RHODES UNIVERSITY
Where leaders learn

Rhodes University
Computer Science Department

Honours year project

Literature Review

Author: Etienne Stalmans
Supervisor: Dr. Karen Bradshaw
June 2010

1 Introduction

Software visualisation (SV) is the process of producing a visual metaphor to represent the operation of a program or algorithm [13]. The use of SV in teaching is not a new concept and numerous studies have been performed to evaluate its effectiveness. The aim of SV in teaching is to increase learners understanding of concepts through simplification. In order for a SV system to be successful, the visual metaphor needs to effectively and efficiently relay the mapped data [4, 8]. Dougles *et al.*, [8] found that "active use of visualizations by students improves their learning process". Compiler generators produce human-readable parsers from user specified grammars. A grammar is used to formalise the syntax and semantics of a language. In order to understand how a language is parsed, it is important to understand the grammar first. Grammars are classified according to hierarchical type. The grammar type determines which parsing method will be used to evaluate it.One such parsing method is recursive descent parsing. Recursive descent parsers are used to parse Type 2 grammars [25], also known as context free grammars. Coco/R is a recursive descent parser generator which is used to teach compiler theory [25]. VCOCO provides limited visualisation of Coco/R operation. This is done by highlighting the current execution point [23]; providing a debugging approach to expert users [2]. Thus VCOCO is not suitable to be used as an education tool [2], as it assumes prior knowledge of compilers and does not contain explanation facilities which beginners require. Further, VCOCO does not allow for the visualisation of syntax trees. Abstract syntax trees provide an easy to follow description of how a grammar is processed. This makes abstract syntax trees very effective for teaching the basic operation of parsers to new users. A suitable visualisation of the abstract syntax trees should lead to easier understanding of compiler operation.

2 Grammars

Grammars are used to formally specify the syntax of a language. A grammar can be described as a structure $<$ N,T,P,S $>$ [25, 27] 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. Productions are syntax equations [25] which relates two strings and describes how they can be transformed into each other. Terminal symbols literal strings which cannot be broken into smaller units. Non-terminals are symbols which can be replaced; therefore they can be composed of terminal and non-terminal symbols. Grammars must conform to certain restrictions in order to used in the automatic construction of parsers and compilers [25]. 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 [1]. Further grammar types can be classified using the Chomsky hierarchy. The Chomsky hierarchy consists of four classes of grammar, Type3, Type2, Type1 and Type0 [25]. Most modern languages can be described using Type2 or context-free grammars [25]. 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 [25].

2.1 LL(1) grammars and attributed grammars

Grammars can be said to be LL(1) complaint when it is possible to evaluate the grammar, from left to right, while only looking one terminal ahead [1, 27]. LL(1) conflicts can occur under three different conditions; these are, explicit alternatives, options and iterations. Mössenböck, Wöß and Löberbauer [27] 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(a) denotes the set of terminal start symbols of the EBNF expression a; 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 [12] defined attributed grammars which could be used to describe the translation of languages [27]. EBNF as described by Wirth [26], is a notation used to describe a language [25]. EBNF defines metasymbols which are used to simplify the expression of grouping, alternatives, optional symbols and much more [25, 26]. The attributes of the grammar consists of non-terminals; these non-terminals can be described as input attributes, which provide context information, and output attributes which provide results calculated during the processing of the non-terminals [27]. The semantic actions are executed as statements in an imperative programming language, during the parsing process [25, 27]. Recursive descent parsers require grammars to be LL(1) compliant [27]. The LL(1) restriction is enforced as the parser needs to be able to select between alternatives with a single look ahead symbol [27].

3 Compiler Generators

Compiler generators are used to construct human-readable parsers automatically [18]. Traditionally hand-built, recursive-descent parsers where used to recognise languages [18, 27]. These hand-built parsers where able to resolve LL(1) conflicts using semantic information or by performing a multi-symbol look ahead [27]. 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 [27]. To solve this problem, recursive-descent parser producing compiler generators where created [27]. Two early compiler generators are JavaCC and ANTLR. JavaCC generates recursive descent parsers which are able to resolve LL(1) conflicts [27]. ANTLR produces recursive descent parsers which process LL(k) grammars, where $k > 1$ [18]. Furthermore the grammars processed by ANTLR use predicates to aid in the resolution of LL(1) conflicts [18]. The ANTLR compiler generator comes with a tree generator, SORCERER [18, 27].

3.1 Coco/R

The compiler generator studied in this paper is Coco/R. Coco/R is classified as a recursive-descent parser as it does a top-down evaluation of grammars. Recursive-descent parsers use a procedure for each non-terminal [1]. By looking at the input it receives, the procedure determines which production to apply. Using an attributed grammar, Coco/R produces a scanner and recursive descent parser [27]. The productions of an attributed grammar get processed from left to right by Coco/R [25]. Coco/R does syntax analysis and semantic actions, defined by attributes, are executed when they are encountered. The user is able to add custom classes, which perform actions such as code optimisation, code generation and symbol table handling [25, 27]. The grammar is attributed with semantic actions which can be used to access the user defined classes. LL(1) conflicts in Coco/R are handled in a similar manner to ANTLR and JavaCC. This is done through conflict resolvers [27]. Conflict resolvers are Boolean expressions which are inserted before the first occurrence of conflicting alternatives [18, 27]. Another method of conflict resolution implemented in Coco/R is the use of semantic actions. The semantic actions help determine which alternative to select and are inserted into the generated parser [27]. Unlike ANTLR, Coco/R does not include a tree generator.

4 Abstract Syntax Trees

Aho *et al.*, [1] 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 1, where the root is 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 [1, 25].

The parsing process evaluates a grammar on a symbol by symbol basis. This process is recursive in nature and can be represented using parse trees [10]. An abstract syntax tree consists of inter-related nodes which represent data structures [10]. 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 [10, 25]. The construction of syntax trees can be used to perform simple type checking [1]. This is done by matching actual types with expected types as the a node is about to be added to the tree [1]. Further analysis can be done once the AST has been constructed; by executing code fragments at each node in the AST [1].

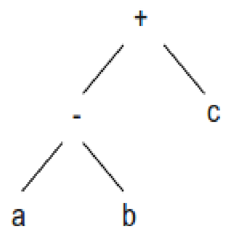

Figure 1: Abstract Syntax Tree for a-b+c

5 Software Visualisation

In order to understand the term software visualisation, it is first required to define visualisation. Visualisation, as defined by the Oxford dictionary, is the "the power or process of forming a mental picture or vision of something not actually present to the sight". Other terms often associated with visualisation are *program visualisation* and *algorithm visualisation* [4]. Program visualisation refers to the various techniques used to enhance the understanding of computer programs [4]. While algorithm visualisation is defined as a the process where-by actual implemented code is visualised [4]. 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". Software visualisation (SV) is used to solve a wide range of problems including algorithm animation and visual programming [13]. In order to be successful, the SV needs to provide a natural and direct mapping from the visual metaphor to the source code and back [13]. The dimensions of SV systems, as described by Marcus, Feng and Maletic [13] are:

- Tasks - why is the visualisation needed?
- Audience - who will use the visualisation?
- Target - what is the data source to represent?
- Representation - how to represent it?
- Medium - where to represent the visualisation?

Representation is an important aspect of SV as this defines how raw data is mapped in to a visual structure and view [13]. Expressiveness and effectiveness are two criteria for evaluating the mapping of data to a visual metaphor [13]. Expressiveness describes how well the visual metaphor represents all the available information. The expressiveness of the SV solution can be described by the ratio between the number of visual parameters and the number of data values [13]. Effectiveness relates to how effective the metaphor is at representing information [13]. Thus it can be seen that the effectiveness of visualisation relates to its semantic richness, simplicity and the level of abstraction [13].

5.1 Effectiveness of Software Visualisation

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 are:

- Help illustrate algorithm operation in lectures
- Help students learn the fundamental algorithms found in computer science
- Help 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 [7]. Software visualisation aims to increase the understanding of how programs work [4, 8, 13]. In investigating the effectiveness of SV, there has been ample evidence that visualisation can significantly reduce the effort required to comprehend a system [11]. In a study conducted by Bassil and Keller [11], 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 [8]. In order 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 aid in the learning process [8, 13]. Cognitive Constructivist theory states that active learning is superior to passive learning [8]. Visualisation allows learners to actively engage with the work by augmenting their view [8]. This view is further enhanced by Bower's [5] 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 [8]. 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 [7]. However, the educational impact of visualisation is dependant on the enhancement of learning with visualisation and the usage of visualisation in the classroom [14, 15]. It has been found that instructors are unwilling to use visualisation in their classrooms [14]. As stated by Naps *et al.*, [14] "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 [15], 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.

6 Related Work

There have been numerous projects which aim to visualise how compilers work. Visual YACC was developed to aid in the teaching of compiler theory [9]. YACC (Yet Another Compiler Compiler) is a compiler generator which generates a

LALR parser. The program takes YACC grammars as input and produces visualisations of both the LR parse tree and LR parse stack [9]. Resler argues that "the study of the inner-workings of a compiler can be greatly simplified through use of a visible compiler" [23]. VCOCO (Visible COmpiler COmpiler) was constructed to generate LL(1) visible compilers [23]. VCOCO provides a visual front-end to Coco/R. The benefit of this approach is that VCOCO is capable of generating compilers which are functionally equivalent to compilers produced by Coco/R, given the same specifications [23]. The VCOCO interface is designed to provide an user with detailed, visual feedback of the compiler generator process. This is accomplished by inserting *hooks* into the semantic actions associated with each production in a grammar [23]. 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 [23]. The operation of VCOCO allows users to trace the compiler generation process from syntax analysis to code generation [23]. VCOCO's method of visualisation offers a debug approach for experts and is not ideal as an education tool [2]. ANTLR (ANother Tool for Language Recognition) is a parser generator which produces human-readable recursive-descent parsers [18]. Like Coco/R, ANTLR uses attributed grammars. However, unlike Coco/R, ANTLR accepts LL(k) grammars, where $k > 1$. ANTLR parsers can automatically construct abstract syntax trees [18]. In order to construct an AST, the user needs to annotate 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. The disadvantage with 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. There have been numerous studies using software visualisation

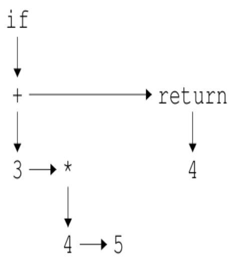

Figure 2: The abstract syntax tree resulting from "if 3+4*5 then return 4;" [18]

in aiding the teaching of programming concepts [7, 14, 15]. Norvell and Lockhart [16] 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 [16]. Norvell and Lockhart used their system as a teaching-aid while instructing an advanced programming course [16]. They found that using program animation they were able to ease the learning process for new students [16].

7 Conclusion

It has been found that software visualisation is effective in teaching programming concepts to new users [8]. However the overall effectiveness is limited due to demonstrators hesitation to use SV systems [8, 14]. Current systems that attempt to visualise compiler generators are too complicated to learn and use by novice users [2]. A successful implementation will need to be easy to learn, easy to use and needs to provide accurate representations of the data being visualised. The compiler visualiser would need to ease the concerns raised by demonstrators, as to increase the use of visualisation in the teaching of compiler theory.

References

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

[2] 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.

[3] 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.

[4] 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.

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

[6] catalin Roman, G., Cox, K. C., and Roman, D. C. A taxonomy of program visualization systems. *IEEE Computer 26* (1993), 11–24.

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

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

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

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

[19] 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.

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

[21] Price, B., Baecker, R., and Small, I. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing 4*, 3 (September 1993), 211–266.

[22] 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.

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

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

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

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

[27] 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.