Debugging Eli-Generated Compilers with Noosa

Anthony M. Sloane

Department of Computing, Macquarie University
Sydney, NSW 2109 Australia
asloane@mpce.mq.edu.au

Abstract. Source-level tools are not adequate for debugging generated compilers because they operate at the level of the generated implementation. It is inappropriate to expect compiler writers to be familiar with the implementation techniques used by the generation system. A higher-level approach presents debugging in terms of an abstract model of the implementation. For example, finite-state machines might be shown while debugging a scanner. This approach is inappropriate for developers who are not compiler experts and even for experts may present more information than is desirable.

An even higher-level approach is used by the Noosa graphical debugger for the Eli compiler generation system. The compiler writer is required to understand a simple execution model that involves concepts that they already have to understand to write Eli specifications. Noosa allows high-level data examination in terms of the input to the compiler and the abstract trees upon which attribution is performed. An event system allows fine-tuned control of program execution. The result is a debugging system that enables developers to diagnose bugs without having to have any knowledge of the underlying mechanisms used by their compiler.

1 Introduction

A variety of methods have been developed for automatically producing compilers from specifications. Using these techniques, a compiler writer can write a high-level specification of compiler functionality and a generation system will produce an implementation that conforms to that specification. Compiler generation has been successful mainly due to the existence of a range of specification notations covering important sub-problems, and the development of efficient methods for implementing these notations (for example, see [4,9,12,14,19]).

The major advantage of generation systems is that they enable a compiler writer to concentrate on the important issues while the responsibility for producing a correct implementation rests with the system. A generation system that is functioning correctly is no guarantee of a correct compiler, however, because the specification may have bugs.

Compiler generation systems typically provide very little in the way of debugging facilities. Source-level debuggers such as Dbx [8] or GDB [16] can be applied to generated compiler implementations, but this approach is largely unsatisfactory because it requires compiler writers to have specific knowledge about the implementation methods used by the generator. Requiring such knowledge defeats the main purpose of a generation system.

This paper describes the Noosa debugger for programs generated by the Eli compiler generation system. Eli generates compilers using a collection of specification notations including regular expressions for token description, context-free grammars for concrete and abstract syntax, and attribute grammars for semantic analysis and later phases.

Noosa presents a graphical view of the execution of a generated compiler in terms of the input being processed by it. The compiler writer can examine how their regular expressions and grammar productions were used to structure the input. Access is also provided to major compiler data structures such as the abstract trees upon which attribution takes place, environment structures and the definition table. Attribute values of any tree node can be examined in a flexible browsing system that is easily extended to new data types. The developer also has access to an abstract event stream produced by the running compiler. User-specified event handlers can be written in the Tcl language. This facility enables the specification of complex debugging operations such as semantic breakpoints and correlation of information from disparate sources.

The goal of Noosa is to provide debugging facilities that operate at the specification level, hiding the implementation details as much as possible. The aim is to achieve a level of implementation-hiding similar to source-level debuggers operating on programs compiled to machine language. In those debuggers the developer is able to interact with their program's execution in terms with which they are familiar. For example, data can be accessed via variable names and the control state is presented in terms of a stack of active routines and the statement about to be executed. Knowledge of all of these aspects can be expected of any programmer familiar with the source language.

Noosa does not present details of the implementation of compiler components. According to this philosophy, if the specification of the compiler includes regular expressions to describe token types then it is appropriate to present information in terms of tokens, where they were located in the input, and the regular expressions that matched them. Detail about the functioning of the finite-state machine that implements the scanner is not suitable because it relies on knowledge that the compiler writer may not have. This approach contrasts with other debuggers for generated components that present a large amount of internal detail.

Noosa can also work in combination with source-level tools. Eli allows arbitrary C code to be included in the generated compiler to implement abstract data types or to code a part of the functionality that is hard to specify. To accommodate debugging of this code, Noosa can be used in conjunction with a source-level debugger. Thus specification-level and source-level debugging can be undertaken at the same time.

Section 2 considers the execution model that should form the basis of a debugger for generated compilers and describes the model used by Noosa. Section 3 describes the basic elements of the Noosa design with reference to the execution model. Section 4 illustrates the Noosa style of debugging by describing typical compiler bugs and how Noosa would be used to diagnose them.

2 Execution Model

The functionality of a debugger is grounded in the facilities it provides for controlling execution of the program and the methods by which the program's state can be examined. For example, in most source-level debuggers breakpoints allow execution to be stopped when specific points in the code are reached. When execution is stopped, the values of program variables can be printed. Some debuggers have more advanced features such as conditional and data-dependent breakpoints or graphical displays of data structures, but the basic features are common to all source-level debuggers.

The kind of execution control and data access provided by a debugger depends intimately on an *execution model* that the debugger shares with its user. An execution model is a description of the structure of a program execution in terms of elementary actions and data items. An understanding of the program code and the execution model used by a debugger is necessary for the user to be able to operate debugger facilities and understand the output from debugging operations. For example, to use breakpoints in a debugger for an imperative language a user must understand the basic units of execution (e.g., statements) and the way execution proceeds (e.g., step-wise execution of statements plus routine calls). The state displayed by the debugger might also rely on the execution model (e.g., a stack of currently active routine calls).

Different debuggers for the same language can have different execution models that reflect the outcomes of design decisions about the kind of debugging that is to be permitted. For example, some source-level debuggers provide facilities for debugging at the machine level such as instruction stepping or the ability to examine the contents of registers. Other debuggers might omit such features on the grounds that machine level details are not relevant for a user of a high-level language. In general, the execution model to be used by a debugger depends on the view of execution that the debugger is trying to present.

In the compiler generation domain there are choices of execution model. If a source-level debugger is used to debug a generated compiler, the execution model is one appropriate to the implementation language. Even if the compiler writer is familiar with that language, they will in general not be experts in the generated code. Thus the use of an implementation language execution model is inappropriate.

At a higher level of abstraction lies a class of execution models based on the methods used to implement compiler components. For example, a generated scanner might use a finite-state machine implementation. An execution model for a debugger operating at this level might include concepts such as finite-machine states, input characters, and legal state transitions. Execution could be presented in terms of the actual transitions performed during scanning, perhaps with a visual representation of the machine.

Some compiler tools offer tracing facilities at this level. For example, parsers generated by YACC [7] and derivatives like Bison [3], can produce a trace of the parsing process. The trace consists of events such as getting a token, shifting a symbol, reducing via a rule, and changing parser state.

More user-friendly alternatives at this level exist in the form of debuggers that present the same kind of information as a trace, but in a graphical, browsable form. For example, Visual Parse++ from SandStone Technology Inc. [6] presents the developer with an extremely detailed view of the operation of generated parsers. Information presented includes depictions of the parse stack, lookahead tokens and three-dimensional views of parse trees . The recently released ParseView debugger bundled with the latest version of the ANTLR tool [13] seems to offer similar features.

The design of Noosa follows a higher-level approach. The execution model used by Noosa does not include anything to do with the implementation of generated components. Of course, compiler writers may well need to know something about component implementations in order to use Eli. For example, writing an Eli grammar may require some knowledge of the LALR parsing method since both of Eli's parser generators use that method and the system will reject non-LALR(1) grammars. Similarly, using Eli's attribute grammar notations may require some knowledge of allowable patterns of attribute dependences and the methods used by generated evaluators. The philosophy behind Noosa is that knowledge of this kind is not needed during debugging.

Some anecdotal support for this position was obtained recently when Noosa was used by final year students in an introductory compiler unit at Macquarie University. The students were able to use Noosa to debug Eli specifications containing regular expressions and context-free grammars. In this unit the students are acquainted with the compilation phases and their purposes, but the implementation techniques used by the tools are only covered in outline form. Thus there is some evidence that such knowledge is not necessary for debugging.

Justification for the Noosa position as a goal can also be obtained from a recognition that the widespread availability of compiler generation systems has resulted in many non-compiler experts attempting to develop compilers (or compiler-like programs). For these users it is important that the debugging system not rely on knowledge that they do not have.

It should be noted that the more advanced features in the debuggers mentioned above are presumably at least partly inspired by the more complex specification notations and implementation methods used in those systems compared to Eli. For example, the ANTLR parsers use multiple symbol lookahead and more advanced debugger support may well be necessary for the user to understand what is happening. Whether this situation is an argument for more complex debuggers or less complex specification notations and implementation methods is unclear. In any event, Noosa operates within the environment provided by the Eli notations and methods.

Figure 1 shows the execution model used by Noosa expressed in pseudocode. An Eli-generated program under the control of Noosa will scan the input attempting to locate tokens and trying to group those tokens into syntactic phrases. When a complete phrase is recognised an abstract tree fragment will be built to represent it and its components. Once the complete input has been recognised the tree is then decorated with attribute values. Usually a side-effect of one or more attribute evaluations will be to produce the compiler output. (To simplify the discussion we ignore the fact that attribution can occur during tree construction and that attribution can produce further trees which are in turn attributed. Both of these aspects are also supported by Noosa.)

```
while there is more input do
    get the next token
    if a complete syntactic phrase has been recognised then
        build an abstract tree fragment for the new phrase
while there are more tree attributes to evaluate do
    perform an attribute evaluation
```

Fig. 1. Noosa's execution model.

The concepts in the Noosa execution model are ones with which an Eli user can be expected to be familiar. The form of tokens is specified by the user using regular expressions or literals in the context-free grammar. The context-free grammar also specifies the valid phrases. The user's attribute grammar describes the abstract tree structure and the attribute computations that must be evaluated. These specification notations and their underlying concepts must be understood before a compiler can be specified using Eli.

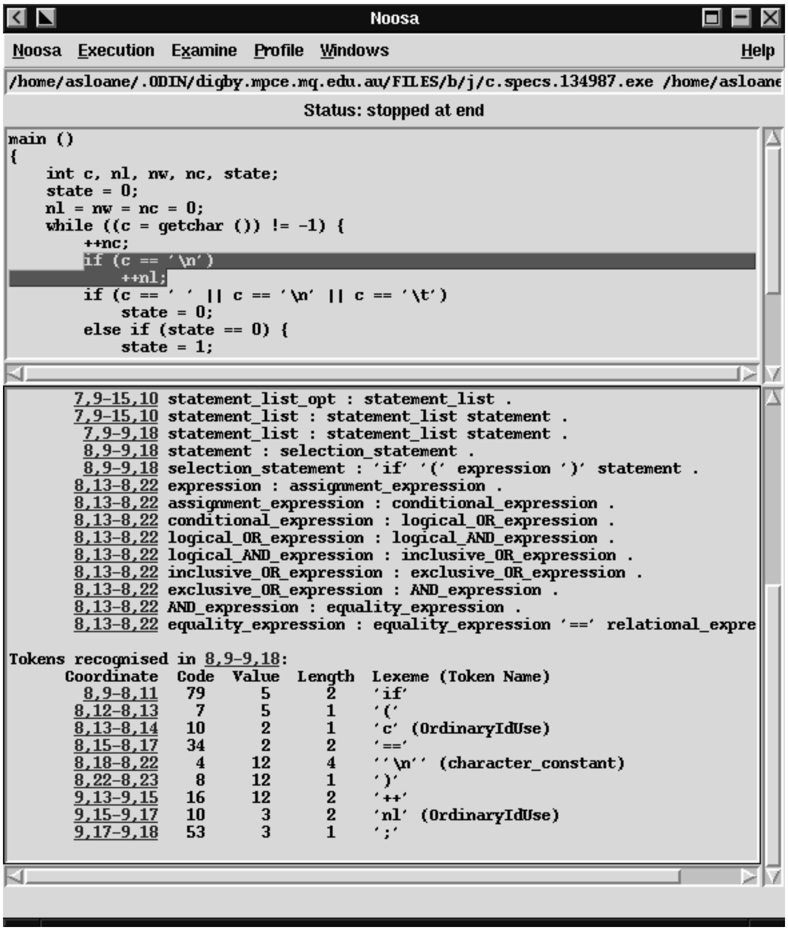
3 Noosa

Noosa's design is based on the execution model presented in the previous section. The input and the abstract tree play a central role in the user interface appropriate to their prominence in the model. Other data items can be accessed via a flexible browsing system designed to be easy to use and extensible to new data types. An event mechanism is used to allow both the debugger and the user to determine which actions are performed by the compiler being debugged and when they occur.

The rest of this section describes the main elements of the Noosa design. The discussion of features is structured according to the relevant elements of the execution model. Some mundane features such as menu invocations, searching in text windows, saving the context of text windows, file editing, on-line help, etc. are omitted. Example screenshots are taken from a debugging session for an Eli-generated C processor.

3.1 Input

The starting point of a compilation is the input that is to be analysed and translated. Thus the main user interface of Noosa features the input (Figure 2).



```
/home/asloane/.ODIN/digby.mpce.mq.edu.au/FILES/b/j/c.specs.134987.exe /home/asloane
```

Status: stopped at end

```
main ()
{
    int c, nl, nw, nc, state;
    state = 0;
    nl = nw = nc = 0;
    while ((c = getchar ()) != -1) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')
            state = 0;
        else if (state == 0) {
            state = 1;
```

```
      7,9-15,10 statement_list_opt : statement_list .
      7,9-15,10 statement_list : statement_list statement .
       7,9-9,18 statement_list : statement_list statement .
       8,9-9,18 statement : selection_statement .
       8,9-9,18 selection_statement : 'if' '(' expression ')' statement .
      8,13-8,22 expression : assignment_expression .
      8,13-8,22 assignment_expression : conditional_expression .
      8,13-8,22 conditional_expression : logical_OR_expression .
      8,13-8,22 logical_OR_expression : logical_AND_expression .
      8,13-8,22 logical_AND_expression : inclusive_OR_expression .
      8,13-8,22 inclusive_OR_expression : exclusive_OR_expression .
      8,13-8,22 exclusive_OR_expression : AND_expression .
      8,13-8,22 AND_expression : equality_expression .
      8,13-8,22 equality_expression : equality_expression '==' relational_expre

Tokens recognised in 8,9-9,18:
      Coordinate  Code  Value  Length  Lexeme (Token Name)
        8,9-8,11   79     5      2     'if'
       8,12-8,13    7     5      1     '('
       8,13-8,14   10     2      1     'c' (OrdinaryIdUse)
       8,15-8,17   34     2      2     '=='
       8,18-8,22    4    12      4     ''\n'' (character_constant)
       8,22-8,23    8    12      1     ')'
       9,13-9,15   16    12      2     '++'
       9,15-9,17   10     3      2     'nl' (OrdinaryIdUse)
       9,17-9,18   53     3      1     ';'
```

Fig. 2. Main Noosa window. The upper text window shows the compiler input (a C word counting program). The lower text window is a transcript of debugging output showing the phrase structure at the equality operator in the highlighted if statement and the tokens in that if statement.

Noosa correctly accounts for input processing that obtains text from multiple sources. For example, if the language has an "include" facility and the compiler expands includes during parsing, the input text window will show the complete input seen by the compiler. This removes any need for the user to guess what the compiler is working with or pre-process the input specially before debugging. The user can ask Noosa for the original source of any part of the input.

3.2 Phrases and Tokens

Syntactic phrases and lexical tokens play a central role in the user's understanding of their compiler. Noosa enables the user to determine which phrases and tokens are recognised by the compiler. Each one is associated with a region of the input text. Thus Noosa's "Phrase" and "Token" commands are invoked relative to coordinates in the input text.

To see the phrases recognised at a particular location in the input the user selects the location with the mouse and invokes "Phrase." This action produces a list of the concrete grammar productions that have been recognised whose coordinate ranges overlap the indicated location. For example, the transcript (bottom text window) in Figure 2 shows productions involved in recognising the equality operator in the highlighted if statement. The productions are listed in order from the axiom of the grammar (not shown) to the most specific. The coordinate range beside each instance indicates the input recognised by that instance.

The coordinate display for phrases is an instance of a "browsable value" (indicated by the underline). Browsable values can be clicked to obtain behaviour dependent on the kind of value. Clicking on a coordinate or coordinate ranges causes the indicated input to be highlighted in the input text window. Thus it is easy to see the input corresponding to a particular recognised production instance.

The "Token" command operates in a similar fashion to "Phrase." To see the tokens scanned in a particular region the user selects the region in the input text window with the mouse and invokes "Token." The transcript lists the relevant tokens (if any). For example, the transcript in Figure 2 shows the tokens from the highlighted if statement. The coordinate range of each token is shown along with the token code, intrinsic value[1], length and lexeme. Tokens which are non-literals and hence specified using regular expressions are also labelled with the regular expression name from the user's specification.

3.3 Abstract Tree

Once the input has been processed, focus switches to the abstract tree constructed during parsing. Noosa has two basic forms of tree display shown in Figure 3. The displays show the region of the tree around the while condition on line six of the input. Each node consists of the non-terminal it represents and the (user-specified) name of the abstract production applied at the node. Nodes are numbered for identification.

[1] The intrinsic value distinguishes different instances of general token classes. For example, the intrinsic value of an integer token might be the numeric value of the integer. For identifiers it might be the index of the identifier string in a global string table.

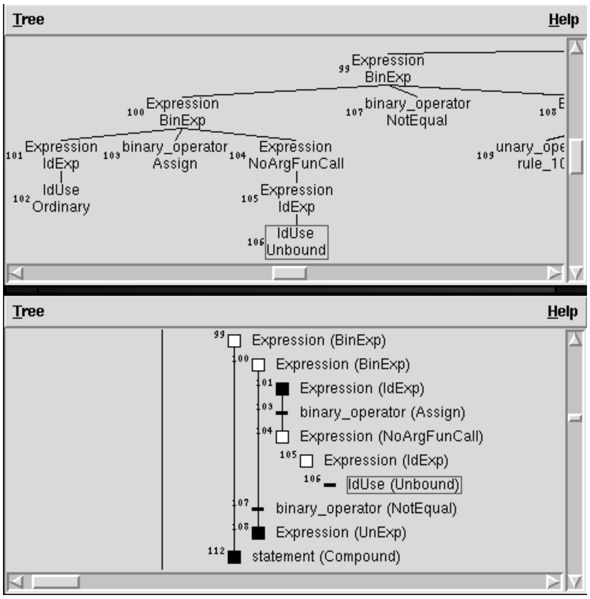


Fig. 3. Abstract tree displays: "complete" tree above and "expandable" tree below. Both windows are focussed on the node representing the `getchar` token in the word counting program.

The upper version of the tree shows every node in a "traditional" tree style. The lower version lays out nodes in an indented style which takes up less space and allows subtrees to be selectively hidden. Siblings are connected by vertical lines. Internal nodes are indicated by squares: white if the node's children are visible, black otherwise. Leaves are indicated by small black rectangles. In both displays the node corresponding to the `getchar` identifier occurrence is highlighted (indicated by the outline around the node).

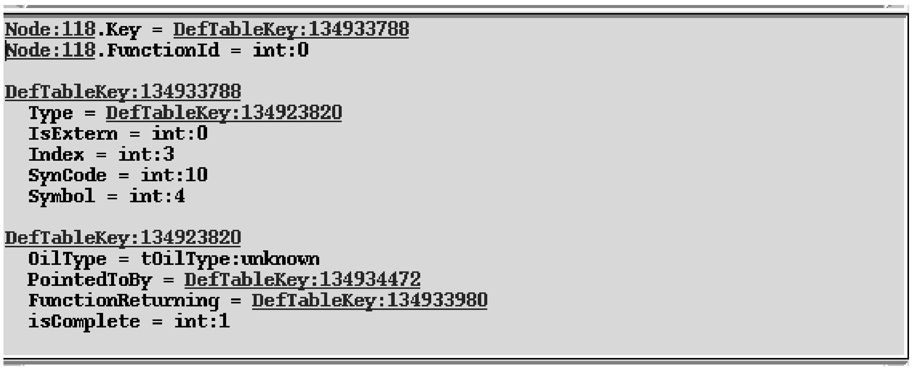
Clicking on a node in a tree display causes the coordinate range of the node and the abstract production applied at the node to be displayed in the transcript. The coordinate range is also highlighted in the input window. Thus it is easy to relate tree nodes to the input text. Browsing in the opposite direction is also supported. Clicking on a location in the input window and invoking the "Node" command in a tree window causes that tree display to focus on the node farthest from the root whose coordinate range contains the selected location. This mechanism is commonly used to quickly focus attention on relevant parts of the tree.

3.4 Attributes

Each node in the abstract tree has a set of attributes associated with it. The user's attribute grammar specifies how the values of these attributes are to be calculated. Eli takes care of traversing the tree and calculating the attributes in an order consistent with the dependences between them.

Using a tree display the user can express an interest in the value of one or more attribute values. The right mouse button on a node brings up a menu of attributes and their types. Each attribute has a setting with options “Show”, “Show, stop” and “Ignore” (default). Selecting “Show” or “Show, stop” causes Noosa to display the value of the attribute in the transcript when it is next calculated. If “Show, stop” is selected, the execution of the compiler will also stop when the attribute is evaluated. The state of the compiler can be examined and a “Continue” command used to resume execution.

Figure 4 shows the transcript window after the user has selected “Show” for all of the attributes of the identifier node representing the occurrence of `nl` in the statement `++nl;`. The `Key` attribute is a definition table key which the compiler is using to represent the variable entity. The `FunctionId` attribute is a Boolean value which is one if and only if this identifier represents a function.



```
Node:118.Key = DefTableKey:134933788
Node:118.FunctionId = int:0

DefTableKey:134933788
  Type = DefTableKey:134923820
  IsExtern = int:0
  Index = int:3
  SynCode = int:10
  Symbol = int:4

DefTableKey:134923820
  OilType = tOilType:unknown
  PointedToBy = DefTableKey:134934472
  FunctionReturning = DefTableKey:134933980
  isComplete = int:1
```

Fig. 4. Attribute value display and browsing. Attribute values are displayed with their node number, name and type. Structured values such as definition table keys can be browsed. Here the first key represents a variable. It has been browsed to reveal the properties of the variable, including the key representing its type. The properties of the type have also been displayed.

In the attribute display some values will be browsable. The node number can be clicked to cause the tree display to focus on that node. This feature is used to switch attention quickly between nodes of interest. Also, the `Key` attribute value is browsable. Since it is a definition table key clicking on it causes the values of its properties to be displayed (middle part of Figure 4). In this case it has four integer properties and a key property that represents the type of the variable. In turn, clicking on that key brings up the properties of the type (bottom part of the figure). Browsing in this way makes it easy to examine the relationships between definition table keys which form the major representational mechanism for information such as the compiler’s type system.

Incidentally, the "unknown" value for the `OilType` property in Figure 4 indicates that Noosa doesn't currently know how to display values of this type. The Eli library module exporting this type can be easily augmented with routines for Noosa to use to obtain a textual representation. It is also straight-forward to make values of new types browsable. This strategy keeps the knowledge of data types with the modules that define them and allows the user to extend the debugging system to support their own types.

3.5 Messages

Often bugs involve the compiler detecting errors when it should not. For example, erroneous regular expressions or context-free grammar productions will usually result in lexical or syntax errors during testing. In Eli each error message is associated with a particular input coordinate. To aid in tracking down this kind of error Noosa displays compiler messages and their coordinates in the transcript. Clicking on the coordinate takes the user directly to the site of the error from which tokens or phrases can be examined as described above. Similarly, a semantic error message can be easily traced back to the relevant tree nodes using the "Node" command.

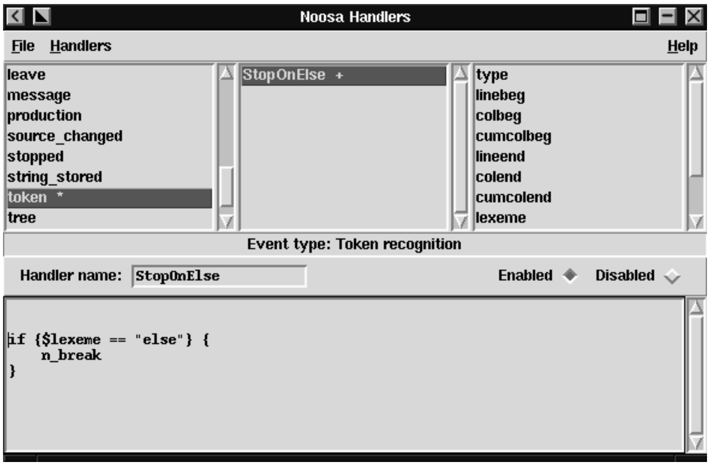
3.6 Events

The capabilities discussed so far have been concerned with data viewing, with one exception. The "Show, stop" command enables execution to be stopped when a particular attribute is evaluated. Noosa also provides a more general event mechanism that allows complex control of the execution. Noosa's use of events is similar to the way they are used in algorithm animation and monitoring systems [1,2,15,17,18]. Abstract event systems have also been used in other debuggers [5,10].

As the compiler executes, it generates a stream of abstract events. Each event represents an action in the execution model. For example, there is an event type to indicate that a token has been scanned. Other events represent the recognition of a concrete production, the construction of the abstract tree, and the evaluation of an attribute. Each event has parameters that distinguish the event instances. For example, a token event had parameters that describe the type of token, source coordinate, lexeme, and so on. Care is taken to ensure that event types and their parameters refer only to user-level concepts.

Noosa uses the event stream to implement the facilities described previously. The stream is also available to the user for more specific control. Event handlers can be specified via the window shown in Figure 5. The left list shows the event types and the right list the parameters of the currently selected event type. In the middle are the names of user-specified handlers.

User-specified event handlers are written in the Tcl language [11]. Tcl is a full-featured, imperative scripting language. Thus an event handler can perform arbitrary actions including querying event parameters, storing data for use by other handlers, and displaying information in the Noosa transcript window.



Handler name: StopOnElse

```
if {$lexeme == "else"} {
    n_break
}
```

Fig. 5. Event handler window. The user is writing a handler for token events so that the compiler will stop each time an `else` token is scanned. Calling the `n_break` procedure achieves this effect.

Space limitations prevent presentation of a complex example, but as a simple illustration, the lower text window of Figure 5 shows a handler for token events which will stop execution each time an `else` keyword is recognised. The handler employs a conventional conditional statement checking the value of the lexeme parameter of the event (available via a Tcl variable). The `n_break` procedure is the published interface to Noosa's execution control mechanism.

4 Debugging Situations

To illustrate the use of Noosa's features, this section briefly considers a variety of typical debugging situations. The situations chosen are intended to be representative of the sorts of bugs that are encountered by Eli users, but are not meant to be exhaustive. In all cases it is assumed that the compiler input is correct; all bugs reside in the user-supplied specifications.

4.1 Lexical Bugs

A lexical error occurs when the lexical analyser is not able to assign a token type to some portion of the input. In the Eli context this means that an error exists in either the concrete grammar literals or the regular expressions describing the non-literal tokens. Eli-generated compilers report lexical errors by pointing to the characters that cannot be scanned.

To diagnose a lexical error with Noosa the user would first browse the error message coordinate to go to the problematic input location. Errors in literals are usually obvious at this point. Otherwise, the "Token" command would be used to examine the token stream around the error location. Usually the erroneous input should belong to one of the tokens surrounding the error and an insufficiently general regular expression is to blame. Less commonly, a non-literal token specification has simply been omitted. Both of these cases are readily diagnosed from the token stream.

4.2 Syntactic Bugs

Syntactic bugs arise in two ways. First, the concrete grammar may not describe the right language. Often focussing in on the problem area will make the reason obvious. Sometimes more information is needed. Eli-generated parsers incorporate error recovery strategies that will possibly insert or delete symbols in an attempt to continue the parse. When this occurs the parser generates a message noting the location from which parsing was continued. Thus the user can easily determine the extent of the problem. In an error situation the "Phrase" command will show the productions that were recognised as part of the error recovery. This information may show that a portion of the input was recognised using a production other than the intended one (leading to the syntax error later). The ability to match production instances to input coordinates is extremely useful in this type of situation.

Alternatively, a syntax error may be reported when the grammar is correct but the scanner is returning the wrong type of token for some portion of the input. For example, a real number may be scanned as three tokens: an integer, a period, and another integer. This may occur because the regular expression for a real number literal erroneously requires an exponent. The "Token" command can be used to diagnose this kind of problem.

4.3 Semantic Bugs

In Eli-generated compilers semantic processing is largely specified using an attribute grammar. Semantic bugs arise if the value of an attribute is incorrectly computed. Thus localising these sorts of errors is a matter of observing the value of attributes. Noosa's attribute value display capabilities are suitable for this purpose.

A semantic bug may be exhibited by an error message or the absence of an error message. For example, a bug in the compiler's handling of type rules may result in an error message for a correctly-typed construct or a missing message for an incorrectly-typed one. In either case the user needs to localise the problem to the portion of the tree that represents the concerned construct. If a message was produced, the coordinate provides access to the appropriate area of the input. If a message is missing, presumably the user knows which part of the input should have produced it. Once the relevant input location is known, the "Node" command can be used to focus attention on the appropriate part of the tree.

Attribute values of tree nodes around the error can be examined to diagnose the problem. This may require multiple executions to focus in on an attribute whose value is incorrect. (Note that attribute storage may be reused, so the value of an attribute computed earlier in the execution might not still exist to be examined at the current execution point.) Usually when this has been done, the user will have narrowed the problem down to a particular computation that is producing the wrong value and the cause of the bug has been located.

Sometimes a semantic bug results from attribute computations being performed in the wrong order. For example, the definition of an applied identifier occurrence might be looked up before relevant defining occurrences have been processed. This kind of error is often due to missing dependencies between attribute computations. It is usually sufficient to display the values of the attributes and to hand verify whether the order is satisfied by the order in which the values are printed. Once this kind of problem has been diagnosed, a fix must be devised in the form of additional dependencies. Noosa does not assist with static dependence analysis because its emphasis is on dynamic information. A similar design decision arises for imperative language debuggers when facilities such as static call graph display are considered.

Complex bugs can be diagnosed with the use of special-purpose event handlers. Since Noosa event handlers are Tcl code they can perform arbitrary computations. In particular, they can store information in global variables for access by other handlers and display information in Noosa's transcript.

For example, Eli's name analysis modules represent scopes as environments holding mappings from identifier names to definition table keys. Suppose that the user wishes to know the tree nodes corresponding to scopes that have at least one identifier defined in them. This can be achieved by three cooperating handlers: 1) one on "an attribute of type Environment has been evaluated" events to record the nodes which have environment attributes, 2) one on "a mapping has been added to an environment" events to increment a per-environment count, and 3) one on the "finalisation" event to print the list of nodes when execution is complete. (For the purposes of this example, it is assumed that only nodes representing scopes have environment attributes.)

Handlers can be saved in files and loaded into other Noosa sessions. Thus high-level debugging functionality can be reused and shared between users.

4.4 Source-Level Bugs

An Eli-generated compiler may have user-supplied code to implement functionality not present in the Eli libraries. For example, abstract data type (ADT) implementations can be provided. Values of user-defined data types can be used in attribute computations.

Problems in user-supplied code can be diagnosed using a source-level debugger. While a debugger can be used independently of Noosa, a cooperative strategy is an advantage because it allows source-level behaviour to be examined in concert with high-level behaviour. For example, attributes of tree nodes may have types which are defined by a user-supplied module. It is useful to be able to examine the values of the attributes in their tree context while debugging the module.

Noosa allows source-level debugging to interoperate with specification-level debugging. In this case the source-level debugger runs as a child process of Noosa and the compiler is a child of that process. The event stream implementation "bypasses" the intermediate process so that Noosa is largely unaffected by the presence of the other debugger. The only complication from the user perspective is a need to be aware of which debugger has control at a given moment. For example, while the compiler is stopped at a source-level breakpoint Noosa has no access to the process, and vice versa. In practice this potential confusion is easily handled.

5 Conclusion

Noosa provides debugging functionality that operates at the level of the user's Eli specifications. Thus the execution of a generated compiler can be understood without knowledge of its internals. The focus is on the main compiler data items: the input and the abstract tree. A text-based browsing facility makes display of other data convenient and extensible. An abstract event stream enables the user to formulate sophisticated queries about the execution. While Noosa is targetted to Eli and its notations, the same general approach should be applicable to other compiler generation systems and specification methods.

Future work will concentrate on extending Noosa to incorporate specification views. This development will aid in the use of Noosa for execution-based specification understanding. Also, it will enable additional debugging capabilities to be triggered from the specifications rather than via the tree. For example, the user might express interest in the value of an attribute in a particular rule context by clicking on an attribute occurrence in that context. Other work is investigating debugging facilities based on the use of program slicing techniques at both the specification-level and on compiler intermediate forms.

Acknowledgements

Much of the functionality and style of Noosa has been influenced by past and current members of the Eli project. In particular, Bill Waite, Bob Gray, Uwe Kastens, Basim Kadhim and Matthias Jung have all contributed valuable suggestions, insights, comments and, in most cases, code to enable Eli and Noosa to interoperate. Thanks to them all. Noosa was implemented by the author with assistance from Tony Vanderlinden and Marianne Brown. The referees also made useful suggestions that improved the presentation of this paper.

References

1. Marc H. Brown. *Algorithm Animation.* The MIT Press, Boston, MA, 1988. 26
2. Marc H. Brown. Zeus: a system for algorithm animation and multi-view editing. Research Report No.75, Digital Equipment Corporation Systems Research Center, February 1992. 26

3. Charles Donnelly and Richard Stallman. *Bison—the YACC-compatible parser generator*. Free Software Foundation, 1.25 edition, November 1995. 20
4. Robert W. Gray, Vincent P. Heuring, Steven P. Levi, Anthony M. Sloane, and William M. Waite. Eli: A complete, flexible compiler construction system. *Commun. ACM*, 35(2):121–131, February 1992. 17
5. David R. Hanson. Event associations in SNOBOL4 for program debugging. *Softw. Pract. Exper.*, 8:115–129, 1978. 26
6. SandStone Technology Inc. Visual parse++. http://www.sand-stone.com. 20
7. S. C. Johnson. YACC – Yet another compiler-compiler. Computer Science Tech. Rep. 32, Bell Telephone Laboratories, Murray Hill, N.J., 1975. 20
8. Mark A. Linton. The evolution of Dbx. In *USENIX Summer Conference*, pages 211–220, June 1990. 17
9. Hanspeter Mössenbock. A generator for production quality compilers. In D. Hammer, editor, *Proceedings of Third International Workshop on Compiler Compilers*, number 477 in Lecture Notes in Computer Science, pages 42–55. Springer-Verlag, October 1990. 17
10. Ronald A. Olsson, Richard H. Crawford, W. Wilson Ho, and Christopher E. Wee. Sequential debugging at a high level of abstraction. *IEEE Software*, 8(3):27–36, May 1991. 26
11. John Ousterhout. *Tcl and the Tk toolkit*. Addison-Wesley, 1994. 26
12. T. J. Parr, H. G. Dietz, and W. E. Cohen. PCCTS reference manual. *SIGPLAN Not.*, 27(2):88–165, February 1992. 17
13. T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Softw. Pract. Exper.*, 25(7):789–810, July 1995. 20
14. Friedrich Wilhelm Schröer. *The GENTLE Compiler Construction System*. R. Oldenburg, 1997. 17
15. Rok Sosic. Design and implementation of Dynascope. *Computing Systems*, 8(2):107–134, Spring 1995. 26
16. Richard M. Stallman and Roland H. Pesch. *Debugging with GDB 4.17—The GNU source-level debugger*. Free Software Foundation, 1998. 17
17. John T. Stasko. A practical animation language for software development. In *International Conference on Computer Languages*, pages 1–10, 1990. 26
18. John T. Stasko. Tango: a framework and system for algorithm animation. *Computer*, 23(9):27–39, September 1990. 26
19. P. D. Terry. *Compilers and compiler generators: an introduction with C++*. International Thomson Computer Press, 1997. 17