Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 141 (2005) 189–194
www.elsevier.com/locate/entcs

# Eclipse Support for Using Eli and Teaching Programming Languages

Anthony M. Sloane[1,2]

*Department of Computing*
*Macquarie University*
*Sydney, Australia*

William M. Waite[3]

*Department of Electrical and Computer Engineering*
*University of Colorado*
*Boulder, USA*

**Abstract**

This demonstration will show Eclipse plugins developed at Macquarie and Colorado to support the Eli Language Processor Generation system and to enhance teaching of programming language concepts and implementation techniques. The plugins support exploration of programming language semantics, integrated development in the Eli system, and high-level observations of compiler execution.

*Keywords:* Language processor generation, integrated development environment, programming language education, debugging.

## 1 Introduction

Our Eclipse work is looking at supporting students and developers working in the programming language domain. We are working with both hand-crafted language processors and those generated by the Eli system [4]. In this demonstration we describe three main initiatives that support:

[1] This work is supported by a 2004 IBM Eclipse Innovation Grant.
[2] Email: asloane@ics.mq.edu.au
[3] Email: waite@cs.colorado.edu


doi:10.1016/j.entcs.2005.02.058

(i) Explorations of the effects of particular language semantic choices (Section 2).

(ii) Integrated development support for language processor generation with the Eli system (Section 3).

(iii) High-level observations of compiler execution (Section 4).

We are using Eli to generate semantic support for each of these projects. This strategy allows us to avoid hand-crafting document models and analysis as is done in most other Eclipse projects that provide sophisticated language support. Section 5 briefly discusses this semantic assistance aspect.

## 2 Exploring Semantics

The *Programming Language Detective* (PLD) is a system in which students can experiment with varying semantics for a language with fixed syntax [3]. Semantic properties such as scoping rules, type compatibility, and parameter passing modes can be varied.

The PLD compiler accepts programs written in a Modula-like syntax called *Mystery* together with a description of the desired semantics. The output is a Java program that can be run under the specified semantic conditions. Note that since the semantics can be varied so much, many static conditions are actually verified at runtime. This implementation detail is hidden from the students.

One kind of exercise supported by the PLD infrastructure requires the students to make a set of semantic decisions that will result in an implementation with certain properties. The desired properties might be specified by a collection of programs and their behaviors, or by a more general description. Students generate implementations from various combinations of decisions and test them against the specification. The final product is a report detailing the set of decisions and why they satisfy the given specifications.

The original PLD implementation is a Web-based system in which students submit programs and configure semantics via HTML forms. While this approach has advantages for student submissions of class exercises, it is less than ideal from a usability perspective. A text field of a Web form provides very little in the way of editing capability, and certainly has no facilities for helping the user to create a syntactically correct program. This leads to considerable frustration, particularly when the syntax is as unfamiliar as the Mystery syntax is to most students.

We have developed an Eclipse plugin to support student use of the PLD. The plugin consists of:

- A Mystery editor with completion support and syntax assistance.
- A launch configuration type that allows Mystery programs to be run from within Eclipse using user-specified semantics. Output goes to an Eclipse console.
- Output processing to present both compile-time and run-time errors using markers in the Mystery editor.

This plugin is also useful for exploring the effects of constellations of design decisions on usability.

## 3 Eli Development Tools

Eli's standard user interface is textual and line-oriented [4]. Users issue requests for derivations of desired products from the specifications that they have supplied. For example, a set of specifications containing regular expressions, a context-free grammar and an attribute grammar can be transformed into an executable for a complete language processor with a single command.

The Eli Development Tools (EDT) plugin provides support for developing and processing Eli specifications in the Eclipse style. At a basic level, specifications can be grouped into Eli projects and manipulated with language-specific editors and outline views. Apart from basic editor features such as completion, syntax assistance and popup tips, we are exploring more advanced capabilities including specification refactoring. EDT also completely hides Eli's textual interface behind an integrated menu-based system for issuing Eli derivations and automatic support for annotating specifications with error messages produced by Eli.

## 4 Observing Compilers

In previous work we have developed the Noosa tool for high-level execution monitoring of Eli-generated compilers [5]. The idea behind Noosa is that we can view the execution of a compiler in terms of concepts such as basic symbols, phrases, or syntax tree nodes without having to know anything about the way in which the compiler is coded. For example, when a compiler writer is developing a lexical analysis specification they work at the level of regular expressions. Thus Noosa provides a way for them to see which input text has matched which regular expressions. Similarly, they must specify a context-free grammar to build a parser. They are not interested in the detail of how the parser works but in which grammar productions match which parts of the input text.

Noosa is successful in a compiler-generation setting because it does not require the developer to have any knowledge of how the generator carries out its task. This approach is also effective in the classroom because although the students may have the compiler code in front of them, they often don't understand how it works. We have previously used Noosa in our teaching and have found that it increased the student's understanding of what the code was doing.

We have developed an Eclipse-based version of Noosa in the form of a plugin that supports a Noosa perspective and a collection of related views. The views provide access to high-level compiler runtime data such as the token stream, the phrase structure or the syntax tree. The plugin is initially aimed at compilers that are hand-written in Java but we are also working on Eli integration.

The Eclipse version of Noosa is integrated into normal development as a launch configuration type. Thus the students can run their Java-based compilers with or without Noosa support. While a student compiler is running, it communicates with the plugin via an event stream that signals when various high-level actions are taken. For example, the compiler's lexical analyser produces an event for each token that is recognised. A reverse communication stream enables the plugin to interrogate compiler data structures such as the abstract syntax tree or symbol table.

Noosa's views support browsing of the information obtained from the running compiler: When a compiler run is launched via a Noosa launch configuration, an editor is created for the input to that compiler run. Selections in the editor trigger other views to update their data to reflect the selection. The lexical view displays the tokens recognised in the selected region of text. Similarly, phrase structure and tree views focus their attention based on user selections. The tree view allows properties of nodes to be examined. Other views allow browsing of compiler data structures and can use data from the main views. For example, a student can select an identifier representation in a tree node and look that representation up in the definition table view to examine the identifier's properties.

Compared to the stand-alone version, the Eclipse-based version of Noosa offers a number of advantages that make the tool easier to use, maintain and evolve. First, integration with other components such as the Java and Eli Development Tools makes the user experience much more seamless. Second, even though Noosa previously used a user interface toolkit, Eclipse provides a much higher level of functionality and support which frees us to add more functionality. Finally, Noosa's communication with the running compiler is more robust and general in the Java setting than using the hand-crafted C

implementation of the stand-alone version.

## 5 Automatically Generating Semantic Assistance

Anyone who has developed a language-specific editor plugin for Eclipse has probably wrestled with how best to provide semantic assistance to the plugin. For example, if the editor needs to do completion on identifiers, how does the plugin analyse the document to find the identifiers? What if the plugin only wants identifiers of a particular kind, for example variable identifiers but not type identifiers? Also, editors may require many different forms of semantic analysis to support operations such as refactoring.

One way to implement semantic analysis in a plugin is to use simple techniques to approximate the information required. For example, a relatively simple scan of the document can reveal identifier-like text by matching against regular expressions. However, this approach is imprecise; it might find keywords that look like identifiers, for example. Also, it doesn't scale to problems such as name or type analysis that need accurate structural information.

A better method is to build an internal representation of the document obeying the semantics of that document and simplifying the particular kinds of analysis one wants to perform. Typically, such a representation can be divided into two components: an *abstract syntax tree* (AST) and a *definition table* (DT). The AST provides the structure of the document, and the DT provides the properties of the individual entities. In the Eclipse literature, such a representation is characterized by the API that it provides. That API is called the *document object model* (DOM) [2]. This approach is followed by the Java Development Tools.

Unfortunately, the programming effort required to implement the DOM is considerable. We felt that we should be able to automatically generate the semantic analysis modules of our plugins. Eli would supply all of the knowledge about how to build good scanners and parsers, how to implement ASTs, how to traverse them efficiently, and how to store program information in the DT. All we would have to do is specify the language-dependent bits of the analyses using notations such as regular expressions, context-free grammars and attribute grammars [1].

Eli-generated support is used in each of the three initiatives described in this paper. For example, the Mystery editor uses Eli support to help perform name completion. Eli-generated parsers incorporate automatic syntactic error recovery, whereas the distributed PLD compiler terminates after reporting a single error. Thus, by incorporating the generated module, we make the Mystery editor plugin considerably more capable than the compiler that it

supports.

## References

[1] "Eli Online Documentation," http://eli-project.sourceforge.net/elionline4.4.

[2] Arthorne, J. and C. Laffra, "Official Eclipse 3.0 FAQs," Addison-Wesley, 2004.

[3] Diwan, A., W. M. Waite and M. H. Jackson, *PL-detective: a system for teaching programming language concepts*, in: *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education* (2004), pp. 80–84.

[4] Gray, R. W., S. P. Levi, V. P. Heuring, A. M. Sloane and W. M. Waite, *Eli: a complete, flexible compiler construction system*, Commun. ACM **35** (1992), pp. 121–130.

[5] Sloane, A. M., *Debugging Eli-generated compilers with Noosa*, in: *Proceedings of the 8th International Conference on Compiler Construction* (1999), pp. 17–31.