RHODES UNIVERSITY
Where leaders learn

Visualisation of Abstract Syntax Trees for Coco/R

Submitted in partial fulfilment
of the requirements of the
Bachelor of Science Honours degree in
Computer Science
of Rhodes University

Etienne Raymond Stalmans

Grahamstown, South Africa
October 27, 2010

Abstract

Compiler theory is a core module in most computer science courses. The tools and techniques used in teaching compiler theory has not advanced to keep pace with modern teaching techniques. One of these teaching techniques is the use of visualisation. Through visualisation students are provided with a visual metaphor of the problem, which aims to improve understanding by encouraging cognitive learning. Coco/R, a popular compiler generator, has no integrated development environment or visualisation facilities. By creating an integrated development environment, which is capable of visualising data structures such as abstract syntax trees and syntax graphs, the effectiveness of Coco/R as a teaching aid is increased. Visualisation is done automatically allowing for a layer of abstraction to be inserted between the user and the problem domain. This ensures that students can focus on learning the material without needing to learn a meta language first. The visual representation of the parsing process can more easily convey the underlying actions and allows students to form a mental model of how compilers function.

ACM Computing Classification System Classification

Thesis classification under the ACM Computing Classification System (1998 version, valid through 2010):

D.2.3[Coding Tools and Techniques]: Program editors
D.2.6[Programming Environments]: Graphical environments
D.3.1[Formal Definitions and Theory]: Semantics, Syntax
D.3.4[Processors]: Code Generation, Compiler Generators, Parsing
D.4.1[Grammars and Other Rewriting Systems]: Grammar Types

Acknowledgements

I acknowledge the financial and technical support of this project of Telkom SA, Comverse SA, Stortech, Tellabs, Easttel, Bright Ideas Projects 39 and THRIP through the Telkom Centre of Excellence at Rhodes University.

I would also like to acknowledge the support of the NRF, who provided financial support that made this year and this project possible.

A special thank-you to Dr Karen Bradshaw, without whom this thesis would never have been completed. Thank-you for all the hours spent providing suggestions, assistance and helping with corrections. I would also like to thank Prof. P Terry for all his input and feedback provided and for peaking my interest in compiler theory and specifically Coco/R.

Thank-you to my family and friends for the moral support and encouragement over the course of the year.

Contents

List of Figures

Listings

Chapter 1

Introduction

1.1 Problem Statement

Modern programming languages are high-level in nature [35], while compiler generation remains a low level operation [37]. Coco/R, a popular compiler generator tool, allows for the generation of compilers for any grammar. This is achieved through high-level abstraction, allowing for simplification of the compilation process. Currently no integrated development environment (IDE) exists for Coco/R, which poses a problem for young programmers who are used to modern development environments with syntax highlighting and code visualisation. Abstract Syntax Trees (ASTs) can be useful in the analysis and comparison of programs [37]. The purpose of this project is to produce an IDE for Coco/R. Furthermore, the project aims to help programmers, new to the field of compilers, understand the compiler generation process and visualise the output of syntax analysis. This will be done through the construction and visualisation of syntax graphs and ASTs.

It has been shown that visualisation can be a useful teaching aid and it is hoped that the user will be able to examine the generated AST and gain a greater understanding of the underlying program. This implementation aims to improve on work done in compiler visualisation by programs such as VCOCO [33] and ANTLR [28]. VCOCO was developed for use with Coco/R and provides step-by-step information about the compiler generation process [33]. VCOCO does not, however, provide visualisation; thus, it is necessary to produce a new tool, capable of visualising the AST and compile time data structures.

1.2 Research Goals

Based on the above problem statement, the objectives of this research are:

- Produce an Integrated Development Environment for Coco/R.
- Visualise abstract syntax trees produced during syntax and constraint analysis.
- Increase understanding of the compilation process for new users.

The development of an IDE for Coco/R will allow for easier development of grammars used in compiler generation. Visualisation will assist in the understanding of compilers and will aid in the optimisation of code generation. The extension of Coco/R should not negatively affect performance or ease of use. Furthermore, the extension should not change the core functionality of Coco/R in any way.

1.3 Motivation for research

Compiler theory is a core module in most computer science courses offered at university level. Subsequently Coco/R is used as a teaching aid for the compiler theory course offered at Rhodes University. Current implementations of Coco/R do not provide an IDE and thus users require third party tools to write the grammar and are required to manually perform the compilation of grammars and the execution of the generated parser. This means students are required to learn how to perform these actions before being able to use Coco/R. With the development of an IDE students will be able to start using Coco/R almost immediately. The IDE will assist in the development of grammars and will make error identification and correction easier for novice users. It is hoped that the visualisations provided by the IDE will help increase student understanding of compiler functionality. The use of visualisation will assist students in gaining a greater understanding of the data structures produced during syntax analysis.

1.4 Document structure and outline

Chapter two contains a literature survey that covers related work and provides a theoretical background upon which the rest of the thesis is based.
Chapter three discusses Coco/R in more detail and explains the basic mechanism behind recursive descent compiler construction.
Chapter four examines visualisation as a teaching aid and identifies the requirements of a good user interface.
Chapter five outlines the IDE produced for Coco/R and highlights the main features thereof.
Chapter six discusses how the AST construction code is incorporated into the existing version of Coco/R. It further explains how the resultant AST is retrieved from the generated parser through the use of reflection.
Chapter seven lists example ASTs generated using the modified version of Coco/R and discusses problem areas identified.
Finally, *Chapter eight* concludes the thesis and suggests further extensions.

Chapter 2

Literature Survey

To gain a firm understanding of the underlying problem domain it is essential to look at the following research areas: *grammars*, *compiler generators*, *abstract syntax trees* and *visualisation*. Specifically we will focus on attributed, LL(1) compliant grammars. These grammars can be used by compiler generators to generate recursive descent compilers automatically. During this compilation the AST is produced. There are numerous uses for ASTs, amongst others, for describing the compilation process. Furthermore, due to the nature of abstract syntax trees they are easy to visualise. Visualisation has been shown to be helpful in the teaching of complex computer algorithms and it is hoped that these visualisation techniques, when applied to compiler theory, will be just as helpful. In order for visualisation to be effective numerous design challenges exist which need to be identified and explored.

2.1 Grammars

Grammars are used to formally specify the syntax of a language. A grammar can be described as a structure $<$ N,T,P,S $>$ [37, 42], where N represents a set of non-terminals, T a set of terminals, P a set of productions and S represents a non-terminal starting symbol. A production is a syntax equation that relates two strings and describes how they can be transformed into each other. Terminal symbols are literal strings that cannot be broken into smaller units. Non-terminals are symbols that can be replaced; therefore they can be composed of terminal and non-terminal symbols. Grammars must conform to certain restrictions to be used in the automatic construction of parsers and compilers. Two main restrictions are that grammars need to be cycle-free and unambiguous. Ambiguity in a grammar occurs when that grammar has more than one parse tree for a given input string [2]. Further, grammar types can be classified using the Chomsky hierarchy, which consists of four classes of grammar: Type3, Type2, Type1 and Type0 [37]. Most modern languages can be described using Type2 or context-free grammars. In a context-free grammar the left-hand side of every production consists of a single non-terminal, while the right-hand side consists of a non-empty sequence of terminals and non-terminals.

2.1.1 LL(1) grammars and attributed grammars

Grammars are said to be LL(1) compliant if it is possible to evaluate the grammar from left to right, while only looking one terminal ahead [2, 42]. LL(1) conflicts can occur under three different conditions; these being, explicit alternatives, options and iterations. Mössenböck, Wöß and Löberbauer [42] described these conditions as follows:

> In EBNF grammars, there are the following three situations where LL(1) conflicts can occur. (Greek symbols denote arbitrary EBNF expressions such as **a[b]C**; FIRST(α) denotes the set of terminal start symbols of the EBNF expression α and FOLLOW(A) denotes the set of terminal symbols that can follow the non-terminal A)
>
> **Explicit alternatives**
>
> e.g. A = $\alpha \mid \beta \mid \gamma$. check that $\text{first}(\alpha) \cap \text{first}(\beta) = \{\} \wedge$
> $\text{first}(\alpha) \cap \text{first}(\gamma) = \{\} \wedge$
> $\text{first}(\beta) \cap \text{first}(\gamma) = \{\}$.
>
> **Options**
>
> e.g. A = $[\alpha]\beta$. check that $\text{first}(\alpha) \cap \text{first}(\beta) = \{\}$
> e.g. A = $[\alpha]$. check that $\text{first}(\alpha) \cap \text{follow}(A) = \{\}$
>
> **Iterations**
>
> e.g. A = $\{\alpha\}\ \beta$. check that $\text{first}(\alpha) \cap \text{first}(\beta) = \{\}$
> e.g. A = $\{\alpha\}$. check that $\text{first}(\alpha) \cap \text{follow}(A) = \{\}$

Knuth [16] defined attributed grammars, which can be used to describe the translation of languages. EBNF, as described by Wirth [41], is a notation used to describe a language consisting of meta-symbols, which are used to simplify the expression of grouping, alternatives, optional symbols and much more [37, 41]. The attributes of the grammar consists of *non-terminals*; these can be described as input attributes, which provide context information, and output attributes, which provide results calculated during the processing of the non-terminals. The semantic actions are executed as statements in an imperative programming language, during the parsing process. Recursive descent parsers require grammars to be LL(1) compliant. The LL(1) restriction is enforced as the parser needs to be able to select between alternatives with a single look ahead symbol.

2.2 Compiler Generators

Compiler generators are used to construct human-readable parsers automatically [27]. Traditionally hand-built, recursive-descent parsers were used to recognise languages [27, 42]. These hand-built parsers where able to resolve LL(1) conflicts using semantic information or by performing a multi-symbol look ahead. The first automatic compiler generators where constructed to produce bottom-up parsers. The bottom-up or LALR(1) parsers are more powerful and are not affected by the same LL(1) restrictions as recursive-descent parsers. However, LALR(1) parsing does not allow for integrated semantic processing as semantic actions can only be performed at the end of productions. To solve this problem recursive-descent parser producing compiler generators where created. Two early compiler generators are JavaCC [3] and ANTLR [27]. JavaCC generates recursive-descent parsers that are able to resolve LL(1) conflicts. ANTLR produces recursive descent parsers that process LL(k) grammars, where $k > 1$. Furthermore, the grammars processed by ANTLR use predicates to aid in the resolution of LL(1) conflicts. The ANTLR compiler generator comes with a tree generator, SORCERER [27, 42].

2.3 Abstract Syntax Trees

Aho *et al.* [2] provide a formal description for Abstract Syntax Trees (AST): "In an abstract syntax tree for an expression, each interior node represents an operator; the children of the node represent the operands of the operator". The expression ***a-b*c*** can be represented by an AST, see Figure 2.1, where the root represents the operator *. The sub expressions ***a-b*** and ***c*** are represented by a sub-tree and leaf node respectively. The grouping of ***a-b***, reflects how operators on the same precedence level are evaluated from left-to-right [2, 37]. A useful feature of an AST is that it can be used to capture the essential features of a program [37]. The AST provides a conceptual interface for the representation of a program's syntax [11]. The parsing process evaluates a grammar on a

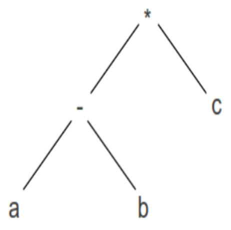


Figure 2.1: Abstract Syntax Tree for a-b*c

symbol by symbol basis. This process is recursive in nature and can be represented using parse trees [14]. An AST consists of inter-related nodes which represent data structures. The construction of the AST is done bottom-up. The parser recursively works its way down each branch, as it exits each method and travels back up the branch, it adds the last visited node to the tree [14, 37]. The construction of syntax trees can be used to perform simple type checking [2], which is performed by matching actual types with expected types as the node is about to be added to the tree. Further analysis can be done once the AST has been constructed by executing code fragments at each node in the AST. Thus, it can be seen that the AST either exactly mirrors or is an exact subset of the control sequence that has created it [11]. The creation of an AST can be performed at different levels and using different techniques. When manually constructing an AST it is possible to define node types according to the structure of the target grammar. As demonstrated by Terry [37], an AST node can be defined using a class structure.

Listing 2.1: Description of BinaryNode type using a class structure [37, p.470]

```
class BinaryNode: Expr
{
 int op;              //infix operator
 Expr left ,right;   //operand subtrees

 //constructor
 public BinaryNode(Type type,int op,Expr left , Expr right)
}
```

A similar approach can be taken to construct nodes representing other type structures such as statements. Once all node types and their structures have been defined, the user can manually perform AST construction. The AST construction can be performed by attributing the user grammar with semantic actions for node creation. As described by Terry [37], the user grammar is attributed so that each component of the tree is synthesised as an output parameter.

Listing 2.2: Attributed grammar showing manual calls to construct an AST node [37, p.475]

```
Condition<out Expr exp>
= Expression<out exp>
  (. if (exp.type != Type.boolType)
          SemError("boolean expression required");
  .).
RelExp<out Expr a> (. int op; expr b; .)
= AddExp<out a>
  [ RelOp<out op> AddExp<out b>
  (. if (!a.type.IsArithType() || !b.type.IsArithType())
       SemError("incomparable operands");

     a = Expr.MakeBinaryNode(Type.boolType, op, a, b);
  .)
  ] .
```

By studying this description we can see how the call to *MakeBinaryNode* is used to construct a leaf node of type boolean. This leaf node has two of its own leaf nodes; with left node *a* and right node *b*. From this it is clear that AST construction can be done by attributing the grammar with semantic actions that construct the nodes of the tree and link them together [37]. Using this method requires that the user creates the AST class first and understands where node creation semantic actions need to be inserted in the grammar. Furthermore, the user needs to understand the nature of each node so as to ensure that the node creation statements are implemented in the correct locations within the grammar [37]. This is a problem when trying to construct an automatic AST generator that is generic enough to work with most user supplied grammars.

The benefits associated with constructing an AST during parsing are numerous and include the following [37, 42]:

- Constructing a tree and then traversing it, allows the compiler to overcome the "declare before use" constraint of languages such as Parva and Pascal.
- If tree construction occurs in the syntatic/semantic analysis phase, it is possible to perform code optimisation. This optimisation can be done before actual code generation is performed.
- Through the use of a tree constructor, we can perform constraint analysis. This allows checking that the expression is defined correctly and storing the value in the symbol table, rather than calling code creation methods.
- Analysis of the tree construct can also lead to the identification of statements that require no code generation at compile time.

To take full advantage of these benefits it is essential that students understand AST construction. Through understanding they will be able to interpret these trees correctly.

2.4 Related Work

Resler [33] argues that "the study of the inner-workings of a compiler can be greatly simplified through the use of a visible compiler". There have been numerous projects that aim to visualise how compilers work. Visual YACC was developed to aid in the teaching of compiler theory [12]. YACC (Yet Another Compiler Compiler) is a compiler generator that generates a LALR parser. The program accepts YACC grammars as input and produces visualisations of both the LR parse tree and LR parse stack. VCOCO (Visible COmpiler COmpiler) was constructed to assist in the generating of LL(1) compilers by providing a visual front-end to Coco/R. The benefit of this approach is that VCOCO is capable of generating compilers that are functionally equivalent to compilers produced by Coco/R, given the same specifications. The VCOCO interface is designed to provide a user with detailed, visual feedback of the compiler generation process. This is accomplished by inserting *hooks* into the semantic actions associated with each production in the grammar. These *hooks* update the *Grammar* window of VCOCO to visualise the progression of the parse process. Furthermore, VCOCO highlights the source code of Coco/R, line-by-line, as it steps through the compilation process. The operation of VCOCO allows users to trace the compiler generation process from syntax analysis to code generation. VCOCO's method of visualisation offers a debugging approach for experts and is not ideal as an education tool [3]. Furthermore, VCOCO does not provide any visualisation of the data structures associated with parsing. ANTLR (ANother Tool for Language Recognition) is a parser generator that produces human-readable recursive-descent parsers [27]. Like Coco/R, ANTLR uses attributed grammars, however, unlike Coco/R, ANTLR accepts LL(k) grammars, where $k > 1$. The parsers constructed using ANTLR are capable of constructing and visualising the abstract syntax tree. Construction of the AST is performed when the user annotates the grammar to indicate root and leaf nodes, as well as what needs to be excluded from the AST. An ANTLR generated AST can be seen in Figure 2.2. A disadvantage of ANTLR's AST generation is that the user needs to manually specify node types and which nodes should be included. This complicates the learning process for novice users, as they are required to learn ANTLR annotations before being able to interact with the AST. Furthermore, ANTLR assumes prior knowledge of how AST's are constructed and thus, is not suitable for learners wanting to learn about AST construction.

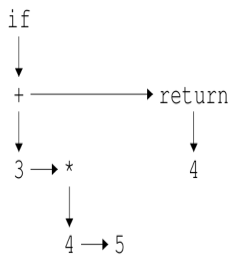


Figure 2.2: The abstract syntax tree resulting from "if 3+4*5 then return 4;" (taken from [27])

There have been numerous studies using software visualisation in aiding the teaching of programming concepts [9, 21, 22]. Norvell and Lockhart [23] created a software system, *The Teaching Machine*, which animates computer programs. The system allows students or instructors to single step through C++ or Java programs; the system then visualises these changes to the virtual machine. Norvell and Lockhart used their system as a teaching-aid while instructing an advanced programming course. They found that by using program animation they were able to ease the learning process for new students. The use of visualisation as a teaching aid is further explored in Section 4.

Chapter 3

Coco/R

3.1 Introduction

Created by Hansperter Mössenböck at ETH Zürich, Coco/R is a compiler generator [37, 42]. Coco/R takes an attributed grammar as input and produces a recursive descent parser and scanner. A modified version of the C# implementation of Coco/R is used in this thesis. This modified version of Coco/R is used as a teaching aid in the instruction of compiler theory to third year students at Rhodes University. The parser generated by Coco/R is LL(1), but allows for LL(1) conflicts to be resolved through the use of semantic checks. These semantic checks are implemented by attributing the grammar's productions. Coco/R produces numerous data structures during compilation, however, none of these are visualised.

3.2 Implementation

3.2.1 Coco/R structure

Terry defines the basic compiler structure [37, p308], consisting of file handling routines that are responsible for the execution of input and output routines. The input routines are used to transmit the source file to the scanner, also known as the lexical analyser. Once compilation is complete the output routines produce source and error listings along with the object code file.

During lexical analysis characters are grouped together into recognised tokens. These tokens are stored for later parsing by the syntax analyser. The parser is responsible for parsing the user grammar and performing constraint analysis. Tokens recognised by the lexical analyser are processed by the parsing routine, and as these are processed the parser checks that syntactic constraints are met. Non-terminals and terminals are stored as symbols in a symbol table, represented as a symbol graph, which is used to keep track of symbol declaration in the user grammar. The parser performs semantic checks to ensure that symbols are not declared twice. Other semantic checks include checking that symbols are declared before use and that there are no attribute mismatches between symbol declaration and usage. Error handling is done during the lexical analysis and parsing phase. The error handling routines interface with the output routines to produce error listings, which users may consult to gain a greater understanding of where syntactic and semantic errors occur. If parsing completes without any errors, Coco/R calls the code generator routine to produce the scanner, parser and driver files for the user defined grammar. The code generator routine is divided into three sections, scanner generation, parser generation and driver generation.

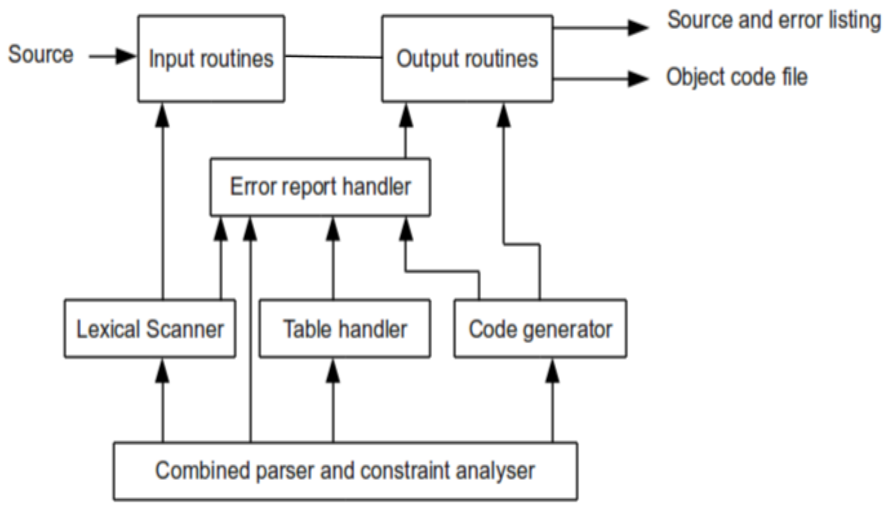


Figure 3.1: Relationships between main components of Coco compiler [37, p308]

3.2.2 Input

The input used for Coco/R consists of a context-free, attributed grammar known as Cocol [37]. Cocol itself can be described using EBNF notation [37]:

> *Cocol* = [*LibraryAccess*]
> "COMPILER" *GoalIdentifier*
> [*ArbitraryText*]
> *ScannerSpecification*
> *ParserSpecification*
> "END" *GoalIdentifier* ".".

The *LibraryAccess* section defines any additional C# declarations, which allows parser methods access to additional facilities in other namespaces. This is done by defining the C# *using <namespace>* directive. Any text included in the *ArbitraryText* section is not checked by Coco/R and is added directly to the generated parser. This is useful for declaring custom fields and methods which are used for semantic actions. Coco/R uses each production specified in the *ParserSpecification* section to generate a matching parsing routine [20].

Scanner Specification

The scanner is required to scan a users input, skip over meaningless characters, and to recognise tokens to be handled by the parser [36]. These recognised tokens are identifiable by a simple integer, unique to the token type. Tokens may be classified as either literals or as token classes. Literal tokens may be introduced directly into productions as simple strings. However, token classes must be named and have a structure that can be defined in EBNF. The parser is able to retrieve the lexene or textual representation of a token from the scanner. Coco/R allows tokens to be declared in any order, where each token is named using a *TokenSymbol* and has a token structure defined using EBNF. Once a token is recognised by the scanner, it is stored as a *Token* object, defined by the following *Token* class:

Listing 3.1: Token declaration within Coco/R *Scanner.cs*

```
public class Token
{
        public int kind;    // token kind
        public int pos;     // token position in the source text
        public int col;     // token column (starting at 0)
        public int line;    // token line (starting at 1)
        public string val;  // token value
        public Token next;  // Tokens are kept in linked list
}
```

Parser Specification

The parser specification is the main part of Coco/R. The user defined *ParserSpecification* consists of productions that provide procedural descriptions of parser actions [37, 42]. Coco/R uses the production section as the syntax of the language to be recognised. Each production is composed of identifiers which are the names of *Terminals* and *NonTerminals*. If an identifier appears in a production and has not been declared as a terminal token, it is considered to be a *NonTerminal*. Each *NonTerminal* must be defined by exactly one production. Furthermore, the productions are used to determine the actions to be taken as each token is recognised. These descriptions can be extended through the use of user specified attributes and semantic actions.

Attributes are used to define parameters of the non-terminal symbols [42, 20] and may be either input or output attributes. These attributes are enclosed in angle brackets < and >. The attributes can be considered to be the parameters of the production where input attributes are used to pass values into the production, while output attributes return values from the production. Coco/R performs checks to ensure that *NonTerminals* with attributes are always used with attributes and *NonTerminals* without attributes are used without attributes.

Semantic actions define statements in an imperative programming language. These actions are added to the produced parser and are executed during the parsing process. The semantic actions, enclosed within `(.` and `.)` are copied to the generated parser as they are encountered and no checks of their validity are performed. Users may add semantic actions anywhere in the grammar but they are mainly implemented at the end of productions [20]. Coco/R uses a one token lookahead and thus productions are processed from left to right [42]. The example in Listing 3.2 shows a production that processes variable declarations and adds the identifiers to a symbol table, *symTab*:

Listing 3.2: LL(1) Conflict Resolution in a Recursive Descent Compiler Generator [42, p.3]

```
VarDeclaration (. Structure type; string name; .)
= Type<out type>
  Ident<out name>        (. symTab.Enter(name, type); .)
  { "," Ident<out name>  (. symTab.Enter(name, type); .)
  } ";".
```

3.3 Grammar checks

Once a user submits a grammar, Coco/R performs several checks to ensure that the grammar is well formed [37]. If these checks fail Coco/R does not produce any code and the user is informed of the checks that have failed. The user then needs to correct these errors and resubmit the grammar. Several checks do not result in errors, but rather in warnings [37]. If a warning occurs, code generation still takes place but the user is informed of the warnings. Checks that result in errors include:

- whether each *NonTerminal* has been defined by exactly one production;
- whether there are no useless productions;
- whether it is a cycle free grammar;
- whether no ambiguity exists and each token can be distinguished from the others.

The following checks result in warnings:

- whether a non-terminal is nullable;
- whether LL(1) conditions are not met, such as at least two alternatives of a production have common elements in the FIRST and FOLLOW sets.

3.4 Coco/R output

If Coco/R successfully compiles the user supplied grammar, numerous files are produced as output. These files contain the source code of various classes,as described by Terry [37, p.241]:

- scanner class, which represents a FSA scanner (*Scanner.cs*);
- parser class able to perform recursive descent parsing (*Parser.cs*);
- driver class used to run the compiled code (*Calc.cs*).

These classes all belong to the *namespace* with the same name as the grammar's goal symbol.

3.5 Coco/R frame files

Coco/R requires *frame* files to function correctly. These frame files provide skeleton code to be used by Coco/R during code generation. The standard frame files used by Coco/R are the following;

- Parser.frame
- Scanner.frame
- Driver.frame

The frame files are used to specify additional code to be included into the generated *scanner*, *parser* and *driver* files and are thus useful for implementing extensions to Coco/R. Support classes required by extensions to the parser generated by Coco/R may be included in the *Parser.frame* file, simplifying implementation and increasing portability.

Chapter 4

Visualisation

Visualisation, as defined by the Oxford dictionary, is “the power or process of forming a mental picture or vision of something not actually present to the sight”[1]. Visualisation is seen as a cognitive activity as it is occurs in the mind, resulting in a mental model, which allows insight and understanding to be gained [34]. Other terms often associated with visualisation are *program visualisation* and *algorithm visualisation* [6]. A difference exists between program visualisation and algorithm visualisation. Program visualisation refers to the various techniques used to enhance the understanding of computer programs, while algorithm visualisation is defined as the process whereby actual implemented code is visualised [6]. Software visualisation encompasses both program and algorithm visualisation. Thus, software visualisation can be defined as “the use of the crafts of typography, graphic design, animation and cinematography with modern human-computer interaction technology to facilitate both the human understanding and effective use of computer software” [6, p214]. Software visualisation (SV) is used to solve a wide range of problems including algorithm animation and visual programming [19]. To be successful, the SV needs to provide a natural and direct mapping from the visual metaphor to the source code and back. Through graphically illustrating how algorithms function, visualisation techniques, such as algorithm visualisation and algorithm animation, can greatly increase understanding of algorithm function. Spence [34, p11] emphasises the value associated with the ability to explore and interactively rearrange the visualised data, this being particularly true when applied to visualisation as a teaching aid.

[1]The Oxford dictionary (2010). [Online].
Available:http://www.oxforddictionaries.com/definition/visualize?view=uk

4.1 Visualisation as teaching aid

The modern classroom has changed and teaching techniques need to be adapted to these changes. Algorithm visualisation and algorithm animation tools are freely available and easy to use [4]. Unfortunately the use of visualisation technology in mainstream computer science education has not yet caught on [10]. There have been numerous studies on the use of visualisation as a teaching aid. Unfortunately these studies have yielded mixed results. The use of algorithm animation in teaching usually receives good informal reviews. However, formal studies have found little statistical evidence of the efficacy of using visualisation as a teaching aid. Palmiter and Elkerton [26] compared the teaching of computer tasks using animation techniques and text-only presentation. Their study showed that the animation group was faster and enjoyed the lesson more. It was found that the results were negated in the long-run, with the text-only group performing better in the delayed test. A further study performed by Reed [31] indicated that the use of visualisation should be combined with external lesson activities to be effective. One of the main areas where software visualisation can be used to enhance the understanding of computer programs and algorithm function is the field of computer science education. Possible uses of SV in education include:

- helping illustrate algorithm operation in lectures,
- helping students learn the fundamental algorithms found in computer science, and
- helping instructors track down bugs in students programs.

The use of SV in the teaching of algorithms and programming to novice students is not a new concept. One such general purpose system *Karel, the Robot* has been used to assist students in studying Pascal [9]. Software visualisation aims to increase the understanding of how programs work [6, 10, 19]. The investigation into SV effectiveness has revealed ample evidence that visualisation can significantly reduce the effort required to comprehend a system [15]. In a study conducted by Bassil and Keller [15], participants rated *better comprehension* as a benefit of SV systems. In the evaluation of SV systems used in education, expressiveness is usually the focal point [10]. For a SV system to be effective as an education tool, it is important that the visual metaphor effectively expresses the underlying data. The metaphor needs to provide an accurate representation of what is being visualised; this will allow students to trust the metaphor and them aid in the learning process [10, 19]. Cognitive Constructivist theory states that active learning is superior to passive learning. This theory can be applied to visualisation due to the cognitive nature of visualisation as discussed by Spence [34]. Visualisation allows learners to actively engage with the work by augmenting their view. This view is further enhanced by Bower's [7] statement "it is important that such simulations are manipulative in order to actively involve the learner rather than... passively watch pre-programmed instruction". The learning can be further enhanced by requiring learners to answer questions about the visualisations. Thus, it can be said that software visualisation offers the advantage of providing high student involvement and allowing for the development of intuitive understanding of concepts in a visual manner [9]. However, the educational impact of visualisation is dependent on the enhancement of learning with visualisation and the usage of visualisation in the classroom [21, 22]. It has been found that instructors are unwilling to use visualisation in their classrooms [21]. As stated by Naps *et al.* [21] "the overall educational impact of visualisation is and will be minimal until more instructors are induced to integrate visualisation techniques in their classes". In a 2002 study of SIGCSE members [22], respondents listed *time required to search for good examples* and *time it takes to develop visualisations*, among the main impediments to the implementation of visualisation in their teaching. From this it is clear that these issues need to be addressed in order for visualisations to become effective teaching aids.

4.2 Testing effectiveness

To determine whether the visualisation of ASTs and other elements of compiler generators offer a benefit to students, studies have been performed. As reported by Hundhausen, Douglas and Stasko [10], visualisations can be used in multiple scenarios. In their study they state that the notion of effectiveness of a visualisation is derived from the scenario in which it is used. Hundhausen, Douglas and Stasko identified a taxonomy of scenarios within computer science education where visualisation can be used including: *Lectures*, *Assignments*, *Laboratories* and *Tests.*

For our study we focused on the use of visualisation in *laboratories* and *assignments.* The use of visualisation in laboratories have allowed students to interactively explore algorithms [10]. Computer science students at Rhodes University are required to attend a formal laboratory session for three hours every week. During these laboratory sessions students are assisted in completing their assignments by the lecturer and tutors, comprising senior computer science students. The students are required to complete additional assignments in their own time for submission at the next practical session. This would provide an ideal situation for testing visualisation effectiveness. In their study of visualisation effectiveness Hundhausen, Douglas and Stasko identified four techniques used to evaluate visualisation effectiveness. These are:

- Anecdotal techniques which provides author analysis of example usage of their visualisation system.
- Programmatic techniques where the actual programs used to produce visualisations are evaluated. Programs are evaluated using different metrics, such as the number of lines of code required to specify a visualisation.
- Analytic evaluation assesses the effectiveness of an interactive system. Users are asked to evaluate the usability of the system and to identify any usability problems the perceive.
- Empirical evaluation requires the collection of data on the students who use the visualisation system.

Chapter 5

User interface design

A user interface aims to present content to a user in a visual manner. Furthermore, the user interface provides mechanisms for user interaction with the content being presented. These two features of the user interface, makes it an integral component when using visualisation as a teaching aid. The usability of software is determined by the quality of the user interface design. The user interface is of even more importance when it comes to educational software. Learner orientation and feedback are two of the core issues when designing the user interface [25]. It is essential that information is presented in a way that is appropriate to the learning material. Furthermore, students need to be able to navigate the software with ease and without needing to learn a whole new system before being able to study the relevant learning material. Thus, being able to easily orientate the user without the use of a help system is essential. Guidance can be accomplished through implicit guidance, whereby the learning system guides the user through content presentation. Another means of guidance is explicit guidance, whereby the user is guided by instructions on how to use the content presented to them. Learning systems require more implicit guidance, which allows the learning to focus on studying the domain content as opposed to learning the system. Therefore, we can say that the main tasks demanded of a learning system is content presentation and facilities for user interaction with the content. This needs to be accomplished in an implicit manner, making the user feel at ease with the system and assisting in the learning process.

When evaluating the user interface, we look at usability. The usability of a system can be measured by the efficiency and effectiveness of the system, as well as the satisfaction of the user [25]. Effectiveness is determined by how many of the overall goals of the system are achieved. Efficiency can be measured by the amount of resources required in order to achieve the system goals. To evaluate usability we need to evaluate the look and feel of the system. Further determinants of usability are access to tools and services as well as communication and user support. The extent to which usability goals are met can be measured by:

- Suitability for the task: how well the system supports the user in completing the goals of the system.
- Self-descriptiveness: the system provides implicit guidance to the user. This allows the user to determine what is required without being explicitly prompted by the system.
- Conformity with user expectations: the user interface is consistent and relates well to the problem domain.
- Error tolerance: the system is able to respond to errors in input and still produce expected results with minimal correction required by the user
- Suitability for learning: the system assist the user in learning how to use the system and creates greater user understanding of the problem domain.

As discussed earlier, a user interface aims to provide a visual representation of content with which users can interact. This visual representation of content can be defined as *information presentation*, the look and feel can be represented by the following presentation attributes [25]:

- clarity: the content is conveyed quickly and accurately;
- discrimination: information can be distinguished accurately;
- conciseness: only relevant information is displayed;
- consistency: information is displayed through a unique design and meets user expectations;
- detectability: the user is guided towards relevant information;
- legibility: information is presented in an easy to read manner;
- comprehensibility: the meaning of information is easy to interpret, unambiguous and recognisable.

5.1 Learning software user interface

The user interface used in learning software needs careful design and consideration. It needs to meet standard interface demands [25], as well as taking into account specific user demands and design considerations determined by the problem domain. When designing a user interface for learning software, it can be assumed that the user is a novice user of the software and that the user will not be a long term user of the software [25]. The nature of learning software dictates that the software will only be used for a time period determined by the time taken to grasp the required concepts of the problem domain. Due to this time limit, it is essential that the user interface is easy to learn and use. A user should be able to easily determine which aspects of the problem domain are covered by the learning software, simply by navigating the user interface. This is known as implicit guidance. The ability of learning software to accurately display content to users is limited by screen size. To overcome the screen size limitation, the user interface needs to be designed to make maximum usage of available screen space. The information presentation by the learning software should clearly reflect the problem domain.

Providing feedback is key in learning environments [8], since it is essential that the user interface provides clear, concise feedback to the learner. As discussed earlier, feedback may be given implicitly or explicitly. In order to prevent interruption to a learner's work process, explicit feedback should be kept to a minimum [25].

5.2 Design and Implementation

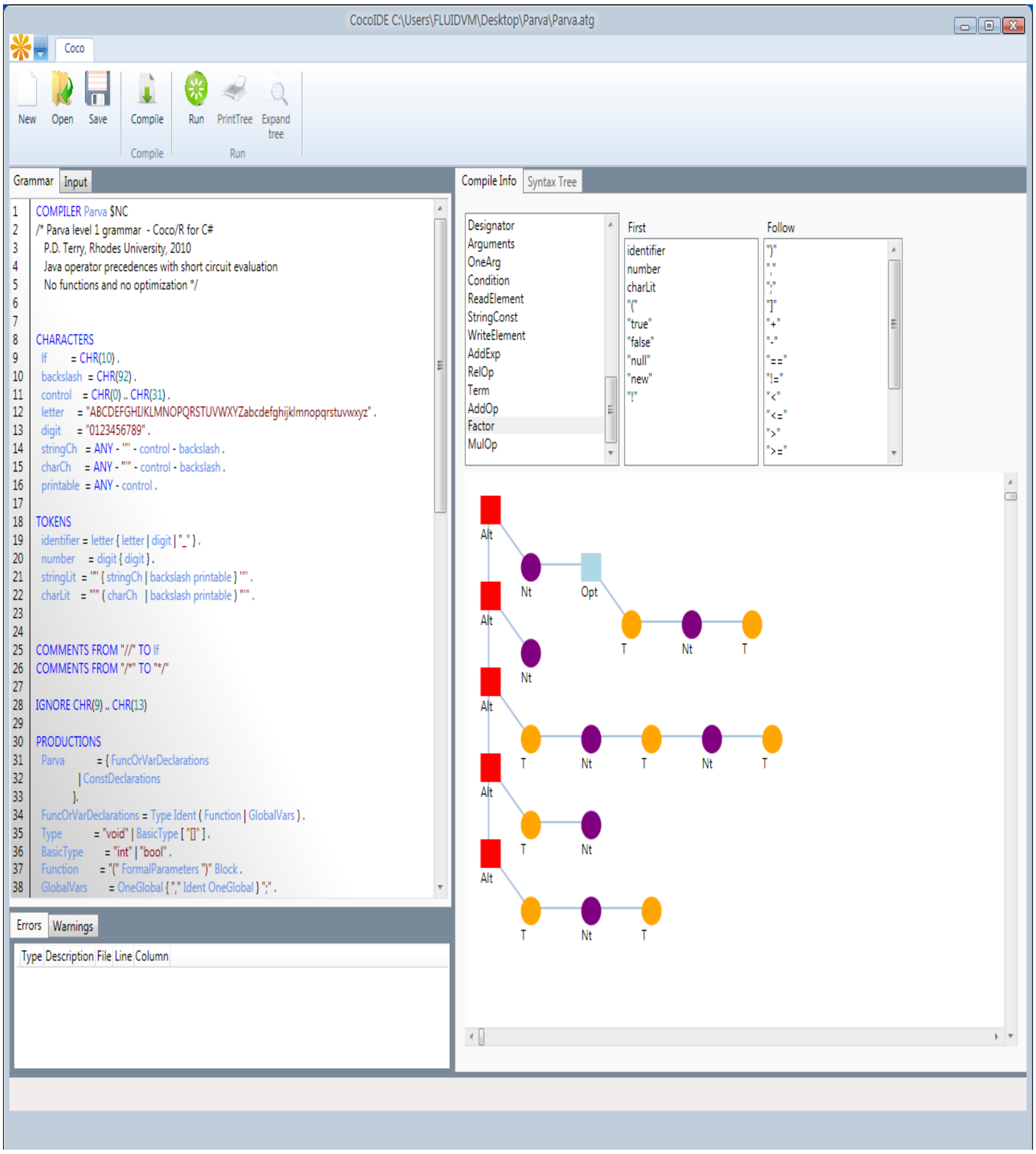
Numerous factors were considered in designing the user interface for the Coco IDE. Since the overall design of the system will greatly impact on the usability, and in turn the usefulness, of the Coco IDE. Learners need to be comfortable working in the new environment. Furthermore, learners need to quickly learn how to use the system. As discussed in Section 5.1, it is important to ensure that the user interface does not hinder user understanding of the underlying problem domain. The learning curve of the system needs to be low, ensuring that students can quickly start working on the problem domain as opposed to first needing to learn the new interface.

The basic design of the Coco IDE matches current software packages that the learners are comfortable working with. To this aim, the Ribbon device from Microsoft Office is incorporated into the Coco IDE. Learners use Microsoft Office or similar products on a regular basis and have become comfortable navigating their way around those systems. Thus, by following a similar design, we could ensure that learners can get to grips with the basic functionality of the system quickly. Important and frequently used actions are available on the Ribbon bar, with large, prominent pictorial buttons, allowing for quick and easy identification of their functionality. Buttons are disabled when their action is not available in the current program state. By disabling the buttons we are able to provide implicit guidance to the user.

Using a tabbed interface has the benefit of allowing more screen space to be available to each panel without cluttering the interface. The left-hand side of the interface is dedicated to user input. The *grammar* tab contains a large text area with syntax highlighting, where the user can input and edit their Cocol grammar. The *input* tab allows for the editing of sample grammars, which can be run and tested using the parser generated by Coco/R. The right-hand side of the interface is used for visualisation. The *compile info* tab visualises the output of the compile process. This includes syntax graphs as well as FIRST and FOLLOW sets. The *syntax tree* tab contains the visualisation of the abstract syntax trees produced when running the generated parser. An always visible feedback pane is placed on the bottom left of the screen. The feedback pane allows for error and warning messages to be displayed to the user in a well structured manner.

5.2.1 Syntax highlighting

Syntax highlighting has the advantage of making code easier to read. Careful text layout can significantly increase readability [11]. Syntax highlighting can thus be combined with text layout to make interaction with source code easier. By highlighting important keywords and control symbols, users are made more aware of these symbol's function. Syntax highlighting also aids in the identification of errors; for example the user is able to identify unclosed comment blocks easily. Furthermore, unmatched brackets and string literals are clearly identifiable. The IDE attempts to use syntax highlighting to convey the meaning of symbol usage in the grammar implicitly. Reserved keywords are highlighted in blue, while user keywords are highlighted in light blue. This makes differentiation between the two types of keywords easier for the user, which is especially true for new users. By colouring the reserved keywords in a prominent colour, users are made aware of their importance. Furthermore, users are able to identify missing grammar structures as missing keywords are more noticeable. Comments are highlighted in light green, which is a standard used in many editors that incorporate syntax highlighting. The use of this



```
COMPILER Parva $NC
/* Parva level 1 grammar  - Coco/R for C#
  P.D. Terry, Rhodes University, 2010
  Java operator precedences with short circuit evaluation
  No functions and no optimization */


CHARACTERS
 lf        = CHR(10) .
 backslash  = CHR(92) .
 control    = CHR(0) .. CHR(31) .
 letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
 digit      = "0123456789" .
 stringCh   = ANY - '"' - control - backslash .
 charCh     = ANY - "'" - control - backslash .
 printable  = ANY - control .

TOKENS
 identifier = letter { letter | digit | "_" } .
 number     = digit { digit } .
 stringLit  = '"' { stringCh | backslash printable } '"' .
 charLit    = "'" ( charCh   | backslash printable ) "'" .


COMMENTS FROM "//" TO lf
COMMENTS FROM "/*" TO "*/"

IGNORE CHR(9) .. CHR(13)

PRODUCTIONS
 Parva          = { FuncOrVarDeclarations
                  | ConstDeclarations
                 }.
 FuncOrVarDeclarations = Type Ident ( Function | GlobalVars ) .
 Type           = "void" | BasicType [ "[]" ] .
 BasicType        = "int" | "bool" .
 Function        = "(" FormalParameters ")" Block .
 GlobalVars        = OneGlobal { "," Ident OneGlobal } ";" .
```

Designator
Arguments
OneArg
Condition
ReadElement
StringConst
WriteElement
AddExp
RelOp
Term
AddOp
Factor
MulOp

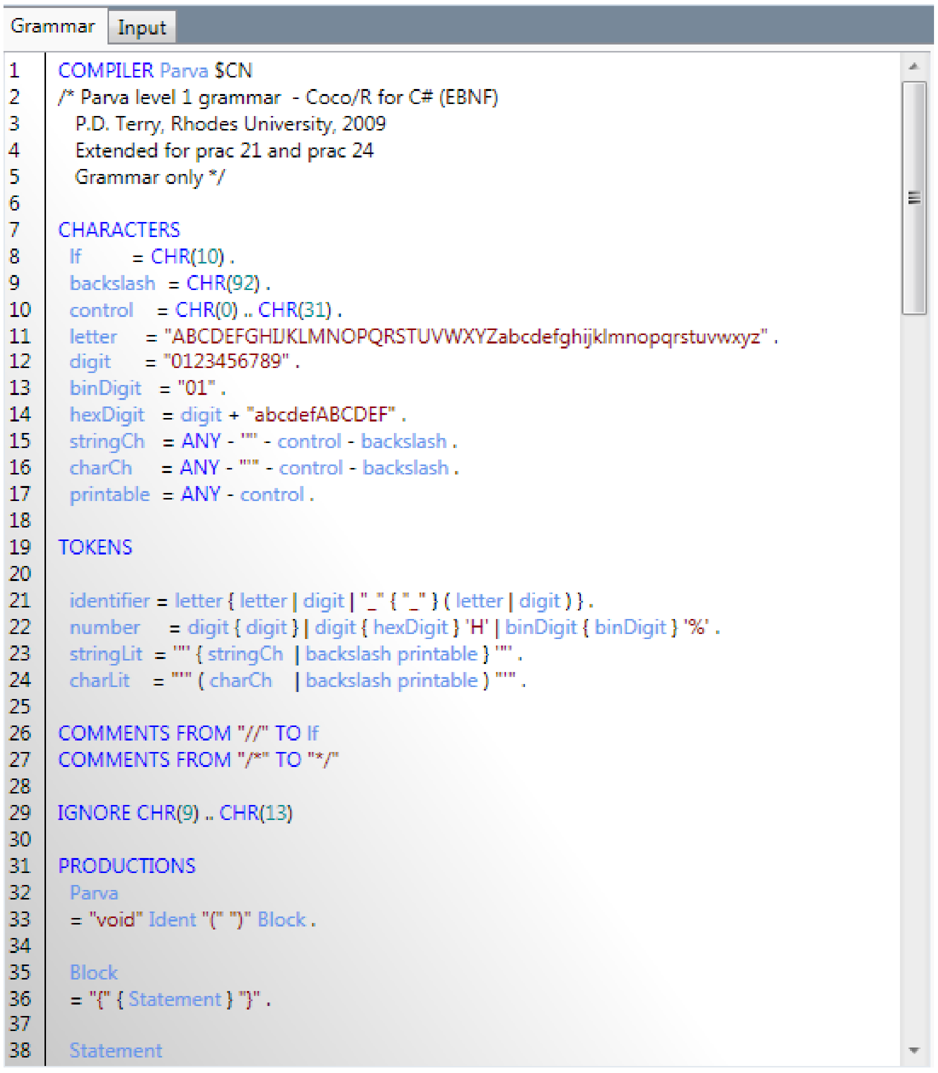
First
identifier
number
charLit
"("
"true"
"false"
"null"
"new"
"!"

Follow
")"
","
";"
"]"
"+"
"-"
"=="
"!="
"<"
"<="
">"
">="

Type Description File Line Column

Figure 5.1: The CocoIDE

convention helps new users easily identify the meaning of the highlighting as they are used to seeing it in familiar software packages. String literals are highlighted in maroon, which is another convention taken from widely used editors. Line numbering allows users to identify the current line as well as simplifying error recovery. By using line numbers, the user can quickly find the problem line as identified by Coco/R.



```
COMPILER Parva $CN
/* Parva level 1 grammar  - Coco/R for C# (EBNF)
   P.D. Terry, Rhodes University, 2009
   Extended for prac 21 and prac 24
   Grammar only */

CHARACTERS
  lf         = CHR(10) .
  backslash  = CHR(92) .
  control    = CHR(0) .. CHR(31) .
  letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit      = "0123456789" .
  binDigit   = "01" .
  hexDigit   = digit + "abcdefABCDEF" .
  stringCh   = ANY - '"' - control - backslash .
  charCh     = ANY - "'" - control - backslash .
  printable  = ANY - control .

TOKENS

  identifier = letter { letter | digit | "_" { "_" } ( letter | digit ) } .
  number     = digit { digit } | digit { hexDigit } 'H' | binDigit { binDigit } '%' .
  stringLit  = '"' { stringCh  | backslash printable } '"' .
  charLit    = "'" ( charCh   | backslash printable ) "'" .

COMMENTS FROM "//" TO lf
COMMENTS FROM "/*" TO "*/"

IGNORE CHR(9) .. CHR(13)

PRODUCTIONS
  Parva
  = "void" Ident "(" ")" Block .

  Block
  = "{" { Statement } "}" .

  Statement
```

Figure 5.2: A user grammar with syntax highlighting

Syntax highlighting is performed in real time. The implementation of syntax highlighting is done by creating a modified *textbox* as a new user control in C#. This modified *textbox* contains methods for recognising reserved words and characters. To achieve this goal, a modified version of the Coco/R scanner is used. This modified scanner, *SyntaxHighlightScanner*, scans the user grammar currently in the *textbox* and classifies tokens according to type. Recognised types are:

- Keywords such as *COMPILER*, *CHARACTERS*, *TOKENS*, *PRODUCTIONS* and *END*.
- Strings are used to identify string literals.
- Comments may be single or multi-line and are identified by in the same manner as comments in Java and C#.
- Identifiers represent *Terminals* and *NonTerminals*.
- Operators are the standard meta-symbols defined by EBNF.

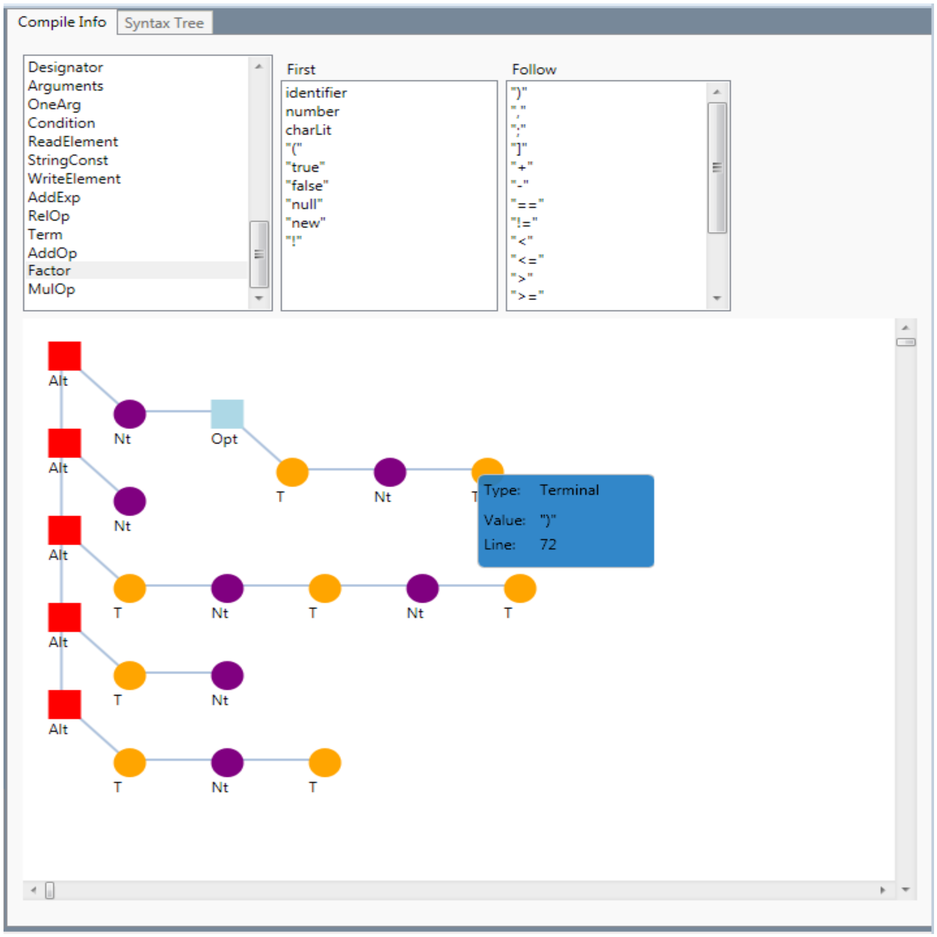
The class *CocoLexer.cs* parses the tokens recognised by the scanner. *CocoLexer* calls the *Scan()* method from *SyntaxHighlightScanner*, which returns the current recognised token. The token returned by the Scan() method is stored as a *CodeToken* object. The *CodeToken* object, defined as a C# class (see Listing A.1) stores the *TokenType* and the token's start and end indices. Once the token has been transformed into a *CodeToken* object, it is added to a token list. The updating of the token list triggers the *textbox* to render the token's text with the correct formatting. Due to the real-time parsing of the user grammar, a possible extension would be to include real-time error checking. This could be done in a similar manner to most modern IDE's where unrecognised keywords are underlined and unmatched brackets are highlighted. This extension could improve the speed of debugging as users are made aware of errors in real-time as opposed to waiting until the compile process has failed.

5.2.2 Syntax information

Syntax information is visualised at compile-time. The *compile info* tab contains compile-time information such as a list of productions and the FIRST and FOLLOW sets. Also included is a graphical representation of the grammar's syntax graphs. During the compilation process Coco/R produces syntax graphs, which can be visualised to provide the user with a graphical representation of the EBNF rules [18]. The use of a visual graph allows the user to see how productions will be processed during the parsing process. Unique symbols are used to differentiate between *Terminal* and *NonTerminal* symbols. Symbols are also used to identify control structures. *Terminal* symbols are represented by a yellow oval and labelled with a small *T*. The NonTerminals are drawn as a purple oval and labelled with *Nt*. The colour differentiation between *Terminal* and *NonTerminal* symbols facilitates quick identification. Control structures are drawn using a rectangle. These control structures are also colour coded to aid their identification. The *iteration* structure is coloured in green and labelled with *Iter*. The optional structure is coloured in light blue and labelled *Opt*. Finally alternative options are represented with a red rectangle and labelled as *Alt*. Further information is made available to the user by hovering over a graph symbol with their mouse pointer. This additional information identifies the value of the symbol as well as the line on which it occurs. The production given in Listing 5.1 is visualised in Figure 5.3.

Listing 5.1: EBNF description of the Factor production in Parva

```
Factor   = Designator [ "(" Arguments ")" ]
           | Constant
           | "new" BasicType "[" Expression "]"
           | "!" Factor
           | "(" Expression ")".
```


Compile Info | Syntax Tree

	First	Follow
Designator	identifier	")"
Arguments	number	","
OneArg	charLit	";"
Condition	"("	"]"
ReadElement	"true"	"+"
StringConst	"false"	"-"
WriteElement	"null"	"=="
AddExp	"new"	"!="
RelOp	"!"	"<"
Term		"<="
AddOp		">"
Factor		">="
MulOp		

Type: Terminal
Value: ")"
Line: 72

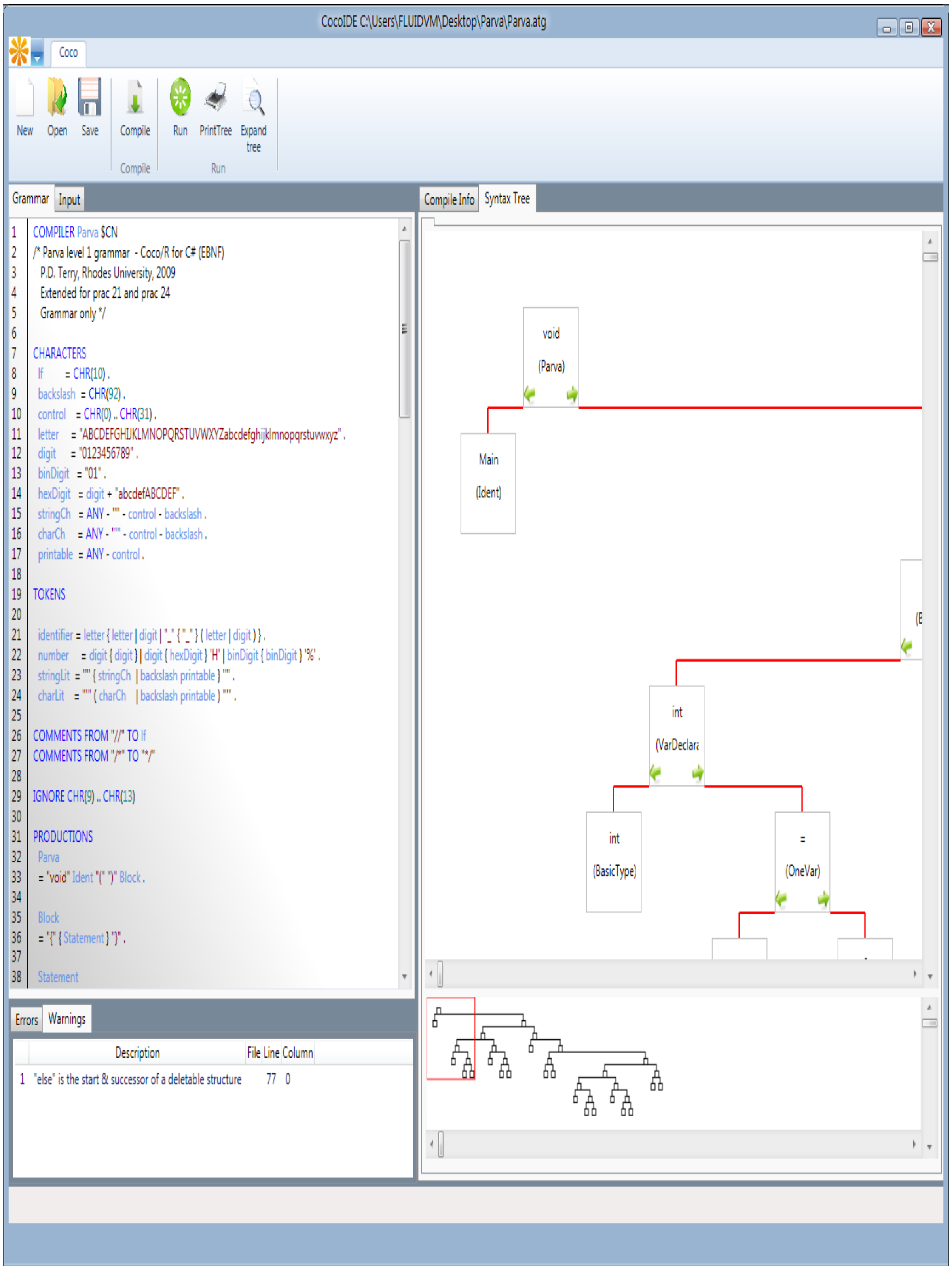
Figure 5.3: Syntax information visualised by the CocoIDE for *Factor* production

Parsing of the Cocol grammar results in the generation of FIRST and FOLLOW sets for each production. Coco/R uses these FIRST and FOLLOW sets to determine whether LL(1) conditions are violated [36]. In the original Coco/R implementation students are required to execute Coco/R with the *-t* flag set to ensure that the FIRST and FOLLOW sets are printed to file, where they can be studied. With the Coco IDE, these FIRST and FOLLOW sets are automatically displayed, along with the associated production and the correct syntax graph. This allows students to view all the related information in one specific location and to comprehend the significance of the FIRST and FOLLOW sets within the grammar. By providing the syntax graph on the same tab as the FIRST and FOLLOW sets, students are able to observe how the contents of the FIRST and FOLLOW sets determine the parsing of the associated production.

5.2.3 Abstract syntax trees

Abstract syntax tree construction takes place during compile time (see Section 2.3). Upon a successful compilation Coco/R produces a recursive descent parser. This parser is capable of constructing a generic AST for user supplied input. Users can choose to test their newly generated parser against sample input by selecting the *Run* command from the menu-bar. The user will be prompted to supply sample input to be used, if none has been defined under the *input* tab.The user supplied input is then parsed by the parser. During this parsing phase the AST is constructed. The Coco/R IDE uses reflection to retrieve the AST object upon successful execution of the *Run* command (see Section 6.5). The object code for the AST is visualised using the *BinaryTreeImage* class. This class walks the AST and draws each node as it is visited. The visited nodes are represented as *TreeNodeControl* objects. The *TreeNodeControl* object has pointers to the left and right child nodes of the node. Further, it has pointers to both the left and right connector lines which are used to link the parent node to its children. The *TreeNodeControl* is represented using a class structure (see Listing A.2).

To construct the visual tree, the *BinaryTreeImage* class first walks the entire tree. This is done to determine the depth of each branch of the tree, which in turn makes it possible to determine how wide each branch will extend. Walking twice, once to determine tree size and the second time to draw the tree, decreases performance slightly but has the benefit of preventing tree branches overlapping each other when the display is created. The tree nodes are drawn by the recursive method *DrawNode*. Within the *DrawNode* method (see Listing A.3), each node visited is added to the *Canvas* object as a *TreeNodeControl*. The



```
COMPILER Parva $CN
/* Parva level 1 grammar  - Coco/R for C# (EBNF)
  P.D. Terry, Rhodes University, 2009
  Extended for prac 21 and prac 24
  Grammar only */

CHARACTERS
 lf        = CHR(10) .
 backslash  = CHR(92) .
 control    = CHR(0) .. CHR(31) .
 letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
 digit      = "0123456789" .
 binDigit   = "01" .
 hexDigit   = digit + "abcdefABCDEF" .
 stringCh   = ANY - '"' - control - backslash .
 charCh     = ANY - "'" - control - backslash .
 printable  = ANY - control .

TOKENS

 identifier = letter { letter | digit | "_" { "_" } ( letter | digit ) } .
 number     = digit { digit } | digit { hexDigit } 'H' | binDigit { binDigit } '%' .
 stringLit  = '"' { stringCh  | backslash printable } '"' .
 charLit    = "'" ( charCh    | backslash printable ) "'" .

COMMENTS FROM "//" TO lf
COMMENTS FROM "/*" TO "*/"

IGNORE CHR(9) .. CHR(13)

PRODUCTIONS
 Parva
 = "void" Ident "(" ")" Block .

 Block
 = "{" { Statement } "}" .

 Statement
```

	Description	File	Line	Column
1	"else" is the start & successor of a deletable structure		77	0

Figure 5.4: The abstract syntax tree resulting from running the user grammar

benefit of declaring each node as a *TreeNodeControl* object is that each node is responsible for itself and thus complies with the principles of object orientated programming. When a tree branch is collapsed, each node is responsible for hiding itself and for informing its child nodes that they need to collapse themselves. This is aided by the recursive nature of the binary tree structure used to represent the AST.

As evident in Figure 5.4, large syntax trees may occupy the whole screen, reducing the user's ability to navigate and view the complete tree. This is a common problem associated with visualisation, namely how to represent large amounts of data on a small screen. Spence [34] discusses multiple solutions to the presentation problem, including *scrolling* and the use of a *context map*. These concepts have been included in our implementation of the AST visualisation. These concepts aid the user in viewing as much of the tree as possible and assist in the navigation of large trees. The first mechanism used is an *expand* feature, which hides the *input* and *feedback* panes. The tree canvas is then expanded to occupy the entire screen, as seen in Figure 5.5.

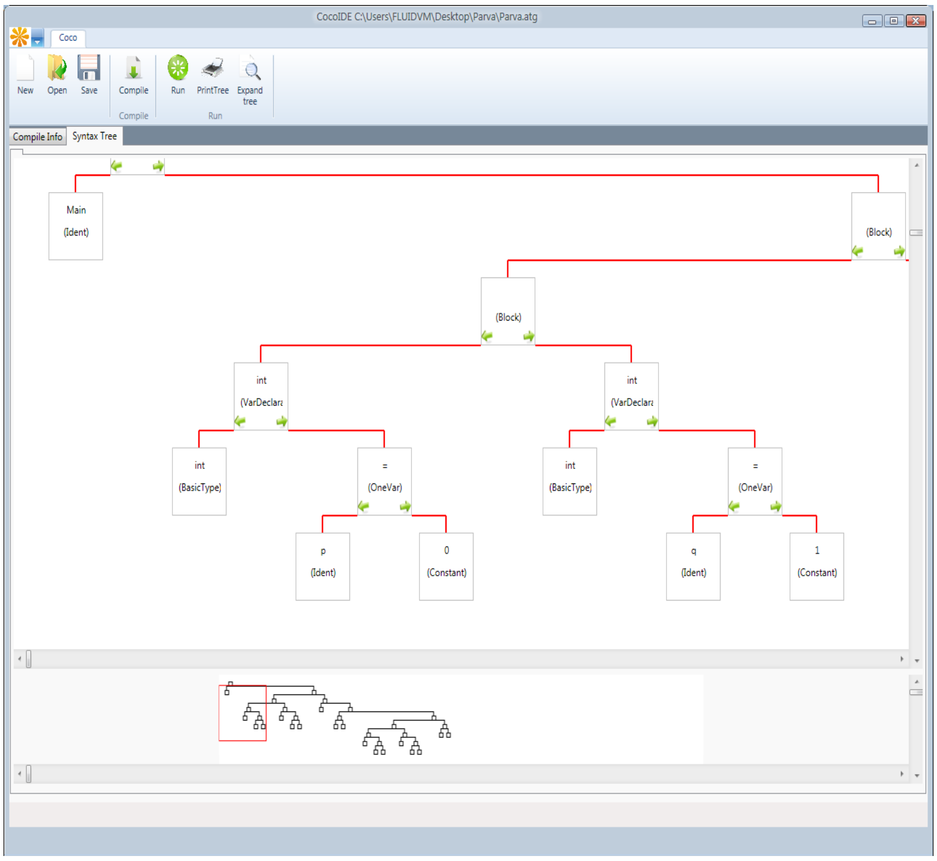


Figure 5.5: The abstract syntax tree view expanded to fill the screen

A context map provides a less detailed display of the visualised data and identifies the part of the visualised data currently visible. The context map has been implemented in the form of a navigation pane, as shown in Figure 5.6. The navigation pane consists of a miniature representation of the AST along with a red rectangle which indicates the current area of focus as related to the main tree canvas. The user is able to drag the mouse along the navigation window and refocus the main tree canvas on a point of interest. The user is also able to collapse the branches of the tree. Collapsing branches of the tree results in them being hidden from the user. The user can select to collapse or uncollapse a branch by clicking on the corresponding parent node's arrow. Users are also provided with the ability to print the AST. The print routine produces a simplified visualisation of the AST. This simplified version relates closely to AST representations students are used to encountering in academic text.

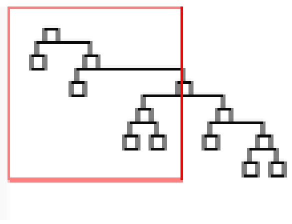


Figure 5.6: Navigation component of the Abstract Syntax Tree visualisation

A further feature introduced by the IDE is the ability to *overlay* two ASTs (Figure 5.7). The user may produce an AST and choose the *overlay checkbox*. When *overlay* is selected, the next AST generated will be drawn in a light shade of blue and inserted as a layer on top of the existing AST. The user is then able to manipulate the transparency of this overlay AST and compare the difference between the two ASTs. This feature is useful to a user who wishes to see how changes made to the grammar affects the AST constructed. The user is also able to see how changing the input parsed by the compiler may result in an AST with the same structure as the original AST, albeit with different values for each node. This should make understanding the parsing process easier for new users who can now visually see how changing the grammar alters the manner in which the input is parsed.

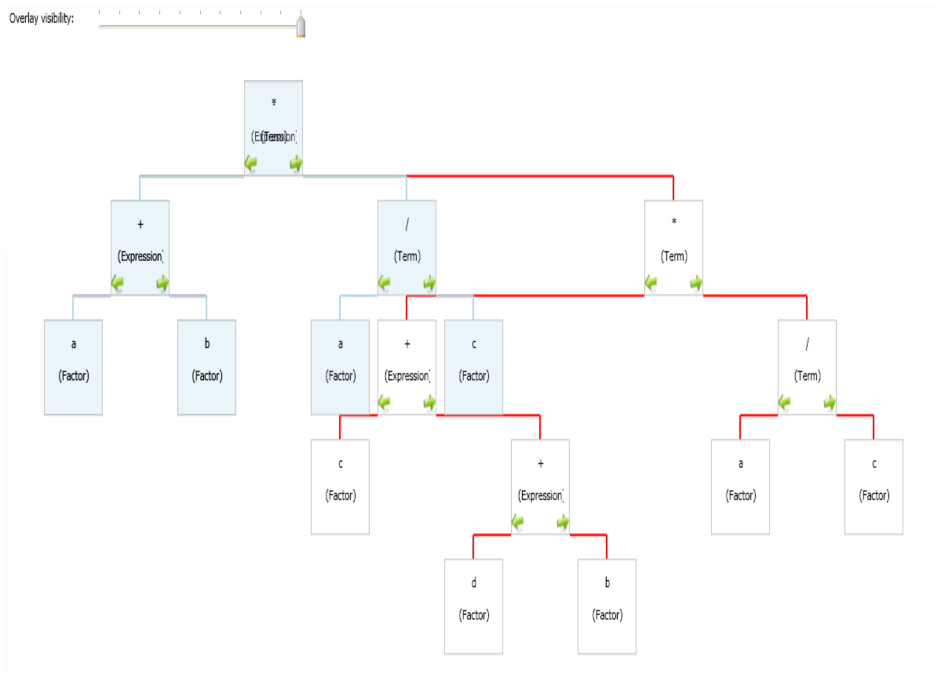
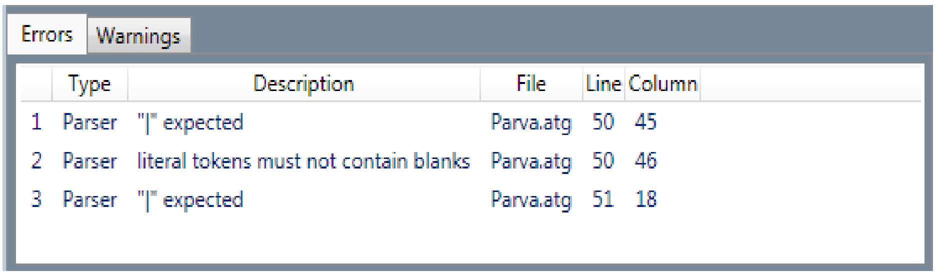


Figure 5.7: Overlay enabled to show the difference between two ASTs

5.2.4 User feedback

As highlighted by Boyle [8], the provision of user feedback is essential. When designing the user interface for Coco/R IDE a key focus was the design of an effective user feedback system. As highlighted in Section 5.1 feedback can be provided implicitly or explicitly. Due to Coco/R being used in an educational environment it is essential that feedback is provided in a manner that is both informative and unobtrusive. For this we created a static feedback section that is always visible and displays error messages in a uniform manner. Elder [11] identifies some best practices for error reporting. He states that the error needs to be described as clearly as possible. Clearly describing errors allows users to understand why an error has occurred, while descriptive error reporting aids in the teaching process. Learners are able to gain greater understanding of the compiler operation by studying error messages. Elder further states that the position an error occurs at needs to be identified as exactly as possible. By indicating the exact position at which an error occurred, users are able to more easily find and correct the error. Providing exact error positions ensures that users are spared the time and effort of trying to locate where an error has occurred. Error location information should be listed by line number and if possible a character count [11].



Errors | Warnings

	Type	Description	File	Line	Column
1	Parser	"|" expected	Parva.atg	50	45
2	Parser	literal tokens must not contain blanks	Parva.atg	50	46
3	Parser	"|" expected	Parva.atg	51	18

Figure 5.8: Feedback to the user

As seen in Figure 5.8, the feedback section is capable of displaying feedback in the form of *Errors* and *Warnings*. Errors are deemed to be critical and will prevent the user grammar from compiling. Language specific, compile time errors, such as errors in semantic actions, are identified by the C# compiler and will be displayed to the user in the feedback section. Furthermore, errors can also occur within a grammar that has been compiled successfully, and these are known as run-time errors. These errors will be handled at run-time and displayed to the user in the error panel. By grouping grammatical, compile time and runtime errors into one central area, users are saved the effort of trying to locate where an error has occurred. Warnings are less critical and do not prevent the grammar from compiling correctly. Users are prompted with a dialogue box if they attempt to run a grammar that produced warnings during compile-time. The user is then able to either, ignore these warnings and attempt to run the grammar, or halt the execution of the grammar and correct the warnings. The error pane layout allows for error information to be displayed in an easy to understand form. The errors are listed in a tabular structure with columns *Description*, *File*, *Line* and *Column*. This layout allows learners to first read what the problem is, and then identify where the error occurs. The *File* column is used to identify errors that occur at run-time, either in the generated *scanner* or *parser* files. The *line* and *column* entries allow the user to locate errors more quickly and without needing to search through every line of the grammar or produced source code to correct the error.

Chapter 6

Design and Implementation

6.1 Introduction

This chapter details the changes made to the original Coco/R compiler generator. We discuss the addition of new data structures and code generation routines. The AST generation routines inserted into the code generated by Coco/R aim not to alter the functionality of the produced parser. The inclusion of ASTs in the new version of Coco/R requires a data structure to store and manipulate the AST nodes, and is discussed in Section 6.3. Coco/R's code generation routines are modified in multiple locations, including the *GenProductions* and *GenCode* methods found in the *ParserGen* class. Output from the original version of Coco/R is compared to the output produced after the modifications have been implemented. By comparing the output, we are able to determine whether the parser functionality has been altered. In Section 6.5 we discuss the methods required to retrieve the generated AST at run-time through the use of *reflection*.

6.2 Incorporating AST's

Abstract syntax tree generation occurs during the parsing phase [18]. Since Coco/R is used to automatically generate a recursive descent compiler, it is necessary to alter the Coco/R compiler construction routines to insert AST generation routines into the generated code. The original implementation of Coco/R parses the following production and produces the code shown in Listing 6.1.

Expression = Term $\{``+"Term|``-"Term\}$.

Listing 6.1: Code generated for simple EBNF expression 6.2

```
static void Expression()
{
        Term();
        while (la.kind == plus_Sym || la.kind == minus_Sym)
        {
                if (la.kind == plus_Sym)
                {
                        Get();
                        Term();
            } else
                  {
                        Get();
                        Term();
                  }
        }
}
```

By attributing the grammar it is possible to insert AST generation statements into the generated parser. However, this approach requires users to have a firm understanding of how the parsing process works. It further requires users to know where AST nodes should be added in order to accurately reflect the syntax of the grammar. By automating the process of attributing the code to include AST generation statements, it is possible to create a layer of abstraction between the user and the tree building process. This should assist new users in gaining a greater understanding of how the AST relates to the parsing of a grammar.

The code generation routines in Coco/R are responsible for creating code that is able to recognise and parse any input that can be described by the user's grammar. This means that code generation needs to be dynamic and unconstrained by what the grammar describes. To achieve this aim, our AST creation code must also be sufficiently generic. *ParserGen.cs*, the class responsible for the parser construction in Coco/R, defines three methods, which are of particular interest when determining where and how AST code generation should take place. These methods are:

- *GenProductions*: This method is responsible for the definition of methods that relate to the NonTerminals defined in the parser specification section of the grammar.

GenProductions generates code to create the method definition, along with the return type and any parameters.

- *GenCode*: This method generates the actual code found within the method created by *GenProductions*. Tokens defined within the production declaration are identified according to their type and determines the that gets generated. This method is used to generate the actual AST creation code.

- *CopyFramePart*: This method is required to copy predefined code from the *Parser.frame* file and insert it directly into the newly created parser. This includes pre-created error handling routines and code to interact with the scanner. This method will not be altered directly but is essential in assuring that all supporting classes are available to our AST creation code.

6.3 Abstract syntax tree representation

The AST generated during the parsing process needs to be represented as a data structure. As discussed earlier (Section 2.3), the AST can be viewed as a collection of related nodes. Every node in the tree should be connected to another node. This relational structure enables a path through the tree, from the root node to any other node, to be found. Hand coded compilers offer the benefit of allowing the creation of node structures to represent each token type. However, Coco/R generated compilers cannot be afforded this level of fine grained design. As Coco/R has no knowledge of the semantic meaning of tokens when they are processed, it is not possible to generate a unique node structure for each token type as they are encountered. Thus, it should be realised that the data structure needs to be sufficiently general to cover all the different kinds of nodes. A node in the AST can be generalised to contain three fields:

- *Type* field, which is used to define the literal value associated with that node.
- *Left-child* and *Right-child* pointers, which provide a link to the left and right subtrees of that node, respectively.
- *Production* field that represents the production or method which the node mirrors.

This data structure can be constructed in C# using an abstract class structure. As can be seen in Listing 6.2, the general node defines the required fields. These fields can be overwritten by the implementation of the actual AST node as seen in Listing 6.3.

Listing 6.2: Definition of the AST node interface

```
public abstract class INode
{
    public abstract string production { get; set; }
    public abstract string type       { get; set; }
    public abstract INode left        { get; set; }
    public abstract INode right       { get; set; }
}
```

The AST node, as defined in Listing 6.3, defines two constructors. The first constructor is used to construct a node with no sub-trees. This constructor is normally used when the node is the final leaf node of a branch. The second constructor is more general and will be used in most cases. This constructor creates a tree node with given name, production and sub-trees. Both constructors result in a tree node object being returned, which can be used directly as the parameter of another *TreeNode* object's constructor. The *TreeNode* class is used by every parser generated using Coco/R. For this reason it is defined within the default *Parser.frame* file. This ensures that the *TreeNode* class will automatically be added to the generated parser. A further modification is to ensure that the abstract class *INode* is always copied to the same namespace as the newly generated parser.

Listing 6.3: Code listing TreeNode class used to represent an AST node

```
public class TreeNode:INode
{
  private string _production; //associated production/method
  private string _type;       //literal value
  private INode _left;        //left sub-tree
  private INode _right;       //right sub-tree

  //constructor for TreeNode with no sub-trees
  public TreeNode(string t, string p)
  {
    _production = p;
    _type = t;
    _left = null;
    _right = null;
  }
  //methods to override the abstract class
  public override string production
  {
     get { return _production; }
     set { _production = value; }
  }

  public override string type
  {
     get { return _type; }
     set { _type = value; }
  }

  public override INode left
  {
     get { return  _left; }
     set { _left = value; }
  }

  public override INode right
  {
    get { return _right; }
    set { _right = value; }
  }

  //constructor for tree node with left and right sub-trees
  public TreeNode(string p,string t, INode l, INode r)
  {
     _production = p; _type = t; _left = l; _right = r;
  }

  //empty constructor
  public TreeNode(){}
}
```

The translation of parser operations into AST nodes is relatively straightforward, but numerous cases need to be considered. A major hindrance is the lack of semantic knowledge. Coco/R generates a recursive descent parser without any knowledge of the semantics. Our AST generation needs to occur in a similar manner, however an AST normally assumes some form of semantic knowledge. This can be problematic in cases where more than two sub-trees exists. Due to Coco/R not being able to judge the meaning of tokens and productions, there is no way of knowing that a multi-branch node needs to be created as opposed to a binomial node. This can also be attributed to the limitation of LL(1) grammars, where they are restricted to a one token look-ahead. An example of this problem is the *if-then-else* production. By including the *if-then-else* production, LL(1) conditions are broken as the production becomes ambiguous due to the optional else, this is known as the dangling else problem [18, p120]. The *if-then-else* production has three possible sub-trees as demonstrated in Figure 6.1. For our implementation we settled on implementing only binomial nodes, which would result in the *if-then-else* production not being represented correctly, as seen in Figure 6.2. The extended version of Parva [37], which allows for the declaration of functions, will results in incorrect ASTs being generated. This is because each function needs to be added as a sub-tree to the root node, the *Parva* class node. As only binomial trees are supported, this will result in incorrect ASTs being produced and sub-trees being overwritten, this is discussed in greater detail in Section 7.2. Restricting the implementation to the use of only binomial nodes has simplified the implementation of automatic AST creation, but results in only strictly LL(1) grammars being correctly visualised.

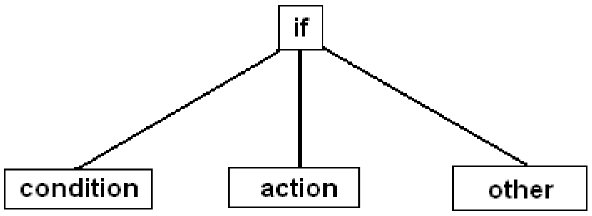


Figure 6.1: The correct syntax tree for *if-then-else*

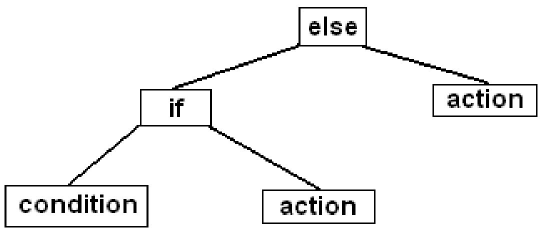


Figure 6.2: The syntax tree produced for the *if-then-else* production

6.4 Code generation

Code generation for the parser occurs within the *ParserGen* class. Once a grammar is parsed correctly, Coco/R calls the *WriteParser* method from *ParserGen*. The *WriteParser* method first validates that the *Parser.frame* file exists, and then proceeds to copy the different frame parts to the new parser file. During the copying of the frame file, our AST *TreeNode* class is added to the parser file. Another addition to the frame file is the inclusion of the *GetTree* method, which is used to return the AST generated during parsing. This provides a common interface for retrieving the AST using *reflection* as discussed in Section 6.5. The next step in parser generation is to call the *GenProductions* method.

6.4.1 Modifications to GenProductions

The method for generating productions, *GenProductions*, is altered in multiple ways. The standard version of *GenProductions* produces a method declaration similar to the example in Listing 6.4. Any attributes defined in the grammar are automatically added to the method declaration as parameters.

Listing 6.4: Method declaration as generated by unmodified GenProductions method

```
static void Factor()
{
                    ...
}
```

As discussed earlier the generated parser is a recursive descent parser. For this reason tree nodes need to be passed back to the calling method after they have been created. For this we used the *out* keyword to pass the new tree node up the call stack. The *GenProductions* method had to be modified to include this behaviour in our newly generated parser. Using the C# version of Ccoo/R, which allows multiple values to be returned by a method, ensures that the expected behaviour of a generated production is not altered. Code is included to perform logic checks, the first of which examines whether the production being generated is our so called *Goal* symbol. If the production to be generated is the *Goal* symbol, no output parameter is added to the parameter list. This is because the *Goal* production defines the root node, which is at the top of the call stack and no calling method exists for it. The second modification is to include code to generate the variables which will be used during node creation within the production. This includes the declaration of the left and right sub-tree nodes and variables to contain the node type and terminal value. The *out* parameter *TreeNode l* needs to be declared as null to satisfy the C# compiler's requirement that all *out* parameters are declared before use.

Listing 6.5: Method declaration as generated by modified GenProductions method

```
static void Factor(out TreeNode l)
{
        l = null;                  //left sub-tree
        TreeNode r = null;         //right sub-tree
        string symVal = la.val;    //literal value of the current token
        string prodVal = "Factor";//node type
        ...
}
```

Once the initial production declaration is complete, the *GenProductions* method calls the *GenCode* method to generate the code within the method.

6.4.2 Modifications to GenCode

The *GenCode* method uses the token type to determine what code should be generated. Code generation techniques are different for each token type and thus it was important to modify the correct routine to achieve AST generation. An AST node needs to be created when a *terminal* symbol is encountered. *NonTerminal* symbols identify calls to other productions and these production calls needed to be modified to include *TreeNode* passing.

Listing 6.6: Code generated using original GenCode

```
static void Factor()
{
        if (la.kind == decNumber_Sym) {
                Get();
        } else if (la.kind == hexNumber_Sym) {
                Get();
        } else if (la.kind == lparen_Sym) {
                Get();
                Expression();
                Expect(rparen_Sym);
        } else { SynErr(11);}
}
```

Listing 6.7: Code generated with modified GenCode method

```
static void Factor(out TreeNode l)
{
        l = null;
        TreeNode r = null;
        string symVal = la.val;
        string prodVal = "Factor";

        if (la.kind == decNumber_Sym) {
                l = new TreeNode(prodVal, la.val, l, r);
                Get();
        } else if (la.kind == hexNumber_Sym) {
                l = new TreeNode(prodVal, la.val, l, r);
                Get();
        } else if (la.kind == lparen_Sym) {
                Get();
                Expression(out l);
                Expect(rparen_Sym);
        } else { SynErr(11); }
}
```

As seen in Listing 6.7, the variable *prodVal* is always initialised as a string literal with a value that is the same as the current production name. The listing further demonstrates how a new *TreeNode* is constructed and returned as the *out* parameter *l*. Other than the insertion of AST node construction code, the original output of Coco/R remains unchanged. This ensures that the core functionality of Coco/R remains the same and functions as expected.

Within *GenCode*, two main sections of code generation logic have been used. The *GenCode* method contains a *switch* block used to determine what code should be generated according to the current token being parsed. The two main cases used for code generation are for *Terminals* and *NonTerminals*. The *NonTerminal* case is used to generate code that is used to call the next production in the recursive descent parser; code generated for a *NonTerminal* token can be seen in line 8 of Listing 6.6. Line 3 of Listing 6.6 demonstrates code generated for a *Terminal* token.

Determining which code routines are generated requires logic rules to be applied to the current token. The first piece of logic to evaluate is to determine if the child node should be added as a left or right sub-tree. We determined that a child node should be added to the left sub-tree if it is the first child node to be added. Once the first child node has been added to the tree, all subsequent child nodes should be added to the right sub-tree. A boolean value *outL* is used as a flag to signal whether code to insert a left child node has already been generated. However, conditions exist where code to insert a left child node needs to be generated more than once. This occurs in Listing 6.8, where the left child node can be added in any one of the conditional statements. Thus the *GenCode* routine needs to be aware that the first occurrence of *outL* is within a conditional statement and that all the alternative conditions can possibly be where the first child node is generated. The code generation routine should also be aware of the possibility that the first left child node could be generated after a conditional. This may occur in cases where the production starts with an optional *NonTerminal* such as:

Expression = [Term] Factor.

To ensure that the parser generator is aware of child node creation in alternative statements, the generation routine for *alternative* tokens has been modified. The alternative routine needs to be aware of nested alternative statements to ensure child nodes are created correctly.

The code generation for AST nodes representing *Terminal* symbols is more straightforward. When a *Terminal* symbol is encountered, code to generate a new *TreeNode* is inserted. Exceptions do exist; a node should not be generated if the next token is a production. This logic prevents meta-symbols from being added to the AST. An AST node should be constructed if the *Terminal* symbol encountered is present within the TOKENS section of the grammar.

Listing 6.8: Dealing with alternatives and conditional statements

```
static void Assignment(out TreeNode l)
{
        l = null;
        TreeNode r = null;
        string symVal = la.val;
        string prodVal = "Assignment";

        if (la.kind == identifier_Sym) {
                Designator(out l);
                if (la.kind == equal_Sym) {
                        symVal = la.val;
                        Get();
                        Expression(out r);
                        l = new TreeNode(prodVal,symVal,l,r);
                } else if (la.kind == plusplus_Sym) {
                        l = new TreeNode(prodVal,la.val,l,r);
                        Get();
                } else if (la.kind == minusminus_Sym) {
                        l = new TreeNode(prodVal,la.val,l,r);
                        Get();
                } else{
                        SynErr(54);
                }
        } else if (la.kind == plusplus_Sym) {
                Get();
                Designator(out l);
        } else if (la.kind == minusminus_Sym) {
                Get();
                Designator(out l);
        } else{ SynErr(55);     }
}
```

6.5 Retrieving the Tree

The abstract syntax tree is created during compile time. Our generated parser is used to parse user input and return the result as well as the AST. The source code generated by Coco/R is not yet runnable and needs to be compiled; for this we use the *CodeGenerator* class, which extends the *CodeDomProvider*. To execute our source code we call the *Run* method. The *Run* method first compiles our source code by executing the *Compile* method. Within this compile method we use the *CompileAssemblyFromSource* method found within the *CodeDomProvider* class. If our compilation is unsuccessful the *CompilerErrorCollection* is returned, which in turn is displayed to the user in the feedback section of the IDE. A successful compilation results in the creation of an executable *.dll* file, which can be executed using *Reflection*[1] and the *Invoke* method.

Firstly the *Scanner* class is initialised by calling the *Init* method, as seen in lines 7-9 of Listing 6.9. The next step is invoking the *Parse* method, which parses the entire input file and returns the AST if there are no errors. The AST is represented as an object when retrieved using reflection, but we know the structure of each *TreeNode* as defined by our abstract *INode* class. The AST is parsed using the *ConvertToNode* method, which uses pre-order traversal to walk the AST and convert each node object into a *TreeNode*. The converted tree is passed into the static *BinaryTreeImage* class, which visualises the tree as discussed in Section 5.2.3. The visualised tree is added to the visualisation tab enabling the user to interact with it.

[1]Reflection is the process of inspecting metadata and compiled code at runtime [1].

Listing 6.9: Using reflection to retrieve the AST

```
private static void Run(CompilerResults compile, string nameSpace,
                        string testGrammer,bool overlay)
{
    Module module = compile.CompiledAssembly.GetModules()[0];
    Type mt = null;
    MethodInfo methInfo = null;

    if (module != null)
        mt = module.GetType(nameSpace + ".Scanner");
    if (mt != null)
       methInfo = mt.GetMethod("Init", new[] {typeof (string)});
    if (methInfo != null){
       methInfo.Invoke(null, new object[] {testGrammer});
       mt = module.GetType(nameSpace + ".Parser");
       methInfo = mt.GetMethod("Parse");
       methInfo.Invoke(null, null);

       methInfo = mt.GetMethod("GetTree");

       object tt = methInfo.Invoke(null, null);
       BinaryTreeImage.Clear();

       object ttmp = ConvertToNode(tt);
       BinaryTreeImage.CreateBinaryTreeImage((TreeNode) ttmp,overlay);
     }
}
```

Chapter 7

Results and Discussion

7.1 Testing

7.1.1 AST generation

We tested the new version of Coco/R to ensure that the core functionality had not been altered. The output from our testing was compared to the output produced by standard Coco/R. Apart from the introduction of AST generation code, the output produced was identical to the original Coco/R output. The next step was to evaluate our tree construction mechanisms. For this we used grammars produced in practicals and tutorials from undergraduate classes in previous years, as well as examples taken from *Compiling with C# and JAVA* [37]. ASTs were hand constructed to compare with our compiler generated ASTs. The test aimed to identify areas where the produced ASTs did not correctly represent the grammar's syntax. Furthermore, the tests ensured that left and right associativity was correctly visualised. The following input was tested using the parser generated from the grammar described in Listing A.4:

(a+b)-(c+(d+b))*(a/c) =

The resulting AST was visualised using the Coco IDE. From Figure 7.1 it can be seen that the generated parser has correctly constructed the AST.

Further tests were performed using more complex grammars. Terry developed a LL(1) grammar to describe a simple teaching language known as Parva [37], and this grammar

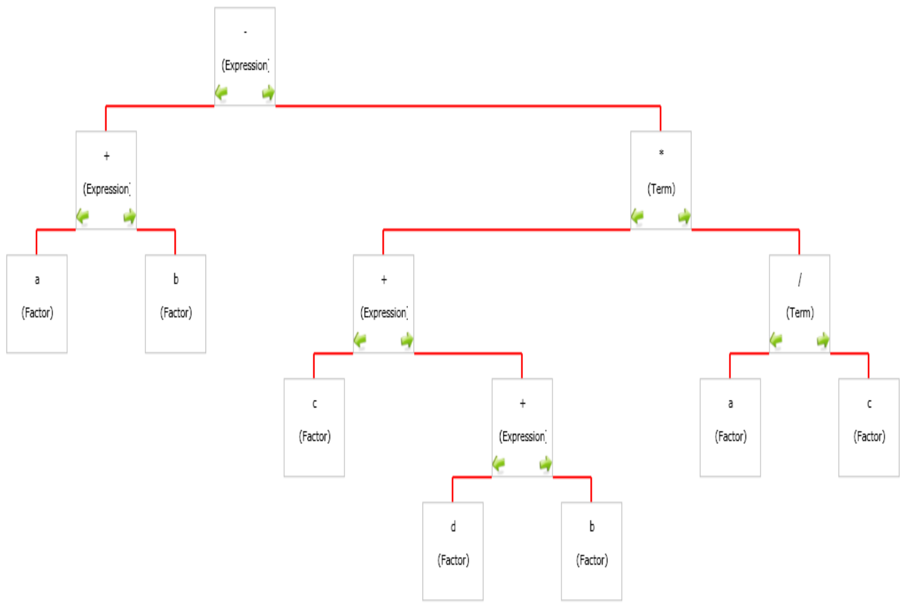


Figure 7.1: AST produced for the expression (a+b)-(c+(d+b))*(a/c) =

is described in Listing A.5. Using this description we tested the accuracy of the AST generation routine. The basic *Parva* program seen in Listing 7.1 results in a correct AST begin generated (see Figure 7.2). It should be noted that the simple *Parva* program is syntactically correct, but is not semantically correct as the variables p and q are never declared. However, the parser produced by Coco/R will parse the simple program correctly as it has no semantic awareness. This demonstrates that the student is responsible for implementing semantic checks in the form of semantic actions.

Listing 7.1: Simple *Parva* program

```
void Main()
{
        while(true)
        {
                if(p>q)
                {
                        p = 1 + 2;
                }
}
```

Once we introduced more complex productions such as conditional statements, the accuracy of the AST was decreased. It was found that new tree branches were overwriting existing branches as opposed to being appended to the existing tree. We were also able to confirm that nodes with multiple sub-trees result in inaccurate ASTs being generated, even though the grammar parses correctly.

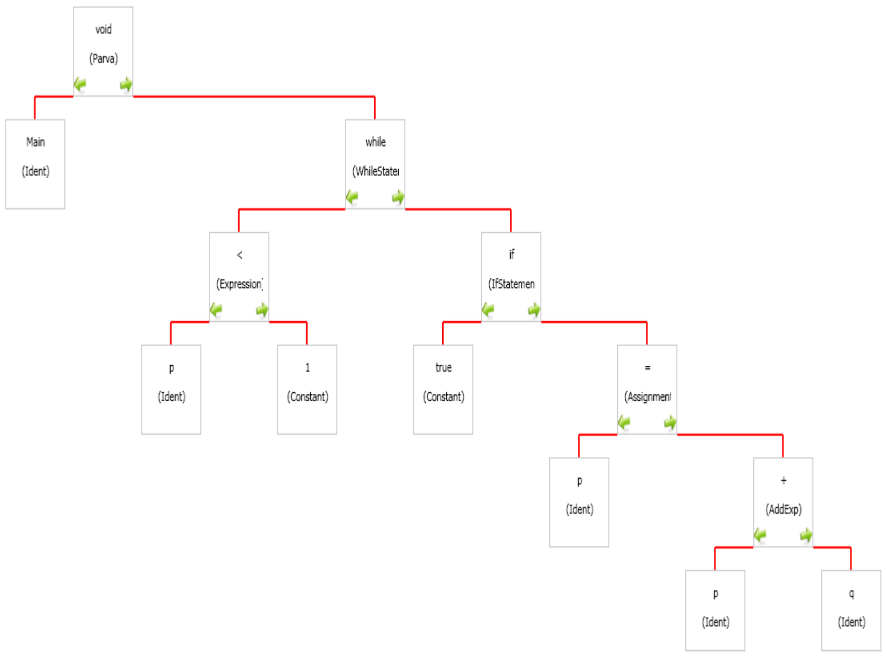


Figure 7.2: AST produced for a simple Parva program (Listing 7.1)

7.1.2 Visualisation effectiveness

A study was planned to evaluate the usability of the Coco/R IDE, and to measure the effectiveness of visualisation as a teaching aid. The study would be aimed at third year computer science students and would be completed during the practical sessions. Students would be asked to complete a pre-practical test, which would measure their understanding of the parsing routine and the syntactic relation to the grammar. Once the test was completed, students would randomly be divided into two groups. During the practical session the first group would complete the practical as normal, while the second group would use the Coco/R IDE. After the practical session, all the students would be required to complete another test. The results from the pre-practical test would be compared to the results of the post-practical test. The difference between the pre-practical and post-practical results for the two groups would then be compared to determine if there was any noticeable improvement in the group that used the Coco/R IDE as compared to the group who completed the standard practical. A final test would be given to the students the following week to determine if there was a long term benefit in using visualisation as a learning aid. The group using the Coco/R IDE would also complete a questionnaire where they rate the ease of use and effectiveness of the IDE. The study was not completed for reasons listed in the discussion to follow (Section 7.2.2).

7.2 Discussion

7.2.1 AST generation

AST generation for simple grammars such as the *Calculator* grammar in Listing A.4 produced syntactically correct trees. Both left and right recursive productions are correctly visualised and the user can clearly see this when looking at the visualisation. This holds true for all grammars that are strictly LL(1) and which require no additional decision making. However grammars that contain productions that result in nodes with multiple sub-trees are not accurately represented by our AST routines. This is due to the binary nature of our *TreeNode* implementation. Coco/R has no advanced knowledge of the possibility of multiple branch nodes being present in the AST. Furthermore, during AST code generation, Coco/R is not able to determine whether a multiple branch child node is required instead of the standard binary node. This can be attributed to the fact that no semantic knowledge exists during the parsing of the Cocol grammar and code generation of our recursive descent compiler. A possible solution to this would be to require the user to insert additional information into the grammar, in the form of semantic actions, which can be used by the code generation routines to identify multi-branch nodes. However this solution defeats the aims of our AST visualisation code, which aims to maintain a layer of abstraction between the user and AST construction.

7.2.2 Visualisation effectiveness

The study was not completed due to the complications encountered when a grammar, which is not perfectly LL(1), was parsed. It was found that trees generated using incorrect grammars tended to be confusing and hard to interpret. Students would be lead to believe that their work was incorrect, even when there were only minor problems with the grammar they had produced. This confirms results from studies discussed in Section 4, where it was observed that a commonly stated problem with visualisation is the time taken to find correct and accurate examples. To accurately evaluate the effectiveness of visualisation, students would have to be provided with working solutions that produce correct ASTs. This would mean altering the intended purpose of the Coco/R IDE, from being used as a learning aid, to be used as a demonstration tool instead.

Chapter 8

Conclusion

The primary aims of this project were to produce an Integrated Development Environment for Coco/R and to visualise the Abstract Syntax Trees produced during the parsing phase. Through these improvements to Coco/R the project aimed to increase the user understanding of recursive descent parser operation.

The design of the Coco/R IDE required careful consideration of the problem domain and needed to take into account the fact that the IDE would be used in a teaching environment. Consequently, the IDE needed to be easy to navigate and learn, ensuring that students could remain focused on the problem domain. Students desire a user interface that has a low learning curve and relates directly to the problem domain. The user interface design was kept unobtrusive and provides feedback implicitly where possible. Through implicit feedback users can gain greater understanding of the material without first having to understand the meaning of the feedback. The visualisations were designed to relate closely to the underlying structure of the data structures produced during parsing. Visualisations are displayed next to the user grammar, allowing the user to identify the association between the visualised data and the grammar.

The creation of ASTs occurs automatically, without the need of user input. By providing a layer of abstraction between the user and AST visualisation, users are able to use the system without any prior knowledge of AST creation being assumed. This feature allows for visualisation to be used as a teaching aid and not as a by-product of the teaching process. The automatic insertion of AST construction code into the parser produced by Coco/R required changes to be made to the original implementation of Coco/R. These changes were made in a manner that did not affect the operation of the parser produced, which will function identically to a parser produced using the original Coco/R.

Providing an IDE with a highly intuitive and usable interface along with visual feedback, users are able to gain a greater understanding of the compilation process. Visualised data structures present information to the user in an easy to understand manner and relate directly to the underlying data structures produced during compilation. The visualisation of ASTs help users identify the difference between syntax and semantics, as well as allowing users to visualise how a grammar is syntactically.

8.1 Possible extensions

The first extension to be considered is the introduction of nodes that can have multiple sub-trees. This will require Coco/R to have some semantic knowledge, which can be implemented in a similar manner to the current semantic action implementation. Users will be able to annotate the grammar to indicate nodes that have more than one sub-tree. However, this solution will remove some of the abstraction obtained through automatic AST code generation and will require greater user understanding of the parsing process and AST construction.

Another proposed extension is improving the visualisation of the parsing process through the introduction of step-by-step visualisation, which visualises each phase of the parsing process. Users should be able to control the speed of the visualisation. This would allow students to gain an even greater understanding of the actions of the parser and scanner.

Bibliography

[1] Albahari, J., and Albahari, B. *C# 3.0 in a nutshell.* O'Reilly Media, 2007.

[2] Alfred V. Aho, Monica S. Lam, R. S. J. D. U. *Compilers Principles, Techniques, & Tools:.* Pearson International, 2007.

[3] Almeida-Martínez, F. J., Urquiza-Fuentes, J., and Velázquez-Iturbide, J. A. Vast: a visualization-based educational tool for language processors courses. *SIGCSE Bull. 41*, 3 (2009), 342–342.

[4] Badre, A. N., Stasko, J. T., Lawrence, A. W., and Lawrence, A. W. Empirically evaluating the use of animations to teach algorithms. In *Proceedings of the 1994 IEEE Symposium on Visual Languages* (1994), pp. 48–54.

[5] Baecker, R. Sorting out sorting: A case study of software visualization for teaching computer science. In *Software Visualization: Programming as a Multimedia Experience, chapter 24* (1998), The MIT Press, pp. 369–381.

[6] Blaine A. Price, I. S. S., and Baecker, R. M. A taxonomy of software visualization. *Journal of Visual Languages and Computing 4* (1992), 211–266.

[7] Bower, R. *An investigation of a manipulative simulation in the learning of recursive programming.* PhD thesis, Iowa State University, 1998.

[8] Boyle, T. *Design for multimedia learning.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[9] Dann, W., Cooper, S., and Pausch, R. Using visualization to teach novices recursion. *SIGCSE Bull. 33*, 3 (2001), 109–112.

[10] Douglas, S. A., Stasko, J. T., Hundhausen, C. D., Hundhausen, C. D., Hundhausen, C. D., Douglas, S. A., and Stasko, J. T. A meta-study of algorithm visualization effectiveness.

[11] Elder, J. *Compiler Construction A Recursive Descent Model.* Prentice-Hall International, 1994.

[12] Elizabeth White, Laura Deddens, J. R. Software visualization of lr parsing and synthesized attribute evaluation. *Software: Practice and Experience 29*, 1 (January 1999), 1–16.

[13] Fisher, C. N., and Richard J. LeBlanc, J. *Crafting a Compiler with C.* Addison-Wesley, 1991.

[14] Howarth, N. Abstract syntax tree design. Tech. rep., Architecture Projects Management Limited, 1955.

[15] IWPC. Software visualization tools: Survey and analysis. In *IWPC '01: Proceedings of the 9th International Workshop on Program Comprehension* (Washington, DC, USA, 2001), IEEE Computer Society, p. 7.

[16] Lemone, K. A., O'Connor, M. A. A., McConnell, J. J., and Wisnewski, J. Implementing semantics of object oriented languages using attribute grammars. In *CSC '91: Proceedings of the 19th annual conference on Computer Science* (New York, NY, USA, 1991), ACM, pp. 190–202.

[17] Linsen, L., Karis, B. J., Gregory, E., and Hamann, M. B. Abstract tree growth visualization.

[18] Louden, K. C. *Compiler Construction Principles and Practice.* PWS Publishing Company, 1997.

[19] Marcus, A., Feng, L., and Maletic, J. I. 3d representations for software visualization. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization* (New York, NY, USA, 2003), ACM, pp. 27–ff.

[20] Mössenböck, H. *The Compiler Generator Coco/R.* Johannes Kepler University Linz.

[21] Naps, T., Cooper, S., Koldehofe, B., Leska, C., Rö, G., Dann, W., Korhonen, A., Malmi, L., Rantakokko, J., Ross, R. J., Anderson, J., Fleischer, R., Kuittinen, M., and McNally, M. Evaluating the educational impact of visualization. In *ITICSE-WGR '03: Working group reports from ITiCSE on Innovation and technology in computer science education* (2003), pp. 124–136.

[22] Naps, T. L., Rö, G., Almstrum, V., Dann, W., Fleischer, R., Hundhausen, C., Korhonen, A., Malmi, L., McNally, M., Rodger, S., and Velázquez-Iturbide, J. A. Exploring the role of visualization and engagement in computer science education. In *ITICSE-WGR '02: Working group reports from ITiCSE on Innovation and technology in computer science education* (2002), pp. 131–152.

[23] Norvell, T. S., and Bruce-lockhart, M. P. Teaching computer programming with program animation.

[24] Oliveira, N., Henriques, P. R., Cruz, D. D., Var, M. J., and Pereira, A. Visuallisa: Visual programming environment for attribute grammars specification.

[25] Oppermann, R. *Handbook on information technologies for education and training.* Springer-Verlag, 2002.

[26] Palmiter, S., and Elkerton, J. Animated demonstrations for learning procedural computer-based tasks. *Hum.-Comput. Interact. 8*, 3 (1993), 193–216.

[27] Parr, T. *ANTLR Reference Manual*, 2.7.4 ed. University of San Francisco, May 2004.

[28] Parr, T. J., and Quong, R. W. Antlr: A predicated-ll(k) parser generator. *Software Practice and Experience 25* (1994), 789–810.

[29] Pauw, W. D., Jensen, E., Mitchell, N., Sevitsky, G., Vlissides, J. M., and Yang, J. Visualizing the execution of java programs. In *Revised Lectures on Software Visualization, International Seminar* (London, UK, 2002), Springer-Verlag, pp. 151–162.

[30] Petre, M., and Blackwell, A. F. Mental imagery in program design and visual programming. *Int. J. Hum.-Comput. Stud. 51*, 1 (1999), 7–30.

[31] Reed, S. K. Effect of computer graphics on improving estimates to algebra word problems. *Journal of Educational Psychology 77*, 3 (1985), 285 – 298.

[32] Resler, D. Using visual compilers in the compiler construction curriculum. In *In Proceedings of the 4th Annual Conference on the Teaching of Computing* (Dublin, Ireland, August 1996), Dublin City University, pp. 195–197.

[33] Resler, D., and Deaver, D. M. Vcoco: A visualisation tool for teaching compilers.

[34] Spence, R. *Information Visualization.* ACM Press, 2001.

[35] Stansifer, R. *The study of programming languages.* Prentice-Hall, Inc., 1995.

[36] Terry, P. *Compilers and Compiler Generators: An Introduction With C++.* Thomson Computer Press, 1997.

[37] Terry, P. *Compiling with C# and JAVA.* Pearson Education Limited, 2005.

[38] Thomas Pittman, J. P. *The Art of Compiler Design Theory and Practice.* Prentice-Hall International, 1992.

[39] Thrembly, J.-P., and Sorenson, P. G. *The Theory and Practice of Compiler Writing.* McGraw-Hill Book Company, 1985.

[40] Von, S., Garcia, M., and harburg germany, T. U. H. Eclipse corner article how to process ocl abstract syntax trees, 2007.

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

[42] Wöss, A., Löberbauer, M., and Mössenböck, H. Ll(1) conflict resolution in a recursive descent compiler generator. In *In JMLC03, volume 2789 of LNCS* (2003), Springer-Verlag, pp. 192–201.

Appendix A

Code Listings

Listing A.1: Definition of CodeToken object

```
public class CodeToken
{
   public CodeTokenType TokenType
   {
     get;set;
   }

   public int Start
   {
      get;set;
   }

   public int End
   {
      get;set;
   }

   public int Length
   {
     get{ return End - Start; }
   }
}
```

Listing A.2: Code listing for TreeNodeControl.cs

```
public partial class TreeNodeControl : UserControl
{
  public int depth;             //keep track of depth of nodes to show
  public TreeNodeControl left;  //left child node
  public TreeNodeControl right; //right child node

  //connector between parent and left node
  public ConnectorLine leftLine = new ConnectorLine();
  //connector between parent and right node
  public ConnectorLine rightLine = new ConnectorLine();
  //reference to self
  public CocoIDE.BinaryTreeImage.ImageNode thisNode;

  public TreeNodeControl() //constructor
  public void setOverlay() //used compare two trees

  //event handler for user collapsing/expanding left child tree
  private void imageLeft_MouseLeftButtonDown
              (object sender, MouseButtonEventArgs e)

  //event handler for user collapsing/expanding right child tree
  private void imageRight_MouseLeftButtonDown
             (object sender, MouseButtonEventArgs e)

  //expand left/right child tree upto set depth
  public void expand(int endDepth)
  //collapse left/right child tree
  public  void collapse()
}
```

Listing A.3: Code listing for DrawNode method in BinaryTreeImage.cs

```
protected static Rectangle DrawNode(ImageNode node, bool isRightChild,
                                    int depth, Rectangle parentBounds,
                                    out TreeNodeControl treeNodeControl,
                                    bool isOnlyChild,bool overlay)
{
   int x, y;
   treeNodeControl = new TreeNodeControl();

   if (node == null) return new Rectangle();

   int offset = (isRightChild) ?
                node.LeftTreeWidth + NodeWidth + Dx
                : -node.RightTreeWidth - NodeWidth - Dx;
   x = (isOnlyChild)? parentBounds.X:parentBounds.X + offset;
   y = depth * (Dy + NodeHeight) - HalfNodeHeight;

   Rectangle bounds = new Rectangle(x, y, NodeWidth, NodeWidth);

   treeNodeControl.label_Type.Content = node.Node._type;
   treeNodeControl.label_Production.Content
                                   = "("+node.Node._production+")";
   treeNodeControl.depth = depth;
   treeNodeControl.thisNode = node;

   if (overlay) treeNodeControl.setOverlay();

   _canvas.Children.Add(treeNodeControl);
   Canvas.SetLeft(treeNodeControl, x);
   Canvas.SetTop(treeNodeControl, y);

   if (_rootNode==null)
   {
      _rootNode = treeNodeControl;
   }
```

```
  if (node.Left != null)
  {
     TreeNodeControl left;
     Rectangle leftBounds = DrawNode(node.Left, false, depth + 1,
                                     bounds, out left,
                                     node.Right == null, overlay);
     Line lh = new Line
     {
       StrokeThickness = 2,
       Stroke = overlay ? Brushes.LightBlue : Brushes.Red,
       X1 = (node.Right==null)? bounds.X:leftBounds.X + HalfNodeWidth,
       X2 = bounds.X,
       Y1 = bounds.Y + NodeHeight,
       Y2 = bounds.Y + NodeHeight
     };

    _canvas.Children.Add(lh);

    treeNodeControl.left = left;
    treeNodeControl.leftLine.horizontal = lh;

    Line lv = new Line
    {
      StrokeThickness = 2,
      Stroke = overlay ? Brushes.LightBlue : Brushes.Red,
      X1 = leftBounds.X + HalfNodeWidth,
      X2 = leftBounds.X + HalfNodeWidth,
      Y1 = bounds.Y + NodeHeight,
      Y2 = leftBounds.Y
    };

    _canvas.Children.Add(lv);

    treeNodeControl.leftLine.vertical = lv;

 }else
  {
      treeNodeControl.imageLeft.Visibility = Visibility.Hidden;
    }

  if (node.Right != null)
  {
     TreeNodeControl right;

     Rectangle rightBounds = DrawNode(node.Right, true, depth + 1,
                                      bounds, out right,
                                      node.Left == null, overlay);
     Line lh = new Line
     {
       StrokeThickness = 2,
       Stroke = overlay? Brushes.LightBlue:Brushes.Red,
       X1 = bounds.X + NodeWidth,
       X2 = node.Left==null? bounds.X:rightBounds.X + HalfNodeWidth,
       Y1 = bounds.Y + NodeHeight,
       Y2 = bounds.Y + NodeHeight
     };

    _canvas.Children.Add(lh);

    treeNodeControl.right = right;
    treeNodeControl.rightLine.horizontal = lh;

    Line lv = new Line
    {
      StrokeThickness = 2,
      Stroke = overlay ? Brushes.LightBlue : Brushes.Red,
      X1 = rightBounds.X + HalfNodeWidth,
      X2 = rightBounds.X + HalfNodeWidth,
      Y1 = bounds.Y + NodeHeight,
      Y2 = rightBounds.Y
    };

   _canvas.Children.Add(lv);
   treeNodeControl.rightLine.vertical = lv;
```

```
 }
 else
 {
    treeNodeControl.imageRight.Visibility = Visibility.Hidden;
 }

 return bounds;
}
```

Listing A.4: A simple calculator grammar

```
COMPILER      Calc  $CN
/* Simple four function calculator
   P.D. Terry, Rhodes University, 2009 */

CHARACTERS
  digit      = "0123456789" .
  symbol     = "abcdefghijklmnopqrstuvwxyz".
  hexdigit   = digit + "ABCDEF" .

TOKENS
  decNumber  = digit { digit } .
  hexNumber  = "$" hexdigit { hexdigit } .
  symbolic   = symbol.

PRODUCTIONS
  Calc       = { Expression "=" } EOF .
  Expression = Term { "+" Term  |  "-" Term } .
  Term       = Factor { "*" Factor |  "/" Factor } .
  Factor     = symbolic | decNumber | hexNumber|"(" Expression")" .

END Calc.
```

Listing A.5: The Parva language grammar

```
COMPILER Parva
/* Parva level 1 grammar  - Coco/R for C# (EBNF)
   P.D. Terry, Rhodes University, 2009
*/

CHARACTERS
  lf          = CHR(10) .
  backslash   = CHR(92) .
  control     = CHR(0) .. CHR(31) .
  letter      = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit       = "0123456789" .
  binDigit    = "01" .
  hexDigit    = digit + "abcdefABCDEF" .
  stringCh    = ANY - '"' - control - backslash .
  charCh      = ANY - "'" - control - backslash .
  printable   = ANY - control .

TOKENS

  identifier = letter
               { letter
                 | digit
                 | "_" { "_" } ( letter | digit )
               } .
  number     = digit { digit }
               | digit { hexDigit } 'H'
               | binDigit { binDigit } '%' .
  stringLit  = '"' { stringCh  | backslash printable } '"' .
  charLit    = "'" ( charCh    | backslash printable ) "'" .

COMMENTS FROM "//" TO lf
COMMENTS FROM "/*" TO "*/"

IGNORE CHR(9) .. CHR(13)
```

PRODUCTIONS

```
  Parva
  = "void" Ident "(" ")" Block .

  Block
  = "{" { Statement } "}" .

  Statement
  =    Block | ConstDeclarations
     | VarDeclarations | AssignmentStatement
     | IfStatement | WhileStatement
     | ReturnStatement | HaltStatement
     | ReadStatement | WriteStatement
     | ForStatement | DoWhileStatement
     | BreakStatement | ContinueStatement
     | ";" .

  ConstDeclarations
  = "const" OneConst { "," OneConst } ";" .

  OneConst
  = Ident "=" Constant .

  Constant
  = number | charLit | "true" | "false" | "null" .

  VarDeclarations
  = Type OneVar { "," OneVar } ";" .

  OneVar
  = Ident [ "=" Expression ] .

  AssignmentStatement
  = Assignment ";" .

  Assignment
  =    Designator
```

```
   (    "=" Expression | "++" | "--" )
      | "++" Designator
      | "--" Designator .

Designator
= Ident [ "[" Expression "]" ] .

IfStatement
= "if" "(" Condition ")" Statement
  [ "else" Statement ] .

DoWhileStatement
= "do" Statement "while" "(" Condition ")" ";" .

ForStatement
= "for" "("
     [ [ BasicType ] Ident "=" Expression ] ";"
     [ Condition ] ";"
     [ Assignment ]
  ")" Statement .

BreakStatement
= "break" ";" .

ContinueStatement
= "continue" ";" .

WhileStatement
= "while" "(" Condition ")" Statement .

ReturnStatement
= "return" ";" .

HaltStatement
= "halt" ";" .

ReadStatement
```

```
= "read" "(" ReadElement { "," ReadElement } ")" ";" .

ReadElement
= stringLit | Designator .

WriteStatement
= "write" "(" WriteElement { "," WriteElement } ")" ";" .

WriteElement
= stringLit | Expression .

Condition
= Expression .

Expression
= AddExp [ RelOp AddExp ] .

AddExp
= [ "+" | "−" ] Term { AddOp Term } .

Term
= Factor { MulOp Factor } .

Factor
=     Designator
    | Constant
    | "new" BasicType "[" Expression "]"
    | "!" Factor
    | [ "char" | "int" ] "(" Expression ")" .

Type
= BasicType [ "[]" ] .

BasicType
= "int" | "bool" | "char" .

AddOp
```

```
  = "+"  |  "−"  |  "||"  .

  MulOp
  = "*"  |  "/"  |  "%"  |  "&&"  .

  RelOp
  = "=="  |  "!="  |  "<"  |  "<="  |  ">"  |  ">="  .

  Ident
  = identifier  .

END Parva.
```