:

> A permissive license whose main conditions require preservation of copyright and license notices. Contributors provide an express grant of patent rights. Licensed works, modifications, and larger works may be distributed under different terms and without source code.

10 Conclusion

We now have a plugin that can be used in IntelliJ IDEA for developing Prolog programs, more specifically GNU Prolog if we wish to launch the files directly in the interpreter. Hopefully this will make the lives of the Logic Programming course students who have less experience using the terminal easier.

This plugin still has a lot of features that could be added or improved, but the main feature we wanted to provide, the ability to launch a file in the interpreter directly is functional and usable but like the rest of the plugin can always be improved.

10.1 Improvement ideas

This plugin has a couple of ideas that can be improved and new features that can be added. The areas of improvement are:

- Lexing and Parsing, this is probability the area of the plugin that could be improved the most and most likely will never be perfect. The lexer could be more fine-grained, which would then help improve the parser and detecting errors. The parser needs to be more precise and provide an easier PSI tree for some of the features that are very dependant on it.

- Error highlighting, this can be improved mostly by improving the parser and displaying the parser errors as well as the lexer errors.

- Code completion, is a feature that is always welcome, specially when learning a new language and we do not know the syntax or all the built-in functions/predicates available. It can also improve efficiency by decreasing the amount of code that needs to be written manually, notably for functors that are used for often.

- Refactoring and finding usages, are features that have become more and more important the larger software projects become. With more and more software having more than 1 million lines of code, according to the site ‘information is beautiful’ [68], even the XBox HD DVD Player software had just over 5 million lines of code. At sizes like that it would be impossible for a human to find all call and rename correctly a function that is spread all over the project, that is where refactoring and finding the usages comes in handy.

There are a lots of other features that can be added to this plugin, for a more complete list of features that can be implemented in IntelliJ IDEs, check out the Custom Language Support [69] page of the IntelliJ Platform SDK DevGuide.

10.2 Goals achieved

When we started this project, we came up with a list of features that we would like to implement, unfortunately we were not able to implement all of these. The features that were not implemented are the following:

- Code completion, the main reason being the difficulty knowing what to suggest. Some ideas were to suggest all the Prolog built-in predicates and/or the predicates in the current file.
- Code formatting, mainly because we were not sure what good code formatting in Prolog should look like, everybody has their own preferences.
- Refactoring, to be able to provide robust refactoring the parser and lexer also have to be robust and not have any ambiguities.

10.3 Personal opinion

This project was very interesting but at the same time very frustrating, mainly because of the poor documentation for the complicated aspects of the plugin development process for the IntelliJ Platform.

The only way to understand how something needs to be implemented relied on reading the implementations of other plugins, and most plugins implement features very differently. It seems the more complicated features have several different ways that they can be implemented and there is no clear indication what the proper or best way is.

There is one benefit of this, I learnt to read a lot of source code and understand it. It also gave me a new appreciation of good documentation and the necessity of it.

In the end, I am pleased with the result, since half way through I sort of lost hope in being able to implement the main objective of the plugin, launching the current file. It was only after reading through a dozen different plugins that I found a solution that I could adapt to our needs.

11 Statement on Oath

I, undersigned, Owen McGill, certify that this report is my own work, based on my personal study and/or research and that I have acknowledged all material and sources used in its preparation, whether they be books, articles, reports, lecture notes, and any other kind of document, electronic or personal communication. I also certify that this report has not previously been submitted for assessment in any other unit, except where specific permission has been granted from all unit coordinators involved, or at any other time in this unit, and that I have not copied in part or whole or otherwise plagiarized the work of other students and/or persons.

12 Bibliography

[1] “GNU prolog.” [Online]. Available: http://www.gprolog.org/[1].

[2] “Daniel diaz.” [Online]. Available: http://cri-dist.univ-paris1.fr/diaz/[2].

[3] “IntelliJ idea.” [Online]. Available: https://www.jetbrains.com/idea/[3].

[4] “IntelliJ idea github repo.” [Online]. Available: https://github.com/JetBrains/intellij-community[4].

[5] “‘LanguageFileType‘.” [Online]. Available: https://upsource.jetbrains.com/idea-ce/file/idea-ce-12c6e6cd02d57c0ab4fd314f62b4ecb94841a0fa/platform/core-api/src/com/intellij/opena

[6] “Spring.” [Online]. Available: https://spring.io/[6].

[7] “Typescript.” [Online]. Available: https://www.typescriptlang.org/[7].

[8] “Javascript.” [Online]. Available: https://www.javascript.com/[8].

[9] “AngularJS.” [Online]. Available: https://angularjs.org/[9].

[10] “SDKs.” [Online]. Available: https://www.jetbrains.com/help/idea/sdk.html[10].

[11] “‘SdkType‘ class.” [Online]. Available: https://upsource.jetbrains.com/idea-ce/file/idea-ce-a00d19098ca9850e1b28a9db178df5a4b3456659/platform/lang-api/src/com/intellij/open

[1]%20http://www.gprolog.org/
[2]%20http://cri-dist.univ-paris1.fr/diaz/
[3]%20https://www.jetbrains.com/idea/
[4]%20https://github.com/JetBrains/intellij-community
[5]%20https://upsource.jetbrains.com/idea-ce/file/idea-ce-12c6e6cd02d57c0ab4fd314f62b4ecb94841a0fa/platform/core-api/src/com/intellij/openapi/fileTypes/LanguageFileType.java
[6]%20https://spring.io/
[7]%20https://www.typescriptlang.org/
[8]%20https://www.javascript.com/
[9]%20https://angularjs.org/
[10]%20https://www.jetbrains.com/help/idea/sdk.html
[11]%20https://upsource.jetbrains.com/idea-ce/file/idea-ce-a00d19098ca9850e1b28a9db178df5a4b3456659/platform/lang-api/src/com/intellij/openapi/projectRoots/SdkType.java?line=36

[12] "'Nvm'." [Online]. Available: https://github.com/creationix/nvm[12].

[13] "'AnAction.java'." [Online]. Available: https://upsource.jetbrains.com/idea-ce/file/idea-ce-32b2fa21845ae8598f946709d2aa98c005add383/platform/editor-ui-api/src/com/intellij/openapi/actionSystem/AnAction.java[13].

[14] "Actions section." [Online]. Available: https://www.jetbrains.org/intellij/sdk/docs/basics/action_

[15] "Haskell idea plugin." [Online]. Available: https://github.com/atsky/haskell-idea-plugin/[15].

[16] "Haskell idea plugin / repl." [Online]. Available: https://github.com/atsky/haskell-idea-plugin/tree/master/plugin/src/org/jetbrains/haskell/repl[16].

[17] "'LanguageConsoleImpl' class." [Online]. Available: https://upsource.jetbrains.com/idea-ce/file/idea-ce-dba03e40ff8fc26feb037493ca72af40c273dfa4/platform/lang-impl/src/com/intellij/execut

[18] "'ColoredProcessHandler'." [Online]. Available: https://upsource.jetbrains.com/idea-ce/file/idea-ce-dba03e40ff8fc26feb037493ca72af40c273dfa4/platform/platform-impl/src/com/intellij/ex

[19] "'GeneralCommandLine'." [Online]. Available: https://upsource.jetbrains.com/idea-ce/file/idea-ce-dba03e40ff8fc26feb037493ca72af40c273dfa4/platform/platform-api/src/com/intellij/exe

[20] "'Gprolog' commandline parameters." [Online]. Available: http://www.gprolog.org/manual/gprolo

[21] "Environment variable." [Online]. Available: http://www.gprolog.org/manual/gprolog.html#sec13

[22] "IntelliJ platform sdk devguide documentation / virtual file system." [Online]. Available: http://www.jetbrains.org/intellij/sdk/docs/basics/virtual_file_system.html[22].

[12] %20https://github.com/creationix/nvm
[13] %20https://upsource.jetbrains.com/idea-ce/file/idea-ce-32b2fa21845ae8598f946709d2aa98c005add383/platform/editor-ui-api/src/com/intellij/openapi/actionSystem/AnAction.java
[14] %20https://www.jetbrains.org/intellij/sdk/docs/basics/action_system.html
[15] %20https://github.com/atsky/haskell-idea-plugin/
[16] %20https://github.com/atsky/haskell-idea-plugin/tree/master/plugin/src/org/jetbrains/haskell/repl
[17] %20https://upsource.jetbrains.com/idea-ce/file/idea-ce-dba03e40ff8fc26feb037493ca72af40c273dfa4/platform/lang-impl/src/com/intellij/execution/console/LanguageConsoleImpl.java
[18] %20https://upsource.jetbrains.com/idea-ce/file/idea-ce-dba03e40ff8fc26feb037493ca72af40c273dfa4/platform/platform-impl/src/com/intellij/execution/process/KillableProcessHandler.java
[19] %20https://upsource.jetbrains.com/idea-ce/file/idea-ce-dba03e40ff8fc26feb037493ca72af40c273dfa4/platform/platform-api/src/com/intellij/execution/configurations/GeneralCommandLine.java
[20] %20http://www.gprolog.org/manual/gprolog.html#sec8
[21] %20http://www.gprolog.org/manual/gprolog.html#sec13
[22] %20http://www.jetbrains.org/intellij/sdk/docs/basics/virtual_file_system.html

[23] "'Program structure interface'." [Online]. Available: http://www.jetbrains.org/intellij/sdk/docs/ba

[24] "IntelliJ platform sdk devguide." [Online]. Available: http://www.jetbrains.org/intellij/sdk/docs/w

[25] "'IElementType'." [Online]. Available: https://upsource.jetbrains.com/idea-ce/file/idea-ce-5a00c10a69088737a364efbc82a082207a598b45/platform/core-api/src/com/intellij/psi/t

[26] "Backus-naur form." [Online]. Available: https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_

[27] "Context-free grammar." [Online]. Available: https://en.wikipedia.org/wiki/Context-free_grammar[27].

[28] "Grammar-kit plugin." [Online]. Available: https://github.com/JetBrains/Grammar-Kit[28].

[29] "'ASTWrapperPsiElement'." [Online]. Available: https://upsource.jetbrains.com/idea-ce/file/idea-ce-dba03e40ff8fc26feb037493ca72af40c273dfa4/platform/core-impl/src/com/intellij/extapi

[30] "Logtalk3 intellij plugin." [Online]. Available: https://github.com/LogtalkDotOrg/logtalk3/tree/m

[31] "Prolog bnf grammar." [Online]. Available: https://gist.github.com/kuoe0/6191706[31].

[32] "'Flex'." [Online]. Available: https://en.wikipedia.org/wiki/Flex_(lexical_analyser_generator[32].

[33] "'Lex'." [Online]. Available: https://en.wikipedia.org/wiki/Lex_(software[33].

[34] "'JFlex'." [Online]. Available: http://www.jflex.de/[34].

[35] "'JFlex' manual." [Online]. Available: http://jflex.de/manual.pdf[35].

[23]%20http://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/psi.html
[24]%20http://www.jetbrains.org/intellij/sdk/docs/welcome.html
[25]%20https://upsource.jetbrains.com/idea-ce/file/idea-ce-5a00c10a69088737a364efbc82a082207a598b45/platform/core-api/src/com/intellij/psi/tree/IElementType.java
[26]%20https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form
[27]%20https://en.wikipedia.org/wiki/Context-free_grammar
[28]%20https://github.com/JetBrains/Grammar-Kit
[29]%20https://upsource.jetbrains.com/idea-ce/file/idea-ce-dba03e40ff8fc26feb037493ca72af40c273dfa4/platform/core-impl/src/com/intellij/extapi/psi/ASTWrapperPsiElement.java
[30]%20https://github.com/LogtalkDotOrg/logtalk3/tree/master/coding/idea
[31]%20https://gist.github.com/kuoe0/6191706
[32]%20https://en.wikipedia.org/wiki/Flex_(lexical_analyser_generator
[33]%20https://en.wikipedia.org/wiki/Lex_(software
[34]%20http://www.jflex.de/
[35]%20http://jflex.de/manual.pdf

[36] "'FlexAdapter' class." [Online]. Available: https://upsource.jetbrains.com/idea-ce/file/idea-ce-32b2fa21845ae8598f946709d2aa98c005add383/platform/core-api/src/com/intellij/lexer/

[37] "'PsiFileBase' class." [Online]. Available: https://upsource.jetbrains.com/idea-ce/file/idea-ce-32b2fa21845ae8598f946709d2aa98c005add383/platform/core-impl/src/com/intellij/exta

[38] "'ParserDefinition'." [Online]. Available: https://upsource.jetbrains.com/idea-ce/file/idea-ce-dba03e40ff8fc26feb037493ca72af40c273dfa4/platform/core-api/src/com/intellij/lang/Pa

[39] "'Annotator' interface." [Online]. Available: https://upsource.jetbrains.com/idea-ce/file/idea-ce-d00d8b4ae3ed33097972b8a4286b336bf4ffcfab/platform/analysis-api/src/com/intellij/lang/annotation/Annotator.java[39].

[40] "'TextAttributesKey' class." [Online]. Available: https://upsource.jetbrains.com/idea-ce/file/idea-ce-5ecc06ee734c5a9c83a92cbf28c9dd3030293fc7/platform/core-api/src/com/intellij/openap

[41] "'ColorSettingsPage' interface." [Online]. Available: https://upsource.jetbrains.com/idea-ce/file/idea-ce-935f73f27a9c63d529d5f6ea38948a5944c7e630/platform/lang-api/src/com/intellij/opena

[42] "'SyntaxHighlighter' interface." [Online]. Available: https://upsource.jetbrains.com/idea-ce/file/idea-ce-32b2fa21845ae8598f946709d2aa98c005add383/platform/editor-ui-api/src/com/intellij/openapi/fileTypes/SyntaxHighlighter.java[42].

[43] "'SyntaxHighlighterBase' class." [Online]. Available: https://upsource.jetbrains.com/idea-ce/file/idea-ce-32b2fa21845ae8598f946709d2aa98c005add383/platform/editor-ui-api/src/com/intellij/openapi/fileTypes/SyntaxHighlighterBase.java[43].

[44] "'DefaultLanguageHighlighterColors' class." [Online]. Available: https://upsource.jetbrains.com/ic
ce/file/idea-ce-5a00c10a69088737a364efbc82a082207a598b45/platform/editor-ui-api/src/com/intellij/openapi/editor/DefaultLanguageHighlighterColors.java[44].

[36] %20https://upsource.jetbrains.com/idea-ce/file/idea-ce-32b2fa21845ae8598f946709d2aa98c005add383/platform/core-api/src/com/intellij/lexer/FlexAdapter.java

[37] %20https://upsource.jetbrains.com/idea-ce/file/idea-ce-32b2fa21845ae8598f946709d2aa98c005add383/platform/core-impl/src/com/intellij/extapi/psi/PsiFileBase.java

[38] %20https://upsource.jetbrains.com/idea-ce/file/idea-ce-dba03e40ff8fc26feb037493ca72af40c273dfa4/platform/core-api/src/com/intellij/lang/ParserDefinition.java

[39] %20https://upsource.jetbrains.com/idea-ce/file/idea-ce-d00d8b4ae3ed33097972b8a4286b336bf4ffcfab/platform/analysis-api/src/com/intellij/lang/annotation/Annotator.java

[40] %20https://upsource.jetbrains.com/idea-ce/file/idea-ce-5ecc06ee734c5a9c83a92cbf28c9dd3030293fc7/platform/core-api/src/com/intellij/openapi/editor/colors/TextAttributesKey.java

[41] %20https://upsource.jetbrains.com/idea-ce/file/idea-ce-935f73f27a9c63d529d5f6ea38948a5944c7e630/platform/lang-api/src/com/intellij/openapi/options/colors/ColorSettingsPage.java

[42] %20https://upsource.jetbrains.com/idea-ce/file/idea-ce-32b2fa21845ae8598f946709d2aa98c005add383/platform/editor-ui-api/src/com/intellij/openapi/fileTypes/SyntaxHighlighter.java

[43] %20https://upsource.jetbrains.com/idea-ce/file/idea-ce-32b2fa21845ae8598f946709d2aa98c005add383/platform/editor-ui-api/src/com/intellij/openapi/fileTypes/SyntaxHighlighterBase.java

[44] %20https://upsource.jetbrains.com/idea-ce/file/idea-ce-5a00c10a69088737a364efbc82a082207a598b45/

[45] "'SyntaxHighlighterFactory' abstract class." [Online]. Available: https://upsource.jetbrains.com/id
ce/file/idea-ce-dba03e40ff8fc26feb037493ca72af40c273dfa4/platform/editor-ui-api/src/com/intellij/ope

[46] "IntelliJ platform sdk devguide / syntax and error highlighting." [Online]. Available:
http://www.jetbrains.org/intellij/sdk/docs/reference_guide/custom_language_support/syntax_high

[47] "'PsiBuilder.error()'." [Online]. Available: https://upsource.jetbrains.com/idea-
ce/file/idea-ce-d00d8b4ae3ed33097972b8a4286b336bf4ffcfab/platform/core-api/src/com/intellij/lang/I

[48] "'Commenter' interface." [Online]. Available: https://upsource.jetbrains.com/idea-
ce/file/idea-ce-12c6e6cd02d57c0ab4fd314f62b4ecb94841a0fa/platform/core-api/src/com/intellij/lang/C

[49] "'FoldingBuilder' interface." [Online]. Available: https://upsource.jetbrains.com/idea-
ce/file/idea-ce-12c6e6cd02d57c0ab4fd314f62b4ecb94841a0fa/platform/core-api/src/com/intellij/lang/f

[50] "PyCharm." [Online]. Available: https://www.jetbrains.com/pycharm/[50].

[51] "Plugin compatibility with intellij platform products page." [Online]. Available:
http://www.jetbrains.org/intellij/sdk/docs/basics/getting_started/plugin_compatibility.html[51].

[52] "Prolog." [Online]. Available: https://en.wikipedia.org/wiki/Prolog[52].

[53] "Alain colmerauer." [Online]. Available: https://en.wikipedia.org/wiki/Alain_Colmerauer[53].

[54] "SWI-prolog." [Online]. Available: http://swi-prolog.org/[54].

[55] "Logtalk." [Online]. Available: http://logtalk.org/[55].

platform/editor-ui-api/src/com/intellij/openapi/editor/DefaultLanguageHighlighterColors.
java

[45]%20https://upsource.jetbrains.com/idea-ce/file/idea-ce-dba03e40ff8fc26feb037493ca72af40c273dfa4/
platform/editor-ui-api/src/com/intellij/openapi/fileTypes/SyntaxHighlighterFactory.
java

[46]%20http://www.jetbrains.org/intellij/sdk/docs/reference_guide/custom_language_
support/syntax_highlighting_and_error_highlighting.html

[47]%20https://upsource.jetbrains.com/idea-ce/file/idea-ce-d00d8b4ae3ed33097972b8a4286b336bf4ffcfab/
platform/core-api/src/com/intellij/lang/PsiBuilder.java?nav=8173:8178:focused&
line=277&preview=false

[48]%20https://upsource.jetbrains.com/idea-ce/file/idea-ce-12c6e6cd02d57c0ab4fd314f62b4ecb94841a0fa/
platform/core-api/src/com/intellij/lang/Commenter.java

[49]%20https://upsource.jetbrains.com/idea-ce/file/idea-ce-12c6e6cd02d57c0ab4fd314f62b4ecb94841a0fa/
platform/core-api/src/com/intellij/lang/folding/FoldingBuilder.java

[50]%20https://www.jetbrains.com/pycharm/

[51]%20http://www.jetbrains.org/intellij/sdk/docs/basics/getting_started/plugin_
compatibility.html

[52]%20https://en.wikipedia.org/wiki/Prolog

[53]%20https://en.wikipedia.org/wiki/Alain_Colmerauer

[54]%20http://swi-prolog.org/

[55]%20http://logtalk.org/

[56] "Panthéon sorbonne university of paris." [Online]. Available: http://www.pantheonsorbonne.fr/[56].

[57] "Warren abstract machine." [Online]. Available: https://en.wikipedia.org/wiki/Warren_Abstract_

[58] "GNU prolog for java." [Online]. Available: https://www.gnu.org/software/gnuprologjava/[58].

[59] "Jan wielemaker." [Online]. Available: https://research.vu.nl/en/persons/jan-wielemaker[59].

[60] "Vrije universiteit amsterdam." [Online]. Available: https://vu.nl/en/[60].

[61] "JPL package." [Online]. Available: http://www.swi-prolog.org/packages/jpl/[61].

[62] "Choose a license." [Online]. Available: https://choosealicense.com/[62].

[63] "Tl;dr legal." [Online]. Available: https://tldrlegal.com/[63].

[64] "MIT license." [Online]. Available: https://opensource.org/licenses/MIT[64].

[65] "Choose a license / mit license." [Online]. Available: https://choosealicense.com/licenses/mit/[65].

[66] "Apache license 2.0." [Online]. Available: https://www.apache.org/licenses/LICENSE-2.0[66].

[67] "Choose a license / apache license 2.0." [Online]. Available: https://choosealicense.com/licenses/ap
2.0/[67].

[68] "The site 'information is beautiful'." [Online]. Available: https://informationisbeautiful.net/visuali
lines-of-code/[68].

[69] "Custom language support." [Online]. Available: https://www.jetbrains.org/intellij/sdk/docs/refe

[56] %20http://www.pantheonsorbonne.fr/
[57] %20https://en.wikipedia.org/wiki/Warren_Abstract_Machine
[58] %20https://www.gnu.org/software/gnuprologjava/
[59] %20https://research.vu.nl/en/persons/jan-wielemaker
[60] %20https://vu.nl/en/
[61] %20http://www.swi-prolog.org/packages/jpl/
[62] %20https://choosealicense.com/
[63] %20https://tldrlegal.com/
[64] %20https://opensource.org/licenses/MIT
[65] %20https://choosealicense.com/licenses/mit/
[66] %20https://www.apache.org/licenses/LICENSE-2.0
[67] %20https://choosealicense.com/licenses/apache-2.0/
[68] %20https://informationisbeautiful.net/visualizations/million-lines-of-code/
[69] %20https://www.jetbrains.org/intellij/sdk/docs/reference_guide/custom_language_support.html

List of Listings

List of Figures

13 Build instructions

This document describes how to build the IntelliProlog.

13.1 Requirements

The requirements for building the plugin are:

- IntelliJ IDEA Community or Ultimate edition
- Plugin DevKit plugin for IntelliJ IDEA
- Grammar-Kit plugin for IntelliJ IDEA
- PsiViewer plugin for IntelliJ IDEA
- JDK for Java 8

13.2 Setting up IntelliJ IDEA

To be able to build the plugin the JDK for Java 8 and IntelliJ Platform SDK need to be configured in IDEA, the steps to configure these can be found at `https://www.jetbrains.org/intellij/sdk/docs/basics/getting_started/setting_up_environment.html`.

13.3 Importing project

Import the project using the `Open` button on the IDEA Welcome screen and choose the folder containing the source of the plugin.

13.4 Generating Lexer and Parser classes

Before being able to run or build the plugin, you need to generate the Parser and Lexer classes. This is done by selecting `Generate Parser Code` in the context menu for the `Prolog.bnf` file for the Parser. For the Lexer select `Run JFlex Generator` from the context menu of the `Prolog.flex`file.

13.5 Running and debugging

If your IDEA is properly set up, you can click the `Run` or `Debug` buttons in the toolbar or the same menu items in the `Run` menu. This will launch a new instance of IDEA with the plugin configured.

13.6 Build the plugin for deployment

To build the plugin make sure that you can run the plugin using the `Run` button.

Building the plugin for deployment is achieved by selecting `Prepare Plugin Module 'IntelliProlog' For Deployment` in the `Build` menu. This will create a jar in the source folder of the project, this can then be provided to people to install locally or uploaded to the JetBrains Plugins Repository[1].

[1] https://plugins.jetbrains.com/

14 Usage

14.1 Installing plugin

The plugin can be installed two ways, depending on if you have the plugin jar file.

14.1.1 Installing using jar file

If you have been provided with the jar file for the plugin, you can use the `Plugin` menu in the IDEA settings window. Click on `Install Plugin From Disk…` and select the jar file. Restart IDEA and the plugin is now ready to be used.

14.1.2 Installing from Plugins Repository

If you have not been provided with the jar file for the plugin or prefer to install the plugin from Jetbrains Plugin Repository, you can use the `Plugin` menu in the IDEA settings window. Click on `Browse Repositories…` and search for `IntelliProlog`, when found just click the `Install` button. Restart IDEA and the plugin is now ready to be used.

14.2 Creating a Prolog project

To create a new project for Prolog, simply click on `Create New Project` on the IDEA Welcome screen. Select the Prolog option from the project types, click `Next`, and then give your new project a name and define the location where it should created, and finally click 'Finish. You now have a Prolog project.

14.2.1 Setting the Prolog SDK (interpreter)

After your project is created, open the `Project Structure` window, select the `Project` menu option. Once your on the `Project` page, click on the `New` button and select the

`GNU-Prolog` option. The file chooser dialog will open at the default installation location on your system for `gprolog`. If you installed GNU Prolog in a different location, navigate to that location. Select the `gprolog` executable and finish by clicking the `Ok` button.

If GNU-Prolog is not automatically selected in the dropdown menu, click the dropdown menu and select GNU-Prolog.

14.3 Using the plugin

14.3.1 Creating a file

Create a file by using the `New` menu and select the `File` option, name it as you like, just make sure to add the `.pl` extension otherwise IDEA will not recognise it as a Prolog File.

14.3.2 Launching Prolog files

To launch a Prolog file, right click in the editor with a Prolog file open, there are 3 options to choose from, the options are:

- `Load <filename> in GNU Prolog interpreter`, this will launch the `gprolog` interpreter and load the selected file.
- `Load <filename> in GNU Prolog interpreter with trace`, this will launch the `gprolog` interpreter and load the selected file with trace enabled.
- `Run Prolog REPL`, this will launch the `gprolog` interpreter without loading any file.

All of the options set the working directory of the interpreter to the directory where the file is located.

14.3.3 Change the syntax highlighting colours

The plugin provides basic syntax highlighting and allows you to change the colours that are used.

To change the colours, open the IDEA Settings, and navigate to the `Prolog` option in `Editor/Color Scheme`. It is possible to change font colour, font weight and fore and background colours to your own liking.