# Intrepid IDE: A Workbench for Domain Specific Languages

June, 2016

**Final Report**

Submitted for the MEng
Computer Science

by

Daniel Scott Fleming (DSF)

Word Count : 14993 words

# Contents

# 1 Introduction

## 1.1 Context and Scope

Software engineering is defined as applying the ideologies of engineering to software development. One main difference is, professionals are designing intangible compositions and deliverables unlike the tangible outcomes of traditional engineering (Mills, 1980).

Many issues plague software engineering such as managing, scheduling, budgeting, and unforeseen bugs. This area of engineering has to take a holistic view of a project including the software's target architectures; end users technical literacy, and domain (Mills, 1980). These issues can result in spiralling development costs and software bottlenecking.

Software bottlenecking is the growing high demand for ever increasing in complexity software that outstrips developers. (Banker, 1987) One model of software development has been proposed as a macroeconomic production using inputs and producing products. This idea suggests that the software development process is coupled with the size and complexity of a number of environmental complexity factors. These include, difficulty in programming for hardware, engineering tool complexity and domain knowledge. Therefore to reduce effort required and by extension, cost of development, the main option is to either reduce the inputs or lower the effect of depended variables such as size, complexity and environmental complexity. Figure 1 shows that the environmental complexity is a software bottleneck, which this research aims to elevate.

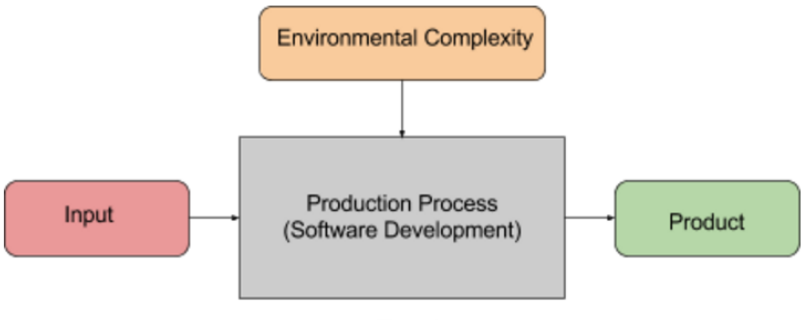



Figure 1: General production process model. (Banker and Datar, 1987).

An effective way to alleviate software bottlenecking in a development process is by using an IDE. IDEs are software that provides a development team with an environment to help write code more eciently. This is made possible through being able to design, create, write, error check, document, and compile their software in a single environment. (Walker, 2015). This contrasts with older methods of writing software such as punch cards; punch tape, and text documents (Tomsett, 2015).

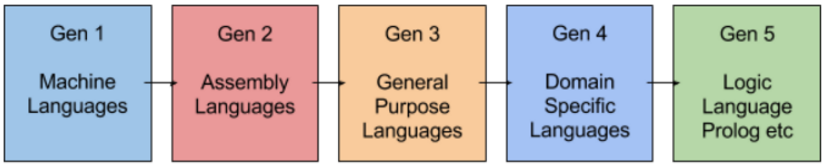



Figure 2: Example detailing the evolution of language generations.

Even though the initial project was to explore creating a new IDE for a GPL, the market for GPL IDEs containing features relevant to this project is highly saturated. It has less room for to

research and contribute towards, as shown in section 4.1. An alternative solution with the aim of reducing environmental complexity would be focusing on producing an IDE for DSLs as opposed to traditional GPLs, such as Java, C#, C++. Domain Specific Languages (DSLs) are an extra layer of abstraction intended to be closer to human readable code (Fowler and Parsons, 2010) IDEs for this domain (Language Workbenches) are still in an early research state, and has much more potential to explore than GPL IDEs. Furthermore, current early DSL IDEs are still plagued with problems of feature creeping. This introduces a significant learning curve for new developers, increasing environmental complexity. This project aims to develop a simple to use language workbench for External DSLs.

## 1.2 Initial Brief

### 1.2.1 Original Brief - Code editor with syntax highlighting and auto-complete

The initial brief involved producing an IDE including syntax highlighting and code completion for a specific GPL. Syntax highlighting displays different terms of text in various colours according to the different category of that term. Syntax highlighting is intended to aid the end user in reading, understanding and writing code. As such, it has no effect on the meaning of the text itself. This form of metadata on text is often referred as secondary notation. Lastly, code completion involves an interface to predict and suggest what the user may type next as they are writing code (Walker, 2015).

### 1.2.2 Revised brief - A Workbench for Domain Specific Languages

The initial brief has been changed as a result of further research into the domain of IDEs. Initial research has concluded that there's a saturation in the traditional IDE market, of which the feasibility of making a competitive IDE was small. Therefore, the proposal is now to research IDEs for external DSLs.

Programmers often avoid external DSLs due to the complexity involved in creating an ecosystem for them. This is exacerbated by existing general-purpose IDEs lack of expandability by offering syntax highlighting, code completion and error detection to support new DSLs. Language workbenches are an emerging software tool to develop DSLs. This research aims to explore this domain, as it is not as saturated as GPL IDEs. The outcome of this project is to develop a simple language workbench application, which involves the research, investigation, design and implementation of:

- A GUI to provide an environment to design and write a new language and to code using the new language.
- A limited end user GUI to provide an environment to write for an included DSL.
- Syntax Highlighting for the DSL implementation code editor
- Basic Code Completion for the DSL implementation code editor.
- Open Source the project to allow wider proliferation of this program.

## 2 Aims and Objectives

*Produce an external domain specific language workbench, with a code editor including syntax highlighting and code completion*

To achieve the aims and objectives stated, this research project's development will be based the spiral life-cycle model, with emphasis on rapid prototyping and iterative risk analysis.

- **Primary Objective 1:** Develop an architecture to produce a lexer and parser from a grammar DSL.
  - Produce a lexer and parser at runtime as opposed to ahead of time complication. This is a key objective, as this allows the user to be able to dene the language syntax, the program will build the lexer/parser. This will then provide a way to at runtime in the same instance parser a program in the language just dened. This provides a list of tokens from tokenization that is used for syntax highlighting and a parse tree.
- **Primary Objective 2:** Develop a GUI to allow the developer to write the DSL Grammar.
  - Including the ability for the grammar to be a context free grammar such as BNF or EBNF. Furthermore it must also allow the developer to code for the just defined language.
- **Primary Objective 3:** Develop a GUI to allow the developer and DSL end user to edit the DSL code, including syntax highlighting and code completion.
  - The user GUI includes a simpler GUI than the developer GUI. With the aim of making easy to use for people outside programming, as is a common use case for many DSLs
- **Primary Objective 4:** Provide an interface to define the colours associated with tokens for syntax highlighting.
  - Based on the non-terminals and terminals of language, allow a developer to set a custom colour for the DSL language text.
- **Primary Objective 5:** Research feasibility of an external DSL IDE and how it fits into the current market.
  - Understand the current marketplace and features sets that users are looking for in this domain.
- **Primary Objective 6:** Must be able to run on the Windows 7 x86/64 and above platforms
  - Windows has the largest market share out of any desktop operating systems, therefore this project must run on a Windows 7 and above.
- **Secondary Objective 1:** Provide Parsing Error information to users.
  - Developer information including incorrect or ambiguous context free grammar and user error information such as language syntax errors.
- **Secondary Objective 2:** Provide Git integration.

- Provide an interface to allow the developer to sync with a Git repository the grammar for their defined language.

- **Secondary Objective 3:** Syntax highlighting for the BNF grammar file.
  - Syntax highlighting to highlight terminals and non-terminals where appropriate for context free grammar used.

- **Secondary Objective 4:** Interface to allow developers to include DSL documentation.
  - Automatically integrated as part of the help section for the user's GUI for programming for the DSL.

- **Secondary Objective 5:** Ability to run program on multiple platforms
  - Have the ability to run the program in Mac OSx, Ubuntu and Windows

# 3 Background - Problem Context and Technologies

## 3.1 Programming languages

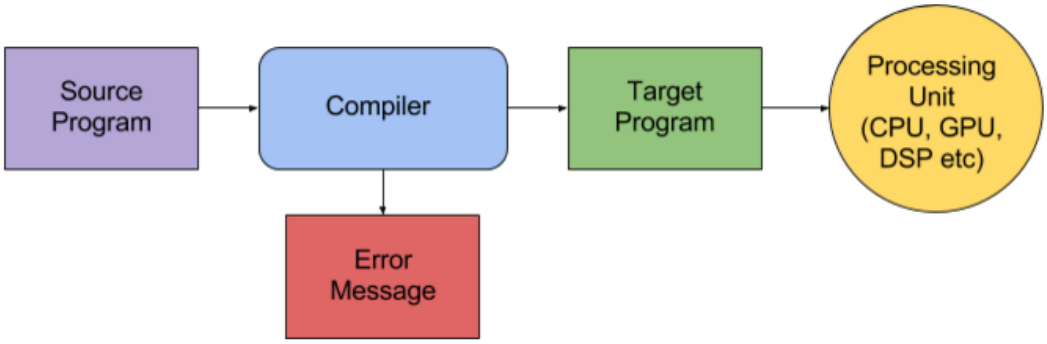



Figure 3: Example of a Compilation Model

A programming language allows communication from a human-readable form to a computer readable form. Programming languages are used to produce programs that manipulate the behavior and state of a computer. Furthermore, they often express algorithms. Programming language have evolved over the years and have many different abstraction levels. The lowest level is machine code. This executes directly on the Central Processing Unit (CPU), or more increasingly with processing pressure from big data and simulations, Graphics processing units(GPU's). ( Hawick, Leist and P Playne, 2010) GPU work under the same instruction, multiple data (SIMD) architectures exploited by higher level GPU compute oriented languages such as CUDA and OpenCL. (The Khronos Group, 2016) This approach contrasts with CPUs, which employ multiple instructions, multiple data architecture (MIMD). Higher levels of abstraction for languages have been created to elevate complexity and time needed to program. (WhatIs.com, 2016) Machine code is a language made up of instruction what are executed directly on the host CPU. Machine code was developed to allow easier coding of machine codes by naming the opcode is used by the instruction set of a CPU. Even then, programming in assembly is extremely complicated, and a detailed knowledge of a specific CPU microarchitecture and instruction set architecture (ISA) is required. The next evolution was general purpose languages such as C, Java. This language family allows a very high-level view of

writing programs. As it is constructed using ASCII logic rules, which are then used to generate an abstract syntax tree (AST) to then transverse to compile and produce machine code for a specific microarchitecture such as x86, AMD x64, MIPS and ARM v8. Different languages exist for various purposes such as:

- To define a system
- To specify a problem
- To deliver instructions
- To code for a specific processor architecture (CPUs vs GPUs)
- To express a problem more closely to the problem domain (Functional vs Logical vs Procedural paradigms)

### 3.2 Syntax and Semantics

The syntax of a language is the structure and sequence of symbols and is mainly dened using a form of language grammar. For example, a syntax rule in Java is that every type declaration must end with a semicolon. The example below illustrates a grammar rule written for ANTLR, to dene what a valid type declaration is. (GitHub, 2016)

```
TypeDeclaration
: classOrInterfaceModifier* classDeclaration
| classOrInterfaceModifier* enumDeclaration
| classOrInterfaceModifier* interfaceDeclaration
| classOrInterfaceModifier* annotationTypeDeclaration
| ';'
;
```

On the other hand semantics is the meaning of the symbols in a specified order, such as what specific words mean in the context of a sentence. This can make it challenging to integrate semantic understanding into a language grammar. In conclusion, a compiler needs to be concerned with both syntax and semantics when compiling a program, whilst a most programming language grammars, which are often context-free, are only concerned with the syntax of a language.

### 3.3 IDEs - Integrated Development Environment

IDEs have become the prominent method of writing code in the current programming field. This is partly due to them lowering the environmental complexity of writing code. This is achieved by providing tool such as syntax highlighting, code completion and debugging. The table below illustrates saturation of the market of IDEs, which support the features being researched for this project.

| **IDEs** | **Syntax Highlighting** | **Code Completion** | **Git** | **Supported Languages** |
|---|---|---|---|---|
| Code Blocks | Yes | Yes | Yes | C, C++ |
| Eclipse | Yes | Yes | Yes | C, C++, Python, Perl, Java, PHP, Ruby. |
| Code Lite | Yes | Yes | Yes | C++ |
| ShiftEdit | Yes | Yes | Yes | CSS, HTML, Java, Ruby, Perl, PHP, Python |
| Netbeans | Yes | Yes | Yes | C, C++, Java, Fortran |
| Visual Studio | Yes | Yes | Yes | C++, C# |
| Intellij | Yes | Yes | Yes | Java, Scala |

#### 3.3.1 IDE Advantages

Advantages of using IDEs include:

- Tracking and providing views for errors, warning, tests, files related to development, code all in the same software. (Atlanta and profile, 2009)
- Collating the compiler, editor, debugger, source control, execution into the same environment. Unlike previous methods of writing code.
- The environment will know the semantics and rules of a language and therefore be able to provide extra Meta data about the code, such as syntax highlighting.
- They can provide an interface to help enforce house coding standards for development companies which will often have their own coding style.

#### 3.3.2 IDE Disadvantages

Disadvantages of IDEs include:

- Feature creeping, from the nature of GPL IDE, they can become verbose in feature sets. This extends the potential use case of the IDEs but can also make it more difficult and daunting for programmers to navigate as well as potentially adding bugs. Example Eclipse.

- Steep learning curve for projects that may involve multiple GPLs, as IDEs tend to only allow single languages to be developed and as a result learning numerous advanced software tools can be time consuming, adding to the environmental complexity.

### 3.4 DSLs - Domain specific languages

Domain specific languages (DSLs) are smaller languages targeted at a specific domain; they are a way of controlling an abstraction. Their aim is to encapsulate the concepts and vocabulary for applications residing in a specific area. They fall into two categories, internal and external DSLs. Internal DSLs use the host programming language to create a semantic model. This is opposed to external DSLs that allows a custom syntax but with the drawbacks of building a parser, maintaining the language and repeating functionality of GPLs and their ecosystems such as IDEs (Fowler and Parsons, 2010).

Internal DSLs have the advantage that they can use features of existing IDEs, due to them being based on a host language. Therefore, they offer the user Syntax highlighting and code completion. This is also a large disadvantage in that they focus on a particular domain, often using alternative command query type interfaces, such as fluent interfaces and function (Fowler, and Parsons, 2010) sequence design patterns to propagate a systematic framework. These patterns often go against traditional design styles that IDEs code completion predicts for, and as a result may offer the user none domain specific options, featuring more GPL patterns. Internal DSLs often employ two different approaches, one is reductionist, where the host languages features are limited and are not exposed to the end user. On the other hand, an extension approach allows all the features of the host language available to the user. This is illustrated in figure 4.

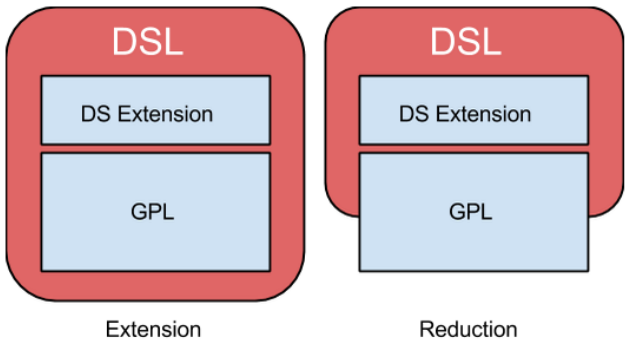



Figure 4: Detailing differences of Reduction and Extension internal DSLs (Husselmann, 2014).

One the other hand External DSLs have the advantage that they are often more domain specific, from the feature of the developer being able to control the whole semantics and design of the language. Therefore, it can be more closely tailored to a domain. This has the adverse effect of complexity through having to create and maintain an infrastructure for the language. Through extension, they also can't take advantage of IDEs, from being a custom syntax and as a result syntax highlighting and code completion. For this project, focusing on external DSLs will be undertaken. This is due to environments needed to code for external DSLs being scarce. Furthermore, a reduction approach internal DSL, often suffer from the same issues as external DSLs. One of the main ways to limiting the use of the host language is to create a custom environment which only provides access to the necessary parts of the host language features. This entails all the overhead involved in creating the environment as for an external DSL. Therefore, this project could be used

to create an environment for an external DSL as well a reduction based internal DSL. For example, a reduction DSL could be programmed in Java, therefore in the proposed project, grammar could be constructed that only forms valid Java code, but does not implement any grammar for the Java features not utilized in the DSL. Therefore, the result is that the DSL is using a host language, but limiting the possible interaction of DSL user and the underlying Java code.

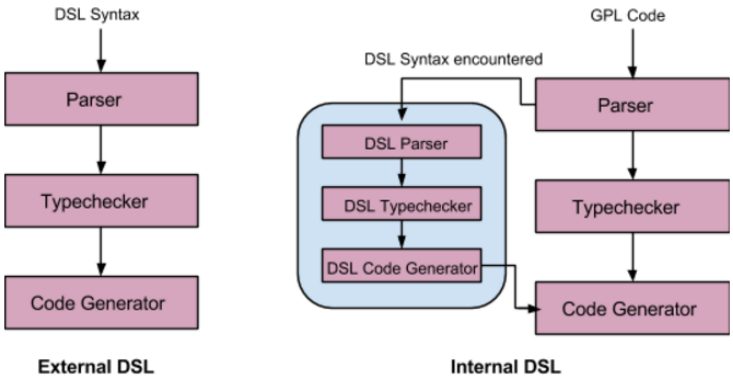



Figure 5: Detailing differences of internal and external DSLs (Husselmann, 2014).

### 3.5 Language Workbenches

Language workbenches is a name coined by Martin Fowler to describe a new type of software tool intended to address shortcomings of DSLs. This is a new area of research, but these tools are essentially IDEs; DSLs. The definition Fowler provides also includes possible features such as (Fowler, 2005):

- Freely define new languages which are integrated.
- Information is a persistent abstract representation.
- They can contain three main parts: schema, editor(s), and generator(s).

The table below shows a selection of the current DSL language workbenches, for this research it was found that there are very few products on the market.

| **Language Workbench Name** | **Syntax Highlighting** | **Code Completion** | **Multiple External DSLs** | **Git** | **Released** |
|---|---|---|---|---|---|
| Intentional Software | Unknown (Not Released) | Unknown (Not Released) | Yes | Unknown (Not Released) | Still in development |
| Meta Programming system | Yes | Yes | Yes | Yes | Yes |
| Rascal (van der Storm, T, 2011) | Yes | No | Yes | Yes | Yes |
| Xtext | Yes | Yes | Yes | No | Yes |

### 3.6 Syntax Highlighting

Syntax Highlighting is a feature of many code editors and are used to display source code in different colours to highlight the various categories of expressions (D'Anjou and Shavor, 2005). Syntax Highlighting is a form of secondary notation. This term encapsulates textural context secondary notation, such as syntax highlighting, indentation, boldness, italics and spacing. Furthermore, secondary notation includes graphical contexts, such as flowcharts. The intended result of this is to improve the programmer's ability to efficiently distinguish key elements of a language and architecture.

```
outputScrollPane1.setViewportView(parseOutputLable);
editorTitleLable.setFont(new java.awt.Font("Tahoma", 0, 18));
editorTitleLable.setText("Output");

outputTitleLable.setFont(new java.awt.Font("Tahoma", 0, 18));
outputTitleLable.setText("Editor");
```

Figure 6: Example of Java syntax highlighting in Intellij.

One advantage of syntax highlighting is that it allows developers to easily detect syntactic errors. For example, in figure 6, the user has started a method call with a ' " ', which is syntactically incorrect, as that indicated the beginning of a string literal. Through syntax highlighting, even in verbose code, this error can be easily seen.

```
GroupLayout layout = new GroupLayout(getContentPane());
getContentPane().setLayout(layout);
layout.setHorizontalGroup(
        layout.createParallelGroup(GroupLayout.Alignment.LEADING)
                .addGroup(layout.createSequentialGroup()
                        .addContainerGap()
                        .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)
                                .addComponent(outputScrollPane1, GroupLayout.Alignment.TRAILING)
                                ."addComponent(OutputScrollPane1)
                                .addGroup(layout.createSequentialGroup()
                                        .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)
                                                .addComponent(jButton2, GroupLayout.PREFERRED_SIZE, 161, GroupLayout.PREFERRED_SIZ
                                                .addComponent(editorTitleLable)
                                                .addComponent(outputTitleLable))
                                        .addGap(0, 492, Short.MAX_VALUE)))
                        .addContainerGap())
);
```

Figure 7: Example of a syntax error being reported in Intellij through use of syntax highlighting.

### 3.7 Code Completion

Code completion is another advantage of IDEs. It provides predictions to developers that suggests the next valid statements for a string of code. This can allow faster access to in scope variables and methods etc. Code completion also allows the user to explore tools and features available such as class and library variables, methods and data types.

Code Completion advantage of allowing the developer to potentially code faster as well as it being less error prone. For example, if the user wants to use a method that they declared, which they may not remember the full name, they can start typing and it should be suggested. Furthermore, this allows the user to explore the tools and features available to them such as variables, methods, and data types in, for example, a library. This can often save time due to the negation of having to search the API documentation for a description, although this can have the adverse effect for thinking a method for example, may do an action indicated by its name that it may not do what the developer expects, which would potentially be explained in the documentation.

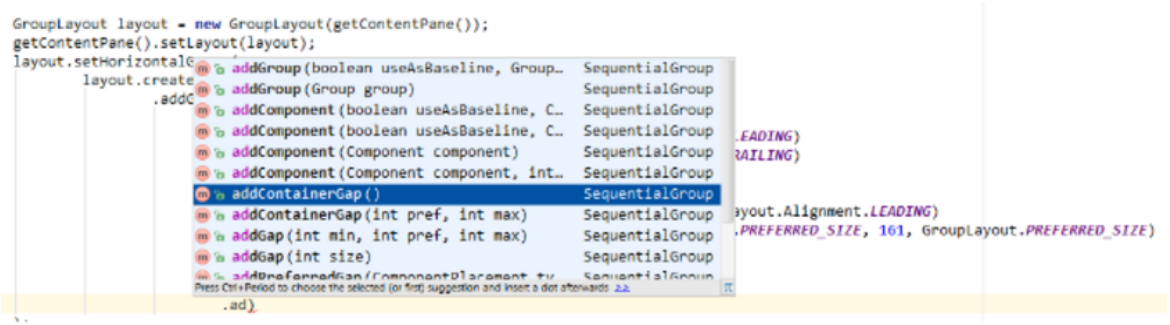



Figure 8: Example of code completion in Intellij.

### 3.8 Lexer

Lexers perform lexical analysis which involves dividing the input into tokens, which are meaningful chunks. Lexical Analysis is the first phase of completion, it accepts a sequence of characters which are partitioned depending on the language specified. The terms that build the language are referred to as lexemes and are often defined using regular expressions. The lexer during lexical analysis identifies the lexemes inputted and replaces them with tokens which act as a unique identifier. During this phase, values of tokens are stored in a symbol table, to ensure the integrity of meaning of the program. The symbol table is accessible by later phases of compilation, to access the data for a specific token (Tomsett, 2015). For example:

$$\mathbf{a = b + c;}$$

The lexer will divide this up into the tokens, with unique identifiers. For the given example, this would produce tokens of a, equal sign, b, plus sign and c. This token stream is then passed to the parser which then works out that b+c is an expression assigned to a (Levine and Associates 2009) .

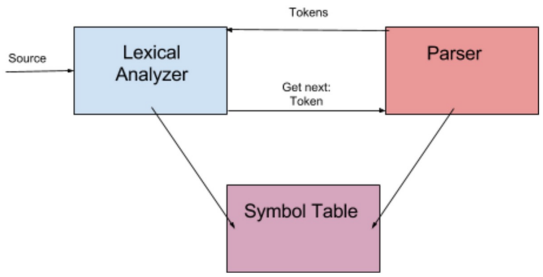



Figure 9: Example showing model of a lexer and parser

### 3.9 Parser

A parser tries to understand how tokens relate to each other. This is important for error checking as it can often find when invalid tokens are in the place where other tokens were expected. Indicating a syntax error in code. The second phase of compilation involves parsing the output of tokens from the lexer. Using the rules of a language, defined by a form of language grammar, the parser

checks the syntactic legality of a stream of tokens. The grammar of a programming language if often defined using Backus-Naur Form (BNF) notation (Tomsett, 2015).

## 3.10 Syntactic Metalanguages

A syntactic meta language is a notation for definition syntax of a language by a number of rules and brings orders to the formal definition of as syntax. A syntactic metalanguage defines the syntax of a language through sets of rules. Each rule names part of the language (called a non-terminal symbol of the language) and then defines its possible forms. Programming languages are constructed using a combination of semantics and language syntax. The semantics provide meaning of every rule that is possible in the language and the syntax provides its structure. A syntactic metalanguage grammar allows the possibility to transform syntactically correct program code, which is often written in ASCII characters, into a abstract syntax tree.

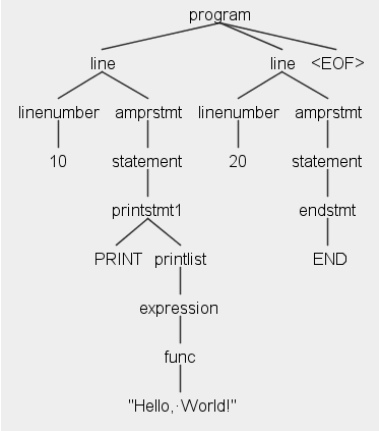



Figure 10: Example of a generated abstract syntax tree from a simple ANTLR BBC BASIC EBNF Grammar

The generated tree is the data structure on which a compiler or interpreter will transverse in order to process the program. This often results in producing machine code and also allow type checking and syntax error checking. Furthermore, it allows the possibility to peek ahead at the next valid nodes on the tree. This is useful in a situation where a statement is not complete, for example when dynamically compiling as a user types code to provide suggestions on the next valid options. Also though traversing the tree, an interpreter can then simulate executing the program. Research conducted over the last few decades have produced different technologies to produce grammars, but the prevalent technology is called context-free grammars. (Meduna, 2014) According to the Chomsky hierarchy (Anon, 2016), language grammar can be categorized into four different levels. (Wikipedia, 2016)

### 3.10.1 Type 0: Unrestricted Grammar

Unrestricted Grammar is a very difficult to automate, but very powerful type of grammar. Although due to its infinite possibilities, it is impossible to prove its completeness. This is a form of a formal grammar which have no restrictions on the left and right side of the grammar rules. (Wikipedia, 2016) This is the most general form of grammar in the Chomosky hierarchy and can generate every type of language possible. This has the result that it is impossible to create a compiler for it, as it would have to account for infinite use cases.

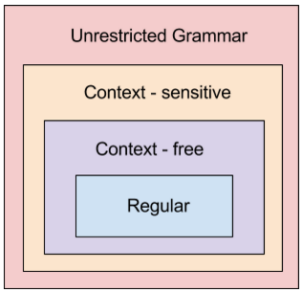



Figure 11: Chomsky Hierarchy of Language grammar

### 3.10.2 Type 1: Context-sensitive grammar

Context-sensitive grammar is what more natural language and programming languages fall into, where the context of the language is necessary. The grammar is constructed with the left and right-hand sides of the production rules contained by a context of terminal and nonterminal symbols. It is possible to use a formal grammar to define a context sensitive grammar. These are called context-sensitive languages. Although due to the more general nature of this form of grammar, as compared to a context-free grammar. It can be very difficult to produce a compiler, but due to some language features not being context free, intermediate mildly context-sensitive languages are often used, such as in processing for natural languages. For programming languages that utilize context-sensitive features, it is often possible to add context sensitive information during the parsing of a context-free grammar.

### 3.10.3 Type 2: Context-free

Context-free grammar are more restricted than the lower level of grammars and tend to be easier to automate and parse. (Seas.upenn.edu, 2016) This makes it especially useful for using in compilers. These forms of grammars use more complex techniques such as recursion and a wider range of syntax. Furthermore they are often defined using a form of BNF or EBNF. Although not all languages can be defined in context-free fashion, as some features require an understanding of context. (Anon, 2016) For example, an EBNF grammar can be constructed using the following syntax:

- Definition : =
- Concatenation : ,
- Termination : ;
- Alternation : |
- Optional : [ ]
- Repetition : { }
- Grouping ( ... )
- Terminal String : ” ” or ’ ’
- Special Sequence ? ... ?
- Comment (* ... *)

### 3.10.4 Type 3: Regular grammar

Regular grammar include no recursion and can be useful for defining words and symbols. As such, it is easy to process maniacally, for example through a finite state machine. Type three grammars can be represented using regular expressions. Regular expressions can consist of:

- Repetition : *, +
- Options : ?, |
- Ranges : [ a-zA-Z ]

## 3.11 Grammar Constructs

Grammar can consist of four main parts, terminals, non-terminals, production rules and a starting symbol. Furthermore this form of grammar only describes the rules of a language, such as what strings and rules of those strings are valid, therefore the grammar has no understanding of the meaning of the strings.

$$G = \{T,N,P,S\}$$

### 3.11.1 Terminals

Terminals are final symbols in a language, such as words. In a programming language context, this would be keywords, identifiers, and numbers.

### 3.11.2 Non-Terminals

Non-terminals are an intermediate representation of a rule. For example, a statement or expression in a programming language would be a non-terminal.

### 3.11.3 Production Rules

Production rules are a form of rewrite rules where the left and side acts as a name that can be used to substitute the rule defined on its right hand side. (GitHub, 2016)

```
classOrInterfaceModifier
: annotation // class or interface
| ( 'public' // class or interface
| 'protected' // class or interface
| 'private' // class or interface
| 'static' // class or interface
| 'abstract' // class or interface
| 'final' // class only – does not apply to interfaces
| 'strictfp' // class or interface
)
;
```

The code above defines a production rule showing what a valid class or interface modifier is in the Java, this is written in EBNF grammar for ANTLR. It states that a class or interface modifier must be made up of a terminal of annotation, followed by one of the following terminals, which are encapsulated in single quotes. Multiple of these production rules come together to define what a valid language can be.

## 3.12 Software Design Methodologies

The spiral model for software development focuses on cyclic concurrent development with an emphasis on risk-driven development (SearchSoftwareQuality, 2015). The aim is to base the current cycle of work on the amount of risk is involved in development. This can help lower overall risk of developing the project as it allows elimination of unachievable goals early on. This style of approach can take on other models, due to its risk analysis focus. Therefore, this project will be focusing on an evolutionary prototyping model, which is a subset of the spiral model.

## 3.13 Java

For this project, a language is needed that is multi-platform. As such the main contender for this is Java, as it runs in a virtual machine. Java is one of the biggest languages in the word, it is a GPL and computing platform which was developed by James Gosling and released by Sun Microsystems in 1995. (Java.com, 2016) The language is concurrent, class and object orientation based and is designed to have as few implementation dependencies as possible. Once compiled, the Java code can run on all platforms that support the Java virtual machine (JVM), without needing to be recompiled. (Oracle.com, 2016) This is due to Java being compiled to a intermediary language called bytecode, which is then run on the JVM for the system it's being run on. For more information please see figure 12. (Safari, 2016) Java is intended to be a high-level language with heavy abstractions to aid in faster development, as such even though its syntax is based on C, it lacks the lower level facilities.

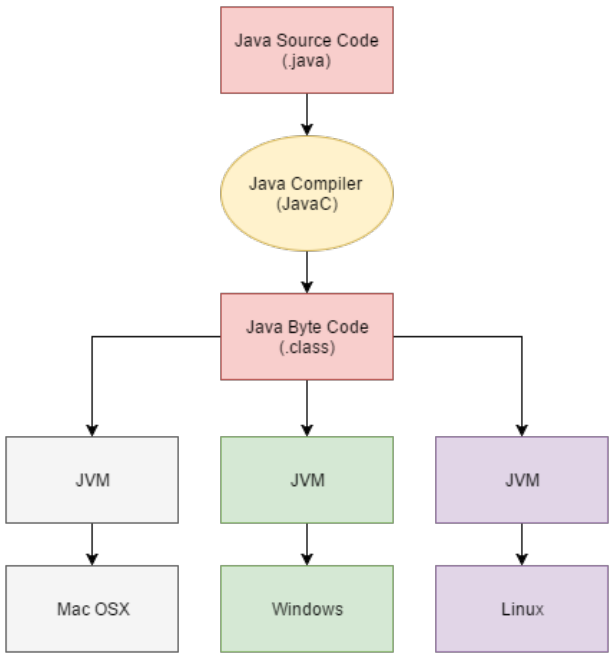



Figure 12: Abstract view of Java's compilation and execution

### 3.13.1 Reflection

Reflection is a advanced language tool often used by programs which require the ability to modify, examine the runtime behaviour of programs running in the JVM. Reflection is an extremely powerful

tool but has many disadvantages as it allows a program to perform operations what wouldn't be possible with normal execution. (Docs.oracle.com, 2016) Some of the disadvantages include:

- **Performance Overhead**
  - Due to the way reflection produces types that are dynamically resolved, there are many JVM optimizations which can't be performed. This has the effect that reflective operations have significantly slower performance than normal operations.
- **Security Restrictions**
  - Reflection requires extra runtime permissions. In certain situations, these permissions may not be present when running under a security manager.
- **Exposure of Internals**
  - Reflection enables code to perform operations that would be illegal on non-reflective code. For example, accessing private method or field is allowed in reflection, but this can result in unexpected side effects. This can often have the effect of rendering code dysfunctional and may destroy its portability. The reflective code breaks standard abstractions and can therefore change it's behaviour with upgrades to the platform.

## 3.14 Graphical APIs

Graphical APIs are important as it is the framework that the program will run on. It's important to choose a API that is the developer is familiar with, indication too having the flexibility to meet the demands of this project. The two main APIs used in the Java ecosystem are Swing and JavaFX and are further detailed below.

### 3.14.1 Swing

Swing was designed to offer more customization and flexibility that its predecessor AWT and its architecture is built around MVC. (Oracle.com, 2016) The platform, however, hasn't evolved in a while and won't continue to be developed, despite its large user base.

### 3.14.2 JavaFX

JavaFX is designed to be a multi-device based GUI toolkit, able to work with a large number of devices and form factors. A large disadvantage of JavaFX over Swing is that it's a much newer platform, meaning developer support and documentation isn't as robust. On the other hand, JavaFX advantages over Swing include (Anderson and Anderson, 2016):

- Better support for animations
- Styled using CSS
- Scenes can be defined using FXML
- Uses a more optimized hardware accelerated graphics pipeline to enhance UI performance

For this project, JavaFX will be used as the GUI tool-kit. This is due to the higher GUI performance of JavaFX, and it's much better support for animations and styles. These are essential advantages when building an IDE with performance intensive real time code completion and syntax highlighting.

### 3.15 Alternative solutions

An automatic approach based parser generator will be required for this project, due to the dynamic language support needed. This makes alternative solutions such as hand writing a parser generator less viable, as the complexity needed would mean too much time on the project would be spent developing it, with no guarantee of a working project. There are currently many candidates for this project, such as Flex and Bison (Levine and Associates, 2009), Byacc and ANTLR (Another Tool for Language Recognition). The table below shows a more in-depth look at possible candidates for this project.

| **Parser Generators** | **Parsing Algorithm** | **Input Grammar** | **Lexer** | **Development Platform** |
|---|---|---|---|---|
| JavaCC (Javacc.java.net, 2016) | LALR | EBNF | Generated | JVM |
| Bison (Gnu.org, 2016) | LR | BNF like CFG | External | All |
| ANTLR4 (Antlr.org, 2016) | LL | EBNF | Generated | JVM |
| CUP (www2.in.tum.de, 2016) | LALR | BNF | External | JVM |
| Beaver (Beaver.sourceforge.net, 2016) | LALR | EBNF | External | JVM |
| Byacc (Invisible-island.net, 2016) | LALR | YACC | External | All |
| Coco/R (Ssw.uni-linz.ac.at, 2016) | LL | EBNF | Generated | .Net, JVM |

As shown in the table above, they're many options for use in the project. A key requirement is that it runs on the JVM, as the project is aimed at being cross compatible with OSs. Furthermore, a generated lexer is necessary to reduce development effort. Also, using a standard context-free grammar such as EBNF is preferred. As such, ANTLR was chosen for this project from meeting all the requirements. Even though JavaCC also meets these requirements, it was found that ANTLR had better documentation and online support, which is important for a time sensitive project such as this.

Antler is a parser generator which uses LL(*) parsing (Bearcave.com, 2015). Using a Grammar DSL to define a language, it generates a recognizer for that language. ANTLR is capable of generating lexers, parsers and tree parsers (Antlr.org, 2015). ANTLR uses use a single EBNF notation to specify lexers, parsers and tree parsers, which contrasts with other solutions. The simplicity of use is an important part of getting developers to adopt new software. This single notation makes ANTLR much easier to use by developers, and as such suits itself well to this project. Furthermore, this project aims to target a wide range potential users and target architectures. Therefore to allow cross compatibility, a JVM-based parser generator will be needed, such as ANTLR. Finally, IntelliJ IDEA and Clion IDEs use ANTLR for their syntax highlighting and code completion.

# 4 Technical Development

## 4.1 System Design

### 4.1.1 System Use Case Diagram

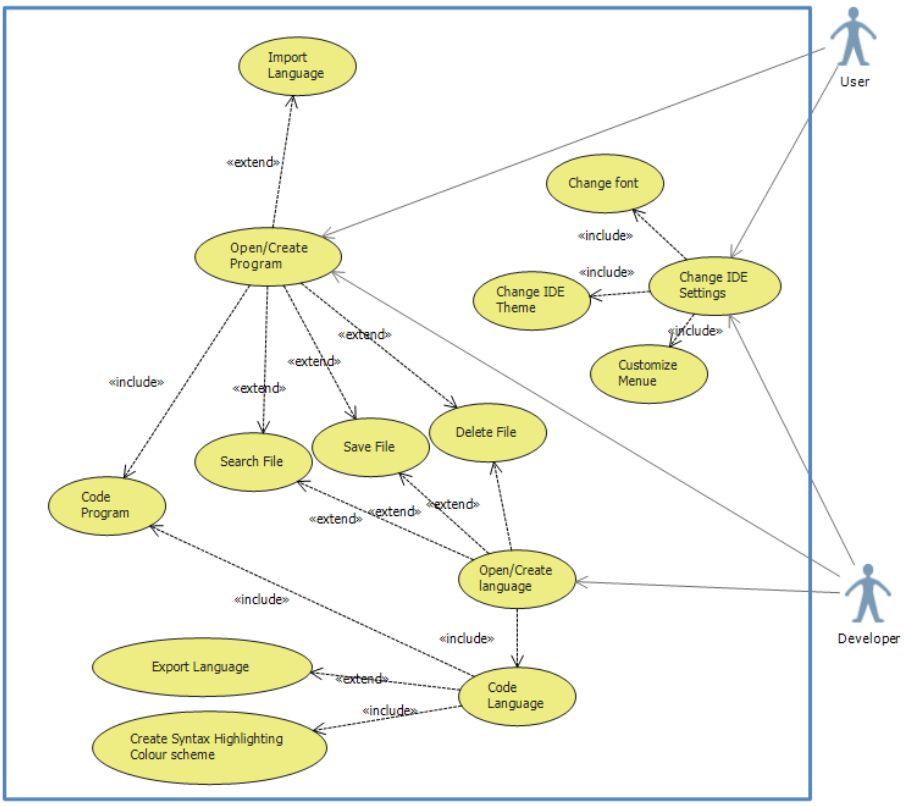



Figure 13: Use Case Digram of system

Figure 13 illustrates a use case diagram for this project. It shows that system has two users and their roles. The diagram illustrates the separation of functionality between the two types of users for this project. The developer actor has access to a complex set of tools to implement a language's grammar, as well as the ability to program for it. On the other hand, the user actor only can import a language and code for it. The system shows the development of this project has to take into account the functionality provided to each user, and as such will need a separation of the user interface to hide functionality from the user actor.

The main focus of this project is to have a seamless environment for a user to be able to write code for an external DSL in an easy way. This means providing some of the benefits of an internal DSL, such as an environment that provides auto-completion and syntax highlighting. Users of a DSL are often not programmers. Therefore, it's important to have a straightforward and easy to use program. Figure 13 shows that the user actor has a limited set of roles in this system, which shows that this design goal has been considered in the system. The second focuses are for a developer to be able to design a small DSL, distribute and test it. The diagram shows that the developer has control over most of the system, which helps demonstrate that design goal has also been considered in the system.

### 4.1.2 Asynchronous Callback Pattern

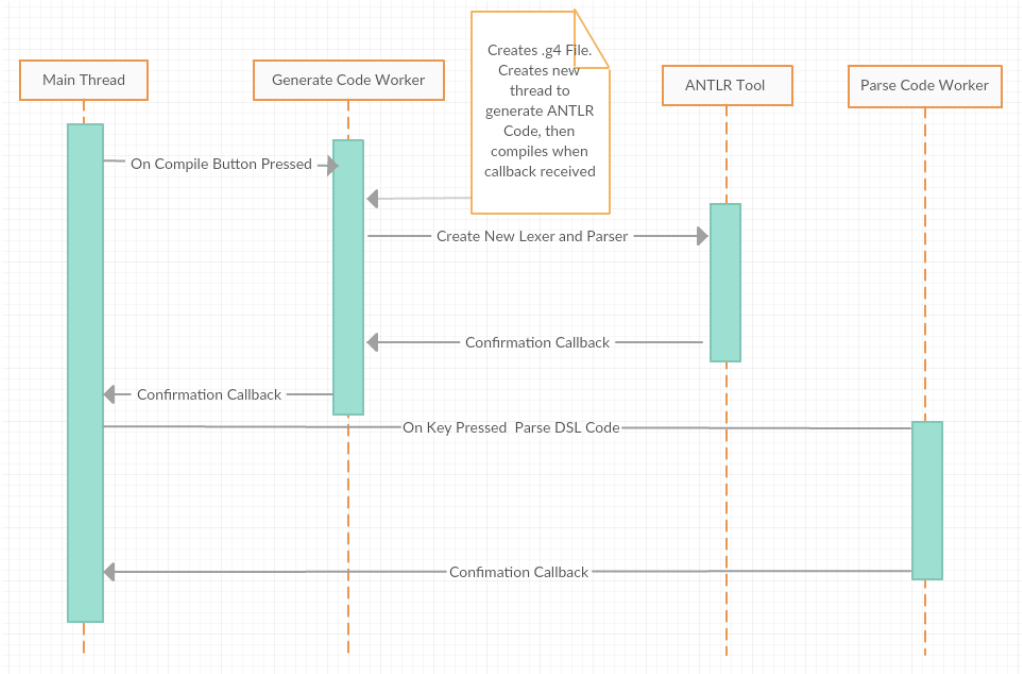



Figure 14: Sequence Digram of threading used for generating code

Figure 14 shows a sequence diagram shows the architecture of the lexer and parser generation. The diagram indicates that the main thread will create a new thread with a callback mechanism which accepts the EBNF text and will the produce a .G4 file, after which the ANTLR tool is used which is encapsulated in its own thread. Upon completion of the ANTLR thread the generate code worker thread will proceed to compile the generated code. Upon completion, the main thread is notified that the task is complete. The callback pattern is utilized numerous times in this project to prevent pausing of the UI thread in the application when a long running task is happening. With the callback architecture, it allows the UI thread to continue to run, and when the thread completes, the UI thread is notified. This allows the UI thread to execute the necessary code and then to update the UI and show the results of the operation. This separation of code on threads is also reflected in the code, due to the threading logic being independent of the business logic, allowing a more layered architecture. This has the benefits of allowing lower coupling between elements in the program. This pattern is also implemented for parsing of code, to ensure the UI thread is not slowed down as the user is typing.

For this design, Java 8's Asynchronous CompletableFuture feature will be utilized. A CompletableFuture is a convenient way to allow callbacks upon completion of the task, including updates on completion progress which is used to notify the user about completion. The worker threads used are executed asynchronously, with callbacks.

### 4.1.3 System Architecture

Using the system specification, the programs class structure and objects that will be needed for the project can be determined. The following elements will be necessary to ensure the program meets the specification.

- For the file management specification, a class can be constructed encapsulating the methods and objects needed to handle input and output of project files, as well as any persistent data in the project. This encapsulation of I/O will ensure minimum repeated code in the program as I/O operations will be common in a project such as what's defined in the aims and objectives.
- A code editor window will be required for the project, with syntax highlighting.
  - A lexer generated at runtime based on the EBNF rules defined in a project will need to be generated. The lexer is provided a block of text and then tokenizes the input. This will return a list of token objects containing the positions in the passed text the tokens are and the type of each token. This will be used to get the information needed for syntax highlighting.
  - A parser generated at run time based on the EBNF rules defined in a project will build an AST, which will be traversed using the tokens provided for by the lexer. The parser will be used to get the syntax errors in the passed text, to then be outputted to the user.
  - The standard tools used for building a robust lexer and parser all use ahead of time compilation, as discovered in the background research for this project. Therefore, the generated un-compiled lexer and parser classes from these tools will need to be compiled and then loaded into the program using reflection as the program is running.
  - Using the information provided for by the lexer and parser, a code complete object can give the user code suggestions

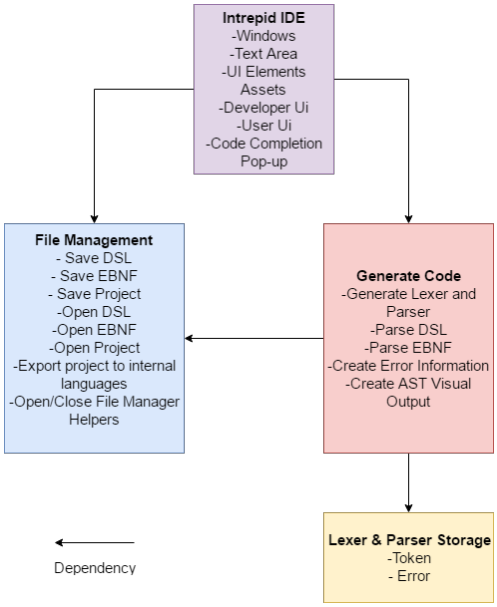



Figure 15: Diagram showing the basic architecture design of Intrepid

As shown in figure 15, to aid in structuring the project, a layered approach encapsulating code according to functionality will be used. Dividing the code into logical sections and following a high cohesion design allows lower coupling of the projects classes, and allows easier editing, maintaining and extending of features.

The file management class contains all the methods required to access code on the devices file system, including the functionality to save projects and DSL code in custom formatted files. These were separated from being coupled to other classes, as the code has been used through the system, as file operations are common for IDEs. Therefore, it was decided to separate the of the classes is imperative, as it allows maximum code reuse.

The Intrepid IDE class contains all the business logic of the project, and is the part of code directly accessed by the user, therefore, this code will be encapsulated as it is conceptually highly linked.

The Generate code classes contain all the method to access and manage the code generation and import part of this project; it was decided to separate this code as it is unique compared to the rest of the project. Also, due to the nature of this code, it may be necessary to focus on improving this section, and lowering the coupling of this code to the rest of the program allows easier updating without affecting the rest of the program.

Finally the Lexer and Parser storage is a collection of storage information of the last parse, this was decided due to the generating of code having multiple temporary storage lists that will often be changed. Having runtime persistent storage of parse history information is useful for code completion and other features.

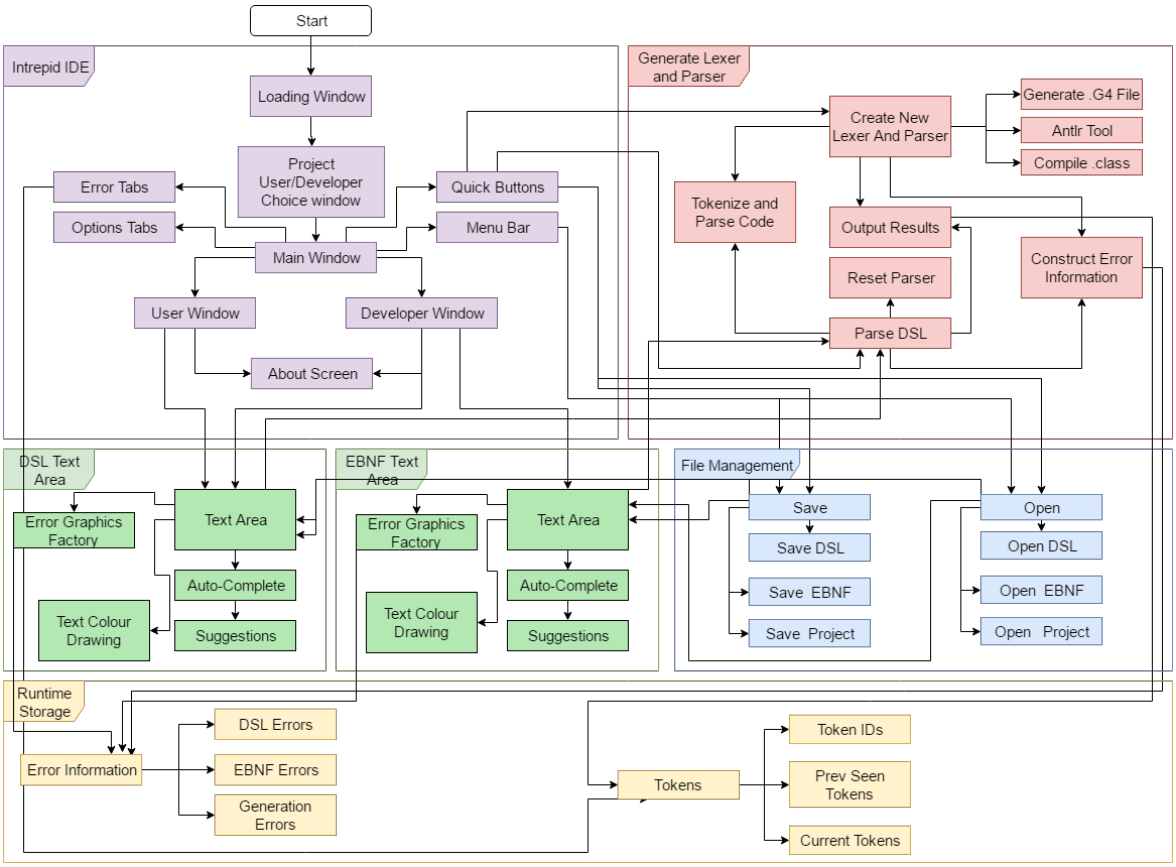



Figure 16: Diagram of a UML code map of Intrepid

Figure 16 shows a code map of Intrepid, which shows a more detailed overview of the relationships between components in the project. It further elaborates on the distinctions between classes shown in figure 15. The errors show the relationship of what feature use other features. Finally, the start is the entry point of the program and where the user will begin. The diagram shows separation of code and that the user only has direct access to the Intrepid IDE UI section of the program. This allows a more modular design, where the separate modules can be updated and modified without affecting the rest of the program. This is due to the high cohesion and low coupling design of the program.

### 4.2 System Specification and requirements

The system specification is intended to identify the main requirements of the software produced during this project. These are constructed using the project's aims and objectives. Furthermore, this list will be utilized in testing the program against, to ensure the specification has been met, as this specification illustrates what actions the end user should be able to do in the program.

- File Management
  - Open Existing language project
  - Open Existing DSL project
  - Create new language project
  - Create new DSL project
  - Save language project
  - Save DSL project
- Syntax Highlighting
  - View DSL code with syntax highlighting in user and developer modes.
  - Change token syntax highlighting colours for a developer language project
  - Real time drawing of colour one valid token has been typed
- Code Completion
  - Display pop-up with useful code suggestions as the developer or user types.
  - Ability to click a suggested text to insert and automatically syntax highlight suggestion if appropriate.
  - Pop-up should follow the users current typing position for fast access.
- Edit Source Code
  - Modify text
  - Cut/ Copy/ Paste text
- Compile Language grammar
- Export project to list of included DSLs for user coding
- Output syntactic and semantic error is DSL code in the DSL errors tab

- Customisation
  - Change the font of code text.
  - Change the size of code text.
  - Allow developer to customize what tokens should be suggesting in auto complete pop-up
  - Allow developer to customize what colours for tokens should be

## 4.3 UI Design

There is a large emphasis on UI design for this project, as it is meant to be a simple and easy to use DSL language workbench. The overall design of the project will be based on a modern flat design, whilst avoiding older Skeuomorphic design principles. Although material design guidelines will also be used, such as using depth as a tool to aid the user to build a spatial model for actions and features (Google design guidelines, 2016). This is intended to aid new users in quickly understanding how to use and navigate the software. Furthermore, pastel colours and an option of a light and dark theme will be included. As shown in the use case diagrams, the system will need two user interfaces for each actor.

### 4.3.1 Initial GUI Concept

Figure 17 shows an initial design for the project; this was constructed during the first cycle of the design development integrating familiar concepts seen in IDEs as indicated in the background section. The design interoperates a cascading menu bar, with the simple placement of objects such as file and edit. The design also interoperates separate tabs for the EBNF and DSL implementation textarea. During the review phase of this cycle, it was found that there was not enough vertical coding space and that a lot of the horizontal space was wasted in this design.

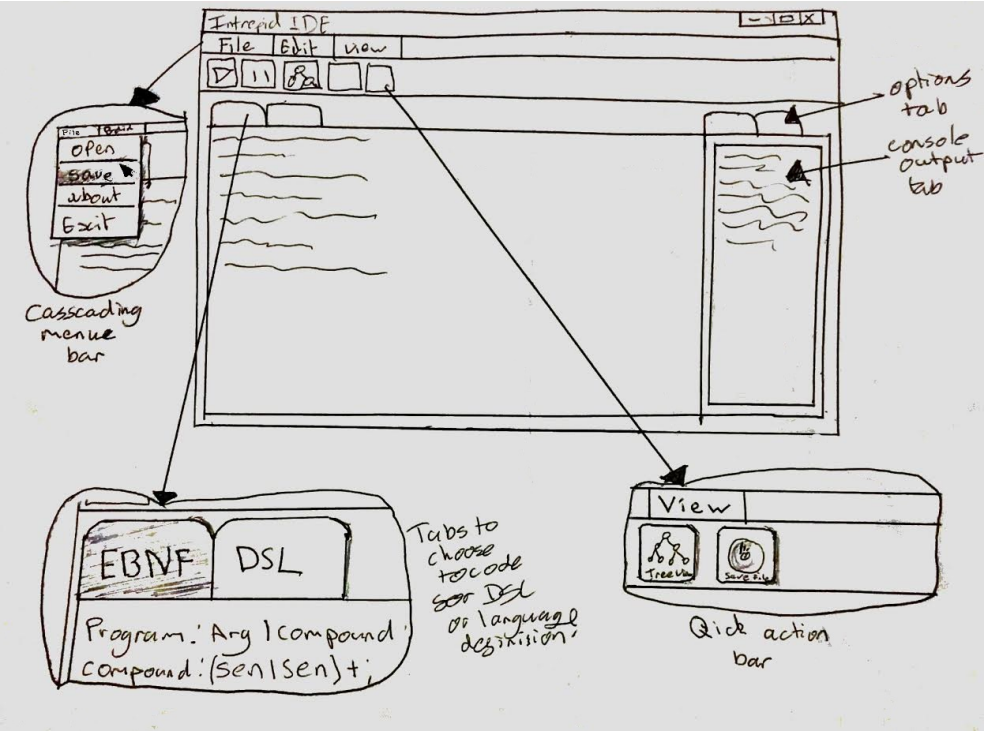



Figure 17: Initial concept drawing for the GUI

### 4.3.2 Concept User GUI

Figure 18 illustrates a concept of the users GUI; it is designed to be as non-threatening and as simple to use as possible. Therefore to avoid visual feature overload, there is a quick and easy to use sidebar that has large colourful options that the user may need to use often. These are using pastel colours and a circular design to stand out from the rest of the program. The design also includes a sidebar on the right-hand side to provide tips and possible help to the user. The main focus of this design to make the user clearly see where the important parts of the program are. As such the largest element is the programming text area, with colour coded tabs above it.

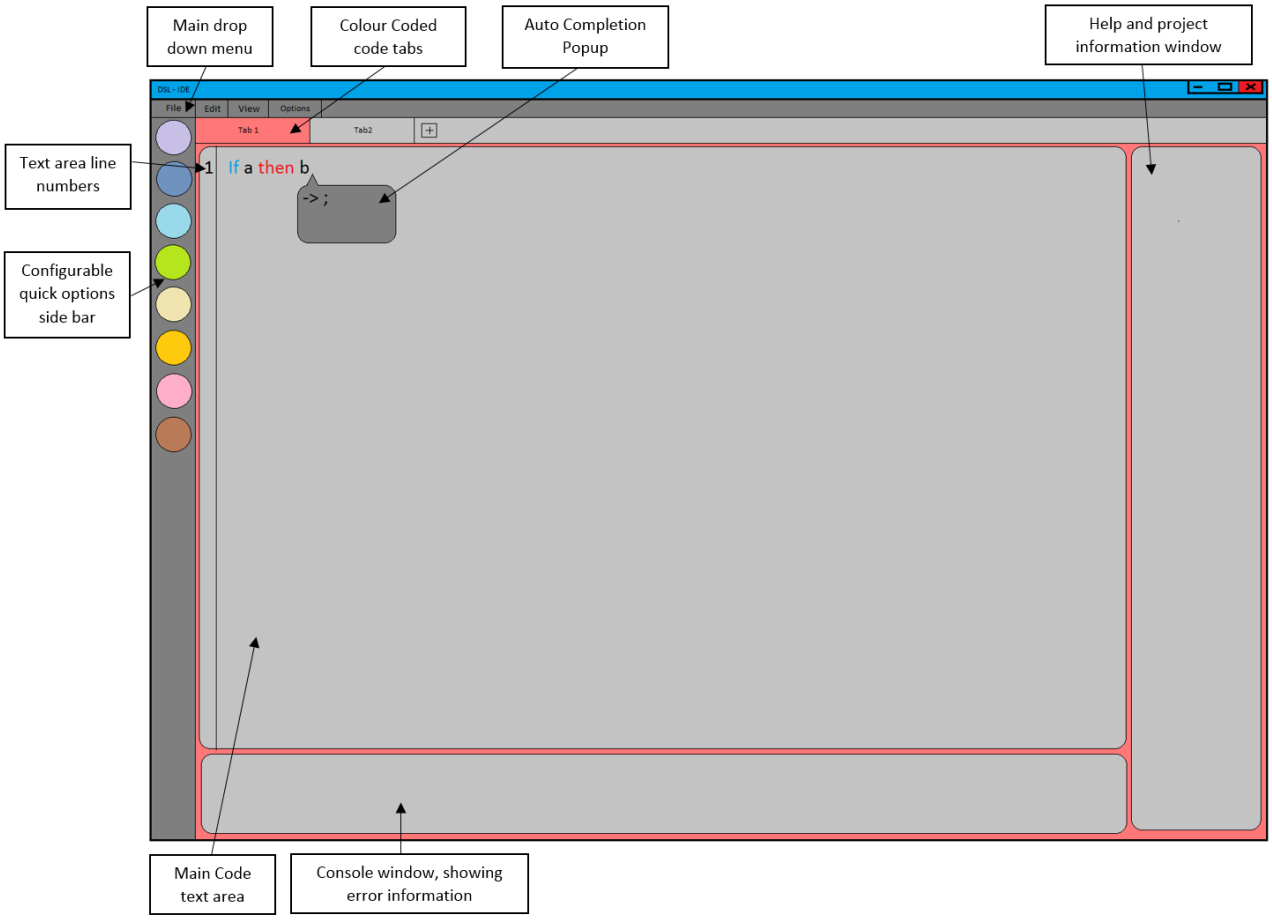



Figure 18: Concept drawing for the user GUI

### 4.3.3 Concept Developer GUI

The developer GUI maintains the same style as the user's GUI, but it includes more complex features such as dual panes by default. The improved developer GUI allows the developer to write and tests the language at the same time. Furthermore, the right sidebar contains information about the current project, such as file directories, project setting, and properties, as well as a help option.

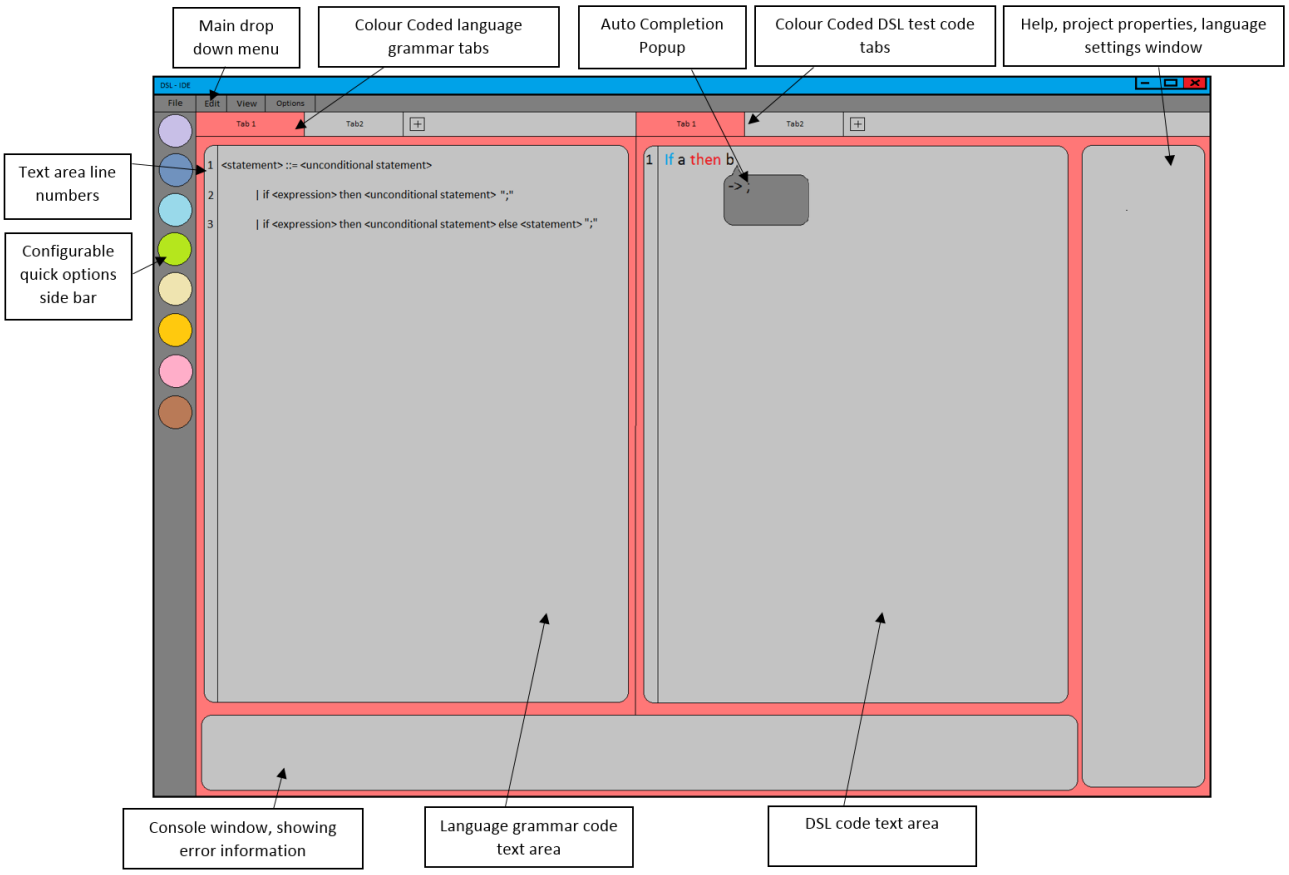



Figure 19: Concept drawing for the developer GUI

During this next cycle of design development, the issues that arise from the first design weighed heavily to the redesign shown in figure 18 and 19. Firstly, the main problem with traditional IDEs and the concept drawing for the first design is that a lot of the vertical space is used with settings. This leaves a minimal amount of space for the coding window. The material design philosophy, utilized for this project, states that elements that are most used should take the largest space and be most prominent to the user. As such, for this design cycle, the decision was made to move the quick action buttons to the left-hand side of the screen. Furthermore, there is now a dedicated small console window at the bottom of the screen. Finally, due to the wasted horizontal screen space, for the developer coding window, it was decided that both text areas can be shown in the same window. This allows for more seamless and transparent transition to coding the EBNF to the language implementation. The decision of moving UI elements to fill horizontal space rather than vertical space was made due to the proliferation of common screen aspect ratios been dominated by wide screen computers as of 2016. According to W3Schools (W3schools.com, 2016), 57% of computers in 2016 utilize a widescreen display. Therefore, it makes sense for this project to cater

for the majority demographic. Most traditional IDEs use design practices catered to vertically stacked UI elements. This is most likely due to long development cycles and that displays from before 2010 were predominantly non-wide-screen.

### 4.3.4 Final Design

This section details the final design of the program. Firstly figure 20 shows the main screen when loading the application. The user is given the choice of coding for a pre-existing language, or to create a new language. The logo is predominate, as for a screen like this, there aren't many options that can be shown, therefore, to fill the space and make proportions correct and professional looking, the logo was added as a centrepiece.

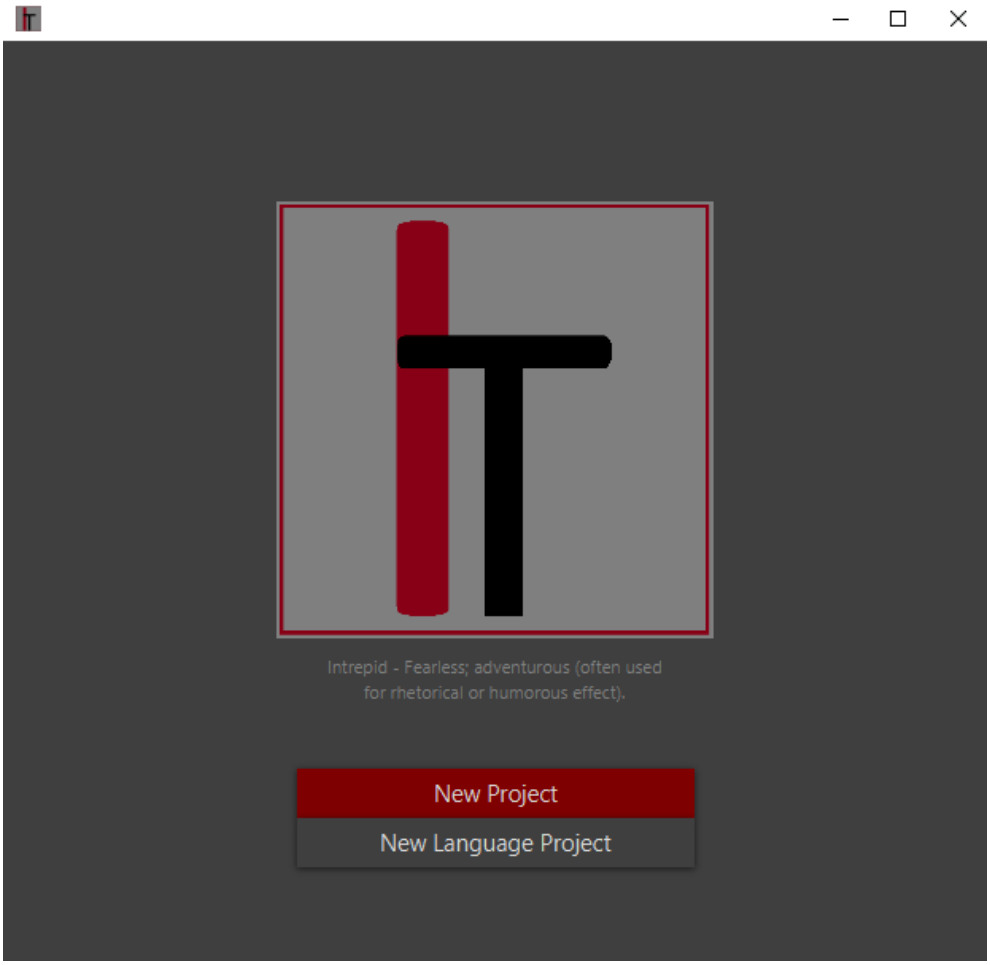



Figure 20: Intrepid's first run option screen

Figure 21 shows the window that appears when the user selects to use a built-in language. It maintains the same UI style as the previous window. As this project is using the material design guidelines, the main buttons on the screen have some elevation to them to make them appear prominent to the user. The choice of grey for the background was chosen to the how it complements the colours featured in the logo.

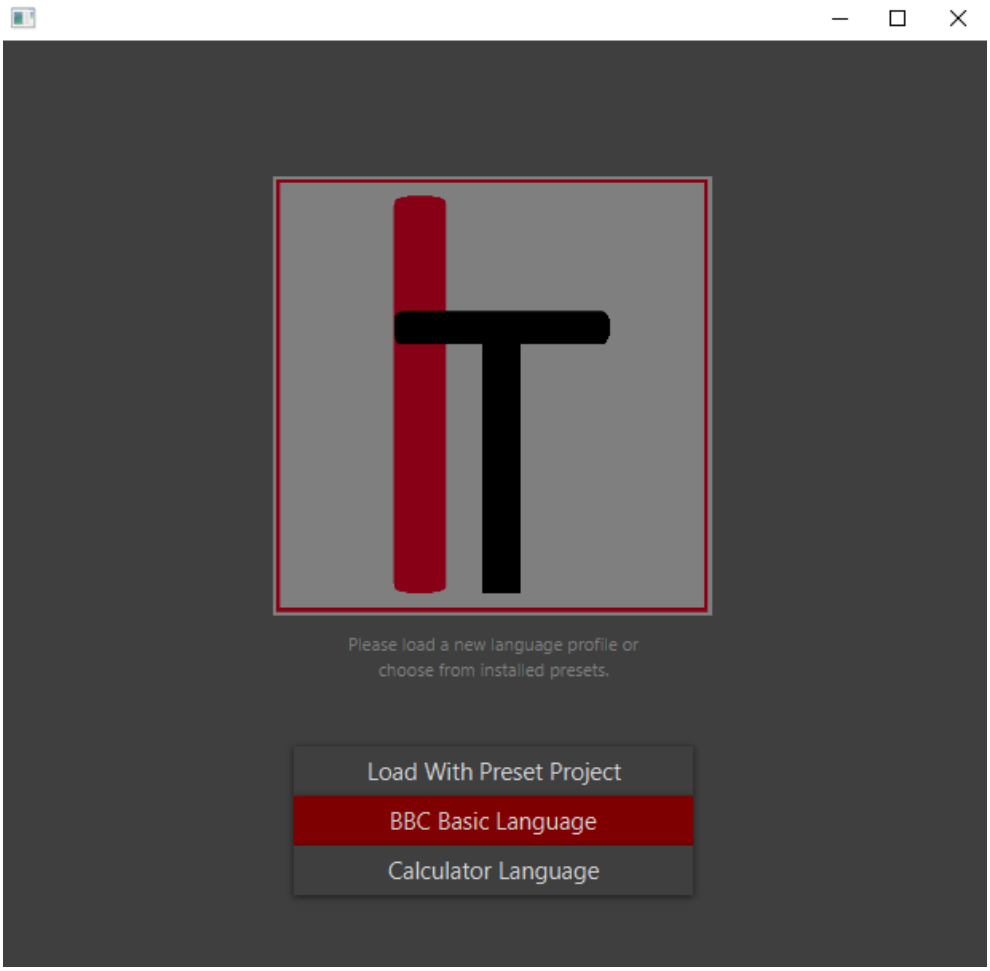



Figure 21: Intrepid's User Language choice GUI

Figure 22 shows the final design for the developer coding window. The design consists of shades of grey and soft colours to highlight frequently used UI objects. A light grey background surrounds all option windows, such as the cascading menu bar, quick actions, and token colour options. This allows easy viewing of text and enhances the definition of each UI object without being too convoluted. Furthermore, there is a consistent red accent colour to the currently highlighted tabs to make it very easy for the user to see quickly what parts of the UI are currently in focus. The white elements of the UI represent dynamic text viewing and manipulation areas, such that text can either be written or text will be displayed. This can be seen in the grammar, DSL and console output sections. All of the setting options combine to make an arc; this is to tell the user subtly that the actions are of a similar theme. The coding areas are surrounded in a dark gray, which contrasts with the solid white of the text areas. Furthermore, the text areas are at the centre of the application and are the largest UI elements. This is intended to tell the user that this is the main focus of this current window. Finally, the coding areas have line numbers and a caret line arrow, to make it easier to track the caret position.

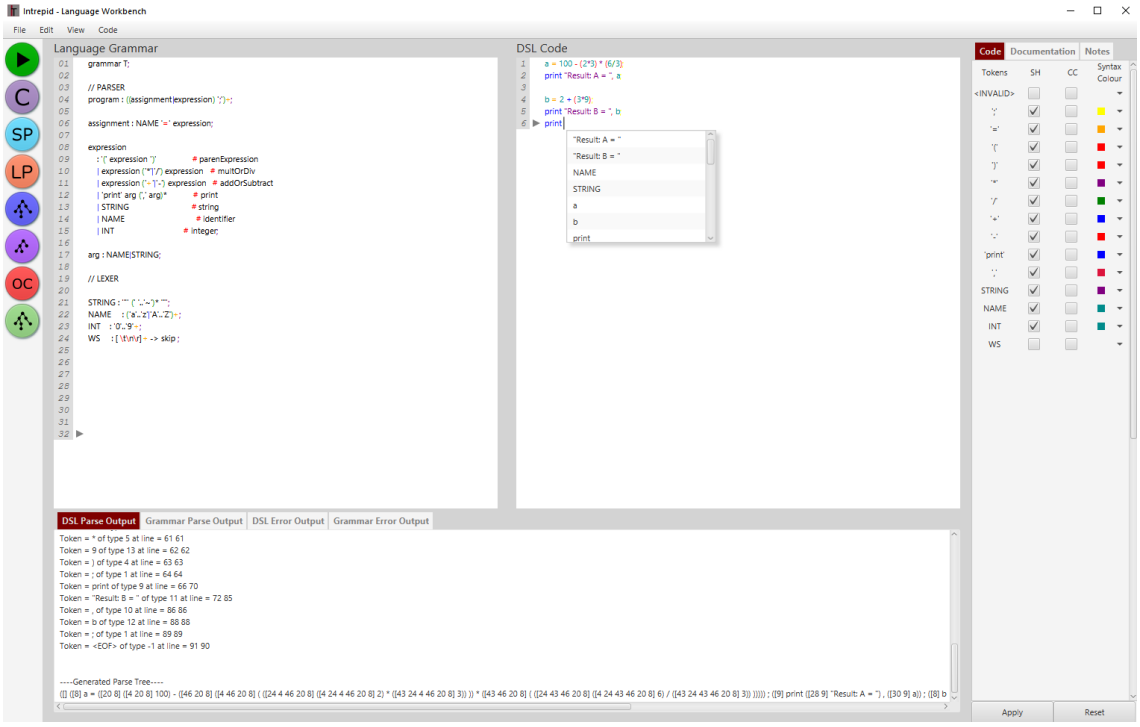



Figure 22: Intrepid's Developer GUI

Figure 23 shows the final User GUI; this GUI is based on a simplified version of the developer GUI and as such follows the same design guidelines. Although the main coding area covers the majority of the screen as from the user's perspective, this is the most important part of this window.

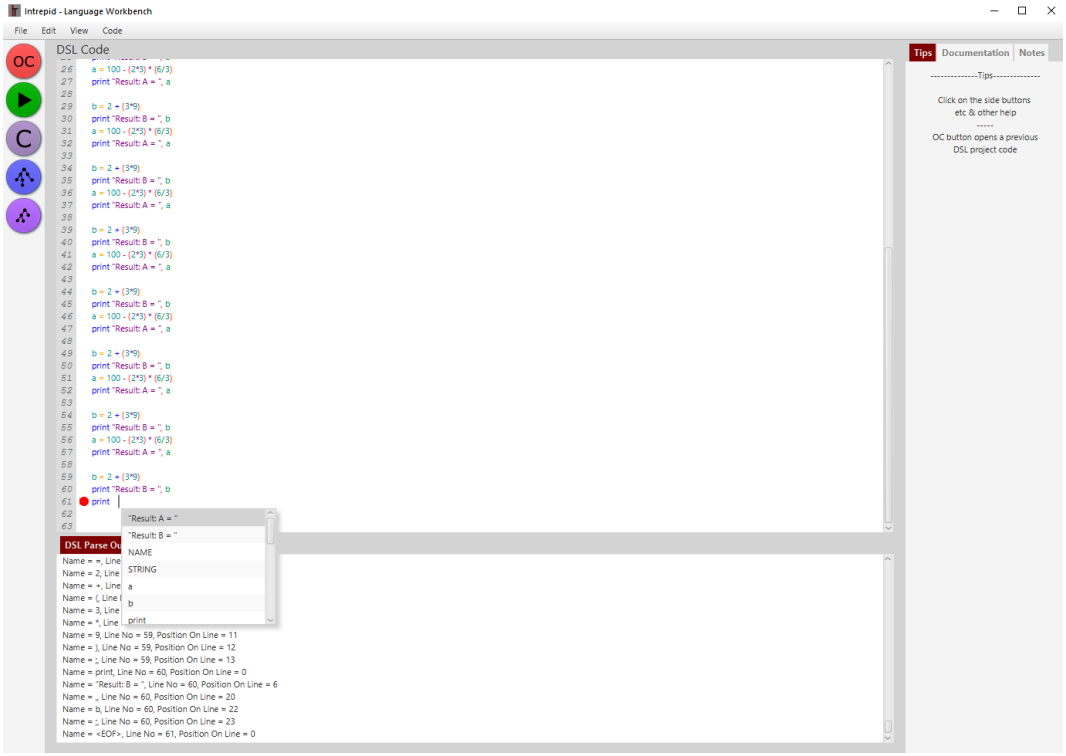



Figure 23: Intrepid User GUI

Figure 24 shows the implementation of the drop down menu buttons; the red highlighting shows the currently focused menu item. The menu has a shadow applied to it. This is intended to show depth to more easily discern the menu and its significance and hierarchy as recommended in the material design guidelines.


Figure 24: Menu bar in Intrepid

Figure 25 shows the main action buttons. These are actions that the user is most likely to need. Therefore, they have been given a circular shape to make them more prominent, as it contrasts with the angle design of the program. Furthermore, the colours are different from each other but use more pastel shades to avoid being aggressive in tone. The buttons also have elevation through the use of shadows to signify further to the user their significance. Finally, all the buttons of hover prompt to make it easy for the user to learn what each button does.


Figure 25: Quick action button bar with tooltips in Intrepid

Figure 26 shows the UI for viewing an AST for the language, as they can be vast it was decided that this should be in a separate window to the rest of the program. This also allows easy side by side comparisons on multi-monitor set-ups for the developer to run through the tree and compare against the language to check for DSL or EBNF errors.

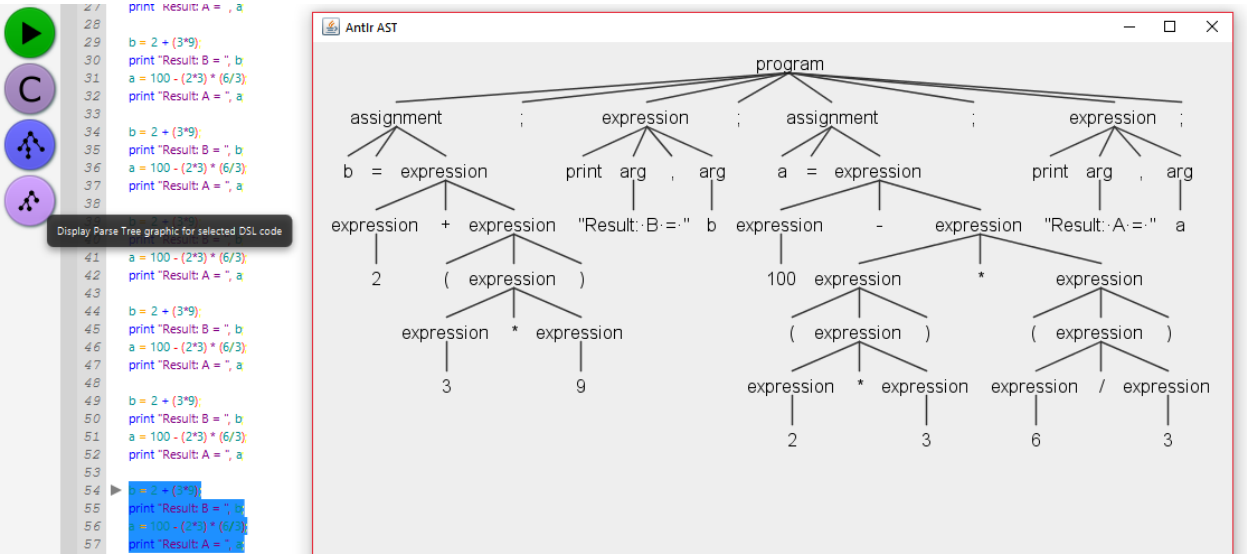



Figure 26: GUI of AST view in Intrepid

## 4.4 System Implementation

### 4.4.1 Jave runtime code generation and import

The main aim of this project is to allow a user to define a new language using a type 2 grammar, and then provide the ability to write the defined language including providing syntax highlighting and code completion. To achieve this goal requires the use of a lexer and a parser generated from a type 2 grammar. Due to the nature of this project, writing a lexer and parser generator is not feasible due to the enormous complexity of such a program. The solution chosen for this project was ANTLR Lexer and parser generator tool. ANTLR was chosen due to its compatibility with Java. This presents a vast technical hurdle for this project due to the nature of the pipeline of code generation and the use of a statically compiled language such as Java, which runs on a virtual machine.

ANTLR is a tool that generates a lexer and a parser based on an EBNF grammar passed to it through a command line or terminal interface. The typical workflow pipeline for ANTLR is to write the EBNF grammar in a text file which is then saved with the .G4 file. This is often done without any environment to aid in the writing of the grammar. After that the user must navigate to where the ANTLR .jar is, open the command line in windows or a terminal in Linux or Mac Osx. Then run the Java file through the command line interface, while passing in the name of the grammar file. After that, its common to import the generated lexer and parser into a working project to then utilize. Furthermore in the context of DSLs, it would then be necessary to program a coding environment for the user to write for the DSL while also implementing features to utilize the lexer and parser for the language. To further illustrate the traditional ANTLR pipeline, please refer to figure 27.

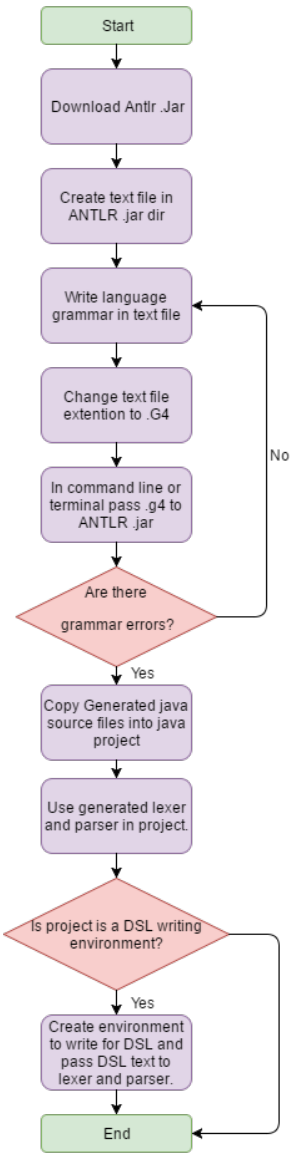



Figure 27: Flow diagram showing process of creating a lexer and parser for use in a basic IDE for a single DSL

The product of this research aims to reduce the pipeline shown in figure 27 and make it much simpler to write a grammar for a language, in addition to producing an IDE trailered for it. Intrepid IDE makes it much easier to write the grammar for a language. Firstly, the program provides an interface to write EBNF which include features such as syntax highlighting, basic code completion and error output. After the language grammar as been written, the next stage is to press the compile language button. This starts the process of code generation for the lexer and parser, including loading them back into the program. When this process is complete, the program outputs a list of tokens and syntax highlighting options for choosing colours the tokens should be assigned when encountered in the DSL text area. This is important due to some facets of a programming language having more significance than others. This isn't possible to be made automatic, as it would require the program to have a semantic understanding of the language, which from a type 2 grammar, is not plausible. Therefore, the choice of colours for tokens is presented to the developer

on the right-hand side of the screen. For a more detailed view of the improved work-flow with Intrepid IDE, please look at figure 28.

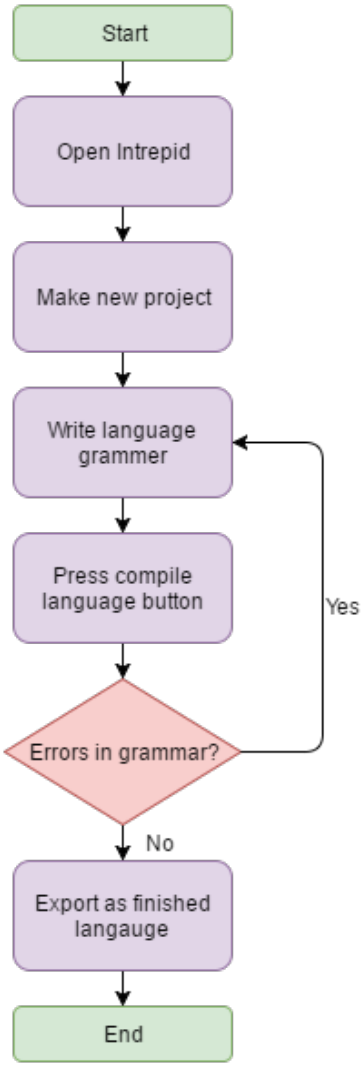



Figure 28: Flow diagram showing process of creating a lexer and parser for use in Intrepid IDE for a single DSL

To realize the goal of achieving the pipeline described in figure 28, the program needs to use ANTLR to build a lexer and parser from the EBNF programmatically, rather than through a command line interface external to the program. To achieve this, the EBNF is passed to a method contained in the CodeGeneration class in Intrepid, which writes a .g4 file to the temporary storage location using a Writer object. The next step is to pass this file to ANTLR. To achieve this programmatically, it was found in the ANTLR documentation (Antlr.org, 2016) that there is a tool called processGrammarOnCammandline() contained in the ANTLR tools class. This is intended to make it easy for developers to write small programs to generate code automatically, without having to use the command line directly every time. This makes it very convenient to use in this program, as given the correct arguments, it's possible to point to where the generated code should be saved. This process is running on a different thread to allow better performance in the program. The final step is to get an instance of the Java Compiler to compile the generated classes; this can

be obtained programmatically through the ToolProvider.getSystemJavaCompiler() method. This returns an instance of the systemsJavaC, which can then be passed the files to be compiled. This will output the generated .class files to the same directory as the .java files. For implementation details, please see the code example below.

```
Writer out = new BufferedWriter(new FileWriter(new
File(FileURL + languageName + ".g4")));
out.write(grammar);
out.close();

Tool tool = new Tool(new String[]{"-visitor", FileURL +
languageName + ".g4"});
tool.processGrammarsOnCommandLine(); //v4 changed
.process to processGrammarsOnCommandLine

JavaCompiler compilers =
ToolProvider.getSystemJavaCompiler();
compilers.run(null, System.out, null, "-sourcepath", "",
FileURL + languageName + "Lexer.java",
FileURL + languageName + "BaseListener.java",
FileURL + languageName + "BaseVisitor.java",
FileURL + languageName + "Listener.java",
FileURL + languageName + "Visitor.java",
FileURL + languageName + "Parser.java" );
```

One issue encountered with the approach described above was that the Java compiler would produce errors in compiling the classes, this was due to incorrect linking of the files. This turned out to be an issue with the order of the passed files to be compiled. It was found that the files have to be compiled in a particular order to be able to have correct linking. This is possibly due to the generated classes having high coupling. To resolve this issue, a significant amount of time was spent looking into the Java classes to find references to the other generated classes, in addition to the order the linking errors were generated. This information was used to deduce the correct compilation order. After conducting experiments with different class orderings, the correct order was found. This was tested against multiple grammars, and the compilation of each was successful.

The next step is to load the compiled classes back into the running program, as Java is a statically compiled language, this step was found to be challenging. However, it was found that though using an advanced language technique called reflection, the classes could be dynamically loaded into the program at runtime. Reflection is a technique in Java that allows Java code to inspect and dynamically call fields, methods, and constructors of an unknown Java file, within security restrictions. (Docs.oracle.com, 2016) The Java reflection API can be used on most public members of an object, although due to the JVM, some access limitations are imposed. The most important methods like constructors, the number of parameters and types are available. Due to the complexities involved in using this approach, the implementation will be detailed in this section. (Pal, 2016)

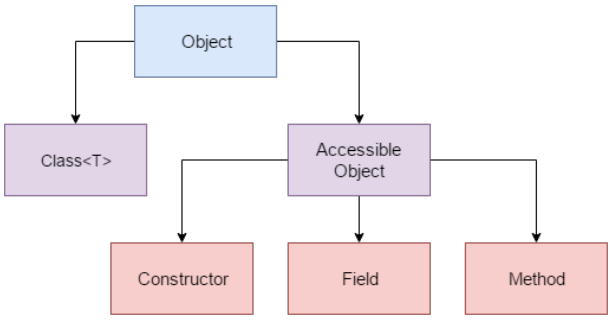



Figure 29: Reflection Class Hierarchy

Through reflection, the following code was used to achieve the objective of dynamically loading in a runtime generated compiled class. The following code example illustrates the loading of the generated class from a URL object. This makes it convenient to load any class off the file system. The URLClassLoader/1 returns a Java ClassLoader object when given a URL to a valid .class file. (Mcmanis, 2016) The ClassLoader is an object which is part of the Java runtime environment, which dynamically loads in Java classes into the virtual machine. Traditionally the Classloader is used to locate libraries, and the contents within them and as well as reading in classes from external libraries used in the program. In this use case, it is used to read and access attributes inside the generated classes obtained from a URL off the file system.

```
URL url = null;
try {
        url = file.toURI().toURL();
} catch (MalformedURLException
e) {
        e.printStackTrace();
        System.out.print(e.toString());
}
URL[] urls = new URL[]{url};
ClassLoader cl = new URLClass-
Loader(urls);
```

Firstly, the name of a class must be known to be able to load an external class into a running programming using reflection. This presents an issue for this project as the name of the generated class is based upon the grammar name defined in the grammar itself. Therefore, when the developer writes code, it's imperative that they use the standard grammar name specified in the user guide in Appendix 12. The next step in reflection is to obtain an object to encapsulate the constructor of the loaded class. It's not possible to access in the tradition way of instantiating a new class directly, as the program does not know what the constructor of the class is. Therefore, the solution found was to use the class loader object to load the class with a given name and then request the declared constructor. This is an important step, as obtaining the declared constructor, given a set of arguments, will search the loaded class for any constructor method that has parameters matching those provided in the declared constructor method invocation. In the example below, any constructor that accepts a char stream will be found by the classLoader and then returned as an object that can be cast to the Constructor object. After that, the constructor is used to call

the newInstance() method, which accepts a char stream. This then returns a new instance of the class. Due to the program not knowing what object the class is, to be able to access its attributes directly would require knowing the names of the methods and member variables contained in the class. Due to every generated ANTLR lexer and parser having different names based on the grammar used to generated them, this isn't feasible. Therefore, one solution was found that the object could be cast to the base object of the lexer that the generated class extends off. This comprises ensures access to parts of the generated class but limits what is accessible. Furthermore, the documentation provided for ANTLR only specifies how to access the generated code directly. Therefore, the techniques needed to obtain the required information to provide the features required for this project had to be found using alternative, undocumented techniques. Finally, the approach described was repeated for the parser object.

```
final String startPoint = "program";
final String lexerName = "Lexer";
final String parserName = "Parser";
final String languageName = "T";

Constructor constructor = (Constructor)
cl.loadClass(languageName +
lexerName).getDeclaredConstructor(CharStream.class);
DescriptiveErrorListener descriptiveErrorListener = new
DescriptiveErrorListener();

CharStream i = new ANTLRInputStream(c);
Lexer lexer = (Lexer) constructor.newInstance(i);
lexer.addErrorListener(descriptiveErrorListener);
TokenString = Arrays.asList(lexer.getTokenNames());
```

The final step in reflection was to invoke the needed methods in the class. In order to access the method which initializes the parsing of code, the base parser object can't be used, as the method is unique to the generated class. In ANTLR, the parser has a generated method called an entry point, which starts the parsing of the token stream provided by the lexer. The problem encountered with this project, is that the entry point is based on the starting point rule name in the grammar. Therefore, the starting point name must be known before generating the language, in order to invoke that method in the class. For the purpose of this project, a compromise was reached in which all languages defined in Intrepid must have a starting point in grammar called "program," this is to ensure access to the starting point method in the generated parser class.

```
try {
      Method entryPointMethod =
parserClass.getMethod(startPoint);
      t = (ParserRuleContext) entryPointMethod.invoke(p);
} catch (Exception e) {
      System.out.println(e.toString());
}
```

The code example above illustrate the main steps in acquiring a base lexer and parser objects to be utilized by the program through reflection. This was a challenging issue to overcome and required a gargantuan amount of time to research reflection as it is an advanced feature of Java. Furthermore, a significant amount of time was needed to test and experiment the code, in order to get the required outcomes. The solution isn't perfect, due to the non-dynamic grammar name and parser entry point name. Although, the compromise is a small issue that could be solved by having a field in the program to allow the user to tell the system the name of the grammar and entry point name. This solution could potentially negate the compromises reached during development.

### 4.4.2 Utilizing the Generated Lexer and Parser

The lexer and parser implementation in this project is essential to achieve the aims and objects of this project. Therefore, this section illustrates the implementation of this technology. The lexer is an object that takes in a string, performs lexical analysis and returns a list of tokens objects. Furthermore, it's secondary function is also to provide the start and end options in the input text for each token, which is stored in each token object.

```
DescriptiveErrorListener descriptiveErrorListener = new
DescriptiveErrorListener();
CharStream i = new ANTLRInputStream(c); Lexer lexer =
(Lexer) constructor.newInstance(i);
lexer.addErrorListener(descriptiveErrorListener);
TokenString = Arrays.asList(lexer.getTokenNames());
CommonTokenStream tokens = new
CommonTokenStream(lexer);
```

The lexer and parser implementation in this project is essential to achieve the aims and objects of this project. Therefore, this section illustrates the application of this technology. A lexer is an object that takes in a string, performs lexical analysis and returns a list of tokens objects. Furthermore, its secondary function is also to provide the start and end options in the input text for each token, which is stored in each token object.

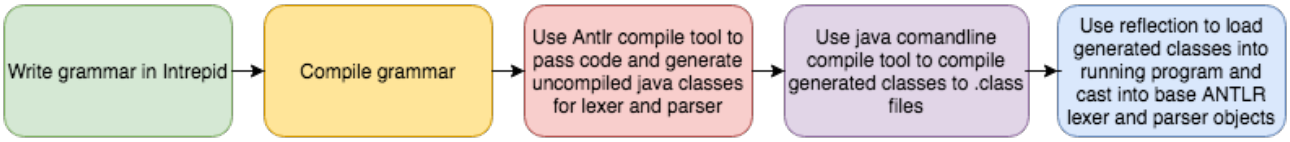



Figure 30: Pipeline of producing lexer and parser objects in Intrepid

For this program, a descriptive error listener class is instantiated, which is an extension to the default error listener that stores all errors encountered while tokenizing the input. Furthermore, the class also stores each error found in a data structure, which is used to then output the errors to the users later in the process. After that, a char stream stores the EBNF text which is then passed to the lexer constructor object obtained through reflection. The new instance method is called on the constructor object which returns an object that is cast into the base ANTLR lexer. The descriptive error listener class is then attached to the lexer. To initiate the tokenizing of the code, the getTokenNames()/0 is called, this returns all the ids of all the token objects found in the EBNF grammar and is stored in a list. The list data structure is used later in the program to

cross-reference against the list of error token ID's to obtain more information, as well as to use for presenting of the tokens that can be syntax highlighted in the DSL text area.

```
CommonTokenStream tokens = new CommonTokenStream(lexer);
Class⟨?⟩ parserClass = c2.loadClass(languageName + parserName);
Constructor parserConstructor =
parserClass.getDeclaredConstructor(TokenStream.class);
Parser p = (Parser) parserConstructor.newInstance(tokens);
p.setBuildParseTree(true);
ExceptionErrorStrategy exceptionErrorStrategy = new
ExceptionErrorStrategy();
p.addErrorListener(descriptiveErrorListener);
p.setErrorHandler(exceptionErrorStrategy);
ParserRuleContext t = null;
```

For this project, a parser is needed to produce an abstract syntax tree to be able to predict the next valid code through peaking at the next vertices on the tree from a given vertex. Furthermore, it is used to generate and find syntax errors in the DSL code implementation. The code example above shows the creation of the parser during runtime. Firstly the CommonTokenStream object uses the passed lexer to get the tokenized output of the code. After that, a generic class is instantiated with the reflected parser class. Then the constructor that accepts the CommmonTokenStream object is stored in a constructor object which is then used to instantiate a base parser object. To ensure an AST is generated the method setBuildParseTree is passed the boolean true value. Furthermore, as described further in the error implementation section, a custom error strategy class is attached to the parser.

```
Method entryPointMethod =
parserClass.getMethod(startPoint);
t = (ParserRuleContext) entryPointMethod.invoke(p);
TokenStream tokenStream = p.getInputStream();
```

The code illustrated above, further described in the Java runtime code generation section, initiates the parsing of the passed token stream. Finally, to obtain the parsed output, the get-InputStream() method returns a token stream. The token stream contains the token objects that have been tokenized. Each token object includes the token id, the string literal, start and end positions. The token streams utilized in this project for syntax highlighting and code completion.

Finally, Figure 31 shows a prototype error and tokenize output for the developer. The console outputs the product of the tokenizing and parsing of the DSL code. The output display is a useful tool to check that the generated parser has parsed code into the correct terminals and non-terminals. Furthermore, the text-based abstract syntax tree is also outputted at the end. The outputted text shows the string literal of the matched text and the id of each token in the parsed text. This output shows also illustrates all the information required for syntax highlighting.


Figure 31: Prototype showing tokens returned from prototype

### 4.4.3 Syntax Highlighting

Figure 32 shows an early prototype GUI, which integrates elements shown in the GUI concept drawings. The prototype allows defining of a language grammar and writing of code for the specified language. Furthermore, it provides the user with syntax highlighting, as well as syntax error information and parsing information. This prototype highlighted Java SWINGs inability to render text in different colours, and as such the only option was to highlight the text. This lead to the development of the JavaFX syntax highlighting panel prototype.

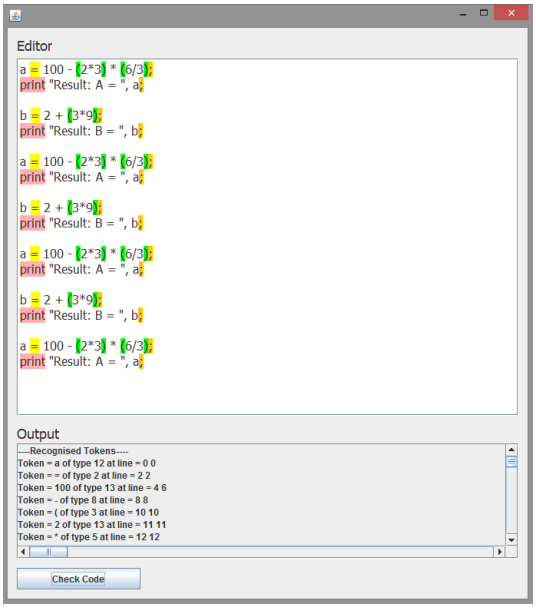



Figure 32: Prototype of UI with basic syntax highlighting using lexer/parser prototype

Figure 32 shows a panel UI designed using JavaFX based RichTextFX external library. (GitHub, 2016) This textarea panel is an extension of the javaFX native panel, which provided a much more robust way of styling each char displayed in the panel in real time. Furthermore, it provides a total line number change callback mechanism and an environment to attach graphics to the side of the panel. This feature was utilized to create a graphics factory that on every line change adds a graphic

presenting line numbers, as well as a current line indicator and line error graphic. RichTextFX was designed to mitigate the feature inept JavaFX and Swing text panels. To implement a basic syntax highlighting implementation, the program stores the last whole word typed, which is then checked against a list of tokens provided by the lexer and assigned the associated colour. After that, it updates the panel with the coloured version of the word to be displayed to the user.

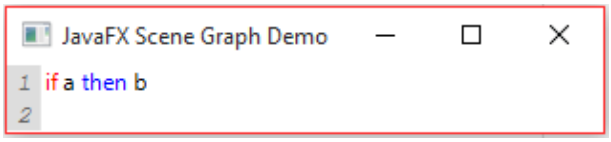



Figure 33: Prototype using JavaFX based RichTextFX with syntax highlighting

The code below illustrates the implementation of syntax highlighting in this project. On every key press, currently typed code is parsed and upon completion executes the code shown below. The parser outputs a tokenStream which is converted into a list of Token objects, which is then iterated through. In each iteration, the tokens ID is used to compare against the list of colours associated with each matched token by the developer. After that, the colour name is obtained using the colourIndex. Then the method called setStyleClass(), which takes in the start and end positions of the token in the original text and finally the name of the colour defined in a CSS style sheet which was associated with the text area instance on initialization. This technique was successful at providing real-time syntax highlighting based on what the user has typed. An experimental prototype using this technology can be seen in figure 33.

```
tokenStreamList.forEach((t) -⟩ {
int colourIndex =
ColourOptionsRow.findIndexOfToken(t.SymbolID);
String ColourCSSName =
ColourOptionsRow.tokens.get(colourIndex).getColour();
if(ColourCSSName.equals("default")){
        DSLArea.setStyleClass(t.StartIndex, t.StopIndex+1,
"black");
}else {
        DSLArea.setStyleClass(t.StartIndex, t.StopIndex + 1,
ColourCSSName);
}});
```

### 4.4.4 Code Completion

Code completion is intended to present a list of words to the user based on what the user is currently typing. As shown through background research, most IDEs implement it through a combination of means, for example through matching letters as the user types and presenting the closest matched words. Also, another technique is to build a list of all non-terminals and then display the closest matches as the user types. A more sophisticated method of providing more syntactically-aware code completion is to use a parser and continuously parse the text and present code suggestions based on the what the vertices that branch off the last valid vertex is. Furthermore, it's also possible to define hard coded code-completion suggestions, under the condition that the language is known as

the programs being built. Due to this projects dynamic nature, code completion presents one of the biggest challenges.

Firstly, to add basic code-completion into the project, the first technique of doing this is to store all the previously used words in a data structure. This was a challenge in itself, as from not knowing the nature of the language that the code completion is intended for ahead of time, it isn't possible to know what the syntactically and semantically significant sections of code are. Therefore, tests were carried out. The first of which was to septate all text though just spaces and present whole words to the user. The technique presented a problem, as many DSLs and GPLs often use symbols to represent chains in syntax, such as method calls in GPLs. As such, this approach of splitting the text proved to only be useful for languages that only separate key language elements through spaces. Otherwise, if the user is typing, the code completion would present whole chains of text depending on the language being used. This error highlights the complexity involved in making a language workbench, as in many features that would be simple to implement for a fixed known language become complicated due to not knowing the exact language that has been defined.

The first experiment was successful, but testing highlighted that it was prone to high degrees of error for non-natural language styled languages. Therefore, the next software design cycle utilized a different approach for separating meaningful text entities though using traditional GPL patterns. This entails that whenever a ".", ";", "=", "+", "/", "*", "%" and "&" are encountered, the code will be split and added to the code suggestions data structure. The result of this experiment showed improvement over the first implementation, as natural language and GPLs styled languages had the text successfully segmented and stored. However a limitation of this method is that if a developer defines an obscure language syntax, that isn't based on natural language or an existing GPL style, then the auto-completion has the potential to suggest syntactically invalid statements. However through testing, it showed a high rate of success. This issue could be mitigated by using a combination of techniques to predict the user's next input.

Finally, the last method of basic code completion was to create a continuously growing data structure of all matched and suggested tokens in the language provided by the parser. Therefore, any token that is suggested that's not in the data structure, it is then added. This new technique sufficiently mitigates the issues of the previous two experiments, due to it not defining critical sections of code through fixed symbols or spaces, but through what the parser itself recognizes as valid tokens. One shortcoming of this strategy is that it may take a few lines of code to be written before the list of possible recommended code encompasses most enough branches of a language to be useful.

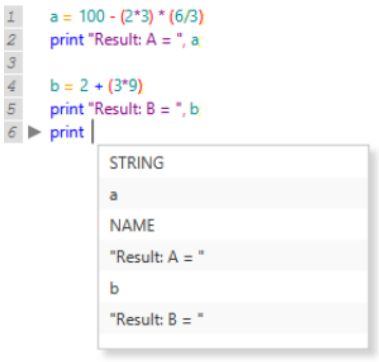



Figure 34: Semantically aware auto completion in Intrepid IDE

Although the previous solutions worked, it isn't fully syntactically-aware, so while it may pro-

duce valid terminals, it won't suggest correct terminals based on the position in the AST the users currently typed code is. This presented a unique challenge as it requires an AST to be built and traversed as the user's typing. This is necessary to check if what the code being typed conforms to the rules of the language. To implement this semi-semantically and syntactically-aware code completion, the parser generated from the loading of the program or when the developer compiles the grammar is stored and reused. This allows the user to type and on every key press, parse the code and check what the next available terminals are. The results of this peaking at the connected vertices on the AST allows the program to present them then as next valid tokens to the user. This approach worked and did suggest the next correct type of token allowed. Albeit, due to how the grammar can be defined, it will present production rule names as valid tokens, when the rule reduces to a regular expression (type 3 grammar). Which is often syntactically incorrect for most languages. Furthermore, due to different styles grammar, the success rate can be radically different. For example, with a syntax that allows very variable length statements where a line of two terminals can be valid, but can be extended further, the method of accessing the position in the tree only returns the shortest route in the tree, and will, therefore, stop presenting the user with code completion suggestions. If this project allowed using ANTLR as intended, this could be solved by creating a custom tree visitor, but this relies on knowledge of the programming language being used, which is inapplicable in this case.

To further develop code completion using the method described above, it was found that a list of token ids could be stored after each parse of the code, in addition to the tokens objects matched in the code. All token objects store the start and end positions in the original text, the corresponding token ID and a literal string of the originally matched text. An algorithm was constructed, to use the data to cross-reference the next logical tokens IDs and the IDs of all the parsed text tokens. If there are any ID matches, then the tokens original text is extracted and added to a data structure. This data structure then is presented to the user in addition to the next suggested tokens based off the AST. After testing, this minimizes the issue with the first implementation of the AST based code completion, as it's more likely to suggest useful code to the user, as well as the proposed code being syntactically correct.

```
final Set<String> matches = new HashSet<>();
for(String tokens : lastWord.split("\\s")) matches.add(tokens.toLowerCase());

Comparator<String> stringComparator = new Comparator<String>() {

    @Override
    public int compare(String s1, String s2) {
        int differenceNo = getDifferential(s1) - getDifferential(s2);
        if((getDifferential(s1) == 0 && getDifferential(s2) == 0) || differenceNo == 0)
        {return o1.compareTo(s2);}

        return - (getDifferential(s1) - getDifferential(s2));
    }

    private int getDifferential(String s) {
        int diff = 0;
        for(String match : matches) if(s.toLowerCase().contains(match)) diff++;
        return diff;
    }
};

Collections.sort(parserSuggestedWords, stringComparator);
Collections.sort(preSeenTokens, stringComparator);
parserSuggestedWords.addAll(preSeenTokens);
parserSuggestedWords = removeDuplicates(parserSuggestedWords);
```

Figure 35: Algorithm sort code completion lists

To further improve code completion, a mix of the two approaches was devised. Whereby the highest priority suggested code will be the code proposed by the parser based on the AST. After that, the previously seen tokens would appear under it. To further increase the usefulness of this feature, Java's comparator feature was utilized, as shown in Figure 35. The comparator will calculate a score of the difference between an input string and a value in the list the comparator was applied too. Then the Collections.Sort will sort the words list by the lowest to the highest difference between strings. This comparator is first applied to the list of parser generated next valid statements. After that, it's then applied to the list of every previously seen token. Finally, the list off every already seen token is appended to the end of parser generated suggestions. This decision was made due to the parser generated suggestion being more likely to be valid when they are available, so are therefore given the highest priority. An alternative approach would be to employ the Levenshtein distance algorithm, but due to time constraints, the comparator solution was used. Although, the Levenshtein distance algorithm could be the basis for future work.

```
// PARSER
program : ((assignment|expression) ';')+;

assignment : NAME '=' expression;

expression
   : '(' expression ')'              # parenExpression
   | expression ('*'|'/') expression  # multOrDiv
   | expression ('+'|'-') expression  # addOrSubtract
   | 'print' arg (',' arg)*           # print
   | STRING                           # string
   | NAME                             # identifier
   | INT                              # integer;

arg : NAME|STRING;

// LEXER

STRING : '"' (' '..'~')* '"';
NAME   : ('a'..'z'|'A'..'Z')+;
INT    : '0'..'9'+;
WS     : [ \t\n\r]+ -> skip ;
```

Figure 36: EBNF Grammar used by the program to generate the results for code completion

The grammar above was used to produce the lexer and parser needed for the current implementation of the code completion feature. In figure 34 the last typed word was "print", which according to the language grammar is a valid terminal. In regards to the print text, the EBNF says that a program can be multiple assignments or expressions with each followed by a semicolon. An expression can be defined as the terminal 'print' followed by an "arg". Finally, an "arg" is defined as being either a NAME or a STRING. Therefore, after 'print' a NAME or STRING is expected. This can be more clearly seen through the AST output for the parsing of 'print' in figure 37. This

shows that the solution of using a combination of techniques can be used to provided meaningful, syntactically-aware code completion for an unknown at compile time language.

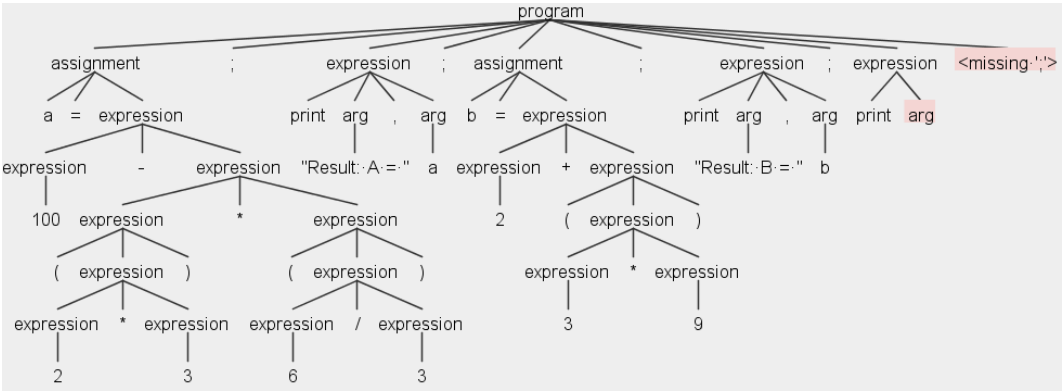



Figure 37: Example of the Intrepid's graphical abstract syntax tree output

### 4.4.5 Antlr Error Reporting

Error reporting is one of the most useful features of IDEs, but during development, it presented numerous challenges. Firstly, error reporting should be as informative as possible. This is to aid in the goal of reducing environmental complexity.

In the process of realizing and overcoming these challenges, the error reporting mechanism went through numerous revisions until the desired result was accomplished. Firstly, ANTLR provided necessary error output information to the debugging console only, which can't be displayed in the running program. Also, the error information provided is cryptic and may not be useful to non-programmers, such as the end users of a DSL. Therefore, an alternative solution needed to be found.

Research into ANTLRs error class structure was undertaken to achieve the goal of providing informative error information. It was found that ANTLR uses a class called BaseErrorListener, which contains methods which get called if there is an error during parsing. The error information reported consists of a recogniser object, offending symbol object, line integer, position in line, the default error message output to the debugging console and the recognition exception. To extract the required information, the BaseErrorListener was extended, and the syntaxError method was overridden. To easily encapsulate individual error information, a custom class utilizing the Java bean pattern was constructed. A list these objects are then stored in the overridden ErrorListener class. The information stored in each error object includes a list of next valid possible tokens, the offending symbol, line and position in the line of error and the string of the actual line in the original text the error occurred on.

To extract the list of next possible tokens, proved to be an issue. The technique described in the ANTLR documentation stated that the method called getExpected/0 should be called on the recognition exception object. It was found that this would cause a Java reflection exception. This is believed to be a result of using reflection, as the running program has no understanding of what is in the loaded class. This issue introduced many challenges through the program, which required some workaround to find the information in ANTLRs class structure. It was found that though using the recognition exception object, then getting the ATN object. The ATN object could then be used to call a method called getExpectedTokens, which then uses the offending state and a context object, which can also be obtained from the recognition exception. This approach

query's the AST generated by the parser on the last valid token. This then returns the name of the connected veracities. This information can be used to tell the user what the next valid tokens are.

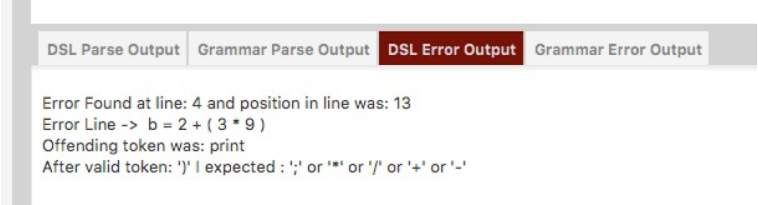



Figure 38: Example of informative error information

Lastly, using the collected error information, it is then possible to extract the line of text the error occurred in and the positions. A custom graphics factory class was constructed that adds a circle bitmap to the code text area, which is then set to invisible by default. If an error occurs, the graphics factory makes the circle visible and attaches an on hover event handler that displays a tooltip containing the error information on the line the error occurred on. The result of this custom graphics factory can be seen in figure 39. This information is also printed out to the error console window. This approach was to aid the user further by providing more helpful error information.

As part of the spiral model software design methodology used in this project, testing was conducted after implementing this feature. It was found that only a third of all the reported ANTLRs errors would be reported to the user, although the debugging window would show that the standard ANTLR error output for the errors. After using the IntelliJ decompiling tool and break pointing through the class structure of ANTLR when an error was detected, it was found that the problem was located in an errorStratagy class. The issue was caused from ANTLR separating syntax errors into three main types, all of which are handled differently in code. The three categories included: reportMissingToken, reportUnwantedToken and reportInputMismatch. For the first two categories, it was casting a null object into a recognition exception. This behavior was undesirable, as the information needed to construct a detailed error information is encapsulated in the recognition exception object. To solve this, a new class called ExceptionErrorStrategy was constructed that overrides these methods and built a recognition exception which is then passed to the attached error listeners. This new class was then attached to the parser and set to be the primary error strategy handler. After testing, it was found that this issue had been resolved. Even though this was a simple solution, the time required to discover and fix the issue resulted in this feature taking an extra time than what was planned to complete.

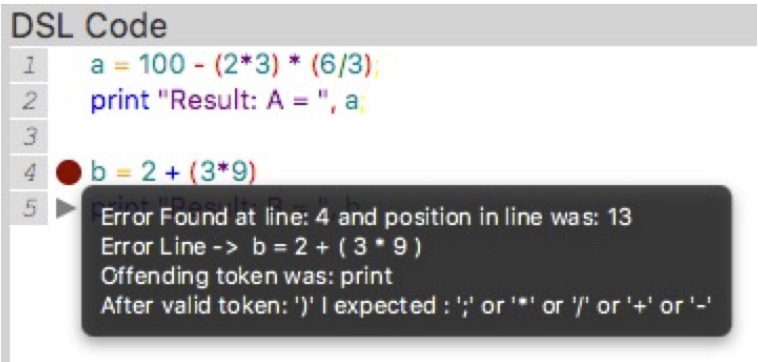



Figure 39: Example of informative error information

## 4.5 Testing Design

System testing is undertaken with every development cycle for prototypes and final product in the spiral model used for this development. This is good to catch bugs early. Furthermore, formal testing has been undertaken to ensure all modules of this project correctly work together. Finally, performance testing on the IDE will be performed. For performance testing data, please see appendix 1.

### 4.5.1 Formal Testing

Firstly, formal testing has been used to evaluate whether the program has bugs and if the system meets the original specification and system specification. Therefore, after the modules have been developed, testing against the specification was have been carried out. This lends itself well to the software design methodology utilized during the project. This means that for every cycle of development, the tests will evaluate whether more cycles are needed before the module can be integrated into the main program. For this section, only the final integrated program will be used, as this will be the program that a user would potentially use. These tests will be based off the specification and the describe behaviour that was displayed. This will help test how robust the program is, as well as the features general behaviour. The specification shows what both users expect the system to be able to do correctly. Furthermore, the specification details all the features of the program and therefore they will be used to test against.

| **Test No.** | **Description** | **Action** | **Expected Outcome** | **Actual Outcome** |
|---|---|---|---|---|
| 1.1 | Open Existing language project | A file containing EBNF grammar, documentation, language preferences | Opens the file, parses correct information and propagates EBNF text area, colour settings and documentation panel | Successful |
| 1.2 | Open Existing DSL project | A file containing just DSL implementation code | Opens file, and propagates the DSL text area | Successful |
| 1.3 | Create new language project | Two text areas to write EBNF and DSL implementation | Opens new developer window, containing two empty code areas, one for EBNF and the other for the language implementation | Successful |
| 1.4 | Create new DSL project | One text area to write for a chosen language | Opens a User UI window containing a black DSL text area that automatically provides syntax highlighting and code completion | Successful |

| | | | | |
|---|---|---|---|---|
| 1.5 | Save language project | A file containing EBNF grammar, documentation, language preferences | Save a file to a user specified location on the hosts file system containing the language setting, EBNF grammar and documentation | Successful |
| 1.6 | Save DSL project | A file containing just DSL implementation code | Save a file to a user specified location on the hosts file system containing the DSL implementation code | Successful |
| 2.1 | View DSL code with syntax highlighting in user and developer modes | DSL text area that colours certain keywords in text | When the user types DSL Code, based on the setting set by the developer the appropriate text will be coloured in real time | Successful |
| 2.2 | Change token syntax highlighting colours for a developer language project | Panel showing tokens found in code with an option to manipulate colours | When a token is ticked and allowed syntax highlighting, the chosen colour will be used to colour the matched text | Successful |
| 2.3 | Real time drawing of colour one valid token has been typed | Text area that can be typed in | The text area will refresh colours of text based on trigger events such as when the space bar is pressed | Successful |
| 3.1 | Display pop-up with useful code suggestions as the developer or user types. | Click able pop-up, which the text can be selected | When there is available code suggestions, a pop-up will appear under the typed text showing code suggestions based on the results from a dynamic parser and previously typed words | Successful |
| 3.2 | Ability to click a suggested text to insert and automatically syntax highlight suggestion if appropriate. | Click able suggestions in a pop-up under currently typed text | When a pop-up appears, when clicked, the clicked text automatically appears in the code area | Successful |

| | | | | |
|---|---|---|---|---|
| 3.3 | Pop-up should follow the users current typing position for fast access | Pop-up follows caret position | As the user types, the pop-up should be as close to where the user has typed as possible | Successful |
| 4.1 | Modify source code DSL text | Editable DSL text area | When the user types in the text area, the text is updated and the colours are updated too | Successful |
| 4.2 | Modify source code EBNF text | Editable EBNF text area | When the user types in the text area, the text is updated in real time | Successful |
| 4.3 | Cut/ Copy/ Paste text | Options in menu bar to copy, cut and paste | When the user highlights text, the copy will copy text to the devices clipboard. When the user selects cut, the selected text is removed from the text area and stored in the clipboard. When the user selects paste, any text in the clipboard is pasted into the selected text area at the caret position. | Successful |
| 5 | Compile Language grammar | Button to compile EBNF grammar | When the developer presses the language compile button, the lexer and parser is generated in the background and the DSL text area colours and recognized tokens are updated | Successful |
| 6 | Export project to list of included DSLs for user coding | Place to store language projects for user to utilize on program start-up | When the developer places the language project in a saved projects folder, when the program is started up and runs in user mode, the language appears natively in the selectable code able languages section | Successfully |

| | | | | |
|---|---|---|---|---|
| 7 | Output syntactic and semantic error is DSL code in the DSL errors tab | Area to show Error information | When the user or developer codes the DSL and error check it, any errors are collated and output to the DSL language error output tab | Successful |
| 8.1 | Change font of code text | Menu option to change font of code text | Present user fonts to choose from and update text areas when chosen | Fail |
| 8.2 | Allow developer to customize what tokens should be suggesting in auto complete pop-up | Panel showing tokens and selectable tick box for inclusion in code completion | When the tick box is checked for a token, the token is allowed to be shown in the code completion pop-up | Successful |
| 8.3 | Allow developer to customize what colours for tokens should be | Drop down box showing possible colours | When the user chooses a colour and ticks apply, the text in the DSL text area is automatically updated | Successful |

# 5 Evaluation

## 5.1 Project Achievements

In this section, the aims and objectives specified in section 2 will be evaluated against the nal program developed during this project. Each point will be assessed to determine whether the requirement has been achieved.

### 5.1.1 Minimum Requirements

The requirements detailed in this section specify the features that must be present in the program to be successful.

- **Develop an architecture to produce a lexer and parser from a grammar DSL.**
  - Intrepid IDE utilizes ANTLR which generates a lexer and parser for a given EBNG grammar .g4 file.
- **Develop a GUI to allow the developer to write the DSL Grammar.**
  - Intrepid IDE uses RichTextFX library, javaFX to provide a GUI to code for the grammar. For performance reasons, this utilized regular expressions to determine colour of text rather than a dynamic lexer and parser on every key press.

- **Develop a GUI to allow the developer and DSL end user to edit the DSL code, including syntax highlighting and code completion.**
  - Intrepid IDE uses RichTextFX library, javaFX to provide a two coding environments in the same window. This is in combination with a dynamic parser for the DSL code side to provide syntax highlighting and code completion. The parse tree generated from the parser is used to predict what code is valid next for use in code completion. Furthermore, the token stream provided for by the parser in combination with colours set for each token ID is used for real-time syntax highlighting of code for the DSL panel.
- **Provide an interface to define the colours associated with tokens for syntax highlighting.**
  - Intrepid Developer UI has implemented a left-hand side panel, which through using the token ids obtained from the lexer, allows the developer to easily assign a colour to a specific token type. This allows the developer to choose the necessary tokens that should be highlighted.
- **Research feasibility of an external DSL IDE and how it fits into the current market.**
  - This was achieved in the background and research phase of the project. The conclusion of the research found that there was a market for the product of this research to be used for external DSLs and reduction based internal DSLs.
- **Must be able to run on the Windows 7 x86/64 and above platforms**
  - Intrepid was written in Java, and runs on the Java virtual machine; this means that any OS with the correct virtual machine installed should be able to run this program.

### 5.1.2 Secondary Optional Requirements

The following requirements are based on the secondary aims and objectives for this project. These are features that are optional to implement during development, and therefore are not required for successful completion of this project, but are recommended to be included as they are useful to the program.

- **Provide Parsing Error information to users**
  - As described in the system implementation section, Intrepid uses a custom error listener and error strategy classes to collect parsing error information and then display them to the user using a graphic factory in the text area, as well as a console error output in the bottom area of the program.
- **Provide Git integration.**
  - This feature was researched but was it wasn't possible to implement in the time constraints of the program. With further time, libraries such as JGit could be used. (Anon, 2016)
- **Syntax highlighting for the BNF grammar file.**

- Intrepid IDE does syntax highlight the EBNF grammar text area. Due to the language being known at runtime, the syntax highlighting is achieved through using regular expressions on known key symbols. The alternative is to use ANTLR to have a dynamic lexer and parser, but it was found that having two running at the same time could potentially be very resource intensive.

- **Interface to allow developers to include DSL documentation.**
  - A text area tab for the developer to write help and documentation is included in the developer GUI and a non writable version is included in the user UI. This is saved in the project file, so is persistent when coding for a language.

- **Ability to run program on multiple platforms**
  - Intrepid was written in Java, and runs on the Java virtual machine; this means that any OS with the correct virtual machine installed should be able to run this program. This was tested on Mac OSX and Ubuntu; the tests showed that the application will run on multiple platforms.

## 5.2 Future Work

The current version of Intrepid IDE is currently feature rich, but theyre many opportunities to expand on currently implemented features and introduce a myriad of extra features during future development.

Firstly a source control system is a valuable feature thats prevalent in many GPL IDE's. This feature allows users regularly to keep track of changes in a project. This would allow features such as committing the newest version of a file along with a description all the changes since the last commit, as well as a full history of the changes of which the user can check all the progress of the project or revert to previous commits. This is especially useful for programmers, as it is common to introduce major bugs when implementing features, where reverting to an earlier version can often be the only plausible option. Furthermore, it is often possible to branch a program from a commit, allowing the user to keep different working copies where the developer can do experiments on code without affecting the main program. Then if successful can merge the experiment into the main branch. This approach has many advantages in that it can minimize the risk of introducing fatal bugs into the main project, as well as saving time from not having to manually backup the project, or keep multiple working copies for experiments. This feature could be implemented using numerous technologies, but the industry standard and most likely candidates for this project would be Git and SVN. To integrate Git in this project, a promising contender is JGit. JGit is a high-level Java library, which makes it much simpler to integrate Git into a Java program. (Anon, 2016) This feature lends itself well to the aim of this project, in reducing the environmental complexity of working with external DSL, though potentially saving time and protecting against data loss and loss of integrity.

Secondly, another method of ensuring data integrity would be a feature that would periodically backup the current project to a cached storage. This would be intended for situations where the program is closed unexpectedly, giving the user the option to restore the previous state on the programs next start up. Due to the time constraints of this project, it wasn't possible to implement the feature.

An additional feature would be the ability to write ANTLR composite grammars. This allows the developer to spread the writing of the grammar into multiple different files. This is often used to

separate lexer and parser information. This feature is intended to be used for very large languages, which often will be verbose definitions for large GPLs. Therefore, this features wasn't included in the main objectives of this project as it would be out of the scope of this projects DSL focus. Furthermore, it is possible to code for those larger languages in Intrepid single EBNF definition file, albeit it would a very verbose single file. To test this approach, during the testing of this program, as BBC basic grammar was used, which contained more than 1100 lines of EBNF and the results showed that it worked successfully in generating a lexer and parser. Also, it also showed that the performance of the program was still acceptable with this size of grammar. Although, to extend Intrepid target addressable market (TAM), this feature could be implemented later and allow the expansion of this program to be used for GPL development as well as DSL development.

Furthermore, it would also be possible to add in support for GPLs by default. This would entail having stored grammar for languages such as Java or C++. Where the user could choose to code for one of those languages without having to write manually or insert any grammar. Also, it would also be possible to utilize the Java platform's built-in Java compiler and the OS platforms built in C/C++ compiler such as GCN to then further compile and run the code for the program. This would then further expand Intrepids Tam, though allowing coding of multiple GPLs in the same IDE. This feature wasn't possible to add in the time constraints for the project, but for future work, it could prove to be a worthwhile feature. Although implementing too many GPL focused features, may introduce the problem of feature creeping, and as such increase the complexity of the program to an extent of increasing the environmental complexity. Therefore, time and user testing would be required to ensure this feature doesn't interfere with the DSL focus of the project.

### 5.2.1 Alternative Computing Platforms

Finally, the mobile computing revolution is currently occurring, where traditional computing tasks are now carried out on mobile devices such as phones and tablets. Many businesses and workflows of companies have transitioned to utilizing mobile technology instead of the old desktop computing paradigm. This has resulted in a decline of the desktop computing market, which therefore has the impact of reducing the total addressable market for this project. For example, according to IDC (www.idc.com, 2016) the desktop market has reduced by 10.3 percent from 2014 - 2015, and this trend appears to be increasing. (Keizer and Keizer, 2016) Therefore to combat this, some minor research was undertaken during this project to assess the feasibility of expanding this project to encompass mobile technology. The main contender for this is Android; this is due Android using the Java platform.

One issue is that even though Android uses the Java language, it has a far different build process than a standard Java program. As seen in the build process illustrated in figure 40. It is possible to include external jars in the program. Furthermore, it is also possible to use reflection on the Android platform. The main issue to be overcome in future work would be to compile the Java classes to the .dex files needed to be used in the program, as opposed to the standard .class files on the desktop platform. A temporary solution would have a desktop tool to compile the classes and have a module system in the add to allow the developer to integrate the lexer and parser in the app. This would then make the app an end user DSL IDE, rather than allow a developer to code and test a language on the mobile platform. This would be an incredibly exciting avenue to explore in future work.

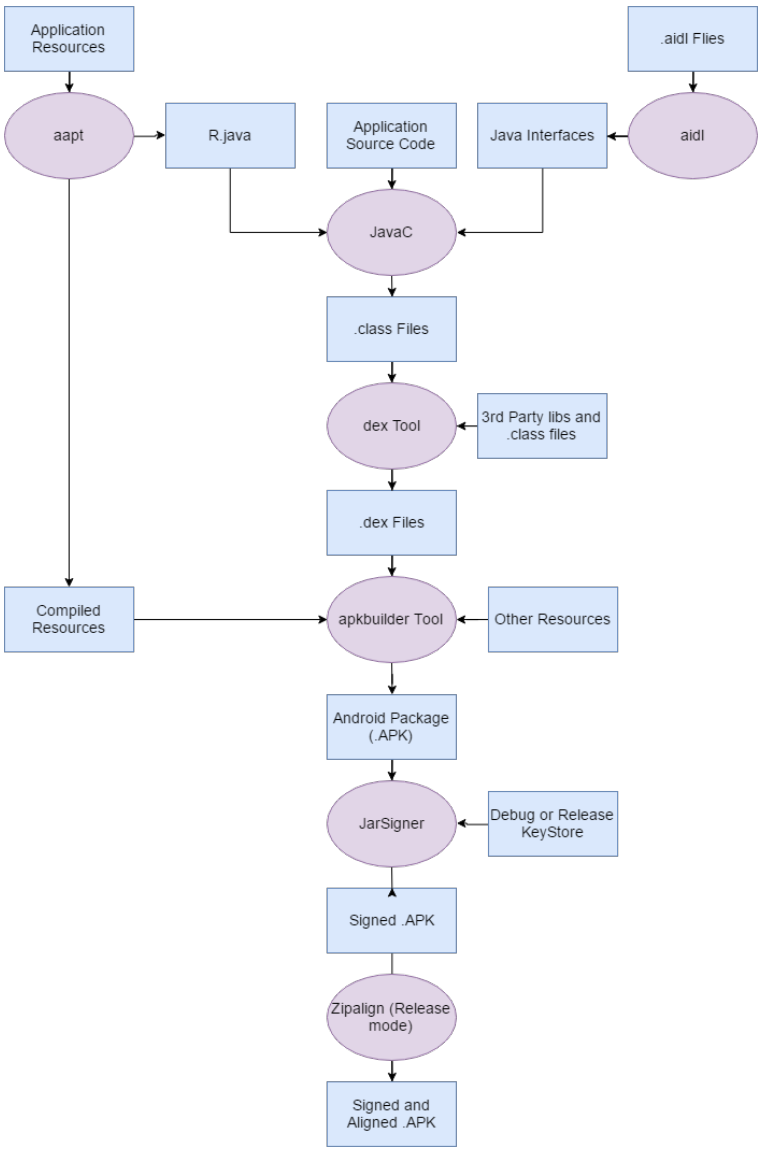



Figure 40: (Androidsrc.net, 2016)

# 6 Appendix

## 6.1 Appendix 0 - Personal Reflection

The following section is an evaluation of the issues encountered during development, as well as what has been learned and to provide a conclusion to this report.

Due to the scale and complexity of this project the developer has undertaken, there has been a significant learning curve. This was due to the knowledge needed to be known by the program at runtime to implement the proposed features. These include a list of tokens for keywords used by the language, as well as a semantic understanding of the language to be able to implement a form of code completion. From the beginning of this project, it has been agreed that this is more of a research endeavour into the feasibility and complexity of creating a dynamic language workbench. As such, a fully working and relatively bug-free program isn't expected for this project, but instead

a research report to determine an avenue in which the problem could be potentially solved, with a proof of concept prototype as a deliverable.

The risk analysis undertaken during this report was a tool with the aim of planning for potential hurdles that could affect the development of this report. For the most part, the risks evaluated were not encountered during the development of this project.

This project was an excellent platform to learn Java, the language and ecosystem were a new platform to learn but aptly suited the versatilely nature of this project due to being supported by all main operating systems. Although, this project did highlight how immature Javas GUI ecosystem was in regard to the Text Area controls.

Furthermore, due to this project's emphasis on an understanding of language construction, a new appreciation for the intricacies of programming languages and semantics have been gained. Especially concerning DSLs, whereby they may not ah-dear to traditional syntax and semantics for a programming language. DSLs grammar can range from a very simple statement procedural syntax to a natural language syntax with a pseudo context-sensitive element. DSLs can also potentially mirror that of GPLs with recursion, nested elements. Although, this is strongly advised against. It was interesting to attempt to strike a balance between these styles of grammar when experimenting with ways to implement code completion and syntax highlighting. Although for a project such as this, it is not feasible to account for every use case due to the numerous possibilities of languages that can be defined using a type two grammar.

The spiral model of application development has been a useful tool in developing this project. It allows iterative refinement on rapid prototypes. This lends itself well to this project, as its tasks can be modularised and have their own spiral development cycles, and can be done concurrently. It was found to ease bottlenecking during development, as tasks such as prototyping parts of the system could be developed in any order and independent of each other. This meant there was little time spent waiting for one task to be completed before moving to the next. Also, the planned weekly meetings with the project supervisor have been a useful tool to base completing tasks by. It was possible to complete a circle of a few different tasks and show prototypes to the supervisor for feedback. Furthermore, the meetings and spiral model helped in quickly finding out if a prototype would work using a set of technologies.

In conclusion, the development of this project has been based on the task plan. The estimations were accurate for most tasks except the code completion.. Though this projects development, the time plan has been used as more of a guide to check progress against, rather than a rigid scheduled. It was useful in being able to check where current progress is at and help ensure the projects progress is where it needed to be. Overall the product of this research has demonstrated the potential reasons for the lack of a product such as what has been researched during this project, from being widely available. These reasons include the complexity involved in using tools such as ANTLR in undocumented ways. Furthermore, the edge cases for a program such as this could take an infinite amount of time to test. Finally, using a statically typed language such as Java may not have been the best design decision for this project in retrospective. This is due to numerous issues with reflection encountered during development, which held back the progress of the project. This project was fascinating, but the scope was out of the range that could be covered during the development period. Therefore, future work will be carried on after the official project end to develop a more polished program.

## 6.2 Appendix 1 - Performance testing

In addition, performance is also very important to IDEs. For example, syntax highlighting and code completion must be real time to aid in the writing of code. As such, performance tables will be made measuring how fast the program is to update text colour and how fast the code completion pop-up appears. This will be tested against the different complexity of grammars as well as how performance scales with DSL code size. Furthermore, different system on chips (SoCs) with varying performance have been used to test performance on. The hardware used was chosen to cover the majority of consumer SoC speeds available. Therefore the SoC's chosen have a varying vertical (Instructions Per Clock (IPC) ⟩ Multi-threading performance) and horizontal (IPC ⟩ Multi-threading performance) nature of the different chip-sets. For example, the i5 4690k represents the best IPC performance that can be expected, although it has low core count. The X5650 represents a more horizontal approach, where the IPC is substantially lower than the i5 4690k, but has much higher multi-threading performance. Finally, the Q9550 represents the worst case scenario, with very poor IPC and multi-threading performance, and is worse than the majority of mid-range consumer laptop system on chips (approximately 2ghz Dual Core hyper-threaded haswell i5/i3). The different CPU's used were:

- Intel i5 4690k - Over-clocked to 4.8ghz, 4 cores, 4 threads, 22nm lithography, 64-bit, Dual Channel DDR3, 6MB Smart Cache, released 2014 (Intel ARK (Product Specs), 2016)
- Intel Xeon x5650 - Over-clocked to 4.136ghz, 6 cores, 6 threads, 32nm lithography, 64-bit, Quad Channel DDR3, 12MB Smart Cache, released 2010 (Intel ARK (Product Specs), 2016)
- Intel Core Quad Q9550 - Over-clocked to 3.8ghz, 4 cores, 4 threads, 45nm lithography, 64-but, Dual Channel DDR2, 12MB L2 Cache, released 2009 (Intel ARK (Product Specs), 2016)

The following results were tests carried out on the rendering speed of colours in the DSL code. The test measured the time it took to use the list of tokens from the text, to then finish drawing them on screen. The tests were repeated ten times and the average score was recorded. The results show only a mild increase is time was taken to draw as the text increases; this is due to updating the colours of the line being edited being updated. The time taken does increase slightly as the code get's larger. This is believed to be a result of the time taken to iterate through the list of tokens to find the line of the changed text to perform the update, as well as the latency of accessing elements from l1 l2,l3, and ram.

| CPU | DSL Lines of Code (Lines) | Average Time Taken to Draw (Milliseconds) |
|---|---|---|
| i5 4690k, 4 threads, 4 cores, 4.8Ghz | 10 | 0.1732 |
| Xeon X5650, 12 threads, 6 cores, 4.136Ghz | 10 | 0.2362 |
| Q9550, 4 threads, 4 cores, 3.8Ghz | 10 | 0.6040 |
| i5 4690k, 4 threads, 4 cores, 4.8Ghz | 100 | 0.1982 |
| Xeon X5650, 12 threads, 6 cores, 4.136Ghz | 100 | 0.4828 |
| Q9550, 4 threads, 4 cores, 3.8Ghz | 100 | 0.7562 |
| i5 4690k, 4 threads, 4 cores,, 4.8Ghz | 1000 | 0.2673 |
| Xeon X5650, 12 threads, 6 cores, 4.136Ghz | 1000 | 0.3620 |
| Q9550, 4 threads, 4 cores, 3.8Ghz | 1000 | 0.9347 |

The following tests were conducted to test for the time taken to produce a lexer and parser from an EBNF grammar, compile the generated code. The tests were repeated 10 times and the average score was recorded. The generating of the code is the most performance intensive process in the application and is heavily multi-threaded. The results show that the Xeon X5650 had the best performance, due to it's super muli-threading performance due to extra cores and hyper-threading technology. Due to this tasks very long completion time on all processors, the whole task is put on a runnable with a callback. This allows the process to not pause the UI thread on the host OS whilst the process is running. Performance optimization for this step and feasible, as the most expensive operation is the ANTLR generating process which is external to the program. The low performance in this area is not much of a concern however, as this is not a regular operation in the program.

| CPU | DSL Lines of Code (Lines) | Average Time Taken to Draw (Milliseconds) |
|---|---|---|
| i5 4690k, 4 threads, 4 cores, 4.8Ghz | 10 | 0736.92 |
| Xeon X5650, 12 threads, 6 cores, 4.136Ghz | 10 | 0481.29 |
| Q9550, 4 threads, 4 cores, 3.8Ghz | 10 | 1371.45 |
| i5 4690k, 4 threads, 4 cores, 4.8Ghz | 100 | 0982.00 |
| Xeon X5650, 12 threads, 6 cores, 4.136Ghz | 100 | 0752.65 |
| Q9550, 4 threads, 4 cores, 3.8Ghz | 100 | 1702.18 |
| i5 4690k, 4 threads, 4 cores,, 4.8Ghz | 1000 | 2382.31 |
| Xeon X5650, 12 threads, 6 cores, 4.136Ghz | 1000 | 1982.30 |
| Q9550, 4 threads, 4 cores, 3.8Ghz | 1000 | 3873.29 |

## 6.3 Appendix 1 - Task List

| # | Task Name | Description | Duration (Days) |
|---|---|---|---|
| 1 | Research Domain | Research current IDE technologies in the market and whether they include features related to this study. | 3 |
| 2 | Improve Proposal | Based on research, evaluate whether this project can add to this area. | 1 |
| 3 | Research DSLs | Research Domain specific languages and how they could be used to improve code maintainability and the tools available for them. | 2 |
| 4 | Research Syntax Highlighting techniques | Research different methods of providing syntax highlighting. | 1 |
| 5 | Research Lexers & Parsers | Compare alternative solutions to generating lexers and parsers to be used for syntax highlighting. | 1 |
| 6 | Prototype Syntax Highlighting | Based off previous research, produce one or more prototypes using different methods and compare them to see which one suits this project. | 3 |
| 7 | Prototype Code Completion | Based off previous research, produce one or more prototypes using different methods and compare them to see which one suits this project. | 3 |
| 8 | Test Prototypes | Review against objectives to whether they meet the specification. | 1 |
| 9 | Based on review, iteratively test and improve prototypes | Review the tests and iteratively improve. Repeat until it meets the specification. | 2 |
| 10 | Interim report | Write the interim report deliverable | 14 |
| 11 | System Architecture Diagrams | Build different forms of UML diagrams to ensure the aim and deliverable of this project have been fully realized. | 1 |
| 12 | Prototype Lexer Generator | Based on previous research produce a lexer generator prototype. | 3 |
| 13 | Prototype Parser Generator | Based on previous research produce a parser generator prototype | 3 |
| 14 | Prototype External DSL Editor | Build a ui that's able to change individual character colonizing to support syntax highlighting of code | 2 |

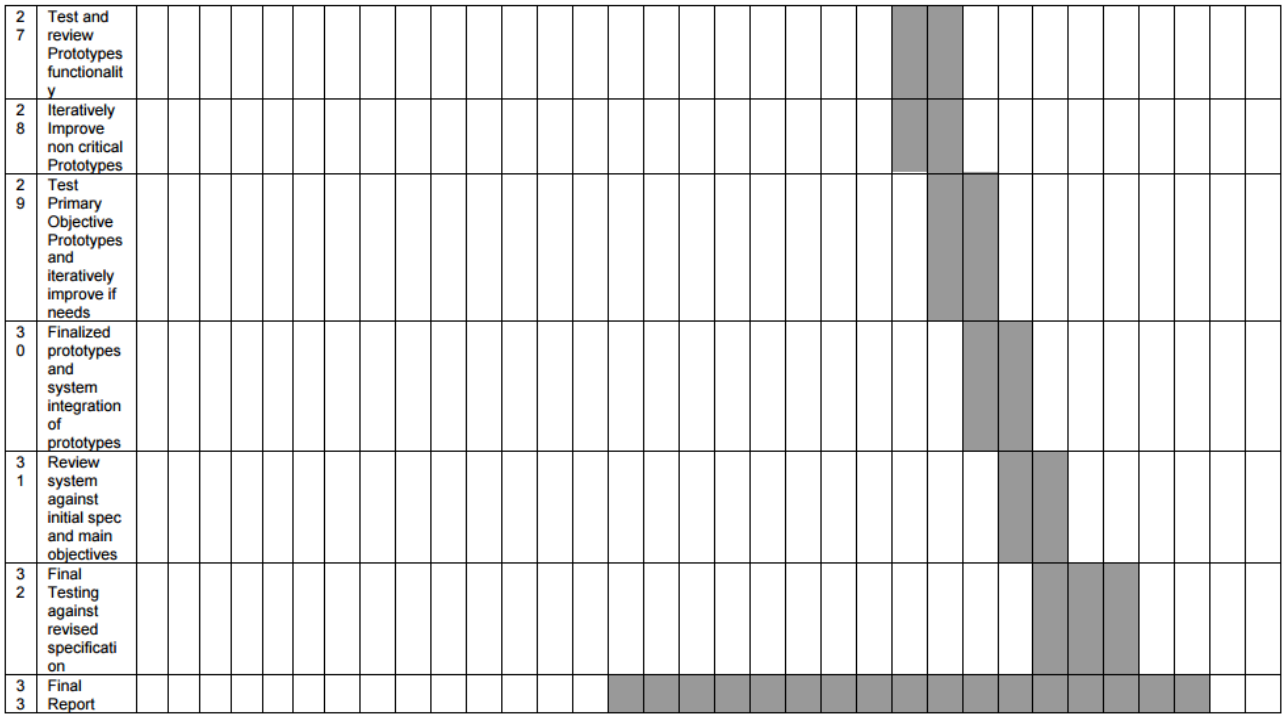
| | | | |
|---|---|---|---|
| 15 | Paper UI Concepts | Construct designs of possible user interfaces, highlighting what are the positives and negative to each design. | 1 |
| 16 | Prototype Developer GUI | Prototype Developer GUI | 3 |
| 17 | Prototype DSL end user GUI | Using previously build lexer and parser, create a editor front end for the BNF notation. | 3 |
| 18 | Prototype Token Colour DSL | Construct and external DSL that uses the list of tokens to allow the programmer to easily assign syntax highlighting colours for a particular token. | 3 |
| 19 | System Testing | Produce tests for the system, that tests valid and invalid data and review the results. | 5 |
| 20 | Review Prototypes | Compare the prototype's functionality against the specification to show whether or not they have been met. | 2 |
| 21 | Iteratively improve prototypes | If the prototype review concluded that improvements are needed, then use a spiral model to iteratively improve and review until it meets the specification. | 3 |
| 22 | Prototype Error Output System | After primary objectives have been met, review secondary and produce and error output system for both the DSL end user and developer. | 1 |
| 23 | Prototype Git Integration | Research and integrate Git integration with the project. | 2 |
| 24 | Prototype Documentation Presentation Interface | Produce an interface for the developer to include a pdf or other suitable format to include as part of the application, intended for the DSL end user to read for help and support. | 2 |
| 25 | DSL for BNF Grammar | Provide a procedural programming style DSL for the BNF grammar | 1 |
| 26 | Test and review Prototypes functionality | Test and review Prototypes functionality | 2 |
| 27 | Iteratively Improve non critical Prototypes | Based on the review of tests, using the spiral model, iteratively improve the prototypes and then review until they meet the objectives. | 2 |

| | | | |
|---|---|---|---|
| 28 | Finalized prototypes and system integration of prototypes | Integrate the prototypes into a main unified system. | 6 |
| 29 | Review system against initial spec and main objectives | Review the unified system against the revised specification. If elements don't match, then improve those area and review again. | 3 |
| 30 | Final Testing against revised specification | Great tests with large coverage and review results, if errors are present then fix them and test again. | 5 |
| 31 | Intro Report | Produce the intro report document | 7 |
| 32 | Final Report | Produce final report, including testing and reviewing of the software process. | 15 |

## 6.4 Appendix 3 - Time Plan

Due to this project being based on the spiral model, tasks are often given multiple time segments which correspond with iterative cycles of development, furthermore some tasks can be done concurrently, due to task dependencies. The following has taken the estimates from the time analysis and has added extra time to tasks to predict the worst case scenario.

| | | University Calendar Weeks | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| # | Task Name | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 |
| 1 | Research Domain | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 2 | Improve Proposal | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 3 | Interim report | | | | | | | | | | | | | | | | D | | | | | | | | | | | | | | | | | |
| 4 | Research DSLs | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 5 | Research Syntax Highlighting | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 6 | Research Lexers and Parsers | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 7 | Prototype Syntax Highlighting | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 8 | Prototype Code Completion | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 9 | Review Prototypes | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 10 | Iteratively Test and Improve Prototypes | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 11 | System Architecture Diagrams | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| `2 | Prototype Lexer Generator | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 13 | Prototype Parser Generator | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 14 | Test and improve Lexer and Parser Generator prototypes | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 15 | Prototype External DSL Editor | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 16 | Paper UI Concepts | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 17 | Prototype Developer GUI | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 18 | Prototype DSL end user GUI | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 19 | Prototype Token Colour DSL | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 20 | System Testing | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 21 | Review Prototypes | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 22 | Iteratively improve prototypes | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 23 | Prototype Error Output System | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 24 | Prototype Git Integration | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 25 | Prototype Documentation Presentation Interface | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 26 | Provide a procedural programming style DSL for the BNF grammar | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |




## 6.5 Appendix 4 - Definitions

**Integrated Development Environment (IDE):** Programming environment, with language tailored features to provide more efficient workflow.
**Software Bottleneck:** The high demand for increasingly complex software that outstrips the capacities of developers. (Banker and Datar, 1987).
**3rd Generation Languages (3GL):** General Purpose High level Languages (GPLS), Java, C++ etc.
**4th Generation Languages (4GL):** DSL, higher level of abstraction over GPLS.
**Domain Specific Language (DSL):** Little languages targeted at a specific domain, incorporating only domain specific features.
**Language Workbench:** Software tool intended to aid development and use of DSLs.
Syntax Highlighting: Form of secondary notation, highlighting key language terms often in different colours.
**Code Completion:** Predictively suggests the next part of a statement, often based from an intermediate representation.
**Lexer:** First phase of a compiler front end, tokenizing input.
**Parser:** Second phase of a compiler, intended to understand the semantics of the language and often produces an intermediate representation such as an abstract syntax tree.

## 6.6 Appendix 5 - System Use Case Diagram

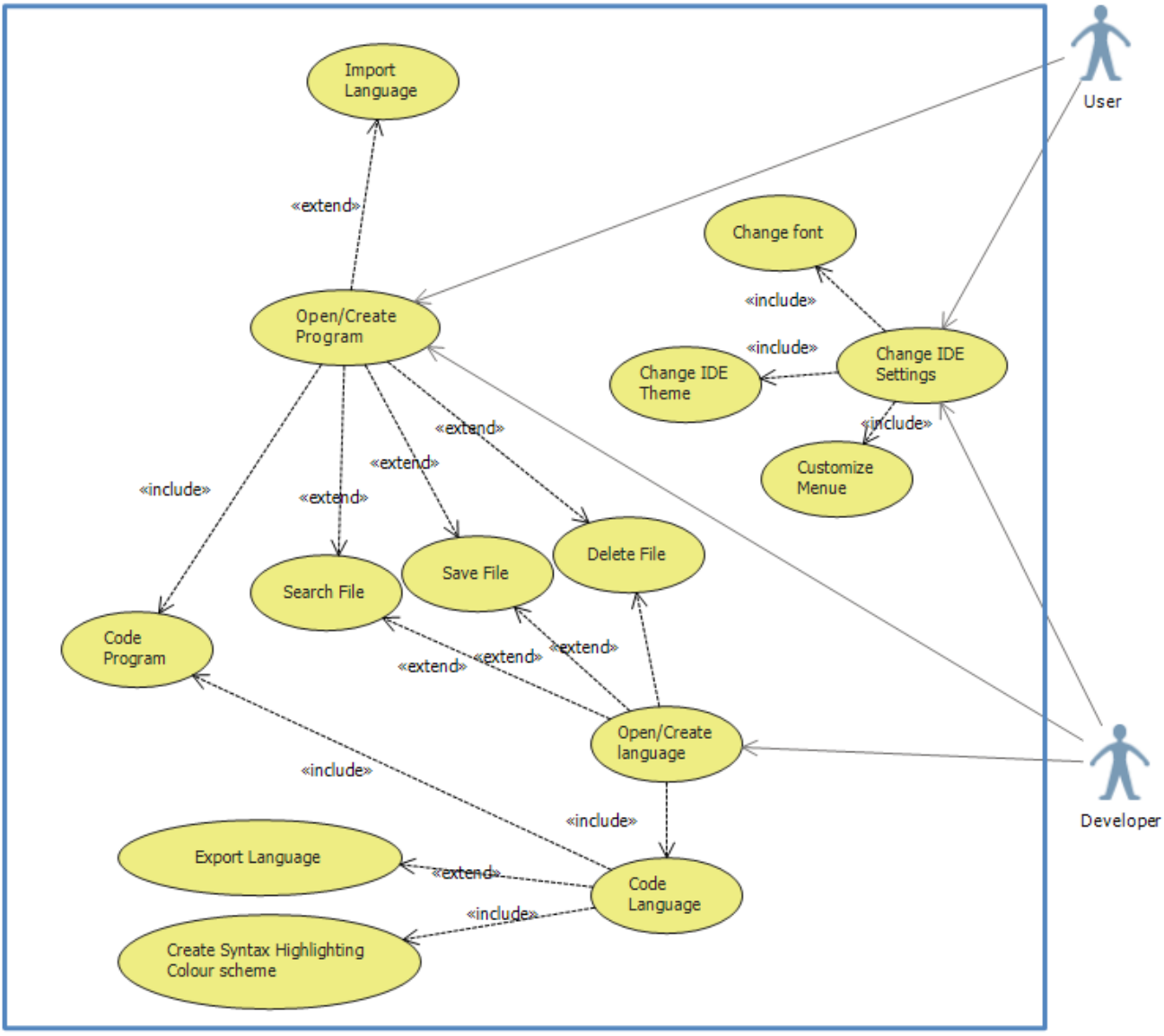



Figure 41: Use Case Digram of system

## 6.7 Appendix 6 - ANTLR Flow Diagram

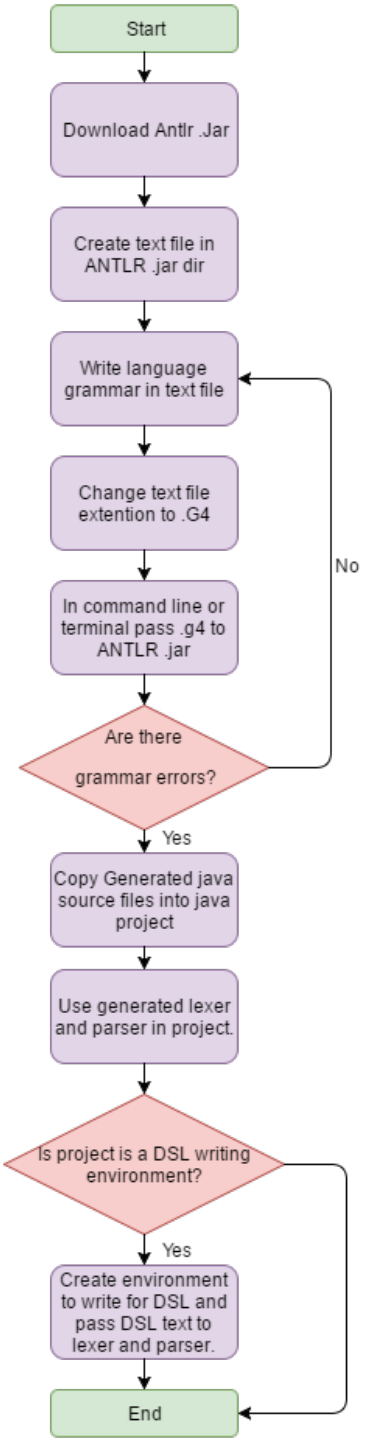



Figure 42: Flow diagram showing process of creating a lexer and parser for use in a basic IDE for a single DSL

## 6.8 Appendix 7 - Intrepid Flow Diagram

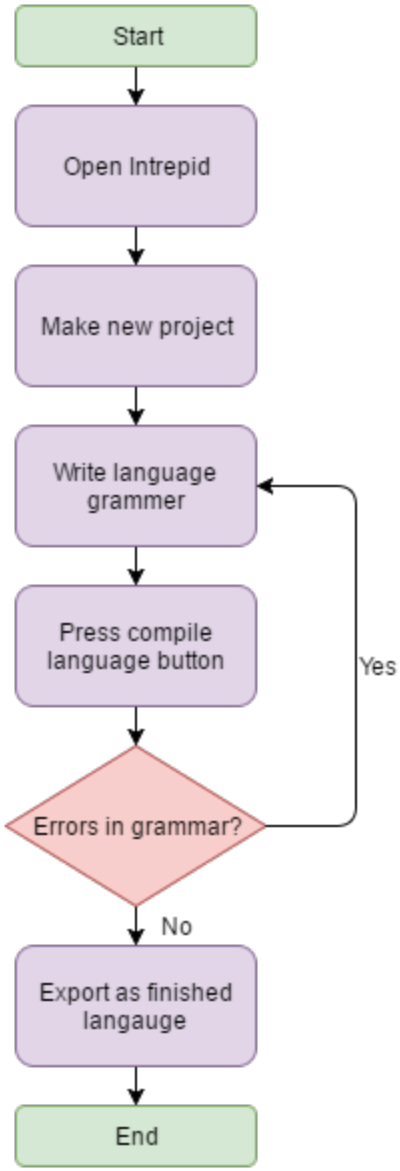



Figure 43: Flow diagram showing process of creating a lexer and parser for use in Intrepid IDE for a single DSL

## 6.9 Appendix 8 - Android Build Process

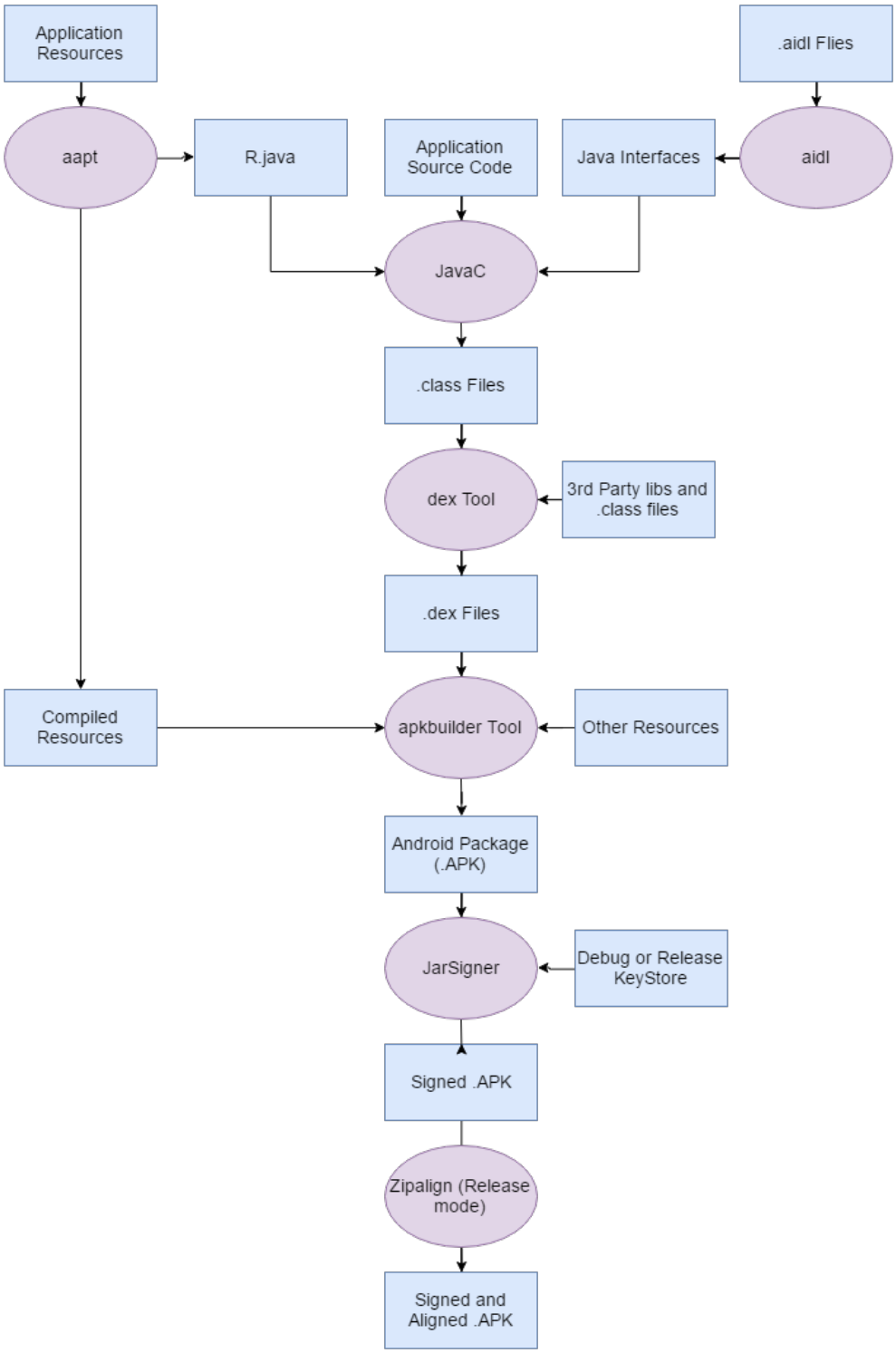



Figure 44: Android Build Process

## 6.10 Appendix 9 - Alternative Android Build Process

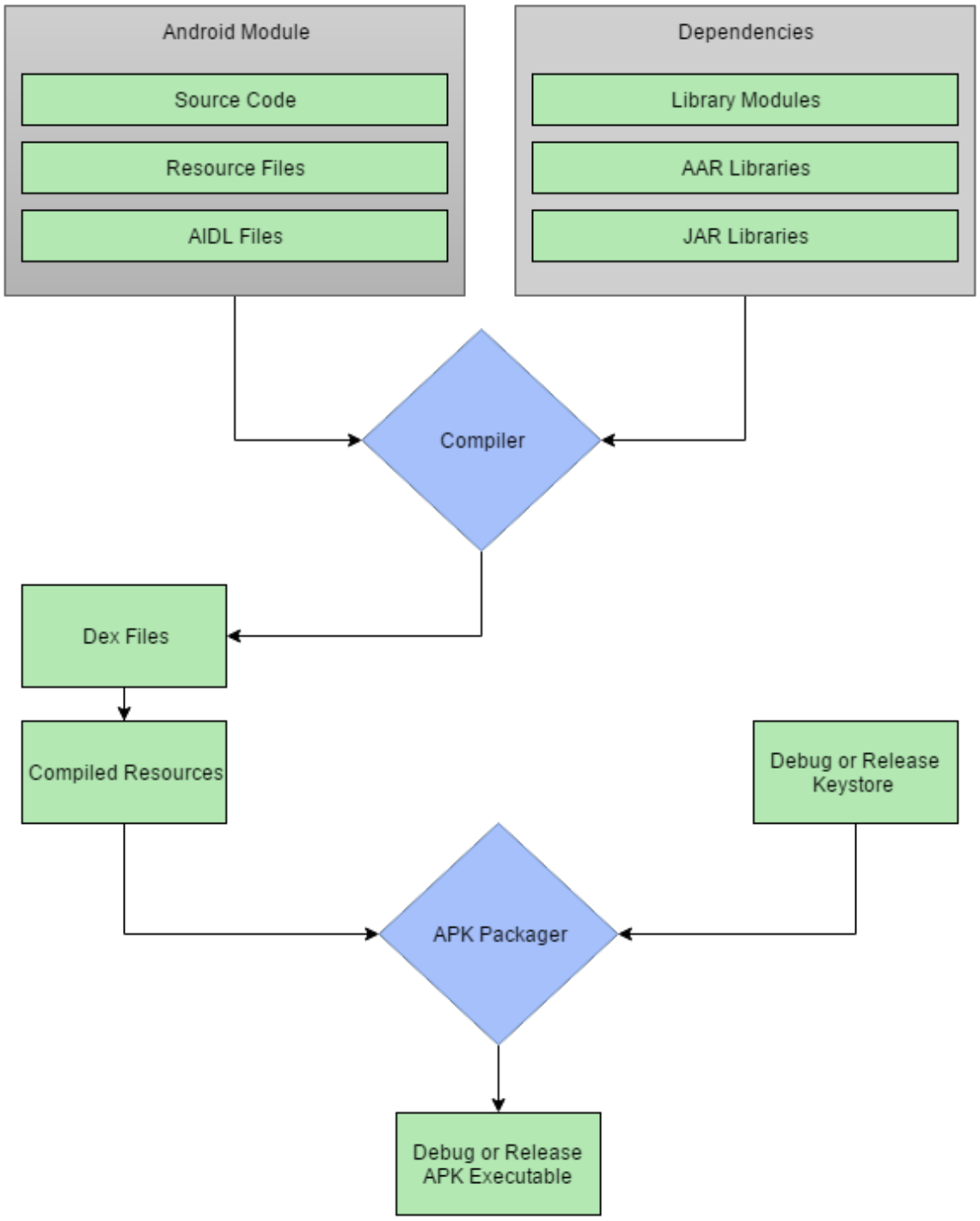



Figure 45: Android Build Process Alternative Diagram (Build, 2016)

## 6.11 Appendix 10 - Final Developer GUI

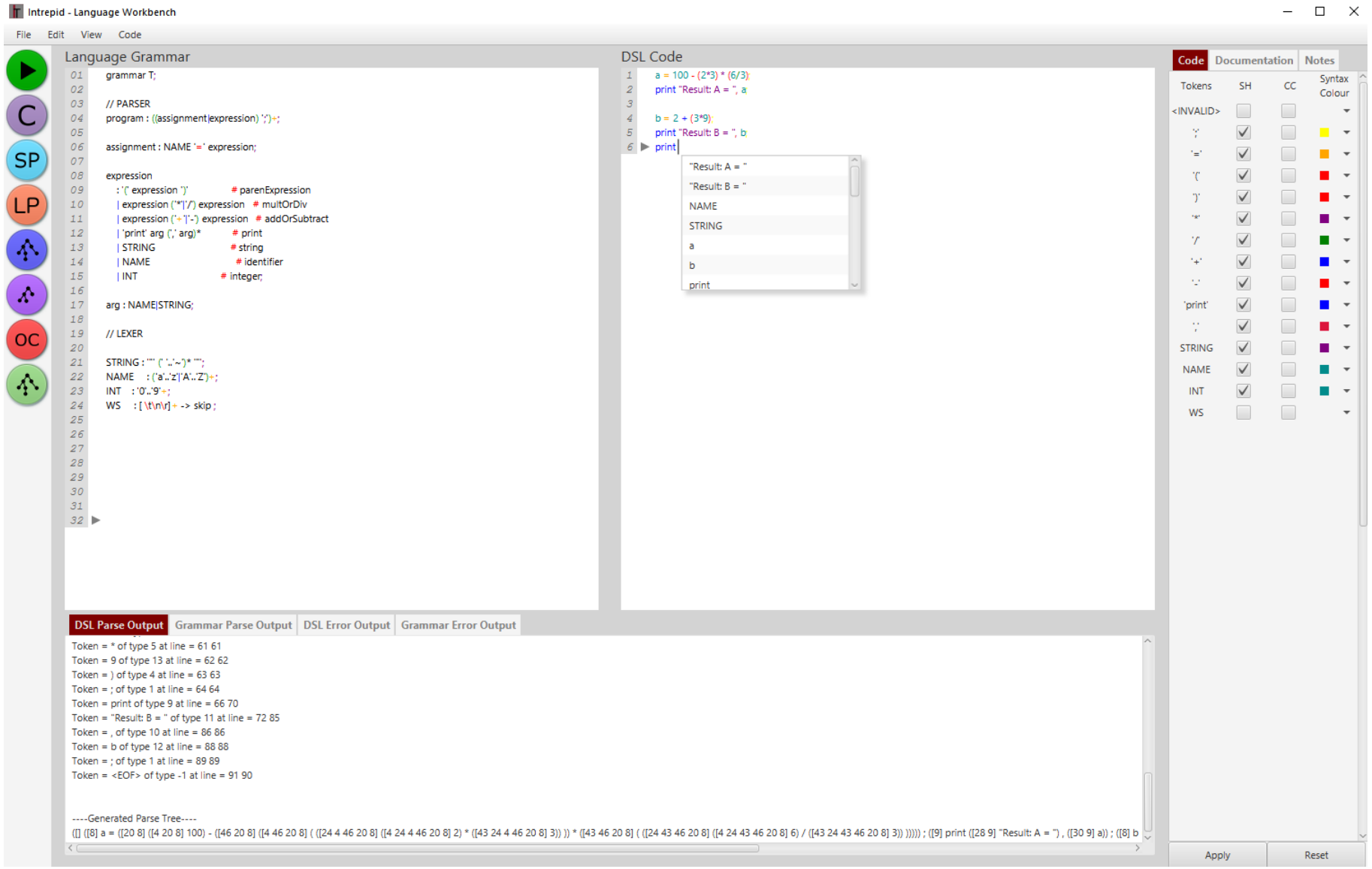



Figure 46: Final Developer GUI

## 6.12 Appendix 11 - Final User GUI

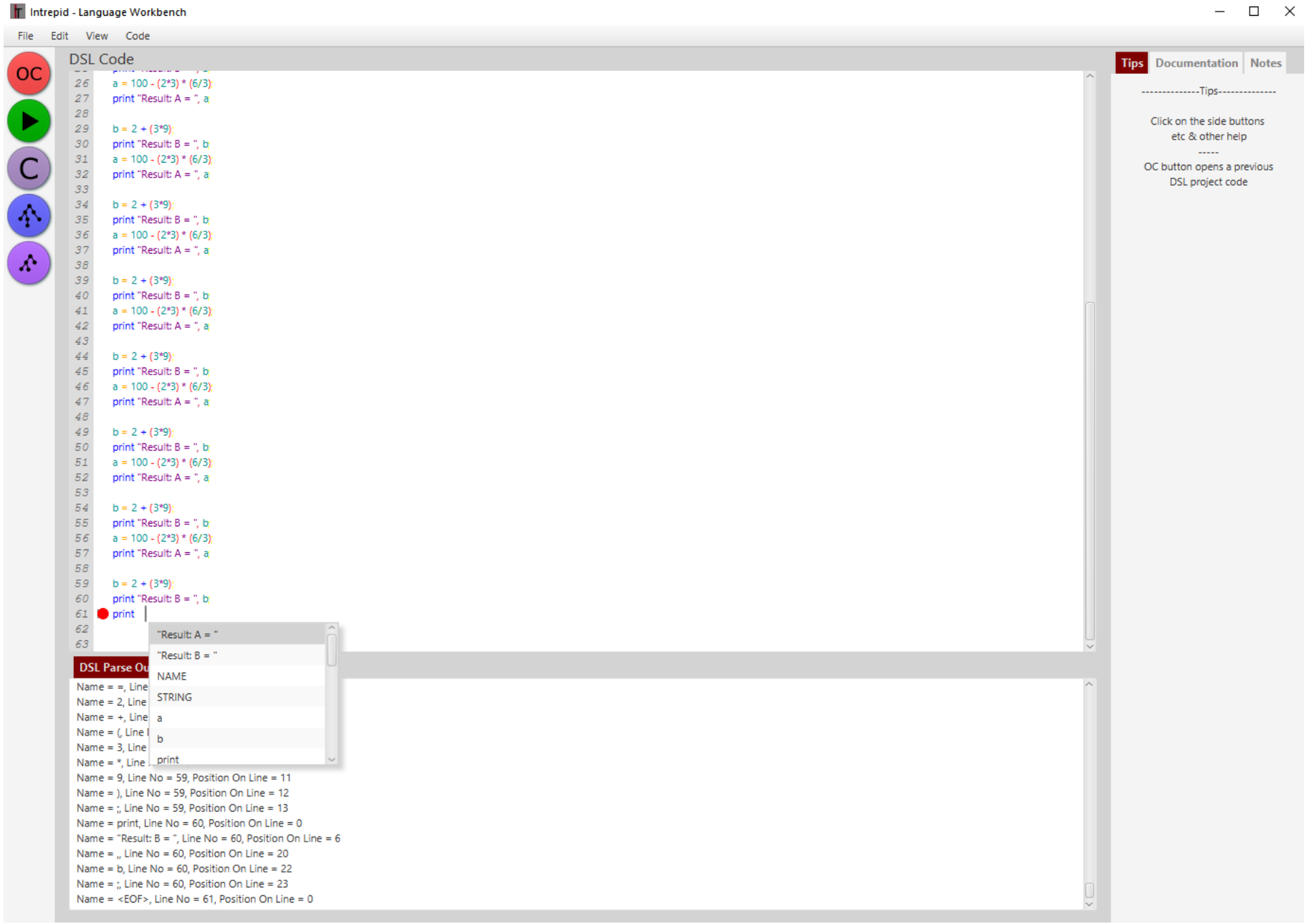
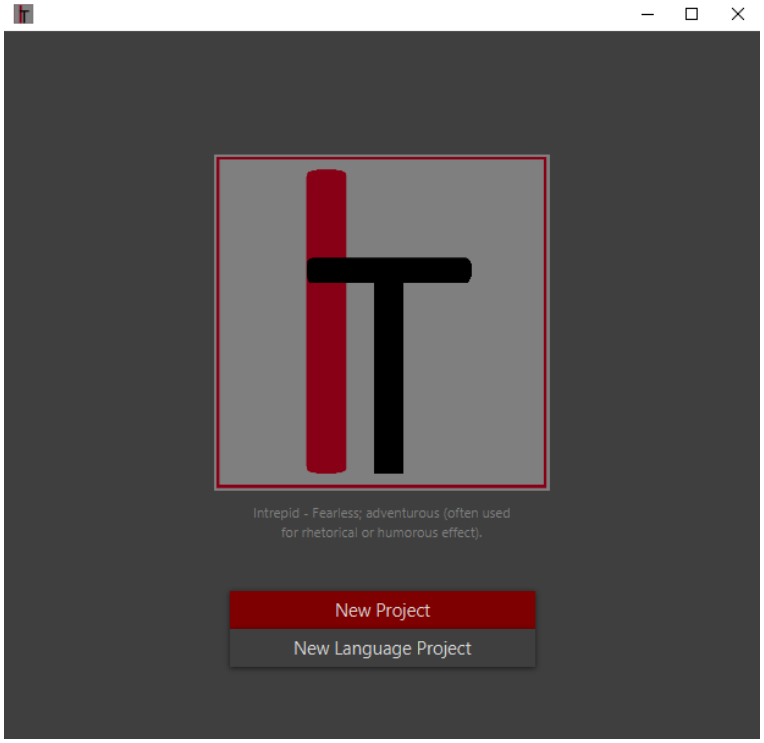



Figure 47: Final User GUI

## 6.13 Appendix 12 - User Guide

### 6.13.1 Introduction to Intrepid IDE - Language workbench

Welcome to Intrepid IDE, this program is intended to allow you to code a language in a simple EBNF grammar code and in the same program window then test the language. Furthermore it's possible to then export the language into the programs default language folder to be fully integrated in the program to provide a native IDE for that language. This user guide is split into two sections, one for language developers and one for language users.



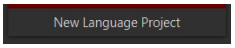

When Intrepid first starts, a screen will appear detailing two option buttons:

- **New Project** - This open will open the user mode of the program, allowing the user to write for a specific language.
- **New Language Project** - This open will open the developer mode of the program, allowing the developer to write a new language.

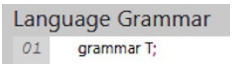
### 6.13.2 Language Developer - Creating a new language project

This window is intended to allow a developer to write an EBNF grammar on the left hand text area and then test it though programming the language on the right hand text area. To create a new project:

- Open Intrepid

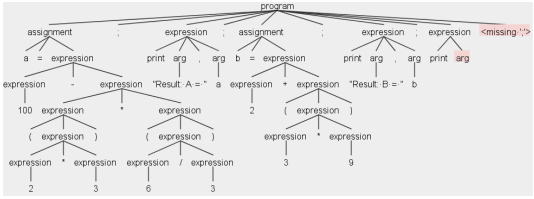



- Press create new new project.

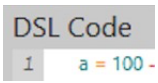



- Write EBNF code in the EBNF text area.

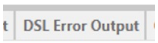
- Press the green compile language button.




- If EBNF errors occur, please refer to the EBNF error console tab to review output.

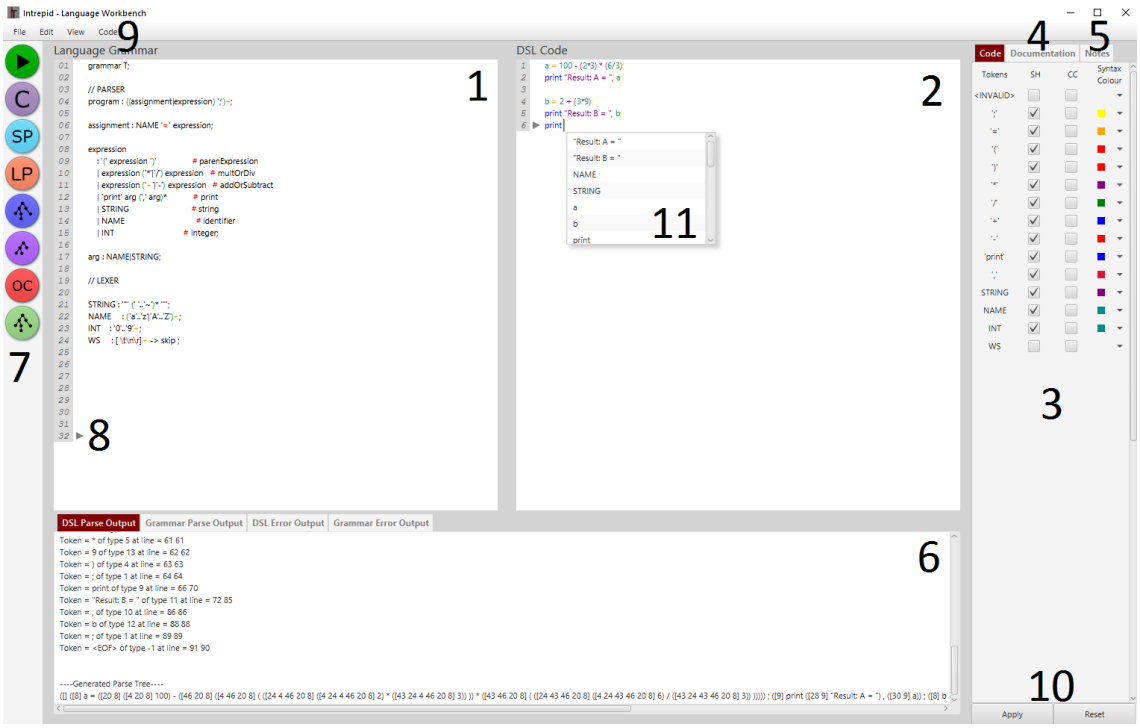



- Test language in the DSL text area.




- If error occur during parsing, check the DSL errors tab to inspect errors or choose the language tree view to see how the code was parsed.
- To save the project, please press file¿save project as. This will open a window to choose the directory to save the project.
- To export the DSL language to the list of included languages, press file¿export language option.

### 6.13.3 Intrepid Developer Tour




- **1 - Grammar Text Area**
  - This is the area in which you should code the EBNF code to produce the language. It's important to note that the grammar name for the language declared at the top of the program must be "grammar T;" and the entry point for the parsing must be named "program".
- **2 - DSL Test Text Area**
  - This is an area in which you can write an implementation of the language defined in the grammar text area. Syntax highlighting and code completion derived from your grammar will automatically be provided. If the language is intended to be deployed with this program to an end user, then this panel will simulate the experience the language end user will have.
- **3 - Syntax Highlighting Control Panel**
  - This is the area in which you can control what tokens in code are colours. The colour drop down box will allow you to pick a specific colour for that chosen token.
- **4 - Documentation Text Area**
  - This is a text area to write any help text and pointers that when deployed, the end user of the language will be able to read.
- **5 - Notes Text Area**

- This is a text area with private notes to aid in developing the language. This text will not be able to be read by the user of the language if deployed.

- **6 - Error / Parse Information Tabbed Panel**
  - These tabs will output parse information about how the code was parse, including how the EBNF was parsed. Furthermore it will also output a collated list of all errors in either the EBNF grammar or in the DSL test code.

- **7 - Quick Actions Button Bar**
  - This is a side panel containing the most used actions in the program for convenience. You check what each button does, please just hover over each button and a tool-tip pop-up will appear.

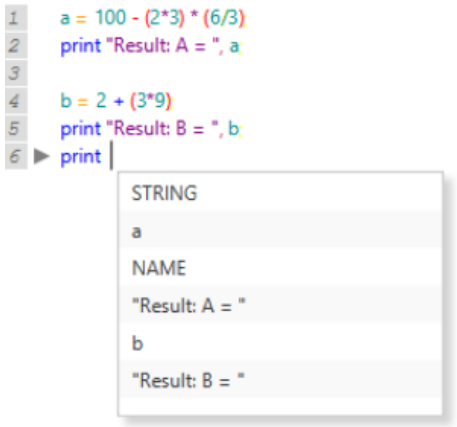

- **8 - Option Drop Down Menus**
  - The option drop down menus contain program wide options such as save/cut/copy and paste.


- **9 - Current Line Prompt**

- The current line prompt is a grey triangle to make it easy to see where the position of the caret is in the text panel. This is available in the Grammar text area and the DSL text area.

- **10 - Apply/Reset syntax highlighting preferences**
  - Apply and reset buttons will redraw the DSL text the colours specified in the colour control panel.

- **11 - Code Completion Pop-Up**
  - This is a pop-up intended to aid the user in writing code through suggesting useful bits of code based on what the user is currently typing. To inset the suggested code, the user has to double click the text they would like to be inserted.

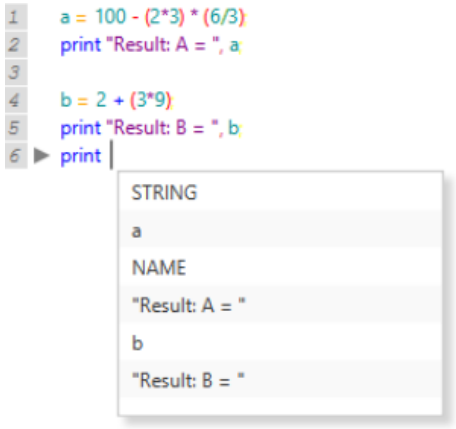


### 6.13.4 Language User - Getting started for writing in a language

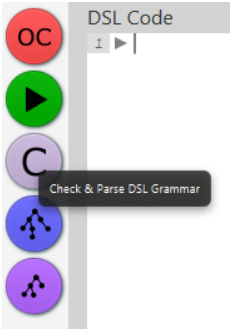
### 6.13.5 Language User - User window tour


- **1 - DSL Coding Text Area**
  - This is an area in which you can write an implementation of the language defined in the grammar text area. Syntax highlighting and code completion derived from your grammar will automatically be provided. If the language is intended to be deployed with this program to an end user, then this panel will simulate the experience the language end user will have.
- **2 - Parsing / Error output**
  - These two tabs contain information about how the code was parsed and any errors that were encounter in the code that was written.
- **3 - Tips Panel**
  - The tips panel contains useful quick tips to quickly navigate and use the program.

- **4 - Developer Made Documentation Panel**
  - The Developer made documentation panel contains documentation text written by the maker of the language being used. It should contain notes and help in relation to the semantics of the language currently being used.
- **5 - Quick Notes Panel**
  - The quick notes panel is a private area to write notes that could be useful whilst programming for the language.
- **6 - Code Completion Pop-up**
  - This is a pop-up intended to aid in writing code through suggesting useful bits of code based on what the user is currently typing. To inset the suggested code, the user has to double click the text they would like to be inserted.




- **8 - Quick Actions Button Bar**
  - This is a side panel containing the most used actions in the program for convenience. You check what each button does, please just hover over each button and a tool-tip pop-up will appear.




- **9 - Option Drop Down Menus**
  - The option drop down menus contain program wide options such as save/cut/copy and paste.


## 6.14 Appendix 13 - Ethics

Issues may arise from using code sources including books and the internet. Therefore any copied and modified code will be referenced to it's original author. Only code available through open source will be used through this process. In addition, no people other than the developer for this project will have input into the source code, except for possible feedback on the programs front end and functionality.

Some ethical issue have arising in the project development, these rate to open source external libraries. This project uses two external libraries, these include RichTextFX ( BSD 2-Clause License and GPLv2 with the Classpath Exception) and ANTLR (BSD License). Both of which are open source libraries, as such the relevant credit has been given in the project to these external solutions, and abide by the licences that both are open sourced under.

### 6.14.1 ANTLR Licence Notice (Antlr.org, 2016)

The BSD License Copyright (c) 2012 Terence Parr and Sam Harwell All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

Neither the name of the author nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIM-

ITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

#### 6.14.2 RichTextFX Licence Notice (Mikula, 2016)

opyright (c) 2013-2014, Tomas Mikula All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

## 6.15 Appendix 14 - Risk Analysis

Potential risks to the project are important to discover in order to more efficiently plan for them. As a result, the table below has been constructed to illustrate potential risks, and a plan of action in case they arise.

| **Risk** | **Severity (L/M/H)** | **Likelihood (L/M/H)** | **Significance (Sev. x Like.)** | **How to Avoid** | **How to Recover** |
|---|---|---|---|---|---|
| Data loss | H | M | HM | Keep Backups | Reinstate from backups |
| Loss of backups | H | L | HL | Multiple Backups | Use alternate |
| RSI | H | M | H | Regular Brakes | Hand Exercises |
| Eye Stain | H | M | M | Low blue light through LUX application and regular brakes | Take a break and look at long and short distance objects |
| Notable Illness | H | M | M | Take care of health such as diet over the project | Seek Doctor consultation immediately |
| Cognitive shortfall | H | L | M | Research Thoroughly | Reach a compromise, and relocate time needed to complete task |
| Lack of Time | H | L | H | Thoroughly research and understand the task before allocating times, therefore higher chance of allocating takes the correct amount of time | Reallocate tasks to free up time, introduce more efficient work flow. Drop optional features from the non critical objectives list |

| | | | | | |
|---|---|---|---|---|---|
| Lower than expected program performance | H | M | HM | Conduct performance testing throughout projects life cycle | Find more efficient and alternative ways to achieve features |
| Report writing takes more time than expected | H | H | H | Build report concurrently with other tasks, and plan out sections of the report | Alter time plan to allocate more time to task |
| Software used has copyright and/or law restrictions | H | L | M | Research technologies thoroughly | Allocate more time to find alternative technologies to achieve tasks |
| Cognitive shortfall | H | L | M | Research Thoroughly | Reach a compromise, and relocate time needed to complete task |

## 6.16 Appendix 15 - Simple Design Diagram

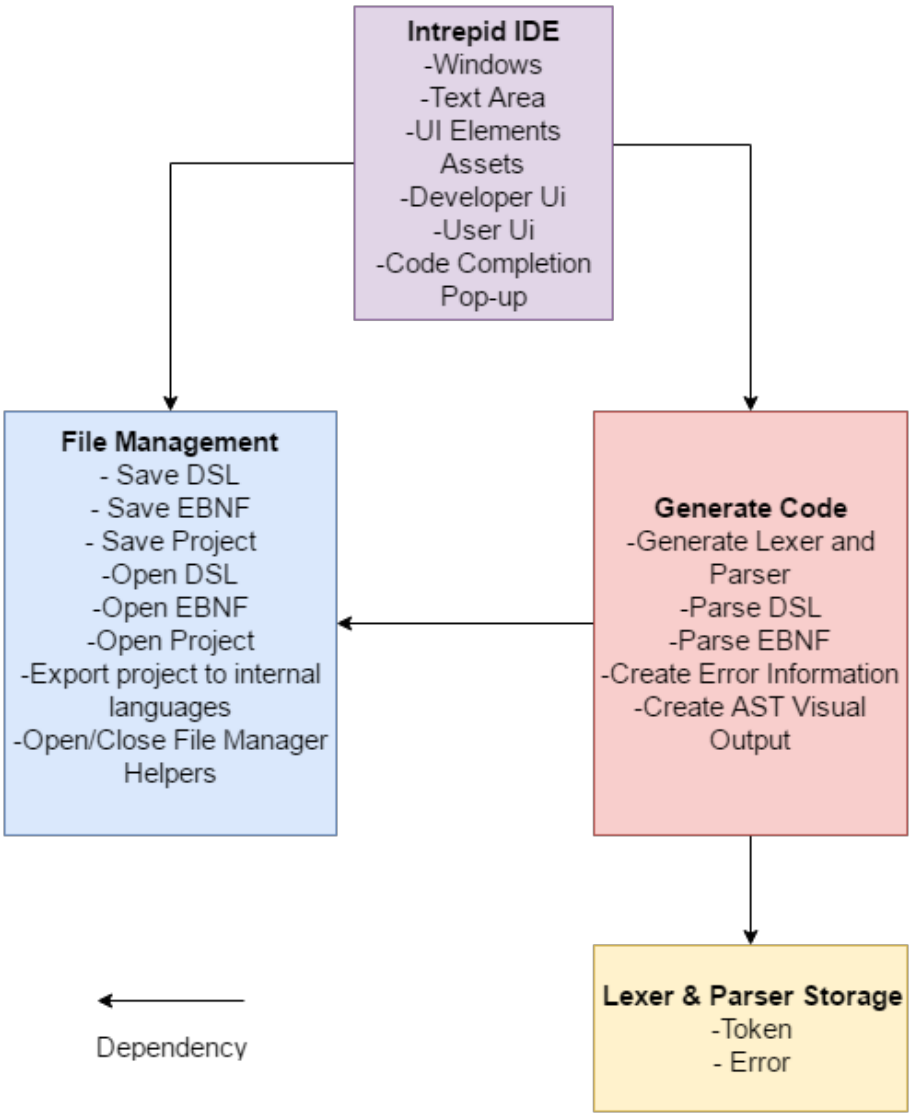



Figure 48: Diagram showing the basic architecture design of Intrepid

## 6.17 Appendix 16 - Code Map Diagram

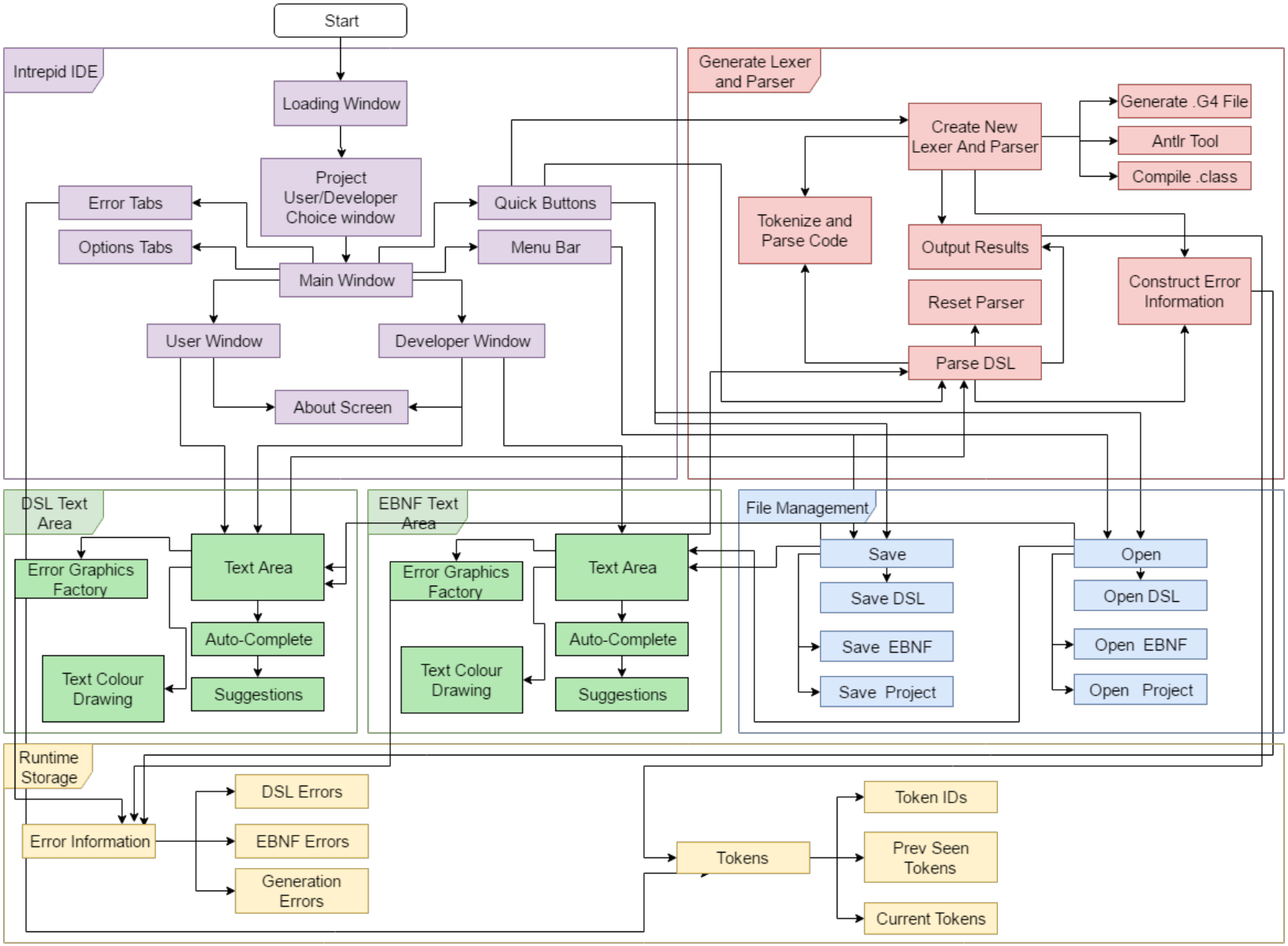



Figure 49: Code Map of Intrepid

# 7 References


A Hawick, K., Leist, A. and P Playne, D. (2010). Parallel graph component labelling with GPUs and CUDA. Parallel Computing, 36(12), pp.655-678.

Fowler, M. and Parsons, R. (2010) Domain-specific languages. 1st edn. United States: Addison-Wesley Educational Publishers.

Wikipedia, (2016). Comparison of parser generators. [online] Available at: https://en.wikipedia.org/wiki/Comparison_of_parser_generators [Accessed 19 Jan. 2016].

www2.in.tum.de, (2016). CUP2 User Manual. [online] Available at: http://www2.in.tum.de/~petter/cup2/#2.1.[Accessed10Jan.2016].

Oracle.com, (2016). A Swing Architecture Overview. [online] Available at: http://www.oracle.com/technetwork/java/architecture-142923.html [Accessed 19 Jan. 2016].

Anderson, G. and Anderson, P. (2016). 20 Reasons Why You Should Move to JavaFX and the NetBeans Platform — What Are JavaFX and the NetBeans Platform? — InformIT. [online] Informit.com. Available at: http://www.informit.com/articles/article.aspx?p=2273822 [Accessed 19 Jan. 2016].

Gnu.org, (2016). Bison - GNU Project - Free Software Foundation. [online] Available at: http://www.gnu.org/software/bison/ [Accessed 19 Jan. 2016].

Antlr.org, (2016). ANTLR. [online] Available at: http://www.antlr.org/ [Accessed 18 Jan. 2016].

Beaver.sourceforge.net, (2016). Beaver - a LALR Parser Generator. [online] Available at: http://beaver.sourceforge.net/ [Accessed 18 Jan. 2016].

Invisible-island.net, (2016). BYACC Berkeley Yacc generate LALR(1) parsers. [online] Available at: http://invisible-island.net/byacc/byacc.html [Accessed 18 Jan. 2016].

Javacc.java.net, (2016). JavaCC Home. [online] Available at: https://javacc.java.net/ [Accessed 18 Jan. 2016].

Google design guidelines, (2016). Elevation and shadows - What is material? - Google design guidelines. [online] Available at: https://www.google.com/design/spec/what-is-material/elevation-shadows.html [Accessed 17 Jan. 2016].

Levine, J. R. and Associates, I. S. (2009) flex and bison. 1st edn. United States: O'Reilly Media, Inc, USA

Boehm, B.W. (1988). A spiral model of software development and enhancement. Computer . Vol. Volume 21(Issue Issue: 5 ), 61-72.

D'Anjou, J. and Shavor, S. (2005). The Java developer's guide to Eclipse. Boston: Addison-Wesley.

D. Banker, R. and F. Kemerer, C. (1987). FACTORS AFFECTING SOFTWARE MAINTENANCE PRODUCTIVITY: AN EXPLORATORY STUDY 1. In: International Conference on Information Systems (ICIS). [online] AIS Electronic Library (AISeL), pp.160 - 175. Available at: http://aisel.aisnet.org/icis1987/?utm_source=aisel.aisnet.org%2Ficis1987%2F27&utm_medium=PDF&utm_campaign=PDFCoverPages [Accessed 6 Oct. 2015].

Sarkar, A. (2015). The impact of syntax colouring on program comprehension. In: PPIG 2015 - 26th Annual Workshop. [online] PPIG, pp.2-20. Available at: http://www.ppig.org/library/paper/impact-syntax-colouring-program-comprehension [Accessed 14 Oct. 2015].

V. Husselmann, A. (2014). Data-parallel Structural Optimisation in Agent-based Modelling.. Doctor of Philosophy. Massey University.

Mills, H. (1980). The management of software engineering, Part I: Principles of software engineering. IBM Syst. J., 19(4), pp.414-420.

Swaine, M. (2015). The Pragmatic Bookshelf — PragPub October 2009 — Language Workbenches. [online] Pragprog.com. Available at: https://pragprog.com/magazines/2009-10/language-workbenches [Accessed 13 Oct. 2015].

Walker, M (2015). Brief History of Programming Languages. [Online] http://intra.net.dcs.hull.ac.uk/student/modules/08348/Lecture%20Materials/08348%2002%20-%20Brief%20History%20of%20\Programming%20Languages.ppt [Accessed 10 Oct. 2015].

Tomsett, B (2015). 2 Lexical Analysis. [Online] http://intra.net.dcs.hull.ac.uk/ Available at: http://intra.net.dcs.hull.ac.uk/student/modules/08348/Lecture%20Materials/Notes%20-%20Compiling%20-%202%20Lexical%20Analysis.pdf [Accessed 10 Oct. 2015].

Tomsett, B (2015). 3 Parsing. [Online] http://intra.net.dcs.hull.ac.uk/ Available at: http://intra.net.dcs.hull.ac.uk/student/modules/08348/Lecture%20Materials/Notes%20-%20Compiling%20-%203%20Parsing.pdf [Accessed 10 Oct. 2015]

van der Storm, T. (2011). The Rascal Language Workbench. 1st ed. [ebook] Amsterdam: CWI, pp.2-28. Available at: http://oai.cwi.nl/oai/asset/18531/18531D.pdf [Accessed 11 Oct. 2015].

SearchSoftwareQuality, (2015). What is spiral model (spiral lifecycle model) ? - Definition from WhatIs.com. [online] Available at: http://searchsoftwarequality.techtarget.com/definition/spiral-model [Accessed 05 Oct. 2015].

Fowler, M. (2005). Language Workbenches: The Killer-App for Domain Specific Languages?. [online] martinfowler.com. Available at: http://www.martinfowler.com/articles/languageWorkbench.html [Accessed 05 Oct. 2015].

BusinessDictionary.com, (2015). What is software engineering? definition and meaning. [online] Available at: http://www.businessdictionary.com/definition/software-engineering.html[Accessed 13 Oct. 2015].

Softwareengineerinsider.com, (2015). What is Software Engineering? — A Common Question. [online] Available at: http://www.softwareengineerinsider.com/articles/what-is-software-engineering.html [Accessed 13 Oct. 2015].

Atlanta, J. and profile, V. (2009). Joseph Powell Mobile Apps Blog: IDE Advantages and Disadvantages. [online] Josephamospowell.blogspot.co.uk. Available at: http://josephamospowell.blogspot.co.uk/2009/09/ide-advantages-and-disadvantages.html [Accessed 13 Oct. 2015].

Bearcave.com, (2015). Why Use ANTLR?. [online] Available at: http://www.bearcave.com/software/antlr/antlr_expr.html [Accessed 14 Oct. 2015].

Saekar, A, (2015). The impact of syntax colouring on program comprehension [online] Available at: https://www.cl.cam.ac.uk/~as2006/files/sarkar_2015_syntax_colouring.pdf [Accessed 29 Jan. 2016].

Ssw.uni-linz.ac.at, (2016). The Compiler Generator Coco/R. [online] Available at: http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/ [Accessed 29 Jan. 2016].

W3schools.com. (2016). Browser Display Statistics. [online] Available at: http://www.w3schools.com/browsers/browsers_display.asp [Accessed 15 Apr. 2016].

Pal, K. (2016). How to get java class information using reflection. [online] Mrbool.com. Available at: http://mrbool.com/how-to-get-java-class-information-using-reflection/28860 [Accessed 24 Apr. 2016].

Antlr.org. (2016). ANTLR 4 Tool 4.5.3 API. [online] Available at: http://www.antlr.org/api/JavaTool/index.html [Accessed 18 Apr. 2016].

Mcmanis, C. (2016). The basics of Java class loaders. [online] JavaWorld. Available at: http://www.javaworld.com/article/2077260/learn-java/learn-java-the-basics-of-java-class-loaders.html [Accessed 27 Apr. 2016].

GitHub. (2016). TomasMikula/RichTextFX. [online] Available at: https://github.com/TomasMikula/RichTextFX [Accessed 28 Apr. 2016].

www.idc.com. (2016). PC Market Finishes 2015 As Expected, Hopefully Setting the Stage for a More Stable Future, According to IDC. [online] Available at: https://www.idc.com/getdoc.jsp?containerId=prUS40909316 [Accessed 14 Apr. 2016].

Keizer, G. and Keizer, G. (2016). Horrendous PC shipment decline in 2015 isn't going to end anytime soon. [online] PCWorld. Available at: http://www.pcworld.com/article/3012553/computers/horrendous-pc-shipment-decline-in-2015-isnt-going-to-end-anytime-soon.html [Accessed 16 Mar. 2016].

Androidsrc.net. (2016). [online] Available at: http://androidsrc.net/wp-content/uploads/2015/01/build.png [Accessed 15 Apr. 2016].

Build, C. (2016). Configure Your Build — Android Studio. [online] Developer.android.com. Available at: https://developer.android.com/studio/build/index.html [Accessed 18 Apr. 2016].

The Khronos Group. (2016). OpenCL - The open standard for parallel programming of heterogeneous systems. [online] Available at: https://www.khronos.org/opencl/ [Accessed 28 Apr. 2016].

WhatIs.com. (2016). What is machine code (machine language)? - Definition from WhatIs.com. [online] Available at: http://whatis.techtarget.com/definition/machine-code-machine-language [Accessed 28 Apr. 2016].

GitHub. (2016). antlr/grammars-v4. [online] Available at: https://github.com/antlr/grammars-v4/blob/master/java/Java.g4 [Accessed 28 Mar. 2016].

Anon, (2016). [online] Available at: http://www.cl.cam.ac.uk/~mgk25/iso-14977-paper.pdf [Accessed 28 Feb. 2016].

Seas.upenn.edu. (2016). CSC 4170 Context-Free Grammars. [online] Available at: http://www.seas.upenn.edu/~cit596/notes/dave/cfg0.html [Accessed 10 Mar. 2016].

Meduna, A. (2014). Formal Languages and Computation. [Place of publication not identified]: Auerbach Publications.
Wikipedia. (2016). Formal grammar. [online] Available at: https://en.wikipedia.org/wiki/Formal_grammar [Accessed 28 Feb. 2016].

Wikipedia. (2016). Unrestricted grammar. [online] Available at: https://en.wikipedia.org/wiki/Unrestricted_grammar [Accessed 10 Feb. 2016].

Anon, (2016). [online] Available at: https://chomsky.info/wp-content/uploads/195609-.pdf [Accessed 28 Feb. 2016].

Docs.oracle.com. (2016). java.lang.reflect (Java Platform SE 7 ). [online] Available at: https://docs.oracle.com/javase/7/docs/api/java/lang/reflect/package-summary.html [Accessed 28 Feb. 2016].

Docs.oracle.com. (2016). Trail: The Reflection API (The Java Tutorials). [online] Available at: https://docs.oracle.com/javase/tutorial/reflect/ [Accessed 28 Mar. 2016].

Safari. (2016). 4. Java Tools [Book]. [online] Available at: https://www.safaribooksonline.com/library/view/client-server-web-apps/9781449369323/ch04.html [Accessed 10 Mar. 2016].

Java.com. (2016). What is Java and why do I need it?. [online] Available at: ://java.com/en/download/faq/whatis_java.xml [Accessed 12 Feb. 2016].

Oracle.com. (2016). The Java Language Environment. [online] Available at: http://www.oracle.com/technetwork/java/intro-141325.html [Accessed 11 Feb. 2016].

Anon, (2016). [online] Available at: https://eclipse.org/jgit/documentation/ [Accessed 10 Jan. 2016].

Antlr.org. (2016). ANTLR v4 License. [online] Available at: http://www.antlr.org/license.html [Accessed 29 Mar. 2016].

Mikula, T. (2016). TomasMikula/RichTextFX. [online] GitHub. Available at: https://github.com/TomasMikula/RichTextFX/blob/master/LICENSE [Accessed 29 Mar. 2016].

Intel ARK (Product Specs). (2016). Intel Xeon Processor X5650 (12M Cache, 2.66 GHz, 6.40 GT/s Intel QPI) Specifications. [online] Available at: http://ark.intel.com/products/47922/Intel-Xeon-Processor-X5650-12M-Cache-2_66-GHz-6_40-GTs-Intel-QPI [Accessed 1 May 2016].

Intel ARK (Product Specs). (2016). Intel Core i5-4690K Processor (6M Cache, up to 3.90 GHz) Specifications. [online] Available at: http://ark.intel.com/products/80811/Intel-Core-i5-4690K-Processor-6M-Cache-up-to-3_90-GHz [Accessed 1 May 2016].

Intel ARK (Product Specs). (2016). Intel Core2 Quad Processor Q9550 (12M Cache, 2.83 GHz, 1333 MHz FSB) Specifications. [online] Available at: http://ark.intel.com/products/33924/Intel-Core2-Quad-Processor-Q9550-12M-Cache-2_83-GHz-1333-MHz-FSB [Accessed 1 May 2016].