School of Computer Science
The University of Manchester
Oxford Road, Manchester
M13 9PL, UK

A GUI FOR THE DESIGN AND REPRESENTATION OF GRAMMARS

Author: A.J. Ashton
Computer Science BSc
Project Report
May 2010

Supervisor: P.J. Jinks

Abstract

Title: A GUI for the Design and Representation of Grammars
Two problems are identified with the use of grammars. A new idea and unfamiliar concepts can present many problems to the inexperienced student. Secondly, their use in computer science to describe programming languages leads to large and complex grammars. There are also a lack of tools available to support their understanding and development.

A novel way of visualising grammars is introduced, accompanied by an implementation in a program. The visualisation uses boxes to represent rules and they are nested to utilise a grammars hierarchical property. The program provides an enviornment to browse grammars written in the popular Backus-Naur Form (BNF). Expanding/collapsing boxes allow rules to be examined with ease. In addition, the visualisations can be used to develop grammars.

The representation is clear and easy to understand, adding a level of abstraction away from the variations in BNF notation. The design enviornment is intuitive to use allowing users to comfortably write or extend their grammars.

Acknowledgements

I would like to extend my deepest gratitutde to Pete Jinks, my supervisor, whose vast knowledge and champion support has influenced this project so much.

I am also grateful for the love and support of my family. A sure truism when I say I love you all.

Last but not least I pay tribute to my friends, especially to whom helped me during the project. I bestow a quintessential thank you.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER

ONE

INTRODUCTION

1.1 Background

LANGUAGES are made up of a set of words and a set of rules which define how the words can be written. *Text* is the written language which should conform to the syntax of the language (e.g. a sentence). A *language syntax* can be thought of as a set of rules to describe a language. A *grammar* describes the syntax for a specific language. In computer science, grammars are used to define the structure of programming languages. To *parse* a language means to check the text is written in a way that conforms to the syntax. Compilers parse text (source code) using the languages grammar to check it is correct. Grammars therefore are used by anyone who designs or extends a programming language. They are not limited to the design of programming languages—flat-files which contain databases data are described using a grammar [25]. Programming language reference manuals are used to understand the syntax of a language. Reference manuals describe the grammar. Programmers therefore are studying its grammar [21].

The scope of this project only considers grammars for programming languages and not that of natural language. From now on, *language* will refer to a programming language that has a formal structure and is described by a formal grammar. A *formal grammar* was introduced and generalised by Noam Chomsky in [4]. *Grammar* will now refer to this formal grammar. Backus-Naur Form (BNF) and its derivatives is a popular notation for describing formal grammars in computer science [48]. Development of grammars in computer science is discussed in chapter 2.

1.2 Motivations

Two problems have been identified when using grammars. Firstly, the learning of grammars. Understanding the concept of a grammar and what it is used for can be a difficult task as it is a new idea to students. There is little evidence of teaching tools that exist to help students understand and construct simple grammars (see Chapter 2).

There are only a few ways grammars are represented, predominantly in textual form where rules are listed in a linear fashion. Grammars for a typical language are quite large, involving a considerable number of rules each having many choices and sub-rules. Providing a way to understand and browse these larger grammars is needed.

1.3 Proposed solution

A novel way of representing grammars using a visual approach is described here. This visualisation will then be implemented in a program to become an effective teaching tool as well as enabling easy browsing of typically large grammars.

1.3.1 Grammar representation

A grammar is a set of rules. These rules are headed by a rule name. The rule body is what makes up the rule. The head and body are separated by what is called an assignment operator, denoted by '='. A terminal is the literal text that can be written in the grammar. A nonterminal relates to another rule in the grammar. A terminal is encapsulated in single quotes to distinguish it from nonterminals. ';' denotes the end of a rule. *Symbol* refers to either a terminal or nonterminal. Concatenation, that is what symbols can appear next to each other is denoted by ','. The start symbol is the starting rule for a grammar, it is usually the first rule. An example of a written grammar is shown in listing 1.1. An example piece of text conforming to the grammars syntax is 'abcb'.

The solution proposed is a visualisation of grammars. It involves a series of boxes within boxes. This mirrors the hierarchical structure of a grammar. A visualisation of the grammar above is shown in Figure 1.1. The start symbol *S* is the outer box, and as you look within each box, a description of that rule is detailed. The represen-

```
S = 'a', B, 'c', T;
T = B;
B = b;
```

Listing 1.1: Example of a grammar.

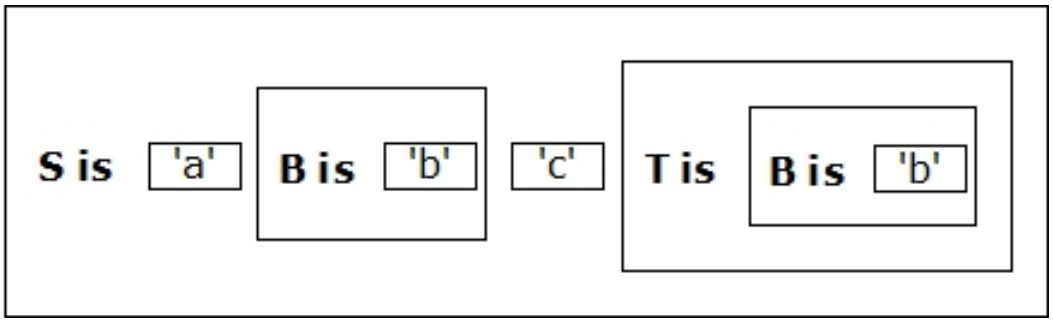

Figure 1.1: Representation of a simple grammar

tation in Figure 1.1 attempts to visualise the grammar in a more natural way. This presents some problems when trying to visualise certain aspects of a grammar. How would one represent choice or recursive rules? In the example, concatenated items appear left to right, one after another. Therefore choice could to be represented as rows of concatenated items, each choice on a new line. Repeating rules could be shown as closed boxes as to avoid visualising the rule indefinitely. The following grammar:

```
S = 'a', B, 'c', S | 'abc';
B = b;
```

can be visualised as shown in Figure 1.2.

This representation could aid students in learning about grammars, providing an alternate view. Visualisations are rather large compared to the textual format. This can lead to large visualisations and will need to be addressed when visualising typical grammars.

1.3.2 The program

Providing a dynamic and interactive learning environment to teach grammars is desirable. Visualisations on their own are not enough to facilitate the learning of grammars, and clearly not suitable for representing large grammars. The solution then is

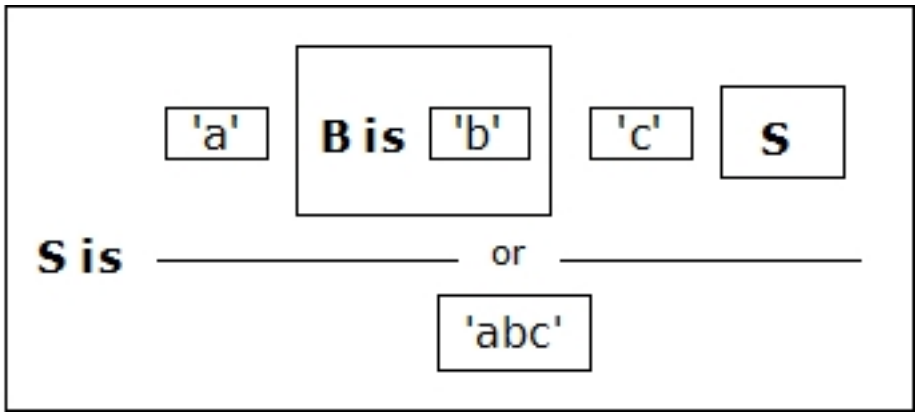

Figure 1.2: Representation of recursion and choice in a simple grammar

to create a program that will implement the given representation and add interactivity. The high level goals of the program can only be accepted if we can understand the users goals. There are two distinct users of my program:

i. The student, who does not understand how grammars work. Their goal is to grasp the concepts contained within grammars.

ii. The competent user, who understands grammars but struggles to comprehend grammars of typical languages which are large and time-consuming to read.

The high level goals of the program are as follows:

- Visualise a grammar and its rules.
- Interactive visualisation where rules can be selected and explored.
- An IDE or design environment where a grammar may be built from scratch or extended.
- Parse an existing grammar in order to visualise it.
- Export the grammar back to its original notation.

1.4 Similar work

There are tools in existence that visualise grammars in some way. Their aim is to help users browse and understand grammars. The tools are briefly mentioned here

and some are analysed in more depth in Chapter 2.

1.4.1 Visualising grammars

YACCVISO [23] is a program which visualises grammars using the classic nodes and connected edges seen in graph drawings.

1.4.2 Teaching tools

A suite of "Visual and Interactive Tools" are being developed by S. H. Rodger [39] in the area of grammars and automata. An automaton is an abstract machine that moves between allowable states. These states could be thought of as words in a language. Therefore, their use is closely related to formal grammars and language design. JFLAP [37] and PÂTÉ [38] have been identified as two tools which involve, but are not limited to visualising the parsing of text using a grammar.

LISA [6] and Visual BNF [13] provide IDEs to design and generate grammars. Both programs generate libraries that will parse your language once the grammar is developed. They both offer visualisations of the grammar and LISA offers simulations of text being parsed.

1.4.3 Managing typical grammars

The BNF Web Club [24] is a research group providing techniques to help users manage and understand typically larger grammars. They provide a web application that lets users:

> "Browse and explore some of your favourite programming languages syntactic rules. See relations between the rules, understand them using both BNF notation and syntactic diagrams"

Their website creates a series of pages each corresponding to the different rules of a given grammar. On each page the description of a rule is detailed and a syntactic diagram is generated. Hyperlinks on these pages make it easy to browse different rules. This technique including syntactic diagrams is discussed in chapter 2.

1.5 Chapter summaries

Chapter 2

The development of grammars in computer science, motivations and a detailed look at some similar work.

Chapter 3

Clarifies the requirements of the project.

Chapter 4

Design decisions made and an overview of the data structures needed.

Chapter 5

A detailed look at how the data structures were implemented and how the visualisations were generated.

Chapter 6

Some results and screen shots from the program.

Chapter 7

We show the program meets its requirements and present a framework for a more complete testing strategy.

Chapter 8

A look at the achievements, improvements and future work of the project.

CHAPTER

TWO

BACKGROUND AND RESEARCH

2.1 Grammars

2.1.1 Why

Every language needs a formal structure otherwise the precise meaning or understanding is lost as the language is ambiguous. We use a grammar to define the syntax of the language such that we can know what is a legal statement and understand what it means. Compilers use a grammar to check the syntax is correct and then to understand the meaning of what was written and compile a correct implementation.

2.1.2 Development

Noam Chomsky described a way of formally describing a natural language by defining a set of rules which can be applied over a string of words. In his paper [5] he introduces the formalisation of a grammar and different types of grammars, called the 'Chomsky Hierarchy'. The hierarchy imposed restrictions on how the rules could be written. It is the type 2 or 'context-free grammar' that is normally used to describe computer languages. BNF is essentially a notation for a context-free grammar and was first publicly introduced to describe the language Algol [36]. Many derivatives of BNF are used to define languages [48].

2.1.3 BNF notations

The first version [36] had the following syntax:

```
<operator> ::= <arithmetic operator>|<logical operator>
<arithmetic operator> ::= + | - | * | /
<logical operator> ::= = | !
```

Nonterminals are encapsulated in angle brackets to distinguish them from terminals. *Symbol* refers to either a nonterminal or terminal item in a grammar. Concatenation (not shown) occurs when two symbols are adjacent - there is no operator. Choice between symbols is denoted by the '|'.

Attempts have been made to standardise the notation resulting in many different versions of BNF. Ironically, the notation used to describe a language does not itself have a definitive syntax. For more information you are referred to [32] for a list of different BNF and Extended BNF (EBNF) notations.

Regular expressions are a way in which we can match strings. Regular expressions have a concise notation that is used to define what we want to match. For more information, you are referred to [42]. EBNF attempts to make BNF more readable by introducing some regular expression techniques in order to reduce the 'clutter' in BNF. Frequent concepts can be utilised with regular expression symbols. For example,

```
S = 'a', B | B;
B = B, 'b' | 'b';
```

could be written in EBNF as,

```
S = ['a'], B;
B = ('b')+;
```

Here '[]' is used for optional items, '+' can be used for 'one-or-more repetition' of symbols. Interested readers may want to refer to N. Wirth's article [48] which first introduced all the concepts from which EBNF notations derive.

The internationally standardised syntax for EBNF is defined by the 'International Organization Standardardization' in the paper ISO/IEC 14977 [12]. This report will give all examples of grammars in this notation. A grammar detailing some of the most used notation is shown in listing 2.1 for your reference. Note the use of ',' to denote concatenated symbols, ' to enclose terminal symbols and ';' to denote the

```
Integer = FirstDigit, Digits;
Digits = '' | Digits, Digit;
(* Integers can't start with a zero *)
FirstDigit = '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';
Digit = '0' | FirstDigit;
```

Listing 2.1: A BNF grammar conforming to the ISO standard.

end of a rule.

2.2 Motivations

This section will detail my motivations and reasoning behind an interactive program for grammar visualisation and design.

2.2.1 Why visualise?

> "The ways people receive information are divided into three categories: visual - sights, pictures, diagrams; auditory - sounds, words; kinaesthetic - taste, touch and smell. Visual learners remember best what they see: pictures, diagrams, flow charts, time lines and demonstrations" [9].

People of university age and older are usually categorised as visual [9]. In [9], Felder describes how to teach "complex processes or algorithms" to students that are visual by using diagrams and information flow charts. Felder also states that topics in general should visualise working processes whenever possible. A *working* grammar is the process of parsing some text. The teaching tools studied in the following chapter all demonstrate this working process. As this area has been covered, we will be looking into developing a way of visualising the grammar before it has started parsing data, helping users to write their first grammars with a visual IDE.

Further research suggests it is not visualisations that have the greatest impact but the level of learner involvement which can benefit students. An analysis of twenty-one experimental studies into learning effectiveness in algorithm visualisation showed that the most successful learners are those "in which the technology is used as a vehicle for actively engaging students" [10]. In [30], they summarise the findings of [10] and find that from the studies which just manipulate representations only three out of nine show a significant result in learning effectiveness. In contrast

to those having more of a focus on learner engagement, ten out of the twelve studies showed significant results. One of the ways to engage the students is to enable them to construct their own visualisations [30]. Providing the ability to build a visualised grammar and giving the user direct access to manipulate and change the visualisations will have a significant effect on learning than just standalone representations.

The writer notes that teaching tools have been beneficial to their learning throughout academic life. It is hoped this project will be successful so students can use this program to aid their learning of grammars.

2.2.2 Managing grammars

To follow from section 1.2, grammars can be large and hard to understand at first. For example, a grammar describing the syntax of the C programming language (ANSI Standard, 1985) is close to 900 lines of code [18]. Moreover, "languages are ever evolving" [20] and the area of language and language development in computer science is described as "lively" [21].

To add support for new features of these growing languages, the grammars that define these languages needs to be extended. There are few tools that give a representation for a large grammar and allow the grammar to be edited direct from the visualisation. Furthermore, the various notations of BNF are only slightly different and somewhat annoying as there is no uniformly used standard [48]. It is hoped that users of typically large grammars can use this program to help maintain and extend existing grammars. The program will add a level of abstraction which will take users away from dealing with BNF notation.

A grammar can be thought of as a hierarchical structure of rules. A study of participants that were tasked with browsing a large hierarchical data structure (1296 nodes) found that an expand and collapse interface was six times quicker than the a 'stable' interface which showed every node [3]. Providing the expand/collapse interface for rules will be a beneficial addition.

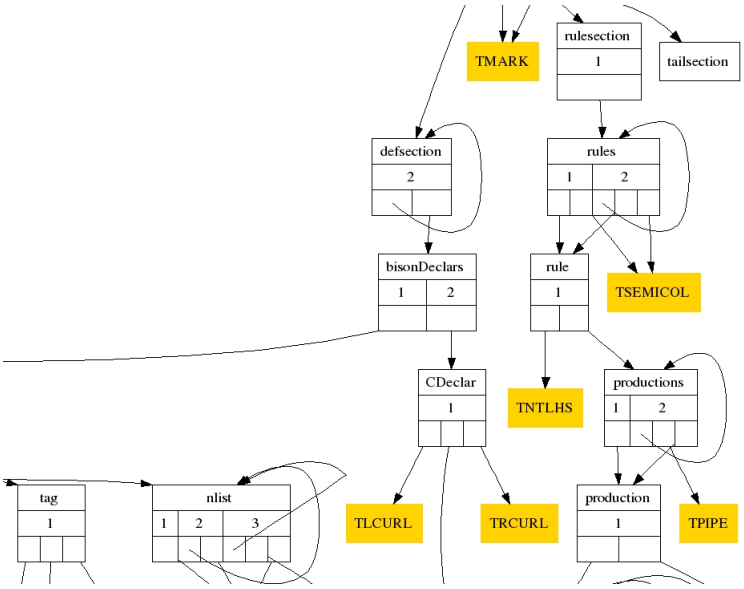

Figure 2.1: An excerpt from a drawing of a grammar in YACCVISO.

2.3 Similar work

2.3.1 Treemaps

A *treemap* is a way of visualising hierarchies in a 2D space. Introduced by Ben Shneiderman in 1991 [15], they were originally intended to show compact visualisations of file systems. Nowadays treemaps are a very popular way of visualising hierarchical data and there are many applications [40]. The solution proposed here for representing grammars is similar to 'nested treemaps'. Since grammars are recursive their application in treemaps is not yet known.

2.3.2 YACCVISO

YACCVISO [23] generates graph drawings of grammars written in YACC [16] or Bison [8]. The graphs can be imported into drawing programs to visualise, manipulate or "play around with" [19]. A excerpt from a drawing is shown in figure 2.1. The numbers represent the choice and inside the choice it represents the concatenation. Each box points to either another rule or terminal (yellow box).

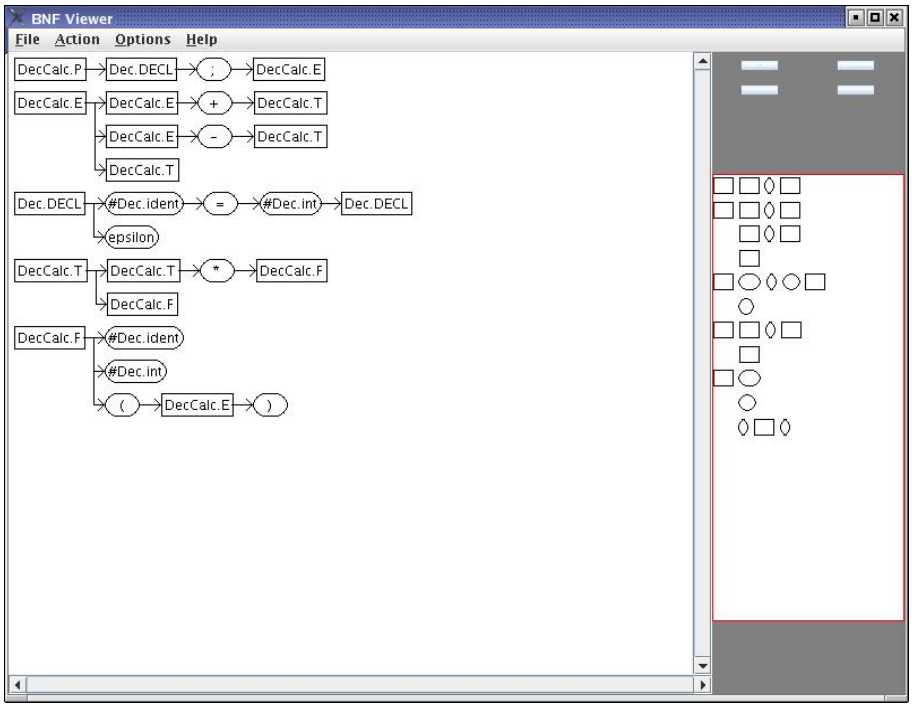

Figure 2.2: The BNF Viewer in LISA

2.3.3 Teaching tools

The tools LISA [6], JFLAP [37] and Visual BNF [13] all have very similar goals and it would be pointless analysing all of them here. They are all attempting to aid users in language development by providing an IDE to help develop grammars. Features such as representing the grammar as automata are used as well. Pâté [38] is a tool that concentrates on grammars and parsing the grammar. LISA and Pâté will been studied below.

LISA offers an IDE to developing a language that can then be used in a Java program. It offers an interactive environment for language development and includes many tools that help the user understand what they have created. LISA has an impressive set of features including generating a parsing tool that can be used in a Java program. The ability to visualise parsing by stepping through a series of commands and equivalent visualisations of a grammar in automata is also available. The ability to design a language in a visual manner is enabled by the use of a finite state automata diagram in which the user can edit the representation. The program can automatically generate a parser for use in programs. The grammar can be presented

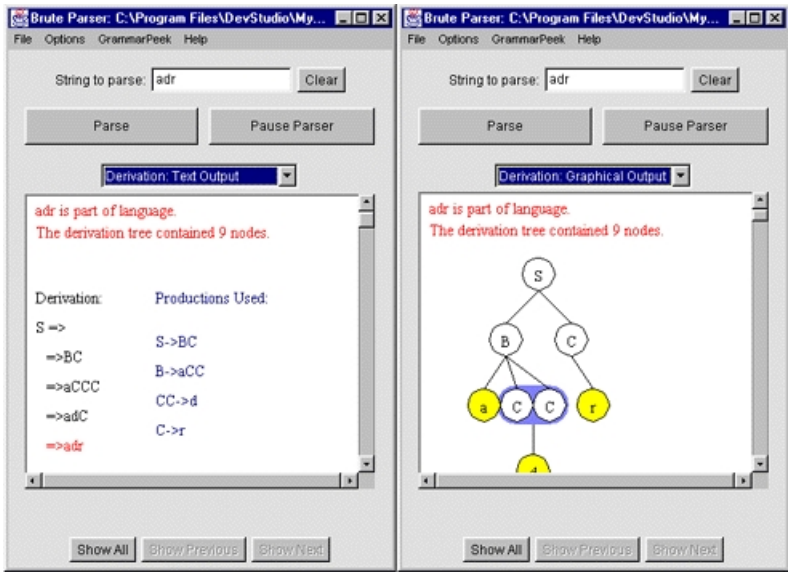

Figure 2.3: Pâté teaching tool demonstrating the parsing of string ‘adr’.

in different formats including BNF, syntax tree, semantic tree and a dependency graph. It is the BNF visualisation that is interesting as this project is looking at visualising grammars written in BNF. See Figure 2.2 where the BNF viewer is helping a user understand how their rule is syntactically defined. The BNF viewer can only be used for individual rules and not the whole grammar. It also shows the used rules within the image from top to bottom, thus understanding a large rule that uses many nonterminals could use a lot of space. LISA aims to provide an all-in-one development environment for language design and as such you must start your language within LISA as the syntax to describe the language is not a recognised standard. The ability to extend a grammar written in another BNF notation is not supported without heavily manipulating your source file to conform to the syntax.

Pâté is a tool belonging to the suite of “Visual and Interactive Tools”. Interested readers are referred to [39] for a list of all the tools and publications which includes JFLAP and Pâté. A excerpt from the website [38] reads:

> “Pâté is a visual and interactive tool for parsing and transforming grammars.”

Pâté is a teaching tool that gives students a clear insight into grammars and how they parse small input. It can show textual or graphical visualisation derivations for a given grammar. Again the user is assumed to have a simple grammar ready

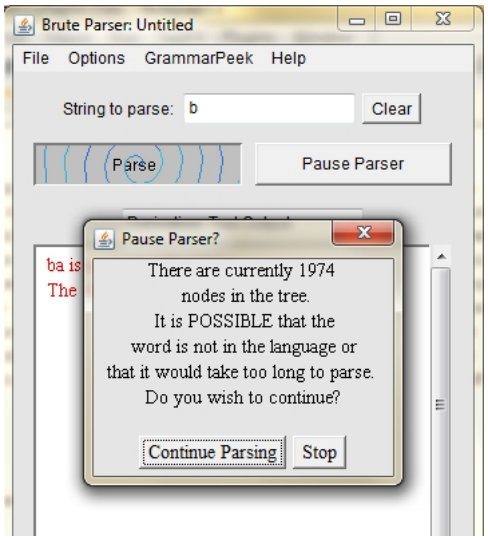

Figure 2.4: Pâté teaching tool attempting to parse string ‘b’.

and the program will demonstrate the process of parsing text one step at a time. In Figure 2.3 we can see the program is limited to simple grammars and would have no application with typical grammars of languages. Nevertheless, for use in teaching students this clear demonstration of how rule productions are used from the start symbol to match the input text is clearly beneficial. The syntax is upper case for nonterminal symbols and lower case for terminals. From performing some tests, parsing simple input can be very slow. The ‘readme’ states that the parser is a brute force parser. Indeed, from testing the parsing of simple text is quick except when the grammar cannot parse the text. The approach is to keep expanding the grammar until it reaches a cut off point. For example, it takes roughly 10 seconds to tell me the grammar,

```
S = A, S;
S = A;
A = 'a', 'b';
```

could not parse the string ‘b’. This can be seen in Figure 2.4.

2.3.4 Representing larger grammars

The BNF Web Club [24] and others [2, 17, 35] utilise web hyperlinks so that the user can ‘jump’ between rules by clicking on a nonterminal rule. The BNF Web Club’s

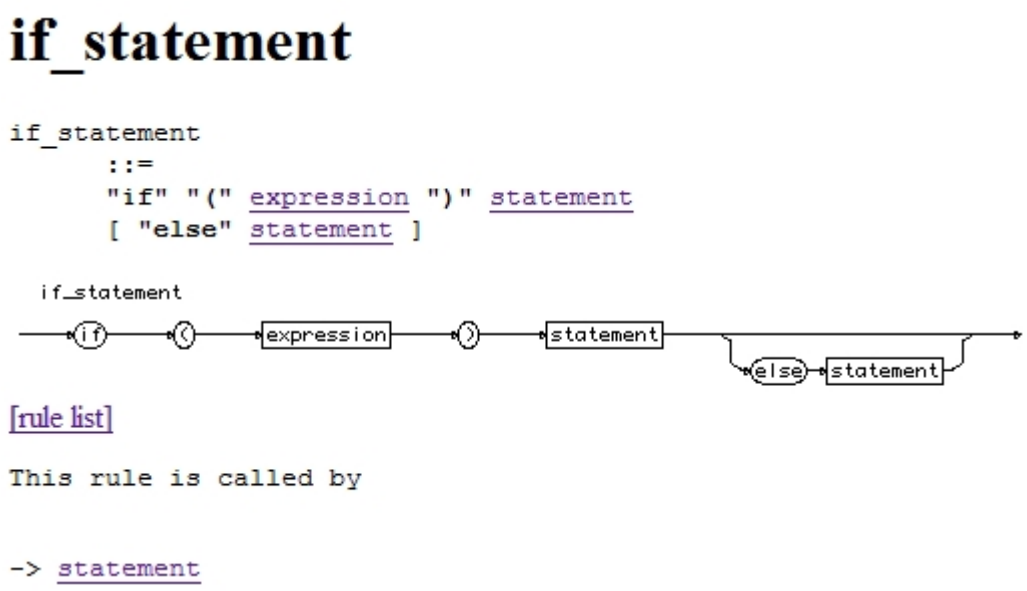

Figure 2.5: The BNF Web Club showing a rule for the Java language.

website has created these ‘hyperdocuments’ from some popular language grammars including SQL, PL/SQL and Java. The web application solves some of the problems of reading larger grammars. If a user wants to know what a rule is they can click it and are taken to a page detailing that rule. This ensures there isn’t too much clutter when looking at typical grammars.

Alongside the hyperlinks the BNF Web Club adds another representation of the rule in what they call ‘syntactic diagrams’. They are shown in Figure 2.5 and are very similar to Figure 2.2. We can use these flow diagrams to quickly and easily understand what is and isn’t allowed in that rule without knowing BNF. Loops and optional items are easily shown by the choice of which route you take, starting from the left and finishing on the right. Terminals have rounded edges whereas nonterminals do not. These diagrams are also known as ‘railroad diagrams’ and there are programs available for converting from (E)BNF to these visualisations [2, 14, 46].

CHAPTER

THREE

REQUIREMENTS

3.1 Approach

C. Larman's book, 'Applying UML and patterns' [22] is a valuable resource for the design of Object Oriented (OO) projects. Larman's book is used throughout the report for guidance and support in design and implementation decisions.

3.2 Requirements

Use cases are a way to discover and record requirements, being described as a "key requirement input to classic OO Analysis and Design" ([22], p64). They should be simple as they are there to emphasise the user's objectives ([22], p65).

3.2.1 Users

The two users are the student and the experienced user of grammars. Here are three use cases.

Student

Understand Grammars

The student has been introduced to grammars but is now looking to reinforce their knowledge by looking at a visualised grammar. The student uses the program to load some of the example grammars predefined in the program. Each predefined

grammar shows a different concept such as concatenation, choice and recursion etc. The student chooses which grammar to look at and is shown a visualisation of the grammar and its BNF equivalent. The user can click closed rules to highlight and read them. When the student is satisfied they can load another grammar or close the program.

Create Grammar

The student has been introduced to grammars and understands how grammars are visualised in the program. The user is going to create their own grammar. The student uses the program to start a new, empty grammar. The student uses the available menus and buttons to create their own grammar. Any action the user wants to do: add, edit or delete is supported by the program. The textual view must match the visualisation at all times to help understanding. The student can save and load the grammar.

Experienced user

The use case for the experienced user of grammars is as follows:

Extend Grammar

The user has this grammar for a language that they wish to extend. The user loads the grammar into the program and immediately the program parses the grammar and visualises it. The user will be able to explore the grammar through the visualisation. The user can also jump to specific rules and hide the rest of the grammar if they are working on a particular section. The user can also change the textual representation and the grammar visualisation will update accordingly. The functions to add, delete and edit symbols and rules will all be accessible from the visualisation. The user can save the grammar in its original BNF notation.

3.2.2 Elicitation

([22], p56-57) categorises requirements into the FURPS+ model.

- Functionality.
- Usability.
- Reliability.
- Performance.

- Supportability.
- Extra requirements (licensing, packaging).

See Appendix A for details of the specific requirements elicited from the use cases categorised according to the FURPS+ model.

3.3 Development plan

Problems are better solved one step at a time and this project is no exception. The development is split into four phases of development that separates the programs objectives and functionality. This approach to a separation of concerns and incremental development is similar to the Unified Process (UP) software design methodology ([22], p18). The UP encourages you to make small time-boxed developments that are first focus on the core architecture and UI then slowly building on your previous working components to create a larger, more functional program. The four phases are to be known as:

1. Visualising grammars.
2. Build and edit grammars.
3. The dual view.
4. Extra features.

These phases are explained in detail in Chapter 4.

3.4 Which technology?

This section introduces different technologies that are available for implementing the program, concluding with and a decision. The ability to draw 2D shapes, enable user interaction and a certain degree of 'higher abstraction' is required to enable efficient development.

Graphics libraries

As the program is going to visualise grammars, it seems necessary to look at the available graphics libraries on offer. Gorgon [44] and HAAF's Game Engine [45] are both 2D graphic libraries that are for .NET [27] and C++ respectively. They both do have the ability to draw 2D shapes and also have libraries for pixel shading, buffering, image display and animation. Predominantly aimed at 2D game programming the level of detail is quite low, allowing for advanced 2D graphics capabilities. OpenGL [31] is a graphics engine for C/C++ developers. OpenGL has 2D and 3D graphics programming but is aimed at creating advanced graphics programs utilising its efficient libraries for rendering the display. These graphics libraries are too low level for the project although they could all be used it is perhaps wiser to utilise some higher level libraries that are available for drawing simple 2D shapes and handling user interaction.

Java 2D graphics package

Java has 2D graphics packages called AWT that allows the creation of simple 2D objects, to colour them and register for events such as mouse click and mouse enter/leave the drawn object. Java is OO which would make programming and debugging much easier as the writer has a good knowledge of OO programming techniques. Java and graphics rendering have tended to be quite slow as the program is being interpreted rather than executed. The capabilities for user interaction is high as the graphics package supports mouse clicks and events.

C# .NET

.NET is a framework that provides a large library of pre-built code to solve common programming problems. C# .NET is a programming language developed by Microsoft. It offers access to the framework and has an impressive IDE for developing large applications. Windows Presentation Foundation (WPF) was released with .NET 3.0 and is the successor to Windows Forms found in .NET programming languages. WPF aims to separate the UI and business logic by having two separate files for developing a GUI component. The UI is now described in a new language called XAML. This separation of concerns is useful as it keeps files focused and ultimately keeps the code more readable. Most GUI applications are built using a hierarchical

structure. Starting from the outer window the programmer adds nested elements such as layout controls, then labels or buttons to build the GUI. The XAML code is a more natural approach to defining GUI applications as it is a hierarchical markup language. Some XAML code is shown here:

```
<UserControl Height="Auto" Width="Auto" BorderThickness="1">
  <StackPanel Orientation="Horizontal">
    <Button Height="20" Content="Button" />
    <Label Content="Label Text" />
  </StackPanel>
</UserControl>
```

There is heavy influence on user interaction and the libraries supports a vast array of drawing tools.

3.4.1 Decision

The graphics libraries are too low level for this project as the visualisations required are not very advanced. The game libraries are also aimed more at helping the user with game physics and player interactivity rather than drawing 2D shapes. The Java AWT package is a high level graphics package that does offer good user interaction with the use of events. Java is also platform independent and non-propriety whereas C# is not. However, C# and WPF provides more functionality and offers a more up-to-date graphics package. The graphics package in WPF is much faster than Java as it uses vector graphics which utilises the computers GPU whenever possible [7]. C# also has good support for large projects with its IDE, Visual Studio and it is free for students [26]. The separation between business logic and the GUI definition in C# will keep the classes readable and focused. Microsoft Windows is installed in nearly all of the computer laboratories across computer science. It is for these reasons C# has been chosen to develop the program.

CHAPTER

FOUR

DESIGN

4.1 Grammar representation

The visualisations have already been shown in Chapter 1, however, that was an abstract view, thus we take the visualisations and move them into the context of a computer program. With a program we have the ability to make it interactive. See Figure 4.1 for a drawing of the proposed visualisation. The grammar it visualises is:

```
Boolean_exp = Number, '==', Number;
Number = '[0-9]+';
```

In the image a number of important design decision were made, they are explained here:

- To distinguish nonterminals from terminals the font will be bold and underlined. The word 'is' will also follow it.
- Terminals are in quotation marks to match the ISO BNF syntax.
- The arrow will show which direction the user should 'read' the grammar. It is the direction of concatenation.
- Each rule can be expanded to view its contents and collapsed again, if desired.
- The default view will show an expanded start symbol and every other rule collapsed. It is left for the user to expand rules.

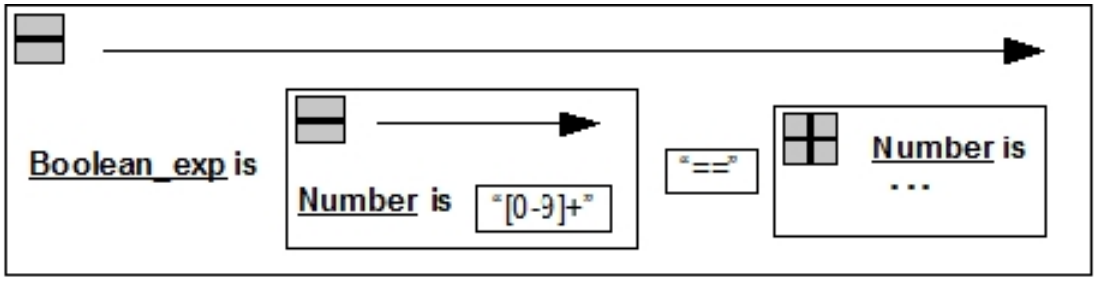

Figure 4.1: Proposed visualisation of a grammar in the program

4.2 Phases

4.2.1 Visualising a grammar

The first phase of development is at the core of the program. It is the ability to visualise a grammar. The core architecture is defined as one of the highest risks to the projects success and so should be developed first [22]. The objective of this phase is to take the different attributes of grammars and visualise them in small, simple grammars. Four grammars have been chosen which will be hard coded into the program. This provides a quick and easy way of testing each visualisation during development and will be kept after development to show students the different grammar concepts.

1. Basic Rule. A simple grammar which tests the most basic principles.

```
S = 'a';
```

2. Concatenation. If the above grammar works the next check is to try two rules and concatenating items.

```
S = 'a', B, 'c';
B = 'b';
```

3. Choice.

```
S = 'a' | 'b';
```

4. Recursion. This grammar tests recursion in grammars.

```
S = 'a', S, 'b' | 'ab';
```

The main areas of development identified for this phase are:

- Data structures. How should the grammar be represented in memory? The data structure had to be developed first as it would define how the other aspects of the program would be developed. It had to support BNF and in the future EBNF. A whole section (4.3) is dedicated to their design.
- Visualisations. Having never used WPF before, the second largest challenge was to understand WPF and then create some user controls that will be used when a grammar visualisation is built. Building separate reusable components meant each control was small and focused on its task. Controls will need to be developed for symbols (terminal and nonterminal) and operators (concatenation, choice). The only interaction (in this phase) will be the expand/collapse buttons on nonterminals.

4.2.2 Build and edit grammars

The second largest phase is providing the ability to build and edit grammars from the visualisations themselves. Inserting more interactivity requires buttons or a menu system. It would seem natural to insert buttons on the visualisations but to reduce the 'clutter' and keep them clean and focused another solution is needed. A right-click menu system would be developed for the visualisations. The main challenge in this phase would be to add and delete branches from the data structure to match the users requests.

Parsing BNF

A parser is required to parse a users BNF grammar in order to visualise it. A compiler-compiler (CC) also known as a parser generator generates source code or a library that will parse text automatically. A CC requires a set of rules and actions to create the required program to parse text. Some popular CCs are YACC [16] and more recently ANTLR [33]. As there are many variants to BNF an element of flexibility is required in the parser. A custom recursive descent parser offers more flexibility and feasibility than learning and using a complex CC. As the syntax for

BNF is small and the need for flexibility a custom recursive descent parser will be written. The user should be able to change the symbols for terminal, concatenation, assignment, choice and end of rule operators. Custom error messages may be more helpful than using the standard CC parse error messages.

4.2.3 The dual view

Representing the visualised grammar next to the text grammar is useful for students to understand what they are creating. The text grammar must match the visualised grammar when a grammar is being built. The dual view will also accept input from the user and the program will visualise what is written. Distinguishing a selected rule from other rules is also desirable.

4.2.4 Extra features

There is an endless list of extra features that could be added to the program. However, some features were identified at the design stage that if time permits will be implemented at the end.

- Support EBNF syntax. Requires extending the data structure and also defining visualisations that distinguish EBNF (Loops, Optional Items) from BNF.
- Export to various outputs such as straight BNF text, EBNF text, image formats, PDF, Lex and Yacc, etc.
- Different views such as finite state automata or state transition diagrams.
- Transformations such as factorising rules, removing left recursion, moving rules around etc.
- Animations of a grammar parsing some text.
- Drag/drop rules or copy + paste rules.
- Support comments or ‘TODO’ feature.

It is hoped that after significant development there will be a better understanding of the feasibility of each feature.

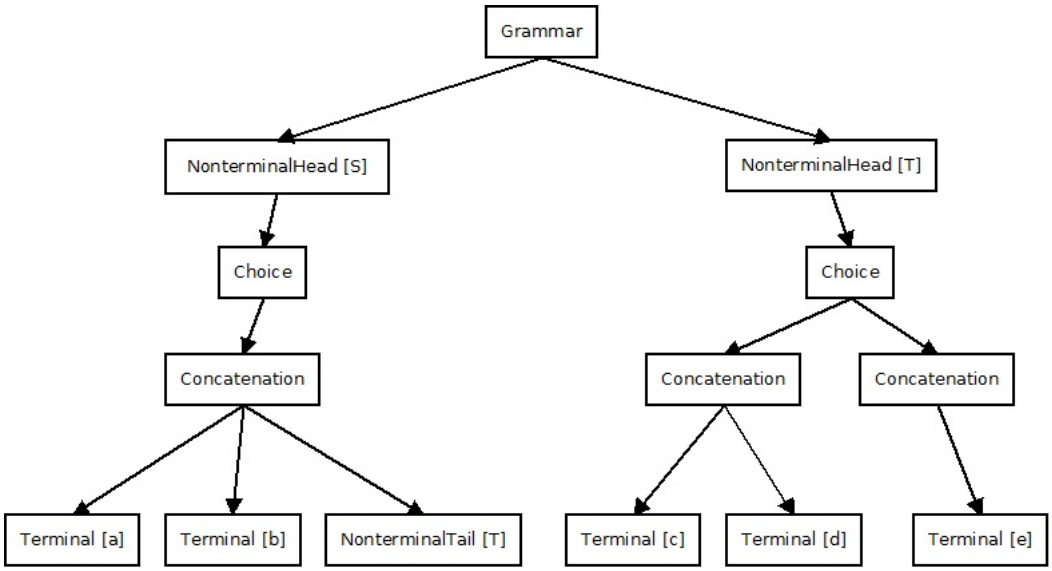

Figure 4.2: Proposed representation of grammar in memory

4.3 The data structures

First we must clarify the elements of a grammar. A grammar is made up of a number of *rules*. Every rule has a nonterminal to denote which rule it is and the rule itself. The rule is made up of *symbols* which are either nonterminal or terminal. Here, we must make the distinction between nonterminals in the rule body and nonterminals at the head of the rule. *nonterminalHead* is the nonterminal at the start of every rule. Nonterminals that appear in the rule are called *nonterminalTails*. The body of the rule is made up of symbols which are separated by an *operator*. The operator can be concatenation or choice, with concatenation having precedence over choice. The rule body contains operators whose children are either operators or symbols. Using this information we can represent grammars using a tree structure.

The following grammar:

```
S = 'a', 'b', T;
T = 'c', 'd' | 'e';
```

can be represented by a tree structure in figure 4.2. In the future when a user is building a grammar it will be easier to append elements to the operators without having to re-arrange the tree structure. Therefore, every rule must contain a choice and concatenation operator even if the rule does not require it.

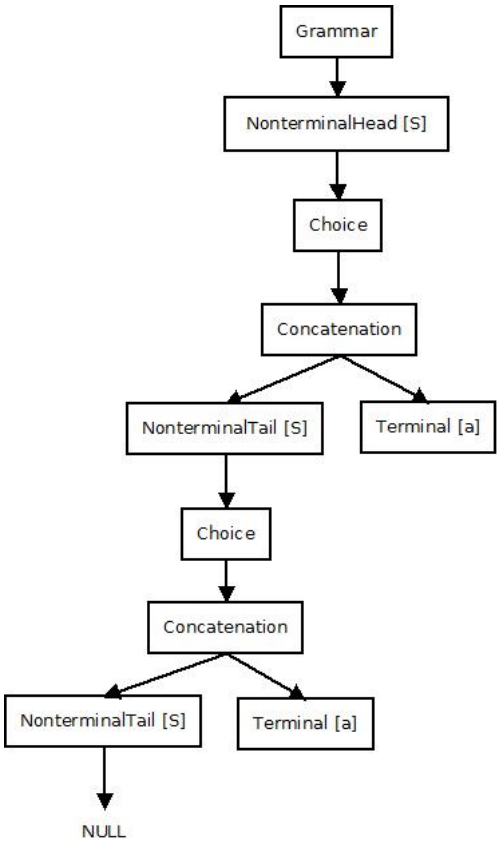

Figure 4.3: Proposed representation of a visualised grammar in memory

The separation of operators and symbols allows for extensions in the future including brackets (scope) and EBNF. The nonterminalHeads always have one child, whereas the operators can have many children. The nonterminalTail classes will need a reference to the nonterminalHead to which it is representing. The 'NonterminalTail [T]' needs to reference the 'NonterminalHead [T]' to know what rule it is. This structure is good for representing the grammar, but this representation cannot be used for the visualised grammar.

Consider the grammar:

```
S = S, 'a';
```

To offer the ability to expand the recursive rule the representation would need to allow nonterminalTails the ability to have children. At this point we have two possibilities,

1. Use one data structure to represent the actual grammar and the visualised grammar. From the structure we could infer the rules and save space at the expense of making the structure less focused.

2. Use two data structures where the first represents the actual or *definitive* grammar, as seen above and a second that represents the visualised grammar. The two structures would be very similar but the distinction must be clear. The second structure, would keep growing as the user 'expanded' the recursive rule `s`.

The responsibilities of each object should be clear, concise and strongly related. This is known as maintaining a "separation of concerns" ([22], p204 and p441). This highly cohesive approach which helps readability of code and minimises the impact of change between connected objects makes it clear that the second approach should be taken. Figure 4.3 shows how the second tree structure is used. This time the nonterminalHead is the start symbol of the grammar and is the single point of entry to the rest of the visualisation. If the nonterminalTail has children, then it is known as 'expanded' and we want to visualise that rule. If it points to null we know that the nonterminal is 'collapsed' and we will show a closed box. Subsequent expansion or collapsion would append or delete to parts the tree. The data to add to the tree can be obtained from the definitive tree structure.

4.4 Program IDE design

An initial design of the layout of the program is presented here. Figure 4.4 shows a drawing of the proposed system. The annotations are explained here.

1. This area is where the main visualisation will be displayed. If the visualisations grow too large for the area, scrollbars will be available to scroll across the image.

2. The user has right-clicked on box 'T' and is presented with a drop down menu as shown in grey. This is where the user can design a grammar through the visualisation interface. Phase 2 will implement this menu system.

3. This section is the BNF representation of the grammar that is being displayed. This section will be implemented in the 3rd phase of development.

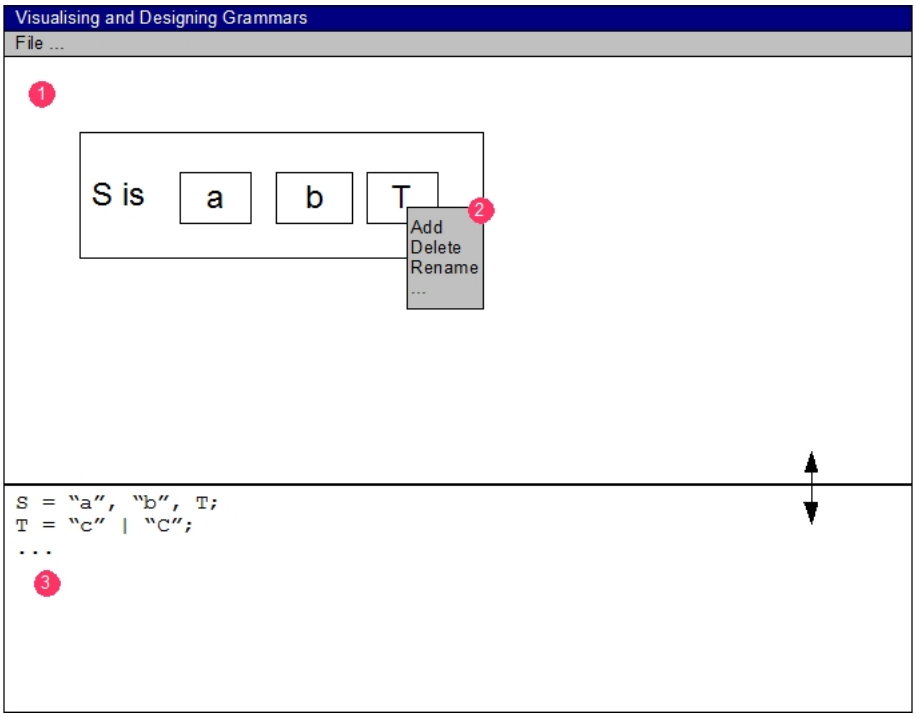

Figure 4.4: Proposed design of the program's layout

4.5 Model-View-Controller

The Model-View-Controller (MVC) is a design pattern that promotes the separation of responsibilities between the model and the view. The data structures and business logic are contained within the model and the UI is the view. When creating a system that has a user interface it is recommended that the MVC pattern is used ([22], p209-210). In the MVC the controller should act as the 'middle-man' passing messages between the model and the view. This modular approach means that in the future, it would be easy to change the view to something else (e.g web-application) without changing the controller or the model. This project is designed using the MVC pattern.

CHAPTER

FIVE

IMPLEMENTATION

5.1 Data structures

The data structures were the most important and complex part of the implementation process. Any limitations in the data structure would result in limitations throughout the rest of the program. This section describes the data structures in more detail; given a grammar we derive a representation in memory for both the grammar and its visualised correspondent.

5.1.1 The definitive grammar

The *definitive grammar* is implemented using a tree structure. Since the tree structure can be made up of a mixture of class types and there will be variables used across all the classes an abstract class was created. The abstract class is called GrammarNodeD. Every class used in the data structure must inherit GrammarNodeD. *D* is used to distinguish this grammar from the visualised grammar seen later on. A UML diagram ([22], p221) of the class hierarchy can be seen in Figure 5.1.

A *NonterminalHeadD* class represents the rule head. The 'S' in `S = A, 'b';`. It has one child of type OperationD. This ensures that every rule must start with an operation that is either ChoiceD or ConcatenationD. The *NonterminalTailD* class appears in the production rule. It has no children but it must reference the rule it is representing in the grammar. A *DefinitiveGrammar* class was made which holds a list of the all the rules, is a wrapper to the data structure and holds a reference to the

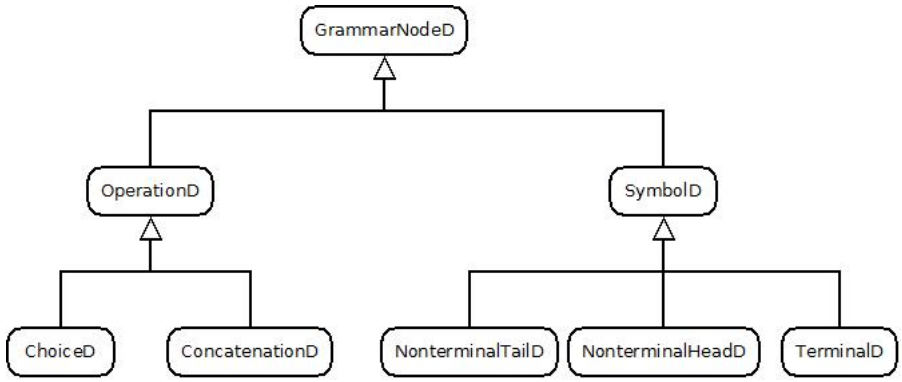

Figure 5.1: Inheritance class diagram of the definitive grammar data structure

start symbol. A diagram which shows how to represent the following grammar:

```
S = A, 'b';
A = A | 'a';
```

is shown in Figure 5.2. The full arrow represents the references to child nodes in the tree. The bold arrow distinguishes the start symbol from the other rules. The dashed arrows represent the reference from the nonterminal to the rule it represents. To reiterate from the design chapter, the decision to put in `ChoiceD` and `ConcatenationD` classes when they are not necessary was to 'future proof' the data structure. In the build and edit phase it will be easier to add children to the operation nodes in the current state than manipulating the tree to add new operation nodes when needed.

Every operation can have 1 or more children. To implement the tree in C# we use a *List* which is a numerically indexed array that can grow (or shrink) dynamically. In the abstract class OperationD there is a list of type GrammarNodeD which we will call it's children. The code for this can be seen in listing 5.1. To expose a private field in C# we create a public field (line 4) instead of creating get and set methods. The code to access a child is `MyChoiceD.Children[i]`.

In section 5.4 a parser which automatically generates the definitive grammar is described.

5.1.2 The visualised grammar

The *visualised grammar* data structure represents what parts of the grammar we want to visualise. We construct the visualised grammar from the definitive grammar.

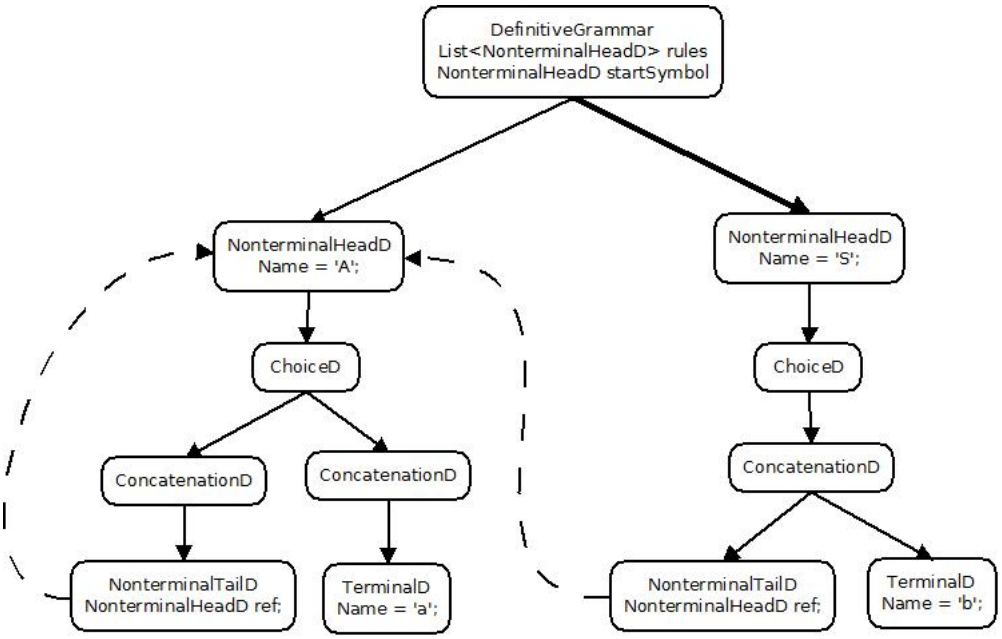

Figure 5.2: Diagram of the data structure for the definitive grammar.

```
1 public abstract class OperationD : GrammarNodeD
2 {
3   protected List<GrammarNodeD> children;
4   public List<GrammarNodeD> Children
5   {
6     get { return children; }
7   }
8   ...
9 }
```

Listing 5.1: The List implemented in ConcatenationD

```
1  public void visit(ConcatenationD concatenationD)
2  {
3    List<GrammarNodeV> childrenV = new List<GrammarNodeV>();
4    foreach (GrammarNodeD childD in concatenationD.Children)
5    {
6      // make children
7      childD.accept(this);
8      childrenV.Add(current);
9    }
10   ConcatenationV concatenationV =
11     new ConcatenationV(childrenV, concatenationD);
12   concatenationD.AddReference(concatenationV);
13   current = concatenationV;
14 }
```

Listing 5.2: A method from the class that constructs the visualised grammar.

The class *VisualisedGrammar* acts as a wrapper to the data structure within. It also has a reference to the nonterminal which will be the outer most box when we visualise - usually the start symbol. The visualised grammar has all the elements of the definitive grammar (Choice, Concatenation, Terminal, etc.) but we need to use a different set of classes. We append these classes with *V* to distinguish them from the definitive grammar. The class diagram is the same layout as the definitive grammar.

To create the visualised grammar we must traverse the definitive grammar. It was decided that we would only traverse the start symbol when creating the visualised grammar. From there the user chooses which rules to expand. To traverse data structures whilst still maintaining the separation between storing data and providing functionality the visitor pattern is used. A *design pattern* is a defined approach to solving some of the most common problems that appear in object oriented programming ([22], p279). The *visitor pattern* is a design pattern that enables traversal of data structures. See ([11], chapter 16) for a detailed introduction.

Listing 5.2 shows an excerpt from the visitor pattern class that constructs the visualised grammar. In line 7 we visit the children first where each child of the concatenationD places the constructed sub-tree in a global variable called current. We add the sub-tree to our List (line 8). The global variable `current` is used as it easier than passing the variable between methods. This is a depth first traversal of the definitive grammar, creating the children before the parent. The visualised grammar is representing the definitive grammar and thus needs a reference to the definitive grammar so the objects know what they are. In lines 10-13 we create

the concatenation class for the visualised grammar, set up references and set the concatenationV to the global variable for use by its parent.

Figure 5.3 shows how the visualised grammar is represented in memory. It also shows the references to the definitive grammar from the previous section. The dotted lines are the two-way references between the visualised and definitive grammar.

Note how the visualised grammar does not store any information that can be obtained from the definitive grammar. The NonterminalTailV must reference the NonterminalTailD so that it knows which rule the nonterminal belongs to. The NonterminalTailD then references the NonterminalHeadD to obtain its name. This complex design will make it easier in the future when we require to make changes to the grammar. E.g. simply renaming the NonterminalHeadD will enable easy propagation of changes throughout the definitive grammar and the visualised grammar.

5.2 Constructing the visualisation

The *visualisation* is the actual drawings of the grammar that will be shown to the user. To draw the visualisations we must traverse the visualised grammar.

Every class in the visualised grammar has its own UserControl. A *UserControl* is a class in C# that has a visualisation or drawing associated with it. The class is separate to the drawing. The class is where we add the business logic and the XAML section is where we define the controls. The UserControl for a Terminal for example is simply a box with a border. Inside the box is a label that displays the Terminal's value. The Terminal class can be seen in listing 5.3. The terminalV reference is needed so that we can obtain the name from the visualised grammar. The visualised grammar will in turn reference the definitive grammar and return the string value.

The XAML Code for the Terminal class can be seen in listing 5.4. The class can access the label 'name' and in the constructor it changes it's value from 'Name Here' to the terminal's value. The concatenation XAML for example, is simply a UserControl called StackPanel. This control appends other UserControl's (e.g. Terminal) next to each other when they are added to the StackPanel. The drawing grows and shrinks according to the number and size of the UserControl's in the StackPanel. Choice is similar but the StackPanel's layout of UserControl's goes from top to bottom instead of left to right, and between each UserControl a Label

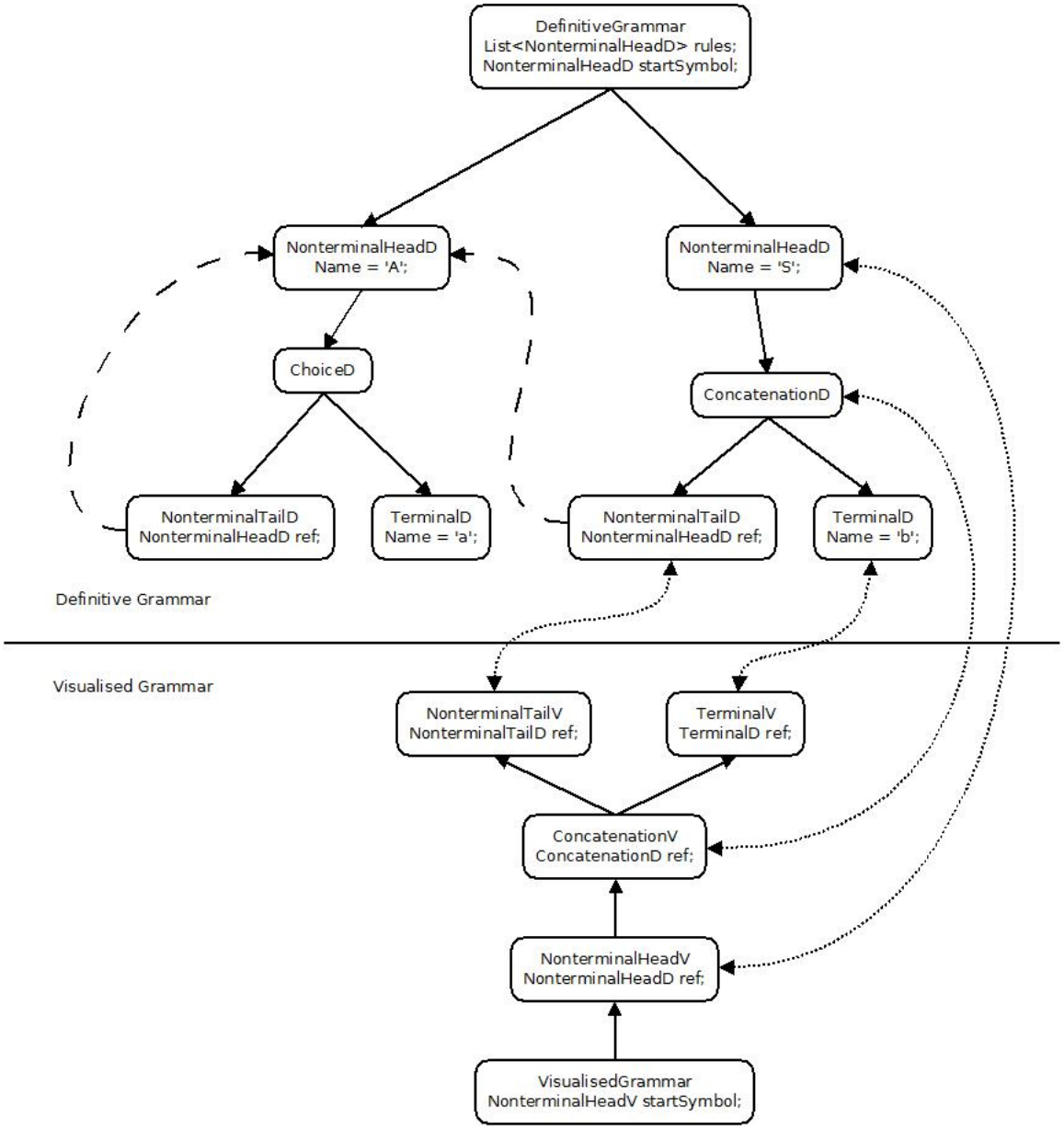

Figure 5.3: The visualised and definitive grammar with their references.

```
1  public partial class Terminal : UserControl
2  {
3    private TerminalV terminalV;
4
5    public Terminal(TerminalV terminalVIn)
6    {
7      InitializeComponent();
8
9      terminalV = terminalVIn;
10     name.Content = terminalV.Name;
11   }
12 }
```

Listing 5.3: The Terminal UserControl class.

```
1 <UserControl x:Class="VADG.Drawer.Terminal" Height="Auto"
2 Width="Auto" BorderThickness="1" BorderBrush="Black">
3   <Label Name="name">Name Here</Label>
4 </UserControl>
```

Listing 5.4: The Terminal UserControl XAML code.

labelled ‘or’ is placed.

The NonterminalTailV is the only exception. It has two UserControl’s to choose from; NonterminalExpanded or NonterminalCollapsed. If in the NonterminalTailV there are no children we do not want to show the rule, we therefore choose to draw the NonterminalCollapsed UserControl. If there are children, we draw the other control. The code for this is shown in listing 5.5. If the nonterminal has children then we must draw the rules contents first (line 5). This places a drawn control of the rules contents in a global variable called drawnItem. Again, it is easier to use a global variable than pass the drawn control around the methods. This is a depth first traversal of the visualised grammar, drawing the child controls of the tree before the parent. To draw the expanded nonterminal, we create the class called NonterminalExpanded with the reference and the drawn control (line 6). Figure 5.4 shows the area that is drawn by the NonterminalExpanded UserControl. The shaded area is the child that has already been drawn.

Upon reaching the end of the tree we simply draw the control and place it in the global variable. When constructing the visualised grammar from the definitive we only traverse the chosen rule (usually the start symbol). It is left to the user to expand whichever rule they want manually.

```
1  public void visit(NonterminalTailV nonterminalTailV)
2  {
3    if (nonterminalTailV.IsExpanded)
4    {
5      nonterminalTailV.Rule.accept(this);
6      NonterminalExpanded control = new NonterminalExpanded(
           nonterminalTailV, drawnItem);
7      drawnItem = control;
8    }
9    else
10   {
11     NonterminalCollapsed control = new NonterminalCollapsed(
           nonterminalTailV);
12     drawnItem = control;
13   }
14 }
```

Listing 5.5: Choosing the correct visualisation UserControl.

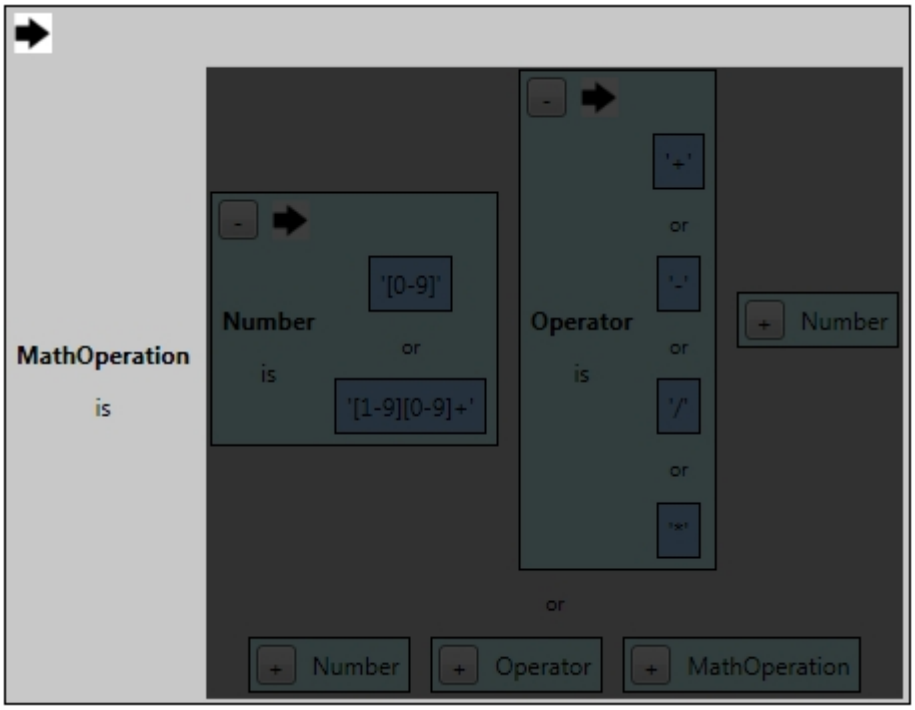

Figure 5.4: The Nonterminal Expanded. The shaded area is not drawn by the outer UserControl.

5.3 Expanding and collapsing

Obviously we want to look at rules and this involves expanding the NonterminalTailV found at the end of the visualised grammar. The algorithm for expanding a rule is as follows,

1. Check the NonterminalTailV to be expanded has not already been expanded (child is null).
2. Use the reference in the visualised grammar to get to the NonterminalTailD and then NonterminalHeadD to find the rule.
3. Use the same visitor pattern described in section 'Constructing the visualised grammar' to traverse the the rule, creating a visualised grammar sub-tree.
4. Append this sub-tree to the NonterminalTailV so now it's child is no longer null.
5. Redraw the visualisation.

Collapsing is simple as well, we just remove the sub-tree from the visualised grammar and redraw. Although care must be taken to dereference all the references that were made (between the definitive and visualised grammar) otherwise the objects will stay in memory, thus resulting in memory leaks and possibly unexpected errors.

5.4 Build and edit grammars

5.4.1 Editing from the visualisation

An *operation* is a action that can be performed to manipulate the grammar. The way in which the program handles operations is that the user selects a control and chooses to add or remove controls relative to the selected control. A doubly-linked data structure is one where the references point both ways between parent and child. The definitive grammar had to be changed so that it would be a doubly-linked tree. When a user clicks a control we need to identify the parent operator so we can add or remove controls to the correct lists. The *add after* operation adds a new symbol after the selected one, in the concatenation. Here is an algorithm for the operation:

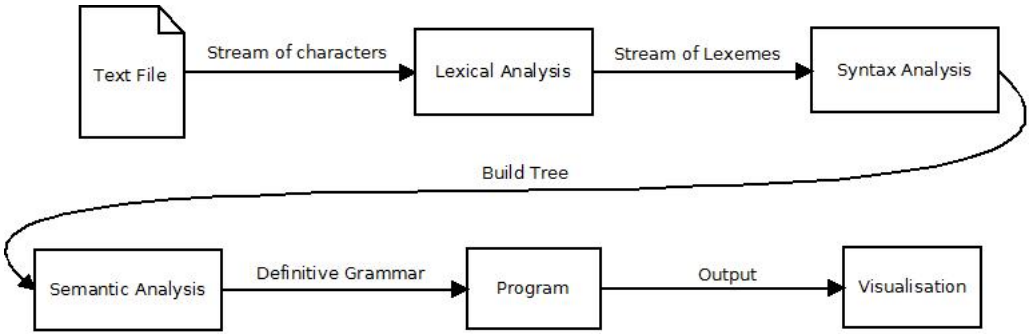

Figure 5.5: An overview of the processes from parsing to visualisation.

1. Select the symbol, *S* which we will be the selected symbol where we want to add after.
2. The user creates a new symbol to add, *D*.
3. Using S identify the associated symbol in the definitive grammar. Then identify its parent.
4. Using the parent, insert D at the correct position in the list. We know a symbols parent will always be ConcatenationD from when we constructed the data structure.
5. Using the visitor pattern we traverse the visualised data structure looking for rules that displays S.
6. If we find a rule that shows S we insert a new visualised node (created from D) into the tree. We keep looking as there may be more than one place where S is shown.
7. Redraw the visualisation.

This tree manipulation was required because the user would like to edit the grammar and not lose their expanded visualisation. The redraw is not noticeable, even for large grammar.

5.4.2 The parser

The recursive descent parser was developed with the help of this [34] book. Figure 5.5 gives an overview of the stages of parsing from a text file right through to

the output. We are interested in the first 4 processes.

Lexemes are the small tokens of useful information we require from the text. The lexical analysis ignores whitespace and comments. It classifies the lexemes into different categories. If for example, it receives a ' character then it will keep reading until it finds another ' character. A lexeme of type terminal is then created. If the lexical analyser does not find a closing ' it throws a lexical analysis error. The lexical analyser uses global variables for the terminal encapsulation character, assignment operator character etc. This enabled the program to change the syntax slightly and enables users to parse from many different BNF variants. This stream of lexemes is then sent to the syntax analyser where we check the lexemes are in the correct order.

```
1 Grammar = Rules, EOF;
2 Rules = Rule | Rule, Rules;
3 Rule  = Nonterminal, AssignmentOp, Symbols, EndofRule;
4 Symbols = Symbol | Symbol, InfixOperator, Symbols;
5 InfixOperator = Concatenation | Choice;
6 Symbol = Nonterminal | Terminal;
```

Listing 5.6: A grammar describing the syntax of BNF.

The grammar for a generic BNF grammar is described in listing 5.6. The rules that have not been identified are the lexemes. These rules can be mapped to methods in the syntax analyser. Starting from the top, we create a method for the start symbol called Grammar. It calls the method `Rules()` followed by `EOF()`. These methods call the sub rule methods from left to right until we reach the lexemes. When we reach the lexemes we are expecting that lexeme in the input stream and error if they do not match. This is called Top Down Recursive Descent (TDRD). The input lexemes are also read from left to right. This is equivalent to an *LL* parser which uses a table driven approach to check the syntax. Any choice requires the code to 'look ahead' k lexemes to determine which method to call. The grammar is said to be an LL(k) where k is the maximum look ahead value. The grammar for the generic BNF is an LL(1) grammar. The code for the rule 'Rules' and 'EOF' is shown in listing 5.7.

Line 3 is where we parse a whole rule and move the current lexeme to one after the rule. The look ahead can be seen in line 4. If the current lexeme is a Nonterminal we assume that there is another rule to parse and recursively call `Rules()`. This recursive call is where the TDRD obtains its name. The accept method (line 10)

```
1  public void Rules()
2  {
3    Rule();
4    if (currentLexeme.Type == LexemeType.Nonterminal)
5      Rules();
6  }
7
8  private void EndOfFile()
9  {
10   accept(LexemeType.EOF);
11 }
```

Listing 5.7: An excerpt from the top down recursive descent parser.

checks the current lexeme type against the one passed. If they match the current lexeme is moved forward one but if it fails an exception is thrown which propagates up the method call list and results in an unsuccessful parse. Exceptions were chosen as they offer more information when an error occurs than the traditional `return false;`.

The syntax analyser also builds a build tree that groups lexemes into rule head and rule body as it parses a rule. The syntax analyser sends these rules to the semantic analyser.

The semantic analyser will check for any rules that are not defined, throwing an error if a rule definition is missing. The semantic analyser processes the build tree to create the definitive grammar.

5.5 Grammar factorisation

The grammar factorisation was more a proof of concept than providing a useful tool. Although factorising duplicate rules into one rule is useful the implementation here proves that the definitive grammar can be used for more than just creating visualisations.

Rules or parts of rules (where the number of symbols is greater than two) that are repeated in a grammar can be factorised into one rule. A factorisation candidate is a subrule that appears more than once in the grammar. A *subrule* is the subset of a rule.

As an example, consider the grammar:

```
1 S = T | 'a', B, 'c';
2 T = 'a', B;
3 B = 'b';
```

The duplicate subrule here is on lines 1 and 2. The factorisation candidate is ‘`'a', B`’.

An algorithm for finding and removing the longest factorisation candidate is shown here. The longest, i.e. subrule with greatest length is deemed the most useful factorisation here. A hash table ([47], p683) is used to find the duplicate rules. A string *S* is used to hold the longest subrule.

1. Traverse the definitive grammar creating string representations of all the subrules that appear in the grammar. Subrules should contain two or more symbols.

2. Add these subrules to a hash table. The key is the subrule. The value is a boolean initialised to false and a string of the subrule. If a collision occurs there are two possibilities:

   (a) If the strings are equal then we have a candidate for factorisation. If the factorisation candidate is longer than S then set S to be the new factorisation candidate.

   (b) If the strings are not equal a collision has occurred. Use a re-hash function ([47], p687).

3. Create the string *G* which is the whole grammar in BNF notation.

4. Choose a new rule name, *K*. Replace all instances of S in G with K.

5. Insert the new rule K with rule body S into G.

6. Use the parser to build the new definitive grammar.

CHAPTER

SIX

RESULTS

6.1 VADG

Figure 6.1 shows a screen-shot from the 'Visualising And Designing Grammars' (VADG) Program. The visualisation corresponds to the BNF grammar written in the 'Text Grammar'. The visualisation is the default view the user sees. The start symbol is expanded but no other rules are.

6.2 Visualisations

Figure 6.2 shows a rule from the ANSI Standard 1985 C grammar [18] being visualised. Some nonterminals are expanded. The controls are centred for each row. The colours for each level help to distinguish between the rules and the rule bodies. The visualisation is primarily aimed at visualising rules and lexemes were an after thought. The unknown lexemes are handled by a control called 'undefined' and is represented as `???` in a nonterminal. This can be seen for the lexeme 'RETURN'. See section B.1 in appendix B which shows a grammar visualisation in more detail.

Functionality for expanding, collapsing and selecting controls is supported. There is no limit to the size of the data structures and thus the visualisation levels can be a deep as possible. The colours were picked arbitrarily and cycle round after 6 levels of drilling down.

To find a rule without traversing the visualisation a useful drop down box is provided. See Figure 6.3. This allows the user to visualise a rule quickly. This is

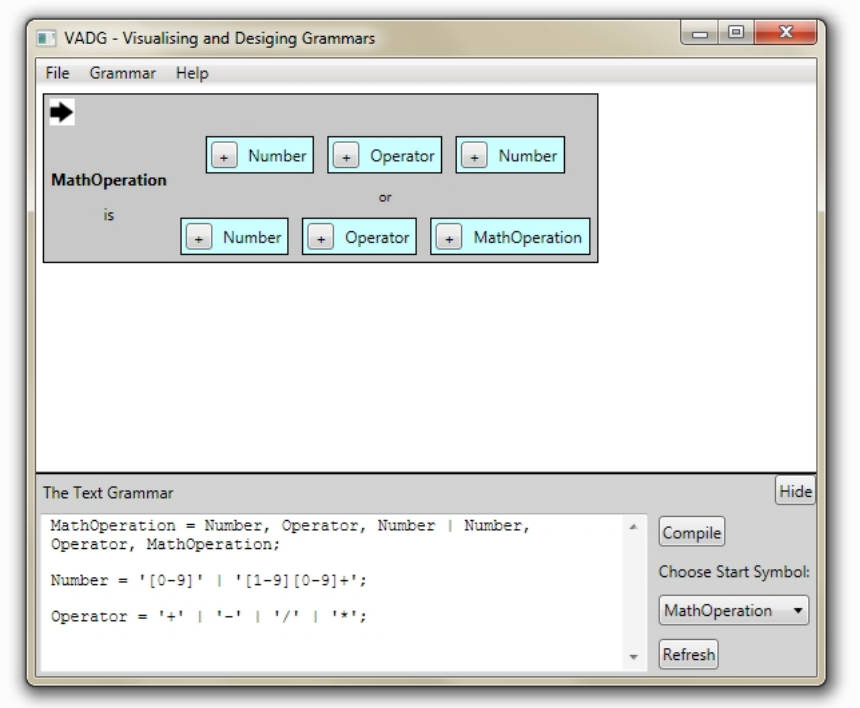

Figure 6.1: The Visualising and Designing Grammars (VADG) program.

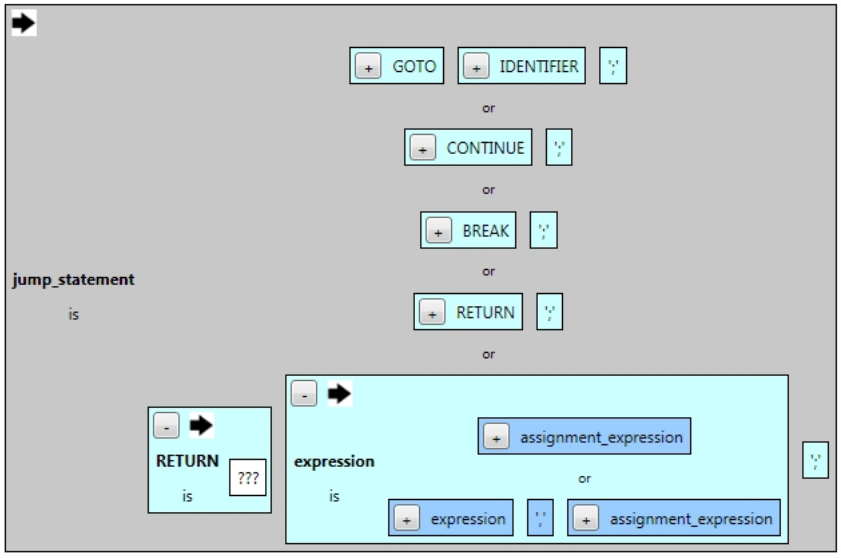

Figure 6.2: The visualisation of a rule from the ANSI C grammar.

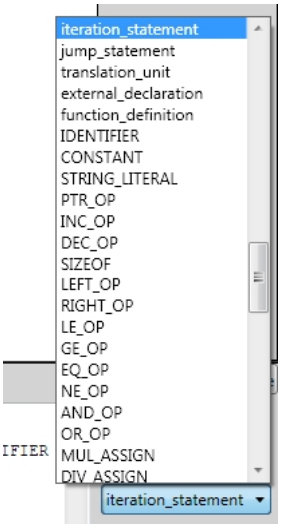

Figure 6.3: A drop down box to choose a rule to visualise.

effectively changing the start symbol in the visualised grammar. This effect can also be achieved by double clicking on a nonterminal control in the visualisation.

6.3 Building grammars

The ability to build or extend a grammar from the visualisation alone is supported in VADG. Appendix B, section B.2 shows an example of a new grammar being built using the IDE.

It should be noted that the developer chose the program for building grammars over writing in a text editor. Often using VADG to generate BNF grammars for testing, demos etc. The program is very quick at generating grammars. This could be that the developer knows VADG very well, although it goes some way to show that there is potential application with the IDE.

6.4 Parsing

The parser is able to parse most BNF grammars found on the Internet. The ANSI C grammar [18] is an example of a grammar that could be parsed. To change the operator symbols a menu system is provided, see figure 6.4. The error messages were helpful, providing information to the user about what symbol the compiler

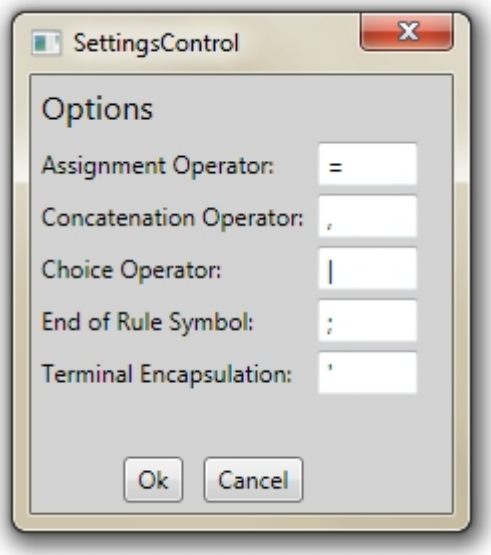

Figure 6.4: A menu to change the operators for the parser.

was expecting and what it received. However, correctly identifying where the error occurs is a problem that has not been implemented yet.

6.5 Grammar factorisation

VADG manages to find the longest duplicated rules successfully. It offers the user to change the grammar and remove the duplication. As an example, it finds a duplicate rule in the ANSI C grammar. This is shown in figure 6.5. The new rule is:

```
NewRule_7302 : IF  '('  expression  ')'  statement;
```

A small tweak is needed here to let the user choose the new rule name instead of the arbitrarily made one.

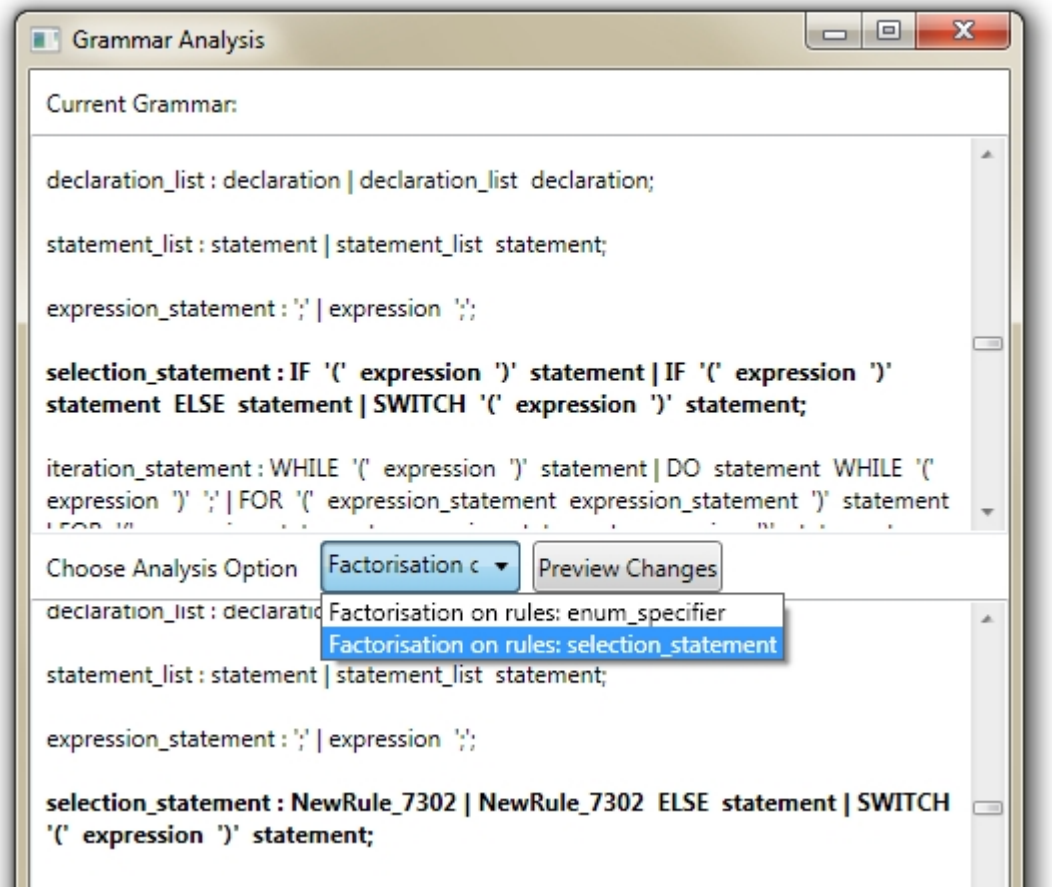

Figure 6.5: VADG factorising a rule

CHAPTER

SEVEN

TESTING

7.1 Introduction

In ([41], chapter 23), Sommerville states that the software testing process has two goals:

1. The discovery of faults or incorrect behaviour in the software.
2. Demonstration that the software meets its requirements.

The aim of this chapter is to show that the testing performed meets the goals stated above. The first goal is met by the way of unit, integration and system testing described below. In chapter 7.4 we show that the testing met the requirements of the system. *System* and *program* are used synonymously in this chapter.

7.2 Unit and Integration testing

Unit testing (also known as component testing) is where the developer tests individual methods, classes or re-usable components ([41], p538). The number of components in the system is huge and no attempt was made to formally test each item. Instead, the developer tested the components as soon as they were were coded as this is the best time to see if the functionality is correct. An example of this was during the development of the visualisations. The developer tested each visualisation on its own with minimal input to ensure the 'look and feel' of the components was correct. This was a quick way to ensure that the visualisations were correct.

This leads us into integration testing, where we need to test that components that work on their own work correctly when integrated together. This was a continual process as components were made or changed they were tested against each other. The developer created a number of 'test grammars' that were hard coded into the system. Refer back to chapter 4.2.1 where the test grammars were chosen. These test grammars were a quick and easy way of checking that all concepts of a grammar worked correctly, they also tested how the parts of the grammar integrated together. They were accessible from a static class called `TestGrammars`, for use anywhere in the system during development.

White box testing is described as testing where the tester can "examine the internal structure of the program" ([29], p11). Instead of printing verbose output, the majority of the white box testing was performed by using the debugger found in Visual Studio [28]. The debugger allows the developer to step through the code one line at a time. This was a much faster and thorough way of testing the code as the developer can see variable values in the source code as well as watch the code execution path.

7.3 System testing

System testing is where we test the whole system after development to check it conforms to the requirements ([1], p129). In ([1], chapter 11) system testing is categorised into a number of categories:

- Sanity testing tests the installation and startup/stop of the system.
- Functional testing tests that the functionality of the system meets the requirements.
- Capacity testing is where the functionality of the system is tested with a high volume of data until the system is "no longer capable of effective operation" ([1], p138).
- Performance testing tests the speed of the system as well as how fast the user takes to complete a task.
- Human factors testing tests the UI in areas such as aesthetics, easy navigation, efficiency and interactivity.

- Documentation testing tests that documents are sufficient for their intended purpose.

Due to resource constraints not all areas have been chosen for system testing. The following areas have been chosen so that some testing can be performed.

7.3.1 Functional testing

The requirements relied heavily on the functionality provided. Functional testing was therefore chosen as it is the most important part of the system. A test case details a specific test. A test case must consist of a set of input procedures and a description of the correct or expected output ([29], p14). Test cases are used for the functional testing. Some functional tests and their results are detailed in appendix C.

7.3.2 Capacity testing

Capacity testing was chosen because in this project it is more important than performance testing as we are not interested in speed, more the usability of the system under load. Test cases were also used for capacity testing. In each test case the volume of input is increased until the system becomes unusable. A system was developed which generated grammars of different sizes, pulling rule and terminal names from an English dictionary. Some capacity tests are shown in appendix C.

7.3.3 Human factors testing

If a system has a UI then "it must be thoroughly tested ... not only for functionality but also for its aesthetics" ([1], p134). This kind of testing is not as straight forward as the previous categories. The testing has a subjective element and therefore we need some subjects. Tests were completed by subjects who had little to no experience with the program. They were chosen based on the use cases identified in the requirements.

- Subject X is a novice grammar student who has had limited exposure to grammars.
- Subject Y has completed a course on compilers and their knowledge of grammars is satisfactory.

Human factors testing was completed by observing subjects perform specific tasks. A 'task sheet' was created which had a list of goals that closely resembled the use cases. The task sheets for subjects X and Y are available in appendix D. Subjects were also briefly interviewed after the tests. Appendix C shows the testing and results in detail.

7.4 Requirements testing

To ensure systematic coverage of testing ([1], p133) suggests to create a requirements validation matrix. This makes it easy to see that each requirement from the use case has been tested in system testing. Refer to appendix A for details of the requirements that were elicited from chapter 3. A requirements validation matrix for the system is given in table 7.1. Note requirements 10 and 12 have not yet been implemented, thus no testing was performed.

Requirement	Use Cases	Test Cases
1	Understand Grammar Create Grammar	Functional tests F1–F5.
2	Understand Grammar Create Grammar Extend Grammar	Functional test F4.
3	Understand Grammar Create Grammar Extend Grammar	Functional tests F5 and F6.
4	Understand Grammar Create Grammar Extend Grammar	Functional test F7.
5	Create Grammar	F8.
6	Create Grammar Extend Grammar	F9.
7	Understand Grammar	Human factors testing, tasks 1 and 5.
8	Understand Grammar	Human factors testing, task 1.
9	Extend Grammar	Human factors testing, task 5.
10	Create Grammar Extend Grammar	Not applicable.
11	Extend Grammar	Functional tests F9 and F10.
12	Extend Grammar	Not applicable.
13	Extend Grammar	Capacity testing.
14	Extend Grammar	Functional test F10.
15	Extend Grammar	Functional test F11.
16	Understand Grammars, Create Grammar	Capacity tests, machine 2.

Table 7.1: Requirements validation matrix.

7.5 Results

The unit and integration testing were some of the best and most efficient ways to discover faults before they made it through to the system testing phase. This is a quick way to test software, unfortunately, there are no results.

The system testing tested three important areas of the system: functionality, load capacity and the usability. The majority of functional tests passed, this is perhaps down to a large amount of unit and integration testing as well as the functionality being so critical to the projects success. The failures were in F9, F10 and F11 (see section C.1.2). Although the functionality is there, the functionality is unacceptable for public use and as such the tests are marked as failed.

The capacity testing shows that the system can be used for grammars up to a size of roughly 10KB. For a reference the ANSI C Grammar is 7.7KB in size and is tested as well. The results show that the system failed to handle grammars of the larger sizes. The grammars grow by factor of 10 each time. Grammars are growing but nowhere near the sizes tested. Although it would be nice to see the system cope with these absurdly large grammars, the program could still be used in the foreseeable future. There may be, however, areas of inefficient code where a simple fix could improve the systems capacity.

The results from human factors testing was overall very positive. Both subjects using the system gave positive feedback. Subject X stated “It’s an intuitive design ... everything is where it should be” and subject Y stated “My impressions [of the system] are generally positive”. Moreover, observing the subjects when they used the system the tester was surprised how quickly they picked up using the programs features. Also noting that subject Y was using the system to its full abilities when confronted with the harder tasks. The testing shows that the design and layout is correct but the system suffers from a lot of minor faults. Both subjects identified faults that seemed to annoy them. Subject Y talking about the the minor issues said they’re “more annoyances than anything else”. Users today expect a faultless UI experience. The subjects managed to complete nearly all of the tasks quickly and without help. The parser stood out as being the one area where significant development is needed. Task 4 was to find the errors in a BNF grammar using the parser. The error messages are unhelpful and misleading. Task 4 was the only incomplete task.

7.5.1 Conclusion

Sommerville's two goals to software testing have been achieved through the use of a broad set of testing. Faults in the system have been identified in all areas of testing, not least surprisingly in the human factors testing where the UI has many, minor faults. The second goal, to demonstrate that the system meets its requirements is clearly demonstrated in table 7.1.

The system has the main areas of functionality in place. This was demonstrated by functional testing as well as the subjects completing the majority of the tasks on the task sheet. The areas for improvement are the parsers error handling and the number of UI issues that need to be ironed out before the system is publicly used.

7.6 Analysis

Testing has been inadequate due to time and resource constraints. The testing covers a number of areas in the system but the number of tests is too small. A great deal more testing is required before any strong conclusions about the quality of the system are made.

Although capacity testing showed that the program is slowing with grammars of typical sizes. The grammars generated bear little resemblance to real grammars. A better approach would be to find real grammars of modern day, larger languages like Java or C#.

The subjects used for human factors testing were the developers friends, thus it could be argued that their answers were biased. Moreover, a sample size of two is inadequate to draw any conclusions towards the usability of the user interface. Testing needs to be performed with many more subjects, perhaps using a questionnaire where subjects can rate the UI on a scale. The *Likert scale* is commonly used for this [43].

The foundations have been made for someone to continue testing the software in the future. There is still more work needed to ensure good, quality testing is achieved. In essence, this section describes a framework toward a complete and systematic approach to testing software.

CHAPTER

EIGHT

CONCLUSIONS

8.1 Achievements

Referring back to chapter 1, the high level goals of the program were:

- Visualise a grammar and its rules.
- Interactive visualisation where rules can be selected and explored.
- An IDE or design environment where a grammar may be built from scratch or extended.
- Parse an existing grammar in order to visualise it.
- Export the grammar back to its original notation.

In reflection of the project the goals have been achieved.

The visualisations are clear, easy to follow and quickly understood. Furthermore, a lack of BNF notation helps to create a clearer visualisation. The expanding/collapsing boxes to browse rules ensures the grammar can be browsed quickly and with ease. Highlighting selected rules in the visualisation and BNF grammar help users understand the larger grammars by comparing the BNF rule and the visualisation.

The IDE where grammars can be built or extended using the visualisation works remarkably well. Providing a right-click menu in the visualisation where actions such as add and remove are available create an environment that allows for easy design of grammars. This IDE works well whether the grammars are small or large.

This is shown in testing where the subjects found the IDE intuitive to use and easy to build grammars.

VADG can import grammars from BNF and its derivatives using the custom built recursive descent parser. The parser is quick, taking about 1 second to parse, compile and visualise a typical grammar.

8.2 Improvements

The parser is indeed quick but it fails to identify where the errors occur when they appear. For example, when there are 'runaway arguments' such as a missing terminal quote the parser has no idea where the error occurred. The error messages are unhelpful and point the user to the wrong lines in the BNF grammar. Significant work is needed to handle lexical and syntax errors that can appear in BNF.

The program suffers from many, minor faults. This was due to the majority of development being devoted to adding features. A structured development cycle that includes feature freezing, significant testing, alpha and beta releases will result in a more robust program.

8.3 Future work

None of the extra features that are listed in section 4.2.4 have been implemented yet. Some interesting future work is presented here that utilises this representation.

8.3.1 EBNF

EBNF is now more widely used than BNF. Although there is nothing in EBNF that can't be represented in BNF, the notation is more concise for loops and optional symbols. A great addition would be including support of EBNF notation.

One way this could be achieved is by merging the syntactic diagrams and the programs expanding/collapsing of rules. A drawing of some potential EBNF visualisations are shown in figures 8.1 and 8.2. The grammar they represent is shown in listing 8.1.

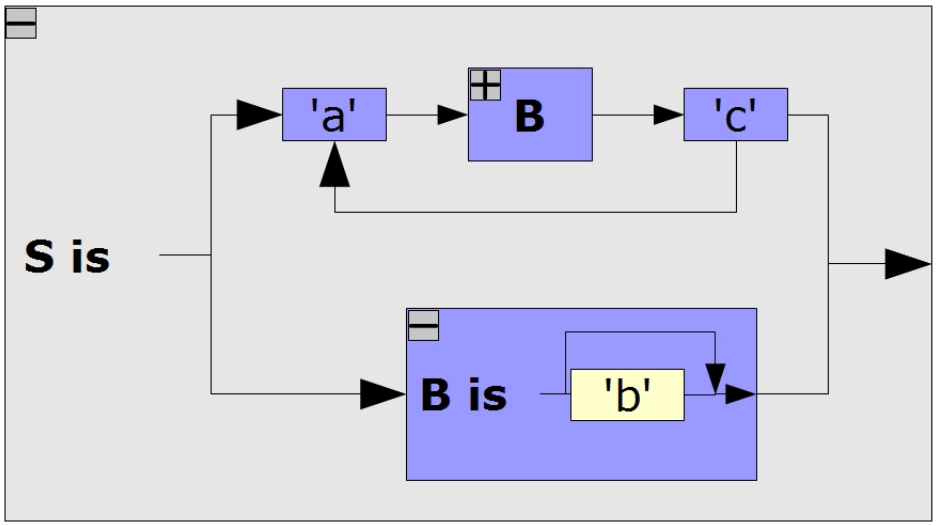

Figure 8.1: Possible visualisation of an EBNF grammar.

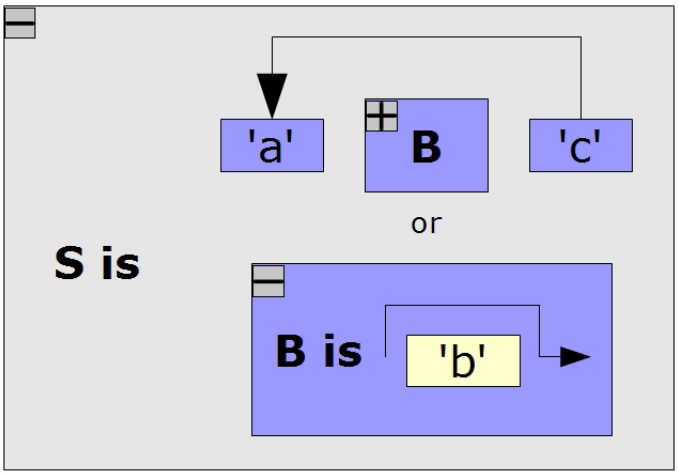

Figure 8.2: Possible simplified visualisation of an EBNF grammar.

```
1 S = ('a', B, 'c')+ | B;
2 B = ['b'];
```

Listing 8.1: An EBNF Grammar.

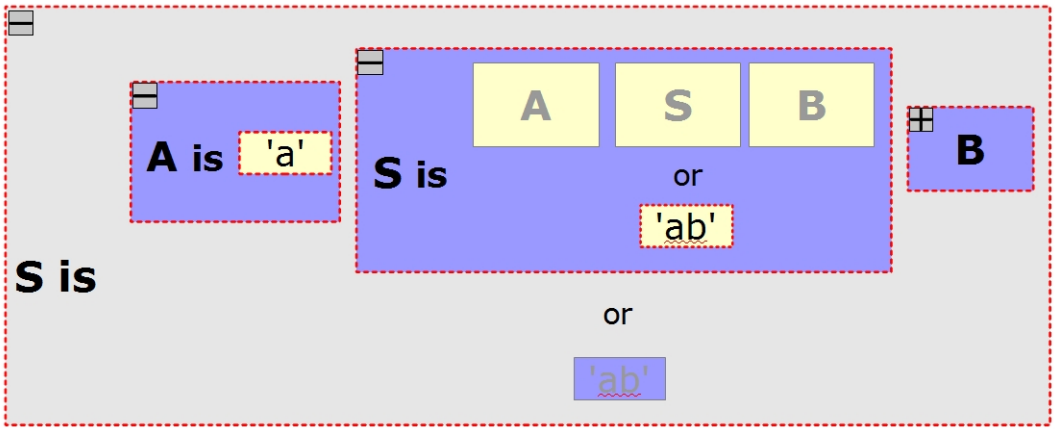

Figure 8.3: A possible visualisation of a parse tree.

```
1 Grammar
2 S = A, S, B | 'ab';
3 A = 'a';
4 B = 'b';
5
6 Text
7 aabb
```

Listing 8.2: A BNF grammar and language that it may parse.

8.3.2 Visualise Parsing

Given the grammar that describes a language and some text that conforms to the languages syntax the visualisations could show how the text is parsed. This is a visualisation of the parse tree. This requires significant work to create a parser generator so that the program could parse a language given its grammar.

In this visualisation of the parse tree the 'choices' that the grammar made are highlighted. The user can browse the parse tree, expanding and collapsing rules like normal. An example is shown in figure 8.3. The grammar and language it parses is shown in listing 8.2. The colours indicate levels as normal. The red, dotted line highlights the parse tree; the highlighted boxes can be expanded down to the terminal symbols. Notice how the rules that aren't used in the parse tree can't be expanded.

8.4 Final remarks

A solution has been introduced for visualising and designing grammars. With significant testing and corresponding bug fixing VADG could be used in the field of language and grammar development. Without further research it is unknown if the program can be an effective pedagogical tool. Nevertheless, students do benefit from different representations, especially visualisations. The representations have been shown to work with BNF grammars and there are exciting possibilities for visualising EBNF or parse trees similarly.

BIBLIOGRAPHY

[1] Imran Bashir and Amrit Goel. *Testing object-oriented software : life cycle solutions*. Springer-Verlag New York Inc., 175 Fifth Avenue, New York, NY 10010, USA, 1999.

[2] Nikolay Botev. Context Free Grammar Visualization Using SVG. On-line. Available from `http://www.svgopen.org/2009/papers/58-Interactive_Documentation_using_JavaScript_and_SVG/`, accessed 2 May 2010.

[3] R. Chimera and B. Shneiderman. An exploratory evaluation of three interfaces for browsing large hierarchical tables of contents. *ACM Transactions on Information Systems (TOIS)*, 12(4):406, 1994.

[4] N. Chomsky. Three models for the description of language. *Information Theory, IRE Transactions on*, 2(3):113–124, 1956.

[5] N. Chomsky. On certain formal properties of grammars*. *Information and control*, 2(2):137–167, 1959.

[6] Computer Architecture and Languages Laboratory (Damijan Rebernak). LISA. On-line, 2006. Available from `http://marcel.uni-mb.si/lisa/index.html`, accessed 29 March 2010.

[7] David Chappell. Introducing Windows Presentation Foundation. On-line, September 2006. Available from `http://msdn.microsoft.com/en-us/library/aa663364.aspx`, accessed 1 April 2010.

[8] Akim Demaille, Joel Denny, and Paul Eggert. Bison - gnu parser generator. On-line. Available from http://www.gnu.org/software/bison/, accessed 2 May 2010.

[9] R.M. Felder and L.K. Silverman. Learning and teaching styles in engineering education. *Engineering education*, 78(7):674–681, 1988.

[10] C.D. Hundhausen, S.A. Douglas, and J.T. Stasko. A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing*, 13(3):259–290, 2002.

[11] J. Hunt. *Guide to the unified process featuring UML, Java and design patterns*. Springer-Verlag New York Inc., 175 Fifth Avenue, New York, NY 10010, USA, 2003.

[12] International Organization for Standardardization. ISO/IEC 14977 Information Technology - Syntactic metalanguage - Extended BNF. On-line, 1996. Available from http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html, accessed 31 March 2010.

[13] Intralogic Corporation. Visual BNF. On-line. Available from http://www.intralogic.eu/, accessed 29 March 2010.

[14] Prof. Dr. Fritz Jobst and Dipl. Math. Frank Braun. Generation of Syntax Diagrams. On-line, 2007. Available from http://www-cgi.uni-regensburg.de/~brf09510/syntax.html, accessed 1 April 2010.

[15] B. Johnson and B. Shneiderman. Tree-maps: a space-filling approach to the visualization ofhierarchical information structures. In *IEEE Conference on Visualization, 1991. Visualization'91, Proceedings.*, pages 284–291, 1991.

[16] S.C. Johnson. Yacc: Yet Another Compiler-Compiler.

[17] Jonathan Leffler. BNF Grammars for SQL-92, SQL-99 and SQL-2003. On-line, July 2005. Available from http://savage.net.au/SQL/index.html, accessed 29 March 2010.

[18] Jutta Degener. ANSI C Yacc Grammar. On-line. Available from http://www.lysator.liu.se/c/ANSI-C-grammar-y.html, accessed 17 March 2010.

[19] Leon Aaron Kaplan. yaccviso - a tool for visualising yacc grammars. On-line. Available from http://www.lo-res.org/~aaron/yaccviso/, accessed 2 May 2010.

[20] S. Klusener and V. Zaytsev. Language Standardization Needs Grammarware. 2005.

[21] R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.

[22] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[23] Leon Aaron Kaplan. yaccviso - a tool for visualising yacc grammars. Technical report, Institute for Computer Languagess, Technical University of Vienna, 2006.

[24] MATIS - Database Laboratory. The BNF Web Club. On-line. Available from http://cuiwww.unige.ch/db-research/Enseignement/analyseinfo/BNFweb.html, accessed 29 March 2010.

[25] M.H. Blaha, W. Premerlani. *Object-Oriented Modelling and Design for Database Applications*. Prentice Hall, 1998.

[26] Microsoft. Dreamspark. On-line. Available from https://www.dreamspark.com/default.aspx, accessed 26 April 2010.

[27] Microsoft. The .NET Framework. On-line. Available from http://www.microsoft.com/net/, accessed 26 April 2010.

[28] Microsoft. Visual Studio. On-line. Available from http://msdn.microsoft.com/en-us/vstudio/default.aspx, accessed 29 April 2010.

[29] G.J. Myers, T. Badgett, T.M. Thomas, and C. Sandler. *The art of software testing*. Wiley, 2004.

[30] T.L. Naps, G. Rößling, V. Almstrum, W. Dann, R. Fleischer, C. Hundhausen, A. Korhonen, L. Malmi, M. McNally, S. Rodger, et al. Exploring the role of

visualization and engagement in computer science education. *ACM SIGCSE Bulletin*, 35(2):131–152, 2003.

[31] Open Source. OpenGL. On-line, 2010. Available from http://opengl.org/, accessed 23 October 2009.

[32] P. Jinks (The University of Manchester). BNF/EBNF variants. On-line, 2004. Available from http://www.cs.man.ac.uk/~pjj/bnf/ebnf.html, accessed 31 March 2010.

[33] Terence Parr. ANTLR - ANother Tool for Language Recognition. On-line. Available from http://www.antlr.org/, accessed 25 April 2010.

[34] P.C. Capon and P.J. Jinks. *Compiler Engineering Using Pascal*. Macmillan Education Ltd., Houndmills, Basingstoke, Hampshire, RG21 2XS, 1988.

[35] Ralf Lmmel (CWI, Amsterdam) and Chris Verhoef (WINS, Universiteit van Amsterdam). Browsable grammars. On-line, 1999. Available from http://www.cs.vu.nl/grammarware/browsable/, accessed 29 March 2010.

[36] J. Rees and W. Clinger. Revised report on the algorithmic language scheme. *ACM Sigplan Notices*, 21(12):37–79, 1986.

[37] Susan H. Rodger. JFLAP Version 7.0. On-line. Available from http://www.jflap.org/, accessed 28 March 2010.

[38] Susan H. Rodger. PÂTÉ. On-line. Available from http://www.cs.duke.edu/~rodger/tools/pateweb/, accessed 28 March 2010.

[39] Susan H. Rodger. Visual and Interactive Tools. On-line. Available from http://www.cs.duke.edu/~rodger/tools/tools.html, accessed 28 March 2010.

[40] Ben Shneiderman. Treemaps for space-constrained visualization of hierarchies. On-line. Available from http://www.cs.umd.edu/hcil/treemap-history/index.shtml, accessed 1 May 2010.

[41] Ian Sommerville. *Software Engineering*. Pearson Education Ltd., Edinburgh Gate, Harlow, Essex CM20 2JE, England, 2004.

[42] The Open Group. Regular Expressions. On-line, 1997. Available from http://www.opengroup.org/onlinepubs/007908799/xbd/re.html, accessed 21 April 2010.

[43] William M.K. Trochim. Likert Scaling. On-line. Available from http://www.socialresearchmethods.net/kb/scallik.php, accessed 29 April 2010.

[44] Unknown. Gorgon. On-line, 2010. Available from http://tape-worm.net/, accessed 23 October 2009.

[45] Unknown. HAAF's Game Enginer. On-line, 2010. Available from http://hge.relishgames.com/, accessed 23 October 2009.

[46] Vincent Tscherter. EBNF Parser & Syntax Diagram Renderer. On-line, 2008. Available from http://karmin.ch/ebnf/index, accessed 1 April 2010.

[47] Mark Allen Weiss. *Data structures and problem solving using Java*. Addison-Wesley, 1998.

[48] Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Commun. ACM*, 20(11):822–823, 1977.

APPENDIX

A

REQUIREMENTS FOR PROPOSED PROGRAM

ID	Identified Requirement	Category	Essential?
1	Provide visualisations of simple grammars	Functional	Yes
2	Allow the user to traverse a grammar	Functional	Yes
3	Display a textual representation that is equivalent to the visualisation	Functional	Yes
4	Highlight rules in the visualisation and textual representations	Functional	Yes
5	Build a grammar from nothing using the visualisation and IDE only	Functional	Yes
6	Save any grammar to a text file in BNF	Functional	Yes
7	Students must be able to understand the visualisations by making the visualisation intuitive to understand	Usability	Yes
8	Pre-loaded grammars must be sufficiently easy to grasp	Usability	Yes
9	Identify and separate each rule easily	Usability	Yes
10	Offer help and advice when constructing a grammar	Usability	No
11	Parse BNF grammars	Functional	Yes
12	Support parsing of EBNF	Functional	No
13	Support large grammars as easily as smaller grammars	Performance	Yes
14	A grammar that cannot be parsed should give as much detail as possible	Reliability	No
15	Ability to support grammar analysis (factorisation, expanding rules, etc.)	Functional	No
16	Program should work in university laboratory	Supportability	Yes

Table A.1: Requirements for proposed program

APPENDIX

B

PROGRAM RESULTS

B.1 Visualising a grammar

The ANSI C grammar visualised in VADG. Notice how statement has been selected. It is highlighted everywhere in the visualisation and in the text grammar.

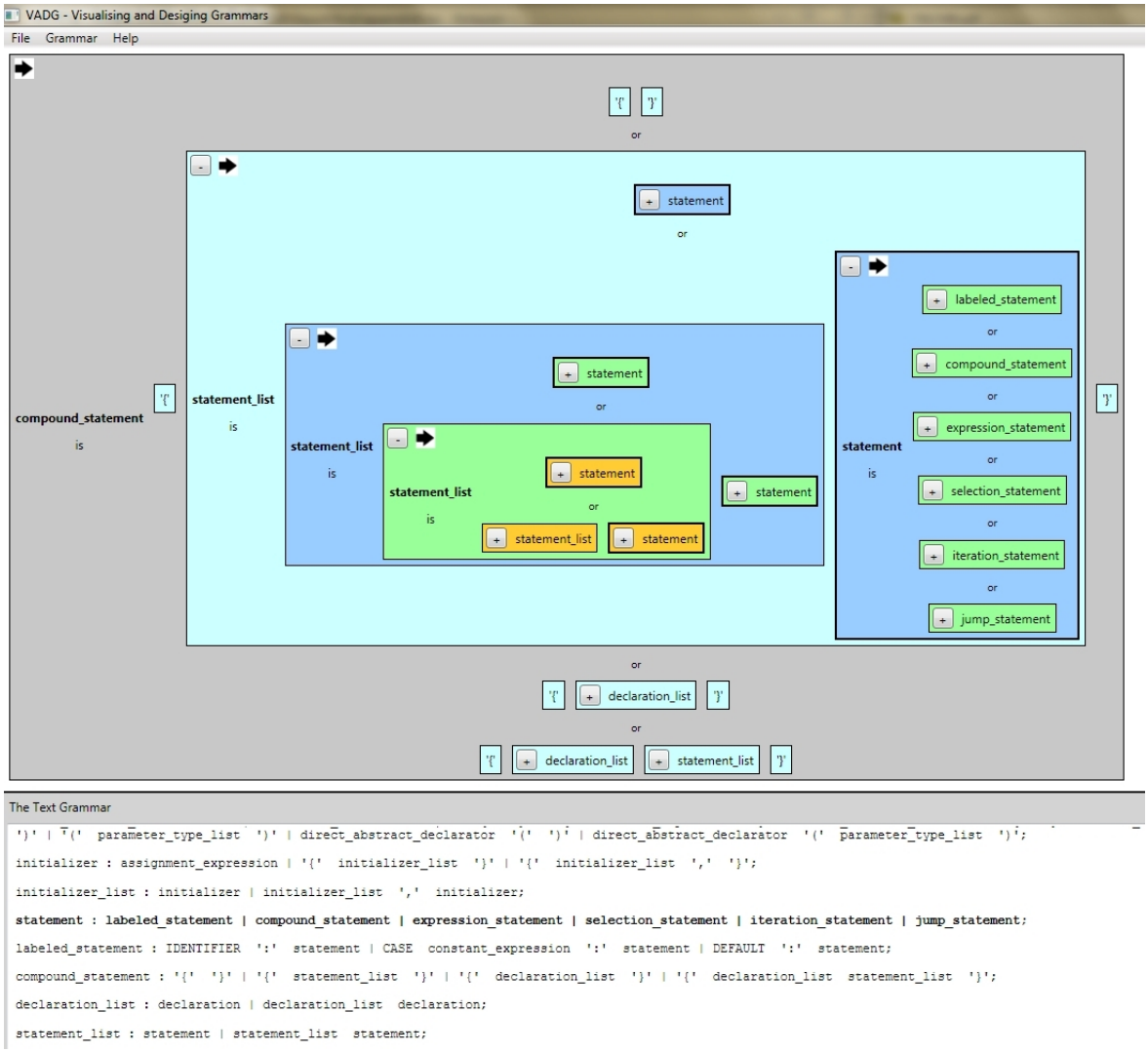

Figure B.1: Part of a grammar being visualised.

B.2 Building a grammar

The figures here demonstrate of a user building a new grammar using the visualisation. The figures are explained here:

1. User selects File, New to start a new grammar.
2. The start grammar is `S = ???;`. The user has clicked on the undefined control and a 'Create New Item' box is displayed.
3. The user right-clicks the terminal, a. The operation chosen is to add a new item after the one selected.
4. The user added a new nonterminal called T. The program automatically adds a new rule, T. Notice how the text grammar matches the visualisation.
5. The user adds a new choice item within the rule T.
6. The user has clicked the undefined item and is changing it to be a nonterminal S.
7. The program recognises S is already in the grammar and does not create a new rule.
8. The recurisve rule is expanded multiple times.

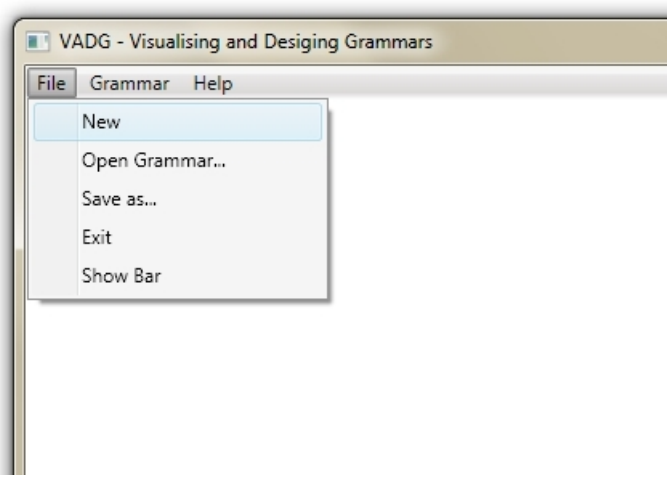

Figure B.2: Starting a new grammar

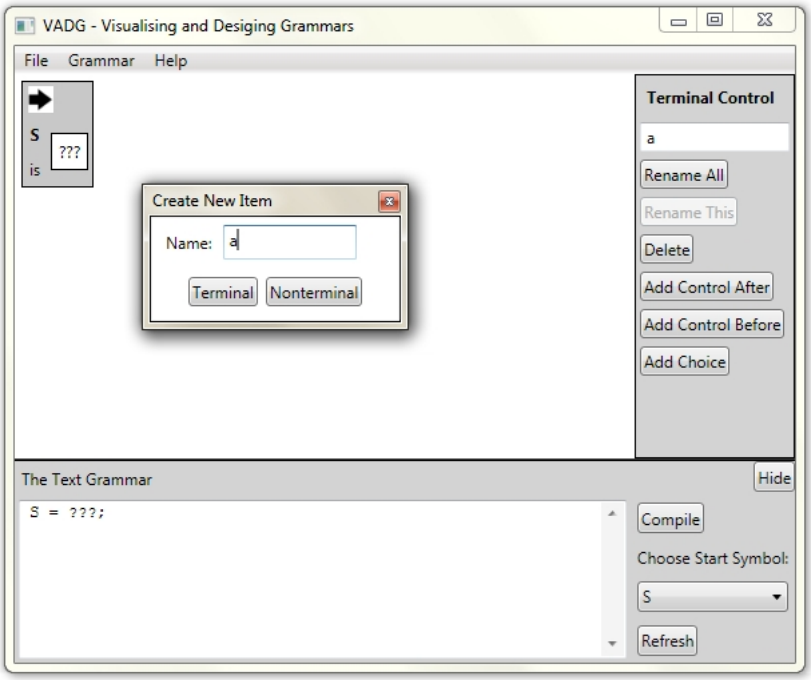

Figure B.3: Adding a new terminal called ‘a’.

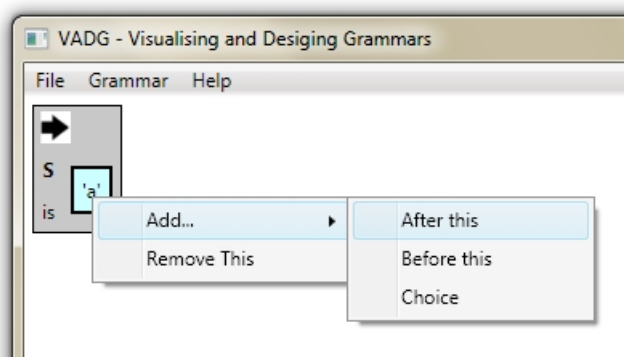

Figure B.4: Adding a new item after the terminal.

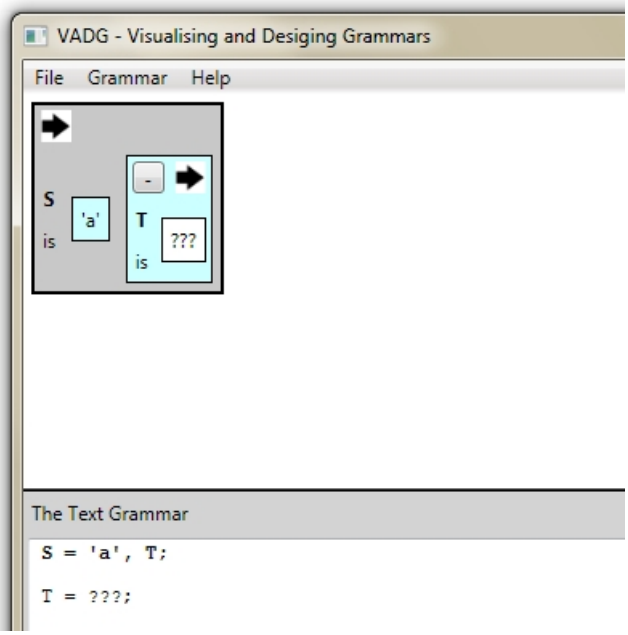

Figure B.5: The text grammar matches the visualised grammar.

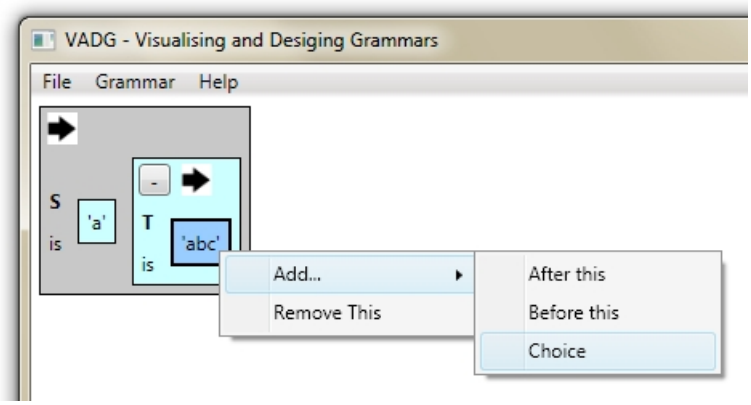

Figure B.6: Adding a choice within the rule T.

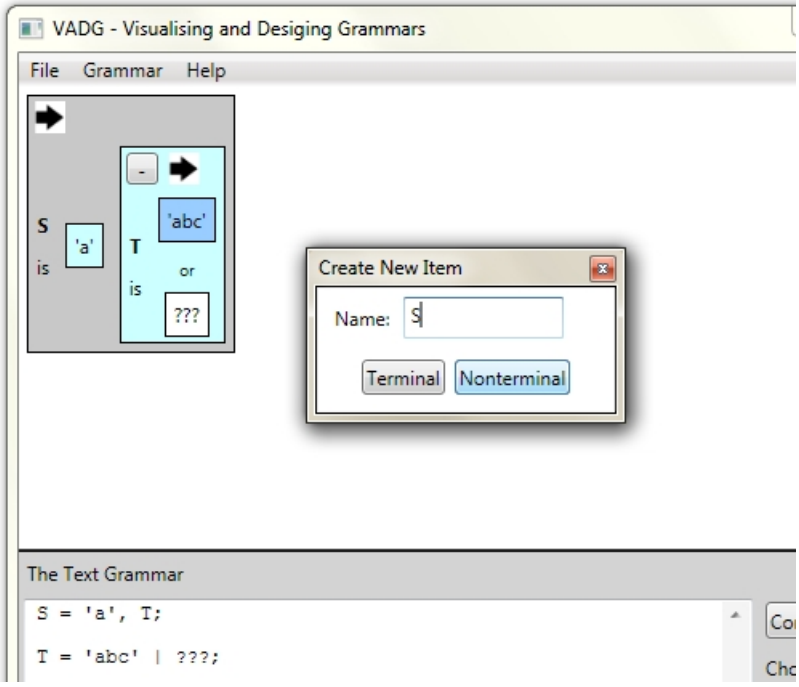

Figure B.7: User adding a new nonterminal rule S.

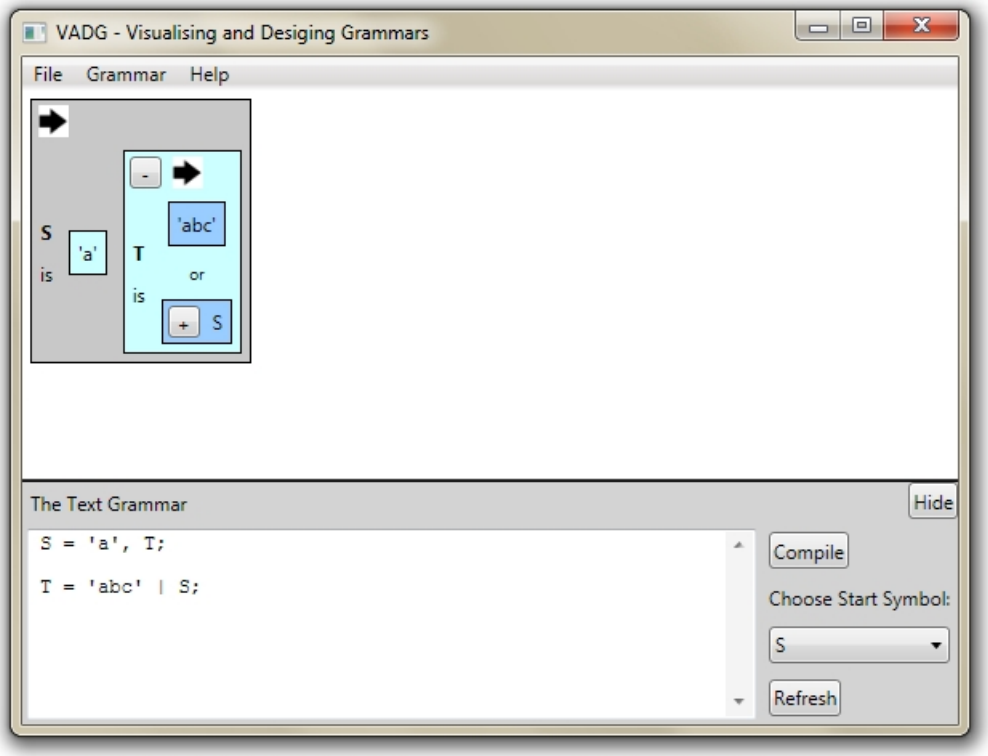

Figure B.8: The resulting program and visualised grammar.

APPENDIX

C

TEST CASES AND RESULTS

C.1 Functional testing

C.1.1 Introduction

Some functional test cases are shown here. Results are shown in table C.1.

C.1.2 Test cases

F1. Concatenation visualised

Description
Prove that a number of items to be concatenated together are shown from left to right on one row in the visualisation.
Pre-requisites
None.
Procedure
Load the test grammar ‘Concatenation’.
Expected Output
The grammar is visualised and the symbols are in the correct order and on the same row.

F2. Choice visualised

Description
Prove that choice is visualised as a number of rows of items. Each row corresponds to a choice in the grammar.
Pre-requisites
None.
Procedure
Load the test grammar ‘choice’ or any grammar that has choice operators in.
Expected Output
The grammar is visualised and the symbols for each choice are represented on a different row.

F3. Recursive rule

Description
Prove that a recursive rule is correctly visualised and can be expanded correctly.
Pre-requisites
None.
Procedure
Load the test grammar ‘recursion’ or any grammar that has a recursive rule. Expand the recursive rule a number of times.
Expected Output

Check the expanded rule is the same each time it is expanded.

F4. Traverse a grammar

Description
Prove that a grammar can be traversed correctly using the visualisation.
Pre-requisites
Access to any grammar in BNF notation.
Procedure
Change the options so the operators match the BNF notation. Open the grammar using File, Open. Expand rules, try and find some rules that do not appear near the top of the grammar.
Expected Output
The program should visualise the grammar, at first only showing the start symbol rule. The program should allow the user to 'walk around' the grammar and find all the rules.

F5. Text grammar equates to visualisation

Description
Prove that the visualisation and text grammar are the same at all times.
Pre-requisites
A small BNF grammar.
Procedure
Open the grammar. Check each rule in the text grammar is correctly visualised by using the drop down box to change the start symbol.
Expected Output
Rules are visualised correctly.

F6. Text grammar always matches

Description
Prove that the text grammar will always match the visualised grammar. When the user is building a grammar.
Pre-requisites
A grammar visualised.
Procedure
Check the visualised grammar matches with the text grammar. Perform all the operations available: add after, add before, add choice and remove on both nonterminal and terminal symbols.
Expected Output
After each operation performed ensure the visualisation and text grammar have updated accordingly.

F7. Rules are highlighted

Description
Prove that when a user clicks a rule in the visualisation the rule is highlighted both in the visualisation and the text grammar.
Pre-requisites
None.
Procedure
Use all the test grammars in turn. In each test grammar, select a rule. If possible, expand a rule and select a symbol from the expanded rule.
Expected Output
The selected symbol should be highlighted by way of a bold border in the visualisation and bold font for the rule in the text grammar.

F8. Build the test grammar

Description
Prove the building of grammars works by trying to build the test grammars from new.
Pre-requisites
None.
Procedure
Load each test grammar in turn. Observe the grammar. Click File, New and attempt to re-create the grammar using the visualisation IDE only.
Expected Output
All test grammars should be able to be built using the visualisation IDE.

F9. Load and Save a grammar

Description
Prove you can load and save a grammar to a text file.
Pre-requisites
A BNF grammar in a text file.
Procedure
Change the operators to match the BNF notation of your grammar. Click File, Open and selected the BNF grammar. Edit the grammar slightly. Save grammar to a new text file.
Expected Output
The saved grammar is in the same format as when it was loaded except for the change made.

F10. Test parser

Description

Prove the parser delivers helpful error messages.
Pre-requisites
A BNF grammar in a text file.
Procedure
Deliberately put some errors in the text file. Attempt to parse the grammar. Try and fix each error from the compiler messages.
Expected Output
The parser provides helpful error messages such that the user can remove all the errors.

F11. Testing the factorisation

Description
Test that the system correctly identifies a duplicate rule that can be factorised.
Pre-requisites
A BNF grammar where there are some duplicate rules.
Procedure
Click Grammars, Analysis. The program should display some rules that can be factorised. Select one and click apply.
Expected Output
Observe that the grammar is indeed factorised and a new rule is appended to the grammar.

C.1.3 Results

Test	Pass?	Test Comments
F1	Yes	None.
F2	Yes	None.
F3	Yes	None.
F4	Yes	The visualisation grows very quickly and a way of keeping the rule you have expanded centred on the screen would be useful.
F5	Yes	None.
F6	Yes	None.
F7	Yes	None.
F8	Yes	None.
F9	No	The grammar does save in the correct notation but my formatting and comments have been lost!
F10	No	The parser finds all the errors but some of the messages are not helpful. The parser regularly fails to identify the correct point of failure nor the correct line.

Test	Pass?	Test Comments
F11	No	As expected for most rules. An error has appeared when factorising rules with choice. The factorisation does not seem to acknowledge that choice has higher precedence than concatenation.

Table C.1: Results from functional testing.

C.2 Capacity testing

C.2.1 Introduction

The capacity tests were tested on two machines:

1. A 2.4 GHz processor, 2.0GB of RAM. Operating system: Windows 7.
2. A 1.6 GHz processor, 1.0GB of RAM. Operating system: Windows XP.

See table C.3 to see the results of the capacity tests.

C.2.2 Test cases

C1. Visualise grammar

Description
Test the program at visualising a grammar.
Pre-requisites
Access to a grammar.
Procedure
File, Open and open the grammar. Use the visualisation, expand and close rules.
Expected Output
The visualisation should work fine. The rules should expand and collapse as soon as the user presses the button.

C2. Edit grammar

Description
Test the program at editing a grammar.
Pre-requisites
Access to a grammar of size.
Procedure
File, Open and open the grammar. Use the visualisation to:

- Add a new rule.

- Change a rule.
- Add some new symbols in a rule.
- Delete some rules.

Expected Output
The program should allow the grammar to be editing without any slowness or delay. If there is a delay, the user should be notified that the program is working.

C3. Factorise a grammar

Description
Test the program at factorising a grammar.
Pre-requisites
Access to a grammar.
Procedure
File, Open and open the grammar. Click Grammar, Analysis. The possible factorisations should be accessible. Choose one and apply.
Expected Output
The grammar should factorise the rule with no delay or slowness.

C.2.3 Results

Machine	Grammar Size	Test	Test Comments
1	ANSI C (7.7KB)	C1	Compilation was instant. Expanding and collapsing rules is instant. There is no delay between when a user selects a rule to it being highlighted.
1	ANSI C (7.7KB)	C2	The operations on the grammar all took less than 1 second. The responsiveness to operations is very quick, almost instant.
1	ANSI C (7.7KB)	C3	The factorisation finding and applying the factorisation is almost instant.
1	100 Rules (9.13KB)	C1	Compilation took 2 seconds. There is a slight delay (1 second) when a user selects a rule to it being highlighted.
1	100 Rules (9.13KB)	C2	The operations on the grammar all took about 1 second. The responsiveness to operations has a very slight but noticeable delay.

Machine	Grammar Size	Test	Test Comments
1	100 Rules (9.13KB)	C3	The factorisation finding and applying the factorisation is almost instant.
1	1000 Rules (89KB)	C1	Compilation took 6 seconds. Expanding a rule is instant. However, Collapsing a rule takes 5 seconds. There is a significants delay (5 seconds) when a user selects a rule to it being highlighted.
1	1000 Rules (89KB)	C2	The operations on the grammar all take about 5 seconds to complete. The responsiveness is unacceptable for editing a grammar. The delay is too long and frustrating.
1	1000 Rules (89KB)	C3	The factorisation finding and applying the factorisation only takes 3 seconds.
1	10000 Rules (892KB)	C1	The program took longer than one minute to compile. Program takes longer than one minute to respond. Other tests abandoned.
2	ANSI C (7.7KB)	C1	Compilation took 2 seconds. Expanding and collapsing rules takes 2 seconds. There is a delay between the user clicking the button and the action happening.
2	ANSI C (7.7KB)	C2	The operations on the grammar take about 3 seconds to complete. The delay is just acceptable.
2	ANSI C (7.7KB)	C3	The factorisation finding and applying the factorisation takes 5 seconds in total.
2	100 Rules (9.13KB)	C1	Compilation took 3 seconds. There is a delay (3 seconds) when a user selects a rule to it being highlighted.
2	100 Rules (9.13KB)	C2	The operations on the grammar all took about 1 second. The responsiveness to operations has a very slight but noticeable delay.

Machine	Grammar Size	Test	Test Comments
2	100 Rules (9.13KB)	C3	The factorisation finding and applying the factorisation takes 10 seconds in total.
2	1000 Rules (89KB)	C1	Compilation took 12 seconds. Expanding a rule is instant. However, Collapsing a rule takes 12 seconds. There is a significant delay (14 seconds) when a user selects a rule to it being highlighted.
2	1000 Rules (89KB)	C2	The operations on the grammar all take about 13 seconds to complete. The responsiveness is totally unacceptable for editing a grammar. The delay is too long and frustrating.
2	1000 Rules (89KB)	C3	The factorisation finding and applying the factorisation takes 30 seconds in total.
2	10000 Rules (892KB)	C1	The program took longer than one minute to compile. Program takes longer than one minute to respond. Other tests abandoned.

Table C.2: Results from capacity testing

C.3 Human factors testing

C.3.1 Introduction

The tasks the subjects had to perform can be found in appendix D. The writer then observed the subjects perform the tasks, offering no guidance. Subject Y had more tasks that tested the compiler and editing a large grammar. An interview with probing was carried out afterwards.

C.3.2 Results

Observations

Subject	Task	Observatory Comments
X	1	Subject tried 'File, Open Grammar' before finding the test grammars in another menu. Subject didn't understand how Choice was represented. 'or' text possibly too small. Struggled with recursive grammar. When prompted, subject expanded S a number of times.
X	2	Subject managed to build the grammars surprisingly quickly with no help. Subject: "That was easy". When subject made an error he easily rectified it. Struggled to make recursive grammar. Subject didn't know how to rename a terminal, the button is "Rename All" should be "Rename this".
X	3	Subject managed to build grammar with no help. Subject did manage to crash program - losing their progress. The 'action bar' was still available when the Subject had deselected a control.
Y	1	Subject found test grammars very quickly. Easily identified what they can parse.
Y	2	Subject clicked 'terminal' for a terminal but didn't realise it was also a submit button when creating a new control. User also double clicks the undefined rule instead of single-clicking.
Y	3	Subject hit 'Enter' once they had typed in a controls name but program did not do anything.
Y	4	The subject did not manage to fix any errors. The compiler error messages told the subject to look on line 17 but the errors were nowhere near there. Error messages showed irrelevant rules. Complete failure.
Y	5	User struggled to find iteration statement. The rules list is not in any order. Once found, managed to extend the c grammar to add the for each loop. Used the other loops in the rule to find the correct rules to use in their new rule. Subject used program to it's full abilities here. Very happy.

Table C.3: Results from observations of human factors testing

Interview transcript with subject X

Interviewer: What was your overall impression when using the system?
Subject X: I think it's easy to use. It's an intuitive design ... everything is where it

should be. I think there are some minor errors with the program, very minor though. Like, Why can't I de-select a control or why can't I use the keyboard?
Interviewer: They're features I still need to add. Are there any major issues with the system?
Subject X: No, as long as you stop it from crashing.
Interviewer: What is your overall impression of the visualisations?
Subject X: I think they do the job. Clean and uncluttered but I think they could look more modern. I like the colours to demonstrate each level but I don't like this grey.
Interviewer: Do you want to choose your own colours?
Subject X: Erm, some people might, I guess.
Interviewer: What do you think is missing from the system?
Subject X: I want a zoom button. Look how big the grammar gets [subject demonstrates].
Interviewer: Even if you couldn't read the text?
Subject X: Yes, I want an overall picture.
Interviewer: Anything else?
Subject X: What about comparing two grammars side by side? One here and here [subject points to top and bottom of program screen]. I know in programming a lot of things are repeated. What about a menu of pre-made grammars [rules]? I could get repeated things from the left and drag and drop them into my grammar.
Interviewer: Thank you for your time. Do you have anything else to say?
Subject X: No, thank you.

Interview transcript with subject Y

Interviewer: What was your overall impression when using the system?
Subject Y: My impressions are generally positive. I think there are some minor usability problems that need to be ironed out. I like the fact I can do everything I want from the builder. Once you've done one rule you can do them all because everything is consistent. It's a shallow learning curve. I think the right-click menu is great for performing actions, makes it much quicker.
Interviewer: Excellent. What is your overall impression of the visualisations?
Subject Y: I like the way I can find out a rule by just clicking the button. I can also explore that rule further, it's handy. I think the concatenation on one line mimics BNF and there is enough separation between choice, on each line is a good idea. It's good that you don't show the BNF operators, I don't want to be bogged down with that. But I can also distinguish terminals and nonterminals. I don't need to remember the syntax.
Interviewer: Any negative aspects of the visualisation, the aesthetics?
Subject Y: I don't care about colours. I'm happy that it distinguishes each level with the colours.
Interviewer: OK, we've covered the positive. Are there any negative aspects?
Subject Y: There's some minor GUI issues here and there, but no problems have

become apparent from using the program. I think it's just those missing features. I want to rename a single rule, not all of them. The minor issues are more annoyances than anything, I'm sure they can be easily fixed and aren't worth mentioning.
Interviewer: Sure, thank you. So, how did you find task 4, where you used the parser?
Subject Y: I think the syntax checking would help people. I mean, I couldn't find the errors but at least I know there are errors in the BNF. I think it defiantly needs some work but also I understand that it's difficult to write it.
Interviewer: OK, so task 5. Editing that C grammar. How did you find it?
Subject Y: I think it would definitely would be harder to do what I did without your program, unless I knew the grammar. I found it easy to know what the rules were because they were right there. I bet in the text grammar they were placed all over the place. I also liked the fact I didn't have to care about syntax. Makes it easier to see the grammar.
Interviewer: Any issues in task 5?
Subject Y: Not really, really it was just my knowledge of for each loops and grammars that limited me, not your program.
Interviewer: Thank you for your time. Do you have anything else to add?
Subject Y: Nope, don't think so. Thank you.

APPENDIX

D

TASK SHEETS

Subject: X

Task Sheet

Please start the program.

1. Find and load the four test grammars that are available in the program. Can you explain what each test grammar can parse? Basic, Long, Choice, Recursive.

2. Can you recreate the following grammars only using the builder? Click File, New for each grammar:

(a)
```
S = 'A';
```

(b)
```
S = A, B, C;
A = 'A';
B = 'B';
C = 'C';
```

(c)
```
S = 'a' | 'b';
```

(d)
```
S = 'a', S, 'b' | 'ab';
```

3. Can you create this grammar only using the builder?

```
MathOperation = Number, Operator, Number | Number, Operator,
MathOperation;
Number = '[0-9]' | '[1-9][0-9]+';
Operator = '+' | '-' | '/' | '*';
```

Subject: Y

Task Sheet

Please start the program.

1. Find and load the four test grammars that are available in the program. Can you explain what each test grammar can parse? Basic, Long, Choice, Recursive.

2. Can you recreate the following grammars only using the builder? Click File, New for each grammar:

```
(a)
S = 'A';
```

(b)

```
S = A, B, C;
A = 'A';
B = 'B';
C = 'C';
```

(c)

```
S = 'a' | 'b';
```

(d)

```
S = 'a', S, 'b' | 'ab';
```

3. Can you create this grammar only using the builder?

```
MathOperation = Number, Operator, Number | Number, Operator,
MathOperation;
Number = '[0-9]' | '[1-9][0-9]+';
Operator = '+' | '-' | '/' | '*';
```

4. Click File, Open and choose the grammar labelled 'c-grammar.txt'.
   a) Change the operators to match the BNF notation.
   b) Using the parser error messages attempt to fix the problems that exist in the grammar by changing the text in the dual view.

5. Open the grammar 'ansi-c.txt'. Find the 'iteration_statement' rule. Using your knowledge of programming and the program visualisations extend the rule so that it has a new loop, of type foreach.