Debugging Framework for Attribute Grammars

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Praveen Kambam Sugavanam

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Adviser
Eric Van Wyk

May, 2013

Acknowledgements

I extend my sincere gratitude to my adviser Professor Eric Van Wyk for his time and support in instilling research interests in me and helping me shape up the work that had been done. His invaluable guidance helped me throughout my tenure in graduate school and his perennial enthusiasm always inspired me. My thanks to Mats Heimdahl and Marc Riedel for serving on my committee.

I would also like to acknowledge Ted Kaminski, Kevin Williams, Ming Zhou, Arvind Narayanan and other fellow students at the Minnesota Extensible Language Tools research group at the University of Minnesota Twin Cities for their feedback and help during discussions.

And finally, I would like to thank my parents for all the support they have given me over the years.

Work on this thesis has been partially funded by the NSF Grants 0905581 and 1947961. Opinions, ndings, and conclusions or recommendations expressed in this thesis should be understood as mine, and not reective of the views of the University of Minnesota or of the National Science Foundation.

Dedication

To my grandfathers.

Abstract

Attribute grammars provide a formal means to specify the semantics of context free grammars. In this work, we propose a method to debug attribute grammars by applying algorithmic debugging to the paradigm of attribute grammars. The technique of algorithmic debugging uses the data flow of a program to debug it rather than stepping through its source code thereby making it suitable for declarative platforms like attribute grammars. In a debugging session, dependencies between attributes are obtained and used to construct an execution tree which is then traversed by the debugger based on interactions with the user. The debugger asks the user questions about the validity of a node in the execution tree and about the search space to be explored. Based on the user's response to the questions the debugger identifies the incorrect equation in a production definition. Further, we also propose a means to improve the debugging process by using a guided heuristic based mechanism which helps in reducing the number of questions to the user regarding the search space to be explored next.

Contents

Acknowledgements **i**

Dedication **ii**

Abstract **iii**

List of Figures **vi**

1 Introduction **1**

1.1 Attribute Grammars 1

1.2 Algorithmic Debugging 3

1.3 Algorithmic Debugging for Attribute grammars 5

1.4 Contributions 6

1.5 Outline of the thesis 7

2 Background **8**

2.1 Attribute Grammars 8

2.1.1 Definitions and Notations 8

2.1.2 Higher Order Attributes 13

2.2 Algorithmic Debugging 16

3 Related Work **20**

3.1 Most Relevant Related Work 20

4 Algorithmic Debugging of Attribute Grammars **23**

4.1 Attribute Dependencies 23

4.1.1 Dependency Graph 23
4.1.2 Reachability 25
4.2 Building the Execution Tree 25
4.3 Traversing the Execution Tree 27

5 Further Improvements **31**
5.1 Two Issues with Basic Approach 31
5.2 Guided Heuristics Based Debugging 32

6 Implementation **37**
6.1 Design 37
6.2 Silver Debugger 40
6.2.1 Guided Heuristics approach 45
6.3 Generic Debugger 48

7 Future work and Conclusion **50**
7.1 Future Work 50
7.1.1 Dynamic Dependencies 50
7.1.2 User Interface 52
7.1.3 Generalization for other Attribute Grammar systems 52
7.2 Conclusion 52

References **54**

List of Figures

1.1 A sample imperative program. 2
1.2 An abstract syntax tree. 3
2.1 A sample imperative program. 11
2.2 The abstract syntax of the sample imperative language. 12
2.3 An abstract syntax tree. 13
2.4 The concrete syntax of the sample imperative language. 15
2.5 A sample execution tree of algorithmic debugging. 17
2.6 A sample interaction of algorithmic debugging. 18
4.1 The *assign* production. 24
4.2 A production dependency graph for the production defined in Figure 4.1 24
4.3 An execution tree for the program in Figure 2.1. 27
4.4 An interaction of algorithmic debugging. 29
5.1 An interaction of guided heuristic debugging. 35
6.1 The component diagram for the debugging framework. 38
6.2 The sequence diagram for a debugging session. 39
6.3 Pseudo code for the debugging algorithm. 40
6.4 The component diagram for Silver debugger. 41
6.5 A sample imperative program. 43
6.6 The execution tree. 44
6.7 An algorithmic debugging interaction for Silver. 45
6.8 An interaction of algorithmic debugging. 47
7.1 The *assign* production. 51

Chapter 1

Introduction

1.1 Attribute Grammars

Attribute Grammars were first introduced by Knuth[1] over fourty years ago as a formalism for describing the semantics of a context free grammar. They have since played an important role in the field of compiler construction [2],[3] and language specification and analysis tools. Attribute grammars have had a significant impact in the development of frameworks for extensible languages and in defining new syntactic and semantic extensions for languages like C and Java to help develop user friendly domain-specific languages. For example, JastAdd[4], used for developing extensible compilers for Java and Modelica is based on attribute grammars (AGs) with reference attributes. Silver[5], is an attribute grammar system that has been used to build composable extensions for languages such as Java[6] and Promela[7].

An attribute grammar definition is used to typically generate the parser and the tree based representation of the program that the parser outputs. This tree is more commonly known as the abstract syntax tree and is comprised of nodes that are decorated with attributes as defined by the productions in the grammar. Figure 1.2 shows the abstract syntax tree of a simple imperative program defined in Figure 1.1 . The node *root* shown in the figure is decorated with two attributes namely *env* and *errors*. Similarly the node *consDecls* is decorated with the *defs* attribute. The figure also shows the two types of attributes on a node, namely *synthesized* and *inherited*. Synthesized attributes like *errors* are computed bottom-up in a syntax tree and the information is passed up

towards the root. On the other hand, inherited attributes like *env* are computed from the top of the tree and their values are passed down to the children of the nodes on which they occur.

```
1          Integer x;
2          begin
3            read(x);
4            x = x+2;
5            write(x);
6        end
```

Figure 1.1: A sample imperative program.

Despite all its uses, one of the major setbacks in the development of attribute grammars has been the lack of a reliable debugging technique. This stems from one of the important characteristics of attribute grammars, which is that they are declarative in nature. The developer of an attribute grammar only specifies the equations that are used to compute the attribute values but does not indicate the order in which this computation would occur. The order of computation is often scattered across the AG definition due the dependencies that are present in the computation of attributes. For example, to compute the value of *errors* on the *root* node of the syntax tree shown in Figure 1.2, one would have to compute the attribute *errors* on the *seq* node first. This recursive dependencies on other attributes leads to an order where the demanded attribute, which in this case is *errors* on the *root* node is the last computed value after all its dependencies. This shows that the execution order of attribute definitions is not intuitive and hence debugging with a traditional step-by-step debugger would be practically impossible. The debugging process could be made easier if the debugger followed the actual data flow in the execution tree as compared to debugging based on the code.

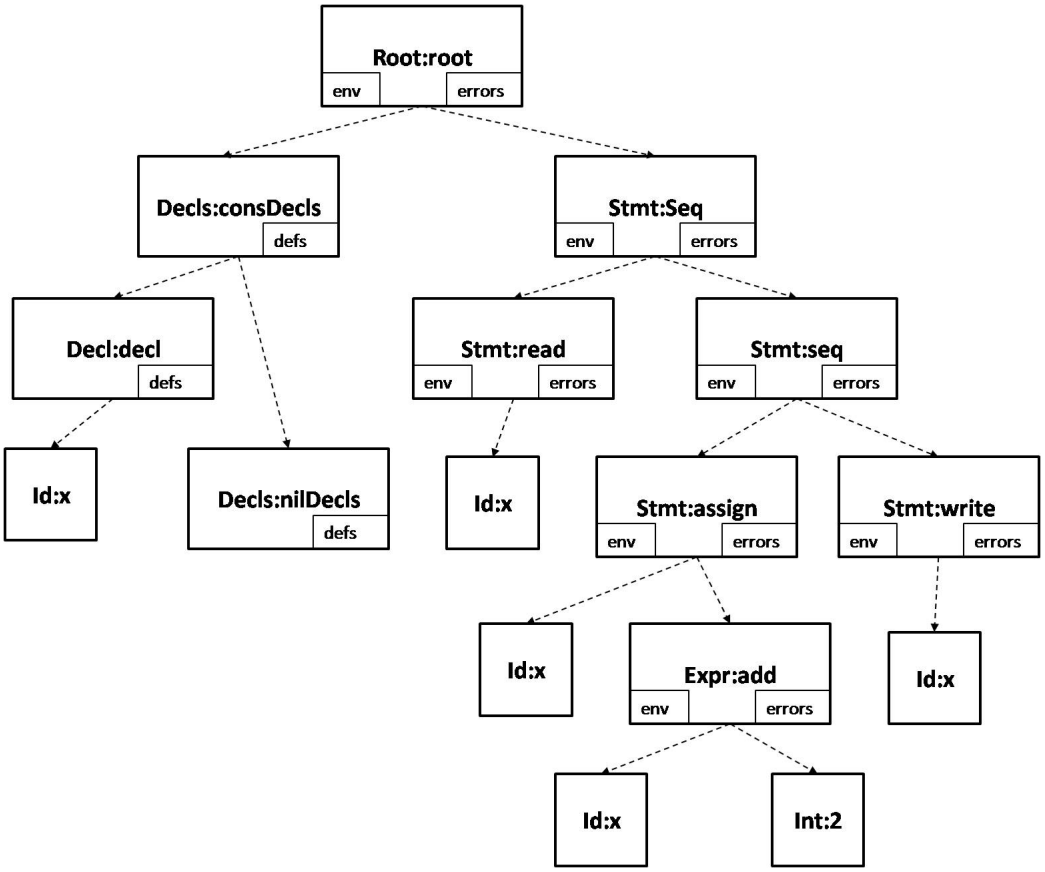

Figure 1.2: An abstract syntax tree.

1.2 Algorithmic Debugging

For any given language, in addition to its features, one of the key characteristics that would make it more user-friendly and adaptable is the presence of a stable and reliable debugger. Traditionally, a debugger is defined as a tool that is used to detect errors in a program by performing a step by step execution of the code and examining the state of the variables in the process. For imperative languages such as C and Java, debuggers like GDB and JDB have used the model of a traditional debugger and made life easier for programmers. Regardless of the fact that the traditional debugging algorithm has been extensively adopted, its use has been restricted to imperative languages. A debugger stepping through the code one statement at a time, cannot be used as efficiently for

declarative languages as they are for imperative ones, the primary reason for this being that the order of computation is not explicitly mentioned in a declarative program. The technique of algorithmic debugging helps overcome this obstracle.

Algorithmic Debugging is a semi-automatic debugging technique that helps the user locate an incorrect computation by traversing through the execution tree of a program. Introduced by Shapiro[8] over thirty years ago, the technique was first used to debug Prolog programs and specifically other side-effect free languages. It has since been extended to other more interesting paradigms like lazy functional languages[9] and imperative languages [10]. Unlike traditional debugging, algorithmic debugging is a post-execution process. The debugger incrementally gains knowledge of the program being examined by asking users about the validity of computations.

Algorithmic debugging relies on the user having an intended interpretation of the program. It is intrinsically a two phase process. In the first phase, an execution tree is built where each node in the tree corresponds to a computation in the program. An execution tree is a tree based structure representing the data flow of the program being debugged. In a typical setting, a node N in the execution tree would refer to a procedure call that is made in the program. The children N in the tree would be the procedure calls that are made from the instance of the procedure referred to in N. In the second phase, the debugger traverses the tree and interacts with the user to gain knowledge about the program being debugged. These interactions are of two types of questions asked to the user namely *correctness* and *search* question. In the *correctness* question, the user is queried about the correctness of the nodes based on their specific input and output values. In the *search* question, he is asked to make a decision regarding the next node to be examined in the execution tree. When all the children of an incorrect node in the execution tree are labelled correct, the debugger can identify the bug at the node corresponding to the incorrect computation. This process of traversing through the execution tree makes algorithmic debugging very suitable to declarative languages where the execution order is not explicitly available by inspecting the program.

1.3 Algorithmic Debugging for Attribute grammars

Considering that the traditional debugging process is suitable only for imperative languages and algorithmic debugging being the preferred method for declarative languages, we propose an algorithmic debugging based debugger for attribute grammars. As described earlier, algorithmic debugging is a two phase process. In the first phase we construct the execution tree for the program being debugged, which in this case is a dependency graph of attribute instances. An attribute instance is defined as an occurance of an attribute at a given node of the syntax tree. Figure 1.2 shows the *root* node of the tree comprising two attribute instances. One belonging to the occurance of *errors* and the other to that of *env*.

In the second phase of the debugging process, we traverse the dependency graph of the attribute instances and at every stage query the user on two things. The first is a *correctness* question where the user is required to answer whether an attribute instance has a correct value, according to the information provided. The second is a *search* question where the user is presented a list of possible nodes that the debugger can examine next and is required to pick a node among them. If the user answers the *correctness* question for a node with a *yes*, that node and its children are assumed to have correct computations. If the user answers with a *no*, based on the response to the *search* question, the debugger would go down to the subtree of that node and repeat the process. When the user has guided the debugger to a state where an incorrect node has only correct (or no) children, we locate the error in the incorrect node.

One unfortunate side effect of algorithmic debugging has always been the number of questions that the user is asked. In the case of debugging attribute grammars, this translates to querying the user about the correctness of nodes (*correctness* question) and asking them about the next node in the list of possible nodes to be examined (*search* question) at every stage. The questions, despite being partially connected to each other could make the debugging process more tedious. Previous work to mitigate this problem for algorithmic debugging in general discusses the ideas of program slicing [11], partition testing [10], correct subcomputations [12] to name a few.

In this work, we propose a guided heuristic based approach to reduce the number of user-debugger interactions by eliminating the *search* question during the process

of debugging. At the start of a debugging session the debugger, to provide a better experience for the user, requests for heuristic information regarding the grammar that is being examined. The heuristic data provided to the debugger is typically a ranking of the possible values of a property of the grammar like attributes, productions or non-terminal symbols. This information is then used during the analysis of the execution tree at every stage to restructure the dependency graph of the program being debugged. By restructuring the dependency graph, the debugger makes an intelligent decision regarding the next node to be examined at each level of the execution tree. This decision made by the debugger directly corresponds to the *search* question that the user would answer regarding the next node to be picked from the list of potential erroneous nodes. For example, at the start of a debugging session, the user could choose to provide a priority ranking for the attributes in the grammar. At every stage of debugging, the debugger would then rearrange the execution tree based on the ranking of the attributes and begin exploring the nodes according to this new order, as opposed to requesting the user to manually pick a node from a given list of nodes. In addition to reducing the number of questions answered by the user, such an approach reduces the number of decisions that the user makes at every step of the debugging process.

1.4 Contributions

In this thesis, we aim to address the problem of the lack of a reliable debugging technique for attribute grammars. In our first contribution, we show how the algorithmic debugging process can be applied to the attribute grammar paradigm to present a stable method to locate errors in attribute definitions. Our second contribution is focussed on trying to make the debugger more user friendly by reducing the tediousness in the debugging process. We propose to reduce the number of interactions between the user and the debugger with a heuristic based mechanism to help the debugger intelligently decide on the path to be examined at every stage.

1.5 Outline of the thesis

The thesis is outlined as follows. In the second chapter we introduce attribute grammars (AGs) and define the basic notations used in AGs. We then describe the technique of algorithmic debugging and its advantages as a debugging tool for declarative languages. Chapter 3 discusses the work done by various groups in the field of algorithmic debugging. The primary contributions of the thesis then follow: In Chapter 4 we first describe the concept of depencency graphs in attribute grammars then, we discuss the application of the algorithmic debugging technique to the attribute grammar framework. We also identify some problems with the basic approach, which we address in Chapter 5 with a guided heuristic based approach to debugging. In chapter 6 we provide implementation details of the debugger developed for the Silver attribute grammar system. We conclude with a few suggestions on the directions that this work can take in the future.

Chapter 2

Background

In this chapter we present the background material necessary for the descriptions in Chapters 4 and 5. We first discuss some formal definitions of attribute grammars with examples to describe the various properties of an attribute grammar. We also examine the different types of syntax trees and the attributes that occur on the nodes of these trees. We then move on to define algorithmic debugging and illustrate the technique with a simple example.

2.1 Attribute Grammars

Attribute Grammars were first introduced by Knuth [1] as a means to express the semantics of languages whose syntax is defined by a context free grammar. They are a declarative formalism aimed at providing semantic meaning to the syntactic statements of a language.

2.1.1 Definitions and Notations

Formally, an attribute grammar (AG) is defined as a triple $AG = \langle G, A, D \rangle$. $G = (V, N, S, P)$ is a context-free grammar, where V is the finite set of all terminals and non-terminals in the grammar; $N \subseteq V$ is the set of all non-terminals in the grammar; S is the start symbol, which appears on the right side of no production and P is the set of production rules that define the language. A is a finite set of attributes, partitioned into sets $A_{nt}(X)$ and $A_{loc}(p)$ for each $X \in N$ and $p \in P$, that define the semantic meaning of

the non-terminals and productions respectively. The set $A_{nt}(X)$ is further partitioned into two disjoint sets $A_s(X)$, the set of all synthesized attributes and $A_i(X)$, the set of all inherited attributes. $D = (T, E)$ is the semantic domain of AG where T is a set of types and E is a finite set of semantic equations.

The sample imperative program written in Figure 1.1 is based on the context free grammar whose abstract syntax definition is shown in Figure 2.2. This description is written in Silver[5], our attribute grammar system. The grammar begins with the definition of the non-terminals *Root*, *Decls*, *Decl*, *Stmt* and *Expr*. The non-terminal *Root* represents the root of the tree to be generated from this grammar. *Decls* and *Decl* refer to multiple and single variable declarations in the program. *Stmt* and *Expr* as expected refer to statement(s) and expression. The figure also shows the production definitions that build the nodes on the syntax tree. The production *root* as shown in the grammar description constructs a node of type *Root*, which is the left hand side of the production with two nodes. One, a list of declarations labelled by the *Decls* non-terminal and the other, a list of statments in the program described in the node labelled by the *Stmt* nonterminal on the right hand side. The goal of this attribute grammar is to list all the errors that occur in a program written in the basic imperative language.

The two main classes of attributes defined on non-terminals are synthesized and inherited attributes. Synthesized attributes are evaluated from the bottom up in the syntax tree. The value of a synthesized attribute of a non-terminal in a production definition is computed from the values of the attributes that occur on the descendants of the non-terminal symbol. Inherited attributes on the other hand are evaluated from the top down. Such attributes pass and propogate information down the tree towards the leaves. Figure 2.2 shows the attribute definitions and the non-terminal symbols on which they occur. The *errors* attribute that is defined on *Root*, *Stmt* and *Expr* is a synthesized attribute that passes information up the syntax tree. At any given node, the *errors* attribute records whether it contains any accesses to undefined variables in the program. To compute the validity of a variable, it would need information regarding all the variables that have been declared in the program. The *env* is an inherited attribute that is a list of strings that contains the names of all the variables that have been declared and is passed down the tree to the individual statement nodes. Based on the formal definition of attribute grammars mentioned above, $A_s(Root)$ contains the

attributes *errors* and *env*, which can be seen at the definition of the attributes in Figure 2.2, that states that the attributes *errors* and *env occurson* the non-terminal *Root*.

Attribute instances are individual occurances of an attribute on a non-terminal symbol at a given node in the syntax tree of a program, based on the computation rules defined in the corresponding production. They decorate the syntax tree with their evaluations. Every node in a syntax tree contains an attribute instance for each attribute defined on that symbol. For example, in the abstract syntax tree shown in Figure 2.3, the *Assign* node consists of two attribute instances. One belonging to the occurance of th synthesized attribute *errors* on the node and the other to the occurance of the inherited attribute *env* on it.

The declarative nature of attribute grammars arises from the fact that the computation equations specified by the programmer does not specifiy any order for their evaluation. For example, in the *root* production of the grammar defined in Figure 2.2, the programmer specifies two equations for computation of attributes. These equations do not have a specified order in which they are executed. Their evaluations are based on the order that they are demanded. In the attribute grammar defined in Figure 2.2, to obtain the errors in the program that is being analyzed, the *errors* attribute at the *Root* node is demanded. As we can see from the evaluation equation specified within the production *root* in the figure, the *errors* attribute on the root is dependent on the value of *errors* on the the *Stmt* node. The *Stmt* in the tree shown in Figure 2.3 refers to a node that is constructed by the *seq* production. The *Seq* production defines the *errors* attribute on the *Stmt* non-terminal being constructed as a concatenation of the values of *errors* on its descendants. Hence to compute the value of *errors* on *seq* we would have to first compute the values of *errors* on the two statements *read* and *assign*. The *errors* on these statement nodes consequently depend on the attribute *env* occuring on to them. Since *env* is an inherited attribute, its value has to be passed down from the ancestor to the current statement node being evaluated. This information dependency cascades to the occurance of attribute *env* on the *root* node. At the *root* production, the value of the *env* attribute on the *Stmt* nonterminal is dependent on the synthesized attribute *defs*. It is these complex dependency structures that determine the evaluation order of the attributes that decorate the syntax trees.

In a demand driven attribute grammar system like Silver[5], we observe that the

attribute that is demanded first, is the last value that is computed during the execution. This is due to the various dependencies both direct and indirect, which have to be evaluated first, before the demanded value can be computed. A contrasting approach to this is to determine statically, the dependencies for an attribute and evaluate them in the order determined. Ordered attributes[13] perform such an evaluation that still maintains the dependency information between the attributes.

```
1 Integer x;
2 begin
3 read(x);
4 x = y+2;
5 end
```

Figure 2.1: A sample imperative program.

```
1  nonterminal Root, Decls, Decl, Stmt, Expr;
2  synthesized attribute errors :: String occurs on Root, Stmt, Expr;
3  inherited attribute env :: [ String ] occurs on Root, Stmt, Expr;
4  synthesized attribute defs :: [ String ] occurs on Decls, Decl ;
5  annotation location occcurs on Root, Decls, Decl, Stmt, Expr;
6
7  abstract production root
8  r::Root ::= ds::Decls s::Stmt
9  {
10   r.errors = s.errors;
11   s.env = ds.defs ;
12 }
13 abstract production consDecls
14 ds::Decls ::= d::Decl rest::Decls
15 {  ds.defs = d.defs ++ rest.defs; }
16 abstract production nilDecls
17 ds::Decls ::=
18 {  ds.defs = []; }
19 abstract production decl
20 d::Decl ::= id::Id
21 {  d.defs = [id.lexeme]; }
22 abstract production seq
23 s::Stmt ::= s1::Stmt s2::Stmt
24 {
25   s.errors = s1.errors ++ s2.errors;
26   s1.env = s.env;  s2.env = s.env;
27 }
28 abstract production assign
29 s::Stmt ::= id::Id e::Expr
30 {
31   e.env = s.env;
32   s.errors = if (existsInList(id.lexeme,s.env) == false)
33              then id.lexeme ++ " has not been defined at location : "++toString(s.location.
                     filename)++"-" ++toString(s.location.line)++" to "++toString(s.location.
                     endLine) ++".\n" ++ e.errors
34              else "";
35 }
36 abstract production skip
37 s::Stmt ::=
38 {  s.errors = ""; }
39 abstract production read
40 s::Stmt ::= id::Id
41 {
42   s.errors = if (checkInList(id.lexeme,s.env) == false)
43              then id.lexeme ++ " has not been defined at location : "++toString(s.location.
                     filename)++"-" ++toString(s.location.line)++" to "++toString(s.location.
                     endLine) ++".\n"
44              else "";
45 }
```

Figure 2.2: The abstract syntax of the sample imperative language.

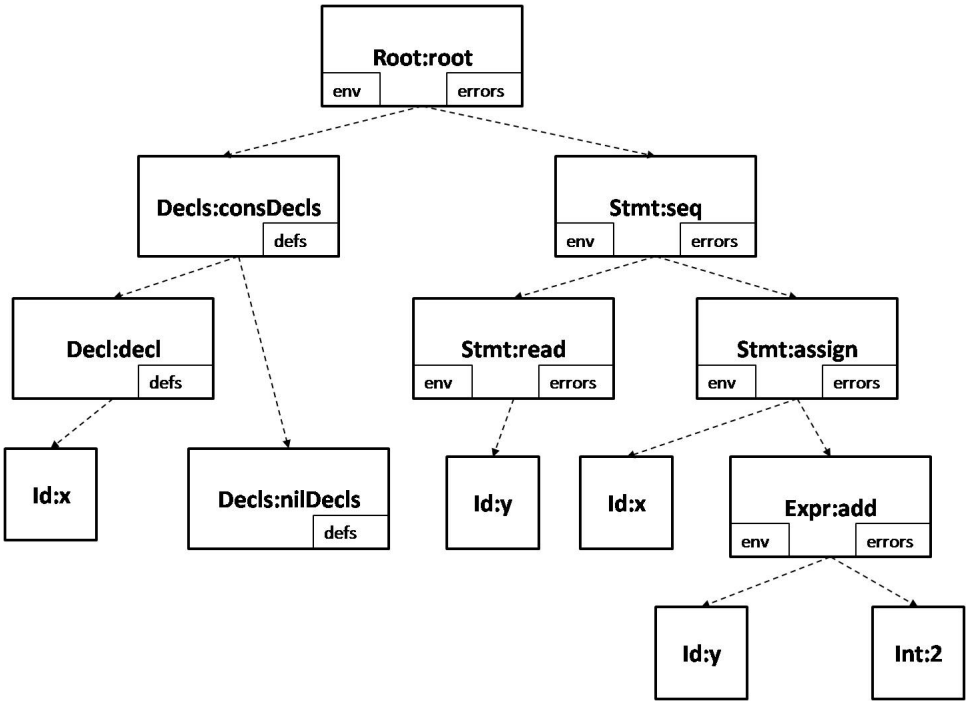

Figure 2.3: An abstract syntax tree.

2.1.2 Higher Order Attributes

Traditionally the attributes on non-terminals were either of primitive type (like string, int etc) or simple data structures containing primitive types (like lists of integers). In 1989 a new type of attributes was introduced by Vogt et al., [14] called higher-order attributes. Higher order attributes are those that contain a tree structure as their value. They are typically of type N where N is a non-terminal symbol defined in the attribute grammar definition. During their computation, these attributes are not supplied any inherited attributes. This results in the attribute having a tree structure with undecorated nodes.

One of the important uses of higher order attributes is in the construction of the abstract syntax tree of a grammar from the concrete syntax. The concrete syntax of a grammar is a context free grammar (CFG) definition which includes all the syntactic details of the language and is used to build the parser. The parser generates a tree structure that is simpler than the CST and is devoid of the syntactic intricacies of

the language. This tree structure is constructed by decorating the concrete syntax tree with higher order attributes that are of types non-terminal symbols defined in a separate CFG definition called the abstract syntax. Figure 2.4 shows a concrete syntax definition for a basic imperative language. This description is written in Silver[5], our attribute grammar system. As discussed above, the grammar contains non-terminal definitions first, followed by attribute definitions. The grammar uses an abbreviated notation where multiple productions constructing the same non-terminal are defined together. The figure shows attribute ast_{Root} of type *Root* defined on the non-terminal R. This is a higher order attribute of type *Root* which is defined in the abstract syntax shown in Figure 2.2. The figures of the grammar definitions omit a specification which requests the parser for the concrete syntax tree first and then demands the AST attribute off that tree, which in this case is ast_{Root} on R.

Apart from higher order attributes, there have been many significant improvements to attribute grammars since its inception like reference attributes[15], forwarding [16] and circular attributes [17] to name a few. In this thesis, however, we will be focussing primarily on primitive and higher order attributes since the extensions to attribute grammars mentioned would not impact the design of the debugger that has been developed.

```
1  grammar sandbox:simpler:host:concretesyntax ;
2  imports sandbox:simpler:host:abstractsyntax as abs ;
3  imports sandbox:simpler:host:terminals ;
4
5  nonterminal R, DS, D, ST, S, E;
6  synthesized attribute ast_Root:: abs:Root occurs on R;
7  synthesized attribute ast_Decls:: abs:Decls occurs on DS;
8  synthesized attribute ast_Decl:: abs:Decl occurs on D;
9  synthesized attribute ast_Stmt:: abs:Stmt occurs on ST, T;
10 synthesized attribute ast_Expr:: abs:Expr occurs on E;
11 annotation location occurs on R, DS, D, ST, S, E;
12
13
14 concrete production root
15 r::R ::= ds::DS 'begin' ss::ST 'end'
16 {
17   r.ast_Root = abs:root( ds.ast, ss.ast,location=r.location ) ;
18 }
19
20 concrete productions ds::DS
21   | d::D ';' rest::DS
22     { ds.ast_Decls = abs:consDecls ( d.ast, rest.ast,location=ds.location  ) ; }
23   | d::D
24     { ds.ast_Decls = abs:consDecls ( d.ast, abs:nilDecls (location=ds.location ),location=ds.
         location ) ; }
25   | --empty
26     { ds.ast_Decls = abs:nilDecls (location=ds.location ) ; }
27
28 concrete productions d::D
29   | 'Integer' id::Id   -- assuming all are integers at this point
30     { d.ast_Decl = abs:decl ( id ,location=d.location) ; }
31
32 concrete productions ss::ST
33   | s::S ';' rest::ST
34     { ss.ast_Stmt = abs:seq ( s.ast, rest.ast,location=ss.location ) ; }
35   | s::S
36     { ss.ast_Stmt = s.ast ; }
37
38 concrete productions s::S
39   | id::Id ':=' e::E
40     { s.ast_Stmt = abs:assign ( id, e.ast,location=s.location ) ; }
41   | 'read' '(' id::Id ')'
42     { s.ast_Stmt = abs:read ( id,location=s.location ) ; }
43   | 'begin' ss::ST 'end'
44     { s.ast_Stmt = ss.ast ; }
45
46 concrete productions e::E
47   | id::Id
48     { e.ast_Expr = abs:varName ( id,location=e.location ) ; }
49   | n::IntegerLiteral
50     { e.ast_Expr = abs:const ( n,location=e.location ) ; }
51   | l::E '+' r::E
52     { e.ast_Expr = abs:plus ( l.ast, r.ast,location=e.location ) ; }
```

Figure 2.4: The concrete syntax of the sample imperative language.

2.2 Algorithmic Debugging

Algorithmic debugging was first introduced by Shapiro [8] as a debugging mechanism for logic programs. It was developed as a semi-automatic method to locate bugs in Prolog programs. The idea of algorithmic debugging was brought about mainly due to two reasons. First, to provide an improvement on the single-stepping trace technique by showing the user only the parts of the program that were relevant to the computation in question. And second, to provide a platform for a more automated method of locating a bug. The technique relied on the user having an intended interpretation M of the program, where M is a set of triples $\langle p, x, y \rangle$ where p is a procedure and y is the output of the procedure for the input defined by x. For example, for a procedure 'square', if the intended output for an input of 4 is 16, it would result in the triple $\langle square, 4, 16 \rangle \in M$. In most practical situations, the interpretation M of the program is the programmer's expected behavior of the code. Shapiro's model was based on one main assertion: A procedure P will be called correct with respect to an interpretation M if and only if all procedure calls made from P return a correct output with respect to M. In other words, if a procedure P returns a result that is not contained in the interpretation M, either one of the procedure calls made by P is incorrect or a computation in P is incorrect. Based on this assertion, Shapiro developed a debugger for Prolog and Prolog-like languages with certain restrictions. His model required that the programs being debugged be loop-free and side effect free.

The technique of algorithmic debugging basically involves two main phases. In the first phase, an execution tree of the program being debugged, is built. The execution tree can be defined as a tree representation of the data flow in the program. In the second phase, the debugger traverses the execution tree and gains knowledge of the program by querying the user about the nodes. The interaction with the user is mainly of two kinds.

1. Questions regarding the correctness of a node (*correctness* question).

2. Questions regarding the next node to be examined (*search* question).

The correctness question is typically of the form $\langle procedure, input, output \rangle$ where given a procedure and it's input in the execution, the user is asked whether the output

returned by it is correct. The user answers the questions with a *yes* or a *no*. This knowledge about a procedure is then used on the assertion discussed earlier which states that if a procedure is known to have an incorrect output and all the procedure calls from that procedure are said to have correct outputs, the error is located at the computation of the incorrect procedure. For example, if p is a procedure whose output depends on one of the procedure calls q_1, q_2,...,q_n made from p, if the user marks the output of p as incorrect for its input and marks the calls to q_i based on their inputs and corresponding outputs as correct for all i, the error is located in the computation of p. If one of the calls to q_i is marked incorrect, the debugger repeats the process for that particular procedure instance. As a corollary, if a procedure instance is known to be incorrect and does not have any other procedure calls from it, the error is identified in the computation of the incorrect procedure. The *search* question regarding the path to be followed is, based on the state of the debugger and the implementation, either answered by the user or automatically by the debugger.

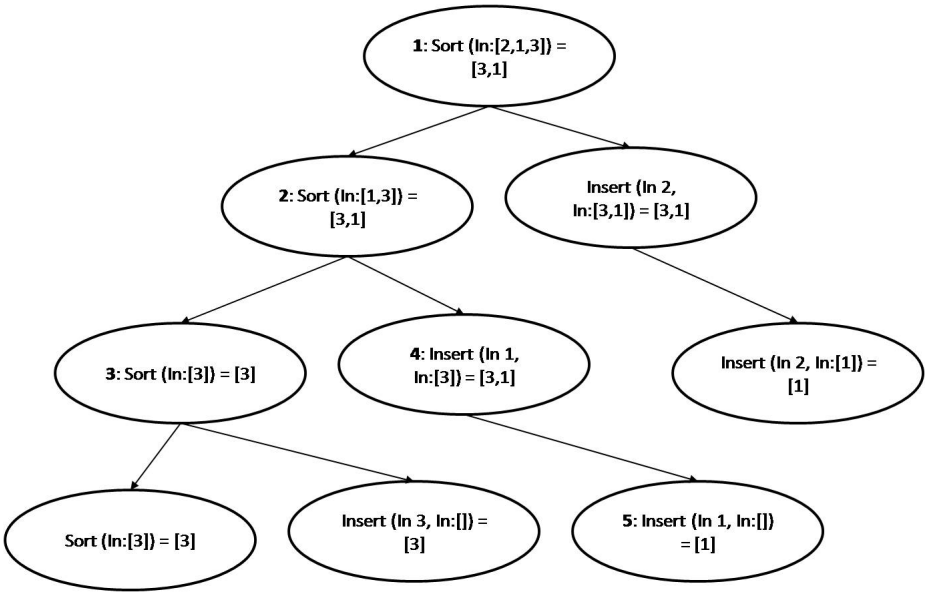

Figure 2.5: A sample execution tree of algorithmic debugging.

```
1: sort(in: [2,1,3], out: sort=[3,1])?
> no
2: sort(in: [1,3], out: sort=[3,1])?
> no
3: sort(in: [3], out: sort=[3])?
> yes
4: Insert(in: element=1, in: [3], out:insert= [3, 1])?
> no
5: insert(in: element=1, in: [], out: insert=[1])?
> yes
An error has been located at the procedure 'insert'
```

Figure 2.6: A sample interaction of algorithmic debugging.

Figures 2.5 and 2.6 depict a sample run of an algorithmic debugger as described by Fritzon et al [10]. The sample program being debugged is an insertion sort algorithm with a bug introduced in the insert procedure. Figure 2.5 shows the execution tree of a run of the program. The root node of the tree shows the input to the sort algorithm to be a list [2,1,3] generating an output list [3,1]. The output is incorrect and does not correspond to the expected output of the sort procedure, which is the list [1,2,3]. I.e., $\langle sort, [2,1,3], [3,1] \rangle \notin M$, the intended interpretation of the program. The execution tree also shows the procedures that are called within each procedure. For example, the sort procedure call at the root node of the tree first recursively calls itself with the list [1,3] as the argument. It then calls the insert procedure with the result of the sort procedure and the first element as arguments. The result of the insert procedure is the final result of the sort procedure at the root. Figure 2.6 demonstrates the interaction between the debugger and the user for a debugging session corresponding to the execution tree in Figure 2.5. The first question asked by the debugger corresponds to the root node of the execution tree. It queries the user whether the output of the sort procedure whose input is the list [2,1,3] is [3,1]. According to the user's interpretation of the code, the

triple $\langle sort, [2,1,3], [1,2,3] \rangle$ is correct and hence any other output for the same input to the procedure *sort* would be considered incorrect. Thus the user responds with a *no*. The debugger then traverses the execution tree asking the user similar questions regarding the validity of the nodes. Once the debugger locates the bug, it immediately reports it with the procedure at which the bug was localized. In the example described above, the *search* question has been automatically answered by the debugger based on the order of the nodes that have been called at each level.

Chapter 3

Related Work

Algorithmic debugging, since its introduction by Shapiro[8], has been a constantly evloving field. The first improvements to the technique was made by Shahmeri and Fritzson [18] when the original idea was extended to suit imperative languages. Nilsson et al[9] further extended the scope of the technique to lazy functional languages. Silva [19] describes a variety of paradigms that use the algorithmic debugging framework today.

In this chapter we look at the various significant implementations and extensions to the idea of algorithmic debugging that has had an effect on our research. The work discussed here is primarily focussed on improving the algorithmic technique by reducing the number of interactions with the user.

3.1 Most Relevant Related Work

One of the first improvements to the basic approach to algorithmic debugging was proposed by Peter Fritzson et al[10]. They developed a system where imperative languages[18] like pascal could be debugged using the algorithmic debugging technique. Another significant contribution made was to improve on the approach to ease the burden of questions on the user. Their goal with this improvement was to reduce the number of difficult questions asked to the user by using previous results of testing done on procedures. For example, in the case of a sort procedure, if a user is asked about the output for an input list consisting of fifty elements, it would be practically impossible for the user to answer. To overcome this they proposed a scheme where if a procedure

has been extensively tested prior to a debugging session, those test results could be used during debugging to help eliminate the need for the user to answer difficult questions.

Another approach to significantly reduce the number of interactions with the user was proposed by Drabent et al[20]. In their approach, the system provides for the user to present an intended formal specification for the program. The debugger would then use this specification to answer the questions intelligently by itself. In the case of it being unable to answer a question, the query would be presented to the user who would give the required information to the debugger.

Hat-Delta[12] an algorithmic debugger for Haskell, combines the ideas of correct sub-computations and program slicing to locate the bug in a program. The method used for to slice a program is based on the assumption that if a part of the source code has been executed many times in a correct execution of the program, it is more likely to be correct than code that has never been executed correctly. This heuristic of the number of times a procedure is executed with correct results is used to eliminate parts of the search tree to eventually reduce the number of questions asked to the user.

Josep Silva and Olaf Chitil [11] present a method to combine algorithmic debugging and program slicing. In their approach they use the augmented redex trail, a compact and detailed tree based representation of the computations in a program, as the execution tree for debugging. During the debugging process, their system aims at reducing the number of questions asked to the user by gaining as much information from the user as possible regarding a potential error. The debugger gives the user an option to isolate an erroneous part of a question asked to the user and slices the execution tree based on the information given. For example, if the user is asked a query like "Is $foo(a, b, c) = d$?". The user can choose to answer that given the arguments, the procedure returns the correct value. But with his knowledge of the program, he could detect that the second argument for the function call to foo should not be b. In this case, the user would mark the second argument as incorrect and the debugger will proceed to explore the subtree rooted at the computation of the second argument, thereby eliminating questions to the user regarding correct computations of the program. A similar technique has been developed by MacLarty et al.[21] in the development of a debugger for the functional logic language Mercury.

The work mentioned above are very similar to our work in that they have improved

upon the basic approach of algorithmic debugging to enchance the user's experience by reducing the number of questions that are asked. But in contrast to our approach described in Chapter 5, all these improvements primarily focus on the *correctness* question discussed in Section 2.2. While some of these approaches focus on reducing the difficulty in answering the question, others address the issue of the large search space leading to more questions regarding correctness, to the user. Our approach while still trying to provide a better experience to the user, focusses on eliminating the *search* question. Section 5.2 describes the heuristic based approach that deals with this issue.

Noosa[22] is a debugging tool for compilers generated by Eli[23] attribute grammar system which follows the traditional debugging mechanism. It provides a means for the users to diagnose a problem in Eli-generated programs by presenting the syntax tree to the user. The user interface in the debugger provides three options to the user at an attribute evaluation namely show, show-stop and ignore. The show option displays the value of the attribute when it has been evaluated and proceeds with the computation. The ignore option, which is the default proceeds with the computation without any special change in display. The show-stop option shows the value of an attribute when it has been evaluated and stops the execution at that state. Noosa uses an event driven mechanism to support the show-stop option of the debugger. Based on these, the user is able to see attribute values at different nodes in the syntax tree which then helps him locate the bug in the grammar definition.

Chapter 4

Algorithmic Debugging of Attribute Grammars

In this chapter we first analyze the dependency graph structures of the attribute definitions in a grammar and discuss production dependency graphs in detail. We then describe how the production dependency graph can be used to build the execution tree for an algorithmic debugging session of an attribute grammar definition. We finally analyze the technique of traversing through the execution tree and describe how the user helps the debugger in locating the errors.

4.1 Attribute Dependencies

4.1.1 Dependency Graph

Section 2.1.1 describes how the evaluation order of attributes is dictated by their dependencies on other attributes. These dependencies when considered over the syntax tree form a directed acyclic graph structure which we call the dependency graph. There are two types of dependency graphs that arise from where the dependencies are picked. The first is a production graph that is derived from the AG definition. The production graph is available before the actual evaluation of the attributes. The second type is the dynamic dependency graph. As its name suggests, it is constructed during the process of attribute evaluation and describes the dependencies more accurately when compared

to the production graph.

Production dependency graph

The dependencies in a production graph are determined by the attribute computation equations defined in the productions of the attribute grammar. The dependency graph $DG(p)$ for a production p contains a vertex $\langle n, a \rangle$ for every attribute a defined on a non-terminal symbol n belonging to the production. An edge E in $DG(p)$ exists from a vertex $V_1(n_1, a_1)$ to a vertex $V_2(n_2, a_2)$ if the computation of the attribute a_1 on the non-terminal n_1 is dependent on the value of the attribute a_2 defined on the non-terminal n_2.

```
1  abstract production assign
2  s::Stmt ::= id::Id e::Expr
3  {
4      e.env = s.env;
5      s.errors = if (checkInList(id.lexeme,s.env) == false)
6                 then
7                   id.lexeme ++ " has not been defined.\n" ++ e.
                        errors
8                 else
9                   "";
10
11 }
```

Figure 4.1: The *assign* production.

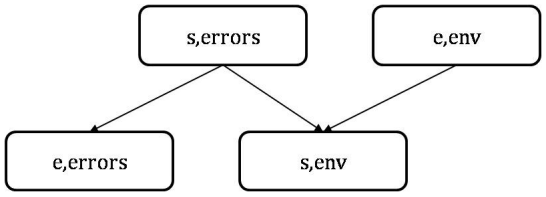

Figure 4.2: A production dependency graph for the production defined in Figure 4.1

Consider the example production defined in Figure 4.1 that is part of the attribute grammar defined in Figure 2.2. A production dependency graph for the production *assign* is shown in Figure 4.2. The graph contains vertices for the attributes *errors* and *env* for both s the Stmt node and e the Expr node. It also depicts the dependency of the attribute *errors* on s on the *errors* attribute on of the Expr node and the *env* attribute of the Stmt node. The *env* attribute of the Expr node depends only on the *env* attribute of the Stmt node. These dependencies directly correspond to the computation rules defined for these attributes in the production definition shown in Figure 4.1.

4.1.2 Reachability

In a graph $G = \langle V, E \rangle$, a path from a vertex u to a vertex u' is a sequence $v_0, v_1 ... v_n$ of vertices such that $v_0 = u$ and $v_n = u'$ and $(v_{i-1}, v_i) \in E$ for $1 \leq i \leq n$. If there is a path l from a vertex v_j to v_k, we say that v_k is reachable from v_j via l. There is always a path of zero length from v_j to v_j. We denote the set of all vertices reachable from a vertex v_j by $reachable(v_j)$.

Let $X_0 = (N_0, a_0)$ and $X_1 = (N_1, a_1)$ be two nodes of the production dependency graph $DG(p)$ for a production p. Node X_0 is reachable from node X_1 if there is a path from X_1 to X_0 in $DG(p)$. I.e., the value of the attribute a_1 in the graph on node X_1 is directly or indirectly dependent on the value of the attribute a_0 on node X_0. The reachable nodes from node X_1 are called the reachable set of node X_1, written as $reachable(X_1)$. Formally, we denote $reachable(X_0)$ to be the set of all nodes that are reachable from the node X_0 in the production dependency graph and thus the set of all nodes that X_0 depends on, where $reachable(X_0) \subseteq V$ with V being the list of all nodes in $DG(p)$ for a production p.

4.2 Building the Execution Tree

The first phase in algorithmic debugging as explained in Section 2.2 involves the building of the execution tree of the program that is being debugged. Debugging attribute grammars presents us with a new approach to the creation of the execution tree. Unlike the traditional algorithmic debugging process [8], we are interested in the definition rules that compute the value of the attributes that decorate the syntax trees of a grammar.

Given our requirements, the data structure that suits our setting most apt are attribute instances since every attribute instance on a syntax tree corresponds to an occurance of an attribute on a non-terminal symbol and belongs to a computation rule defined for the attribute.

Building an execution tree of attribute instances is a non-trivial process. Section 2.1.1 gives us a brief description of the occurances of attribute instances on syntax trees. Every node in a syntax tree contains an attribute instance for every attribute that occurs on that non-terminal symbol. The user starts a debugging session by providing the debugger the erroneous attribute a_e and the root node N_r of the syntax tree on which this attribute occurs. This information is used by the debugger to build the root node of the execution tree (used interchangably with attribute instance tree). The node constitues the attribute instance of the attribute a_e on N_r. The debugger then uses the production dependency graph $DG(p)$ for the production p that defines the node N_r, to extract information regarding the attribute values on which a_e depends. With the list of dependencies obtained from the graph, the debugger explores the syntax tree to locate the attribute instances that correspond to the dependencies. These attribute instances are added as children to the root node of the tree that is being built. The debugger then recursively repeats the process until all the dependencies have been extracted from the production graph and added to the execution tree. The tree that has been built contains all the attribute instances that could have impacted the erroneous value of a_e. The possibility of multiple attribute instances being dependent on a single attribute instance makes the execution tree a directed acyclic graph. If A_e is the attribute instance for the attribute a_e on N_r, the nodes in the execution tree belong to the set $reachable(A_e)$ i.e., the set of all nodes on which A_e directly or indirectly depends.

Formally, the execution tree for a debugging session can be defined as the directed acyclic graph $DG(t, a)$ where t is the syntax tree whose root node contains the erroneous attribute a. A node N_e in $DG(t, a)$ is an attribute instance on the tree t and can be represented as the pair $\langle n_t, a_t \rangle$ where n_t is a node on the tree t and a_t is an attribute occuring on node n_t. There exists an edge between node $\langle n_1, a_1 \rangle$ and node $\langle n_2, a_2 \rangle$ in $DG(t, a)$ if there exists an edge between the nodes $\langle n_1, a_1 \rangle$ and $\langle n_2, a_2 \rangle$ in production dependency graph $DG(p)$ of the production p that defines n_1 in the execution tree.

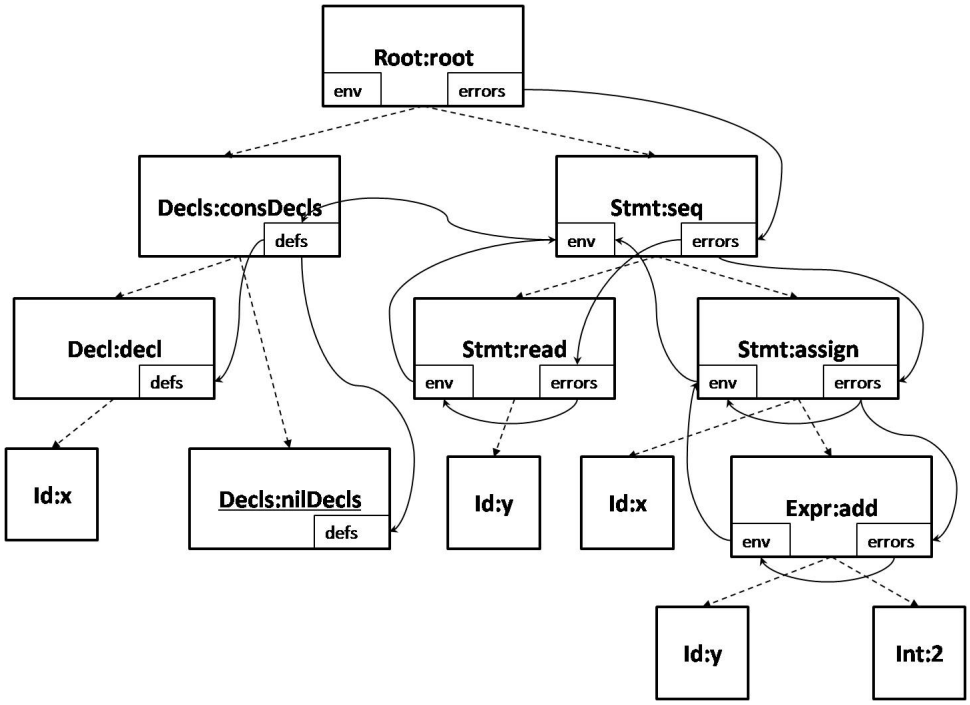

Figure 4.3: An execution tree for the program in Figure 2.1.

Figure 4.3 shows the execution tree for the imperative program in Figure 2.1. The solid lines in the figure correspond to the edges in the execution tree built to debug the *errors* attribute at the root node. The dotted lines correspond to the edges in the abstract syntax tree of the program. The root of the execution tree is the instance of *errors* on the *root* node of the abstract syntax tree. An edge in the execution tree, for example, from *errors* on *root* to *errors* on *seq* denotes the dependency of *errors* on *root* on the same attribute on *seq*.

4.3 Traversing the Execution Tree

According to the algorithmic debugging process, in the second phase the debugger traverses through the execution tree that has been built. We first start from the root node of the attribute instance tree that has been built. The debugger uses the information present in the attribute instance to ask the user about the correctness of the value that it has been evaluated to. At every stage during debugging when the user is asked about

the correctness of an attribute instance, he/she has an option to reply to the question with a *yes* or a *no*. Based on the user's answer, the debugger uses the assertion defined by Shapiro[8] to decide on it's next action. If the user replies with a *yes*, the debugger marks that attribute instance as correct and does not proceed to examine the subtree of that node in the attribute instance tree. It then proceeds to analyze other possible nodes that have the potential to be erroneous. If the user marks the root node of the execution tree to be correct, the debugging session is ended. The second option for the user, i.e., replying to the question with a *no*, forces the debugger to investigate the subtree of the node for potential errors. Once the debugger decides to explore the subtree of a node, it presents the user with a second set of questions. The user is asked to pick a specific node for the debugger to analyze, from a list of nodes that could have impacted the error in the attribute value. When the user picks an attribute instance node from the list presented, the debugger performs the same routine of querying the user about the correctness of the instance. A bug is located in the following two scenarios where an incorrect attribute instance A does not have any incorrect dependencies.

Consider the execution tree in Figure 4.3. It corresponds to a debugging session for the attribute *errors* on the AST of a program described in Figure 2.1. The execution tree shows the dependency structure of the attribute instances on the AST. The solid edges in the figure correspond to the dependencies between attribute instances, which form the nodes of the execution tree. The *errors* attribute on the *root* node is dependent on the *errors* attribute of the *seq* node. This is then dependent on the occurance of *errors* on both *read* and *assign*. On the *assign* node, the instance of *errors* is dependent on the value of the *env* attribute that has been passed down to it from the *seq* node and on the *errors* attribute that is synthesized from the *Add* node. The debbuger first starts the session by asking the user if the value of *errors* on *root* is correct. If the user responds to this with a *no*, it proceeds to the only child of this attribute instance in the execution tree, which is the occurance of *errors* on the *seq* node. The debugger then queries the user if the value of *errors* on *seq* is correct. If the user again responds with a *no*, the debugger examines the execution tree and presents the user with a choice for the next node to explore. Since the current attribute instance is dependent on the occurances of *errors* on *read* and *assign*, the user can choose which one he/she wants to explore first. Once the user picks his choice, the process of querying the user is repeated until

the debugger has found an incorrect evaluation of an attribute instance with all correct children. It then reports the bug at the definition of the node on which the incorrect instance occurs.

```
1: The value of the attribute errors on node root from line  1 to line 5
   is y is not defined on line 3?
> no
2: The value of the attribute errors on node seq from line  3 to line 4
   is y is not defined on line 3?
> no
3: Choices for the next instance to be examined are:
A: errors on node read
B: errors on node assign
> B
4: The value of the attribute errors on node assign from line  4 to line 4
   is ?
> no
5: Choices for the next instance to be examined are:
A: errors on node add
B: env on node assign
> A
6: The value of the attribute errors on node add from line  4 to line 4
   is y is not defined on line 4?
> yes
7: The value of the attribute env on node assign from line  4 to line 4
   is [x]?
> yes
An error has been located at the definition of errors on production assign
```

Figure 4.4: An interaction of algorithmic debugging.

Figure 4.4 describes an interaction between the user and the debugger for a debugging session for the *errors* attribute on the *root* node of the AST shown in Figure 2.3. At first, the debugger queries the user about whether the value of the attribute *errors*

on the *root* node is "y is not defined on line 3". The user replies with a *no* since this value of *errors* does not match his interpretation of the grammar. The debugger then examines the execution tree and finds only one dependency instance for this node. It proceeds to ask the user about the value of the attribute *errors* on the *seq* node. The user again replies with a *no*. At this point, the debugger presents the user with two options for the next node to be visited. The first one belongs to the occurance of *errors* on *read* and the second belongs to the occurance of *errors* on *assign*. The user makes a decision about which node to explore and replies with his choice, which in this case is *errors* on *assign*. The debugger then repeatedly queries the user about the correctness of attribute values on nodes of the syntax tree, following the algorithmic debugging technique and locates the error in the definition of *errors* on *assign* since the values of the attributes on all its children were marked correct by the user but the value on it was marked incorrect.

Chapter 5

Further Improvements

The basic algorithmic debugging approach although accurate in finding and localizing errors, has a few flaws. In this chapter we discuss the two critical issues with the technique. The first being the number of interactions between the user and the debugger, and the second being the usage of static production dependency graphs in building the execution tree. We then proceed to describe a means to tackle the first issue resulting in a more intelligent debugger and lesser number of forced interactions for the user. The second issue is overcome by using dynamic dependency graphs. This is described in Section 7.1.1 on Future Work.

5.1 Two Issues with Basic Approach

The first major issue is the tediousness of the process of debugging for the user. The number of interactions that the user is forced to have is one of the most important criterions in deciding the usabitility of the debugger. The process of debugging attribute grammars described in Section 4.2 requires the user to answer two questions at each stage. One regarding the correctness of the attribute instance and the second regarding the path to be explored next in the execution tree that was built. The second question requires the user to make a conscious decision by comparing the various options provided and picking the most probable erroneous child. Considering the execution tree in Figure 4.3 discussed earlier, if at the *seq* node, the user is asked the correctness of the *errors* attribute and the user responds with a *no*, the debugger proceeds with asking

the user about the node to be examined. In this case, the *errors* attribute on *seq* is dependent on two attribute instances. The user is asked to make a decision about which path to explore first. Based on the user's decision the debugger then proceeds to examine the chosen node. With some help from the user, the debugger could automatically make an intelligent choice about which node to explore next, once the correctness of an instance has been established.

Another issue with the debugging approach described in Section 4.2 is the use of static production dependency graphs to build the execution tree. Using static graphs has many advantages like being able to establish the dependency structure at each production before the actual execution of the program but has it's share of disadvantages too. Static graphs often lead to over approximation of dependencies. This means that the debugger could explore paths that it might not be required to, based on the program being executed. A solution to this problem is to use dynamically generated dependency graphs. Dynamic graphs provide a more accurate dependency structure. Section 7.1.1 on Future Work will discuss this approach in detail.

5.2 Guided Heuristics Based Debugging

An unfortunate side effect of algorithmic debugging has been the number of questions that the user is required to answer. As discussed in Chapter 3, previous work to mitigate this problem includes program slicing [11], correct subcomputation [12], partition testing [10] etc. Partition testing described by Fritzson [10] tries to overcome the issue of the user having to answer difficult correctness questions by using the results of previously done exhaustive testing of procedures, during the debugging session. Silva et al. [11], combine the process of algorithmic debugging with program slicing to give the user semantically connected questions. Their technique provides a means for the user to give information to the debugger regarding the specific parts of a query that are incorrect. For example, a user could specify that it was the input parameter to the procedure call that has the wrong value. The debugger would then slice the execution tree based on this information and thereby reduce the search space. Davie and Chitil [12] have tried to combine a 'computation comparison' method with program slicing. Their technique uses heuristics regarding the number of times a procedure is executed with correct results and

combines them with program slicing to reduce the size of the execution tree and thereby reducing the number of questions asked to the users. These efforts focus primarily on reducing the number of questions asked to the user regarding the correctness of a node.

Our focus in reducing the number of user interactions with the debugger rests on reducing number of the *search* questions about where the debugger should proceed next. We propose a guided heuristics based approach to algorithmic debugging. Our method aims at imposing an ordering on the dependencies of a node in the attribute instance tree. Such an ordering is performed based on obtaining information from the user prior to the start of the debugging session. The debugger then picks the next node to be examined based on the ordering that has been performed. The heuristic information provided to the debugger is a priority based ranking of the various components of the attribute grammar. These components include attributes, productions and non-terminal symbols. Such an ordering of the dependencies in the execution tree at every step eliminates many of the *search* questions and drastically reduces the number of interactions in complex debugging sessions.

A tie in the ordering of the dependencies occurs when two attribute instances result in the same ranking based on their attributes. To resolve such a tie, we propose secondary and tertiary priorities of the grammar. These priorities could be rankings of other components of the attribute grammar like productions and non-terminal symbols, assigned by the user at the start of a debugging session. In the case of a tie occuring, the debugger would then use the secondary priority to order the tied attribute instances and if it still ends up with a tie, the tertiary priorities would be used to resolve it. In the situation that an ordering results in a tie and there are no further priorities that the debugger can use to resolve it, based on its implementation it either reuses the dependency ordering provided in the equation defining the attribute or queries the user to pick the next node to be examined.

Consider the execution tree in Figure 4.3 that belongs to a debugging session for the *errors* attribute on the abstract syntax tree shown in Figure 2.3. In a guided heuristic approach by ranking the attributes, the debbuger first scans the execution tree to get the list of all attributes that are being used to present it to the user for prioritizing them. Let the priorities assigned to the attributes be *errors*, *env*, *defs* in order of descreasing priority. Let us consider the debugger's examination of the attribute

instance pertaining to the occurance of *errors* on the *assign* node. After querying the user about the correctness of the attribute value and getting a negative response, the debugger is required to make a decision about the next node to be explored. The occurance of *errors* on *assign* is dependent on two attribute instances. The first is the occurance of *env* on the same *assign* node and the second is the *errors* on the *add* node which is a child to *assign*. At this point the debugger uses the priorities assigned to the attributes to make a decision. Since *errors* is given a higher priority than *env*, the execution tree is rearranged to have the attribute instance of *errors* on *add* appear before the instance of *env* on *assign*. The debugger then picks the first unexamined node in the reordered list, which is *errors* on *add* and asks the user regarding the correctness of the instance. In the case that there is more than one attribute instance with the same attribute and hence the same ranking, the debugger resolves the tie by either choosing the first attribute instance in the list according the the production dependency graph or by querying the user about the node to be examined. Section 6.2.1 discusses the implementation of the heuristics based approach for the Silver[5] debugger.

The example shown in in Figure 5.1 describes an interaction of a guided heuristic based debugging session corresponding to the execution tree in Figure 4.3. At the start of the debugging session, the user is asked to first provide a prioritized list of the attributes. He is then asked to provide a secondary heuristic by listing out the priorities of the productions used in the debugging session. Once this information is secured, the debugger proceeds as per discussed in Figure 4.4. The first major difference in the interaction occurs at question 3. At this stage previously, the user was required to pick the next node to be examined in the execution tree. With the heuristic information, the debugger first reordered the list of attribute instances based on the attributes. In this case, the two instances end up in a tie since both the instances belong to the occurance of *errors* on *assign* and *read*. At this state, the debugger uses the secondary priority given by the user. Since *assign* is ranked higher than *read*, the debugger reorders the tree accordingly and picks the attribute instance pertaining to the occurance of *errors* on *assign*. A similar reordering is performed before question 4. The dependencies in question here belong to the occurance of *env* on the *assign* node and that of *errors* on *add*. Since the attribute *errors* was given a higher priority by the user, the debugger reorders the execution tree such that the instance pertaining to the occurance of *errors*

on *add* is picked first.

```
The list of attributes used in this session are: errors, defs, env.
What is your order of the attributes:
> errors,env,defs
The list of productions used in this session are: seq, read, assign, add,
consDecls, decl, nilDecls
What is your order for the productions:
> seq, assign, read, add, decl, consDecls, nilDecls

1: The value of the attribute errors on node root from line  1 to line 5
   is y is not defined on line 3?
> no
2: The value of the attribute errors on node seq from line  3 to line 4
   is y is not defined on line 3?
> no
3: The value of the attribute errors on node assign from line  4 to line 4
   is ?
> no
4: The value of the attribute errors on node add from line  4 to line 4
   is y is not defined on line 4?
> yes
5: The value of the attribute env on node assign from line  4 to line 4
   is [x]?
> yes
An error has been located at the definition of errors on production assign
```

Figure 5.1: An interaction of guided heuristic debugging.

The heuristic information presented above can be considered as a sample mechanism to provide information to the debugger. Two attribute instances could be differentiated by various aspects of the grammar like the non-terminal symbol that it resides on, the production that built the non-terminal, the attribute that decorates the non-terminal on this attribute instance to name a few. The implementation of such a heuristic based debugger could choose any of these as the primary means to distinguish or prioritize

attribute instances. This can be generalized more by requesting the user to provide a ranking for the various properties of the grammar first and then prioritize each individual values of these properties in the grammar. Chapter 6 discusses one such implementation of the guided heuristic based debugger.

Chapter 6

Implementation

An important goal in the implementation of the debugger was to create an algorithmic debugger that is not specific to a single language or language paradigm. The design of the framework is such that it caters to the generic nature of the goal. In this chapter we first discuss the design of the algorithm debugger and then describe specific implementations of the debugger for Silver[5].

6.1 Design

The algorithmic debugging framework that has been implemented is modular in design. This modularity provides the framework a flexibility to be extended to other attribute grammar systems and language paradigms other than attribute grammars. For a language to be able to use the framework, we would need the execution tree of the program that is to be debugged. This execution tree, based on the paradigm of the language could be the data flow graph of the executing program or a procedure call trace.

The debugging framework consists of three main components. Figure 6.1 shows the components of the debugger. The first component is the implementation of the basic algorithmic debugging technique (*Debugger*), the second is the *ExecutionTreeNode* interface. The implementation of the interface is specific to the language that is being debugged. It provides the execution tree of the program that is currently being examined in the debugging session and also a means for each node to give the list of its dependencies. These methods have been consciously made generic in order to cater

to different types of tree nodes and to both static and dynamic dependencies. The final component is the dependency picker (*NextNodePicker*) for the debugger. The *NextNodePicker* is a pluggable component which helps in picking the next node in the execution tree for the debugger to examine. The default implementation of the *NextNodePicker* requests the user to pick the next node to be examined from a given list of dependencies. As shown in the figure, the debugger class contains an instance of *ExecutionTreeNode* and *NextNodePicker* which it uses during the debugging session.

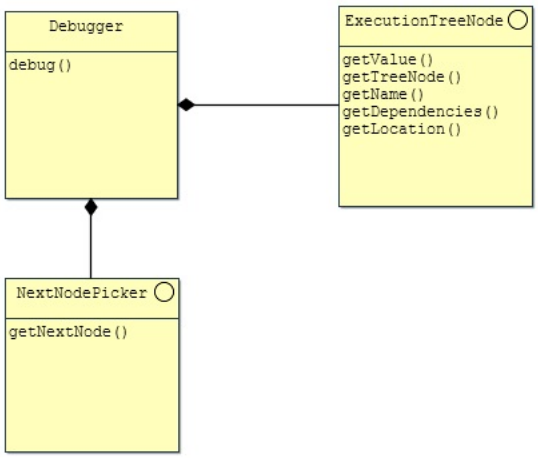

Figure 6.1: The component diagram for the debugging framework.

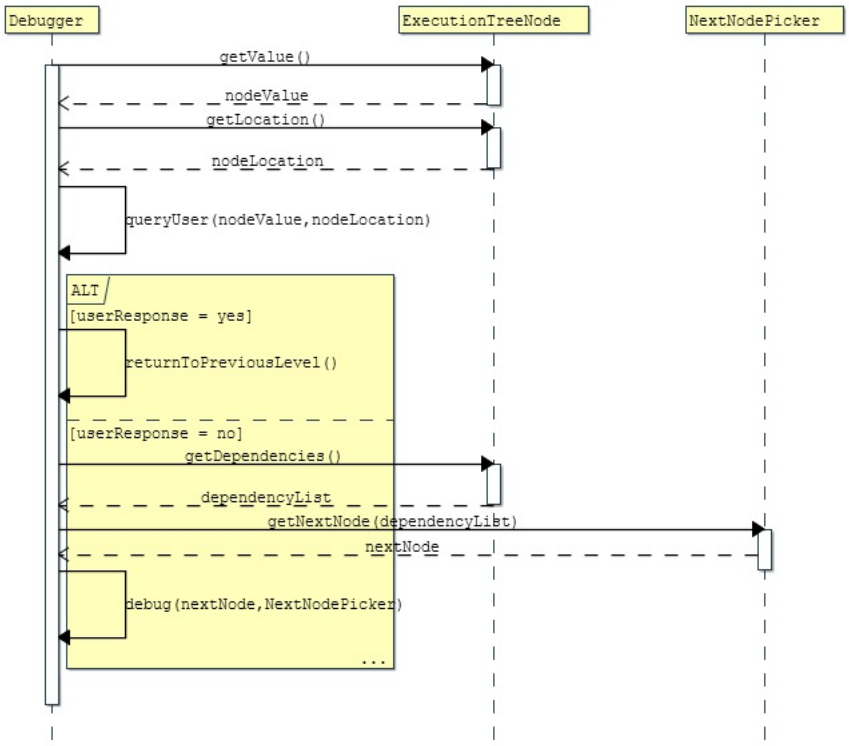

Figure 6.2: The sequence diagram for a debugging session.

Figure 6.2 shows the sequence diagram at every stage of a debugging session according to the generic algorithm described in Figure 6.3. The debugger first uses the *ExecutionTreeNode* instance passed to it to get the value at that node and the location to which the node corresponds to in the program. With this information, it queries the user for the correctness of the node. If the user responds with a *yes*, it returns to the previous level in the debugging session or in the case of a root node, end the session. If the user responds with a *no*, the debugger gets the list of dependencies from the *ExecutionTreeNode* and passes them on to the *NextNodePicker* which then returns it the next node to be explored based on the algorithm used (query the user or pick with hueristics). The debugger then repeats the process with this new node.

```
debug(executionTreeNode, nextNodePicker)
{
   nodeValue = executionTreeNode.getValue()
   nodeLocation = executionTreeNode.getLocation()

   userResponse = queryUserForCorrectness(nodeValue,nodeLocation)

   if (userResponse = "yes")
      return true;
   else if (userResponse = "no")
   {
      dependencyList = executionTreeNode.getDependencies()
      nodeToExamine = nextNodePicker.getNextNode(dependencyList)
      return debug(nodeToExamine,nextNodePicker)
   }

}
```

Figure 6.3: Pseudo code for the debugging algorithm.

6.2 Silver Debugger

The algorithmic debugging approach for attribute grammars described in Chapters 4 and 5 has been implemented for the Silver[5] attribute grammar system based on the debugging framework discussed above. Figure 6.4 shows the components in the implementation of the debugger for Silver. The Silver debugger consists of its custom implementation of the $ExecutionTreeNode$ interface. $SilverAttributeInstance$ represents an attribute instance node of the execution tree that the debugger traverses during a session. It contains a reference to the syntax tree node on which it resides, the name and value of the attribute on that node. It also maintains a list of dependencies on which the current attribute instance depends. This is based on a static dependency graph.

The Silver debugger also contains four implementations of the $NextNodePicker$. The first ($GetDependentByChoice$) follows the basic approach where the user is required to pick the next node to be examined. In this implementation, the user is provided with a list of attribute instances marked by their name and the node on which they reside, from which he picks the next node. The second ($GetDependentByHeuristics$) is the guided heuristics based approach. The implementation provides for the user a means to rank the attributes in the grammar according to the order in which they should be examined. The dependency picker then follows the approach discussed in Section 5.2 to return the next node to be examined by the debugger. The third implementation is the $GetDependentsByWieght$. It helps the debugger choose the node which has the most number of children. The final implementation of the $NextNodePicker$ is a random picker ($GetRandomDependent$) where the dependencies are chosen in a random order for them to be examined by the debugger.

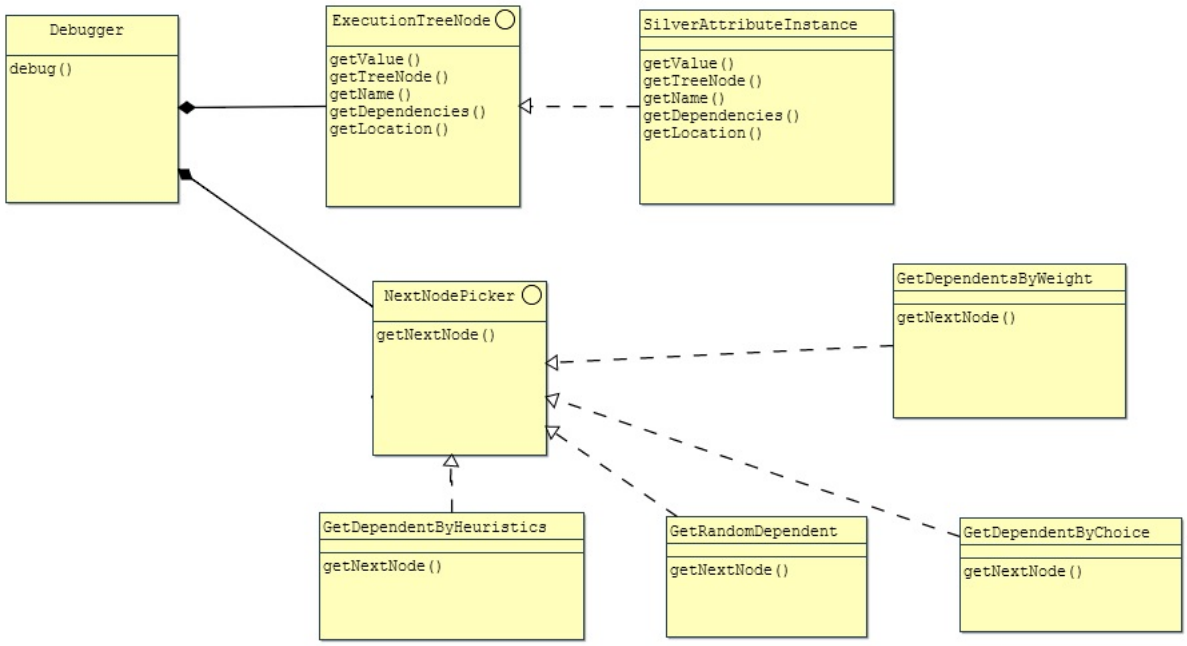

Figure 6.4: The component diagram for Silver debugger.

To start a debugging session, the user selects the incorrect attribute name (a_e), the

root node of the syntax tree on which the attribute value was found to be incorrect and a specific instantiation of the *NextNodePicker* that is to be used. The Silver debugger builds the execution tree for the session in a two step process. In the first step, the debugger parses a static definition file to build the production dependency graph for each production in the grammar. Once it has the dependency graph, in the second step, it creates the *ExecutionTreeNode* object for the attribute being debugged on the root node of the syntax tree. The *ExecutionTreeNode* object contains a reference to the node on the syntax tree to which it belongs. This node is later used to build the execution tree as the debugger explores more of the tree. It also contains the value of the attribute at the tree node and the location of the the node in the real program. This location is computed by mapping the node being examined to the node in the concrete syntax tree (CST) that created it. Since the CST of the language is directly associated with the program being parsed, each CST node is associated with the corresponding location on the program. Once the *ExecutionTreeNode* object is created, the debugger queries the user regarding the correctness of the value of the attribute instance. If the user responds with a *yes*, the debugging session is ended since the attribute value on the root node is correct. If the user responds with a *no*, the debugger uses the static dependency information of the production belonging to this attribute instance to create the *ExecutionTreeNode* objects for all the dependencies. The debugger then uses the *NextNodePicker* that has been instantiated for this session to determine the next node to be examined. It then recursively follows the algorithmic debugging technique as explained in Section 4.2 to locate the bug.

Consider the program in Figure 6.5 written according to the attribute grammar definition in Figure 2.2. The execution tree built by the debugger for a debugging session of the *errors* attribute on the *root* node is shown in Figure 6.6. These figures are the same examples used in Figures 4.3 and 2.1. The dotted lines in the figure represent the abstract syntax tree and the solid lines correspond to the execution tree constructed by the debugger based on dependencies over attribute instances. A typical interaction of the debugging with a default *NextNodePicker* is shown in Figure 6.7. At first, the debugger queries the user about whether the value of the attribute *errors* on the *root* node is "y is not defined on line 3". The user replies with a *no* since this value of *errors* does not match his interpretation of the grammar. The debugger then

examines the execution tree and finds only one dependency instance for this node. It proceeds to ask the user about the value of the attribute *errors* on the *seq* node. The user again replies with a *no*. At this point, the debugger presents the user with two options for the next node to be visited. The first one belongs to the occurance of *errors* on *read* and the second belongs to the occurance of *errors* on *assign*. The user makes a decision about which node to explore and replies with his choice, which in this case is *errors* on *assign*. The debugger then repeatedly queries the user about the correctness of attribute values on nodes of the syntax tree, following the algorithmic debugging technique and locates the error in the definition of *errors* on *assign* since the values of the attributes on all its children were marked correct by the user but the value on it was marked incorrect.

```
Integer x;
begin
  read(y);
  x = y+2;
end
```

Figure 6.5: A sample imperative program.

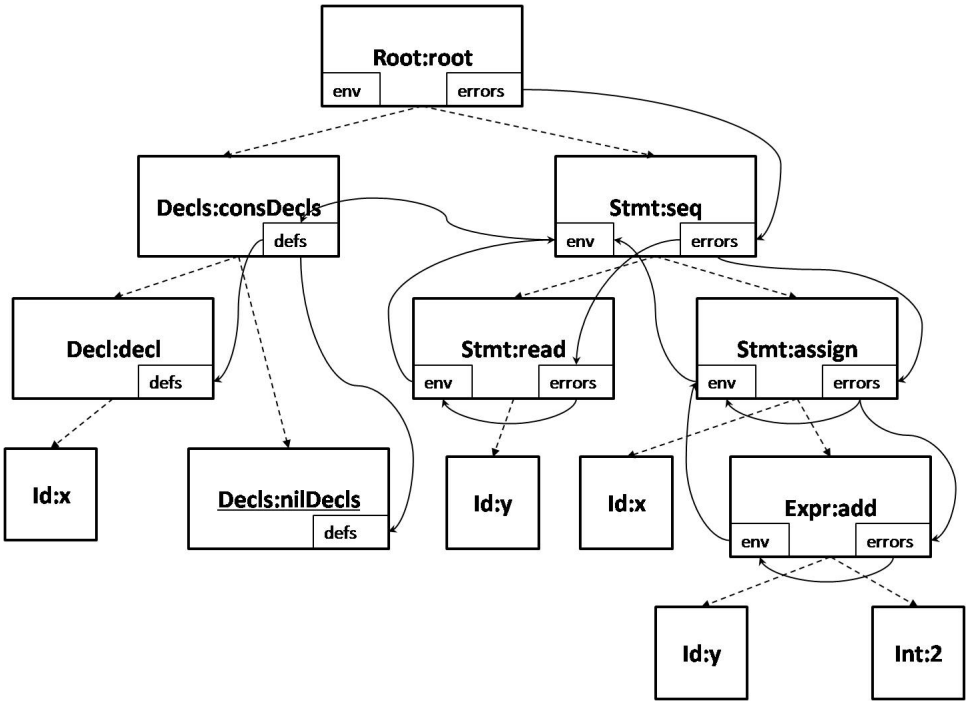

Figure 6.6: The execution tree.

```
1: The value of the attribute errors on node root from line  1 to line 5
   is y is not defined on line 3?
> no
2: The value of the attribute errors on node seq from line  3 to line 4
   is y is not defined on line 3?
> no
3: Choices for the next instance to be examined are:
A: errors on node read
B: errors on node assign
> B
4: The value of the attribute errors on node assign from line  4 to line 4
   is ?
> no
5: Choices for the next instance to be examined are:
A: errors on node add
B: env on node assign
> A
6: The value of the attribute errors on node add from line  4 to line 4
   is y is not defined on line 4?
> yes
7: The value of the attribute env on node assign from line  4 to line 4
   is [x]?
> yes
An error has been located at the definition of errors on production assign
```

Figure 6.7: An algorithmic debugging interaction for Silver.

6.2.1 Guided Heuristics approach

The guided heuristics based approach described in Section 5.2 is implemented in the Silver debugger as a custom $NextNodePicker$. The $GetDependentByHeuristics$ class shown in Figure 6.4 is an implementation of the $GetDependent$ interface and implements the technique. When the debugger first requests for the next node to explore, the instance of $GetDependentByHeuristics$ explores the entire execution tree to get a list

of all the attributes that are used in the computation of the erroneous attribute, either directly or indirectly. It then presents this list of attributes to the user for him/her to rank. The user then ranks each of these in order based either on intuition or an assessment of the complexity of the computation involved. It then gets the list of all the dependencies for the current attribute instance that has been examined and reorders them based on the priorities given by the user. In the case of two different attribute instances having the same attribute name (and hence the same priority), it maintains the order provided by the static dependency graph. After the restructuring, the highest ranked unvisited attribute instance is returned to the debugger for it to be examined.

```
The list of attributes used in this session are: errors, env, defs.
What is your priority for attribute-errors [1-3]:
> 1
What is your priority for attribute-defs [1-3]:
> 3
What is your priority for attribute-env [1-3]:
> 2
1: The value of the attribute errors on node root from line  1 to line 5
   is y is not defined on line 3?
> no
2: The value of the attribute errors on node seq from line  3 to line 4
   is y is not defined on line 3?
> no
3: Choices for the next instance to be examined are:
A: errors on node read
B: errors on node assign
> B
4: The value of the attribute errors on node assign from line  4 to line 4
   is ?
> no
5: The value of the attribute errors on node Add from line  4 to line 4
   is y is not defined on line 4?
> yes
6: The value of the attribute env on node assign from line  4 to line 4
   is [x]?
> yes
An error has been located at the definition of errors on production assign
```

Figure 6.8: An interaction of algorithmic debugging.

Figure 6.8 shows an interaction of a debugging session using the guided heuristics method. The implementation of the *NextNodePicker* for this example requests the user to prioritize the attributes used in the session and in case of a tie it uses the order provided by the dependency graph. As seen in the figure, at the start of the session

the user is provided a list of attributes to rank. The user ranks the attributes in the order: *errors* first followed by *env* and finally *defs*. The interaction with the debugger after the prioritization is very similar to the one shown in Figure 6.7. At question 3, the debugger asks the user to pick a node to examine presenting a list of possible dependencies. The reason for this query as can be seen in the figure is that the ordering of the depepndencies has resulted in a tie due to both the attribute instances containing the same attribute *errors*. A noticable change from Figure 6.7 is at question 5. The debugger no longer asks the user to pick a node to examine since it has used the priority assigned by the user to order the execution tree accordingly. In this case, since *errors* has a higher priority than *env*, the attribute instance of *errors* on *assign* is examined first. The debugger then follows the usual querying mechanism and locates the bug in the definition of *errors* on the *assign* production.

6.3 Generic Debugger

The algorithmic debugging framework that has been implemented is modular in design. The modularity provides the framework a flexibility to be extended to other attribute grammar systems and language paradigms other than attribute grammars. For a language to be able to use the framework, we would need the execution tree of the program that is to be debugged. This execution tree, based on the paradigm of the language could be the data flow graph of the executing program or a procedure call trace.

To be able to use the debugging framework for a language, the language designer would have to implement their custom $ExecutionTreeNode$. This implementation would, as in the case of attribute grammars, help in building the execution tree as the debugger traverses from one node to the next, interacting with the user. As shown in Figure 6.1 the implementation of $ExecutionTreeNode$ would contain a means for the debugger to query for the list of dependencies as which have been obtained from a dependency graph specific to the language being debugged. The language designer could also implement their own $NextNodePicker$ to suit the needs of the language. The picker could choose to apply the guided heuristic mechanism explained in Section 5.2 to rank the properties of the language.

An example of such an implementation could be a toy functional language. The

execution tree of such a language would contain $ExecutionTreeNode$ nodes that refer to function calls made during the execution of the program. The *value* inside each node would be the output for a specific instance of a function call pertaining to the inputs provided to it. An implementation of the guided heuristic approach to this language could ask the user to rank the functions in order of priority. The debugger would then use the custom $NextNodePicker$ to impose the order on the list of dependencies at each stage of the debugging process to make a decision on the node to be explored next.

Chapter 7

Future work and Conclusion

7.1 Future Work

In this section we look at the possible directions to extend this research.

7.1.1 Dynamic Dependencies

One of the issues of the basic algorithmic debugging technique was the usage of static production based dependencies. Using such a dependency graph results in over-approximation of dependencies. This is directly related to the size of the execution tree that is built for the debugging session and hence the number of questions that the user is required to answer before the debugger locates the error in the program.

```
1  abstract production assign
2  s::Stmt ::= id::Id e::Expr
3  {
4      e.env = s.env;
5      s.errors = if (checkInList(id.lexeme,s.env) == false)
6                 then
7                   id.lexeme ++ " has not been defined.\n" ++ e.
                        errors
8                 else
9                   "";
10
11 }
```

Figure 7.1: The *assign* production.

This can be overcome by using a dynamic dependency graph that is computed as the program is being analyzed by the attribute grammar system. The dynamic graph would only contain dependencies that are used in the actual computation of the attribute. For example, in the *assign* production shown in Figure 7.1, if the actual computation of the *errors* attribute on the *Stmt* node chose the *else* branch, its final value would have only one dependency, the occurance of *env* on the *Stmt* node. As opposed to a production dependency graph that would over approximate the dependencies to have both the occurance of *env* on *Stmt* and the value of *errors* on *Expr* irrespective of whether the *if* or the *else* branch is chosen. Our implementation of the debugger for Silver currently does not use a dynamically generated dependency graph primarily due to the modifications required at the attribute grammar system to generate it. Future work could therefore extend the implementation of Silver to use generate dynamic dependencies which the debugger could leverage on to provide a better experience to the user.

Another approach to solving the issue would be to use an attribute evaluation trace. The dependency information for the evaluation of attributes can be extracted from such a trace and used in the building of the execution tree during debugging. Sloane[24] describes one such library which could be used in building the dynamic dependency

graph for a debugging session.

7.1.2 User Interface

The current implementation of the debugger for Silver is purely based on the command line interface and suits Silver's current interface. But with Silver soon moving to an eclipse based IDE, a natural transition for the deugger would be to be packaged with eclipse to provide a user an all-powerful IDE to develop, test and debug attribute grammars.

7.1.3 Generalization for other Attribute Grammar systems

The modularity of the current implementation of the debugging framework has been discussed in Chapter 6. This opens the door for the debugger to be used by other attribute grammar systems and other language paradigms. A good starting step for such a generalization of the debugger would be to implment a custom $AttributeInstance$ as described in Section 6.3 for AG systems like Kiama[25] or JastAdd[4].

7.2 Conclusion

We believe that a stable debugging framework is as important to a language paradigm as are its features. The traditional step-by-step debugger, while good in its use for imperative languages, is not suitable for declarative paradigms. The technique of algorithmic debugging overcomes this issue by debugging with the help of an execution tree of the program instead of a step through of the source code.

In this thesis we have presented an algorithmic debugging scheme to debug attribute grammars. We looked at how an attribute instance on the node of a syntax tree of an analysis constitutes a node in the execution tree for a debugging session and how the debugger interacts with the user to gain knowledge of the program and locate bugs semi-automatically. We further improved upon the original idea of algorithmic debugging to reduce the number of interactions of the debugger with the user thereby making the debugging experience less tedious. We looked at how this can be achieved by the user providing heuristics or priorities regarding certain properties of the attribute grammar like non-terminal symbols, productions and attributes which the debugger could use to

make intelligent decisions regarding exploring the search space in the execution tree. The proposed debugging scheme was implemented for Silver, our attribute grammar system.

References

[1] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[2] Uwe Kastens, Brigitte Hutt, and Erich Zimmermann. *GAG: A Practical Compiler Generator*, volume 141 of *Lecture Notes in Computer Science*. Springer, 1982.

[3] Thomas W. Reps. *Generating language-based environments*. Massachusetts Institute of Technology, Cambridge, MA, USA, 1984.

[4] Torbjörn Ekman and Görel Hedin. The jastadd system — modular extensible compiler construction. *Sci. Comput. Program.*, 69(1-3):14–26, December 2007.

[5] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: an extensible attribute grammar system. *Electron. Notes Theor. Comput. Sci.*, 203(2):103–116, April 2008.

[6] Eric Van Wyk, Lijesh Krishnan, Derek Bodin, and August Schwerdfeger. Attribute grammar-based language extensions for java. In *Proceedings of the 21st European conference on Object-Oriented Programming*, ECOOP'07, pages 575–599, Berlin, Heidelberg, 2007. Springer-Verlag.

[7] Yogesh Mali and Eric Van Wyk. Building extensible specifications and implementations of promela with AbleP. In *Proceedings of 18th the International SPIN Workshop on Model Checking of Software (SPIN 2011)*, volume 6823 of *LNCS*, pages 108–125. Springer Verlag, July 2011.

[8] Ehud Y. Shapiro. *Algorithmic Program DeBugging*. MIT Press, Cambridge, MA, USA, 1983.

[9] Henrik Nilsson and Peter Fritzon. Algorithmic debugging for lazy functional languages. In *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, PLILP '92, pages 385–399, London, UK, UK, 1992. Springer-Verlag.

[10] Peter Fritzson, Nahid Shahmehri, Mariam Kamkar, and Tibor Gyimothy. Generalized algorithmic debugging and testing. *ACM Lett. Program. Lang. Syst.*, 1(4):303–322, December 1992.

[11] Josep Silva and Olaf Chitil. Combining algorithmic debugging and program slicing. In *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '06, pages 157–166, New York, NY, USA, 2006. ACM.

[12] Thomas Davie and Olaf Chitil. Hat-delta — one right does make a wrong. In Colin Runciman, editor, *Hat Day 2005: work in progress on the Hat tracing system for Haskell*, pages 6–11. Tech. Report YCS-2005-395, Dept. of Computer Science, University of York, UK, October 2005.

[13] Uwe Kastens. Ordered attributed grammars. *Acta Informatica*, 13:229–256, 1980. 10.1007/BF00288644.

[14] H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, PLDI '89, pages 131–145, New York, NY, USA, 1989. ACM.

[15] Görel Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3), 2000.

[16] Eric Van Wyk, Oege de Moor, Kevin Backhouse, and Paul Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 128–142, London, UK, UK, 2002. Springer-Verlag.

[17] Rodney Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *Proceedings of the 1986 SIGPLAN*

symposium on Compiler construction, SIGPLAN '86, pages 85–98, New York, NY, USA, 1986. ACM.

[18] Nahid Shahmehri and Peter Fritzson. Algorithmic debugging for imperative languages with side-effects (abstract). In *Proceedings of the Third International Workshop on Compiler Construction*, CC '90, pages 226–227, London, UK, UK, 1991. Springer-Verlag.

[19] Josep Silva. A comparative study of algorithmic debugging strategies. In *In Logic-Based Program Synthesis and Transformation*, pages 143–159. Springer, 2007.

[20] Wlodzimierz Drabent, Simin Nadjm-Tehrani, and Jan Maluszynski. Algorithmic debugging with assertions. In *META*, pages 501–521, 1988.

[21] Ian Douglas MacLarty. *Practical Declarative Debugging of Mercury Programs*. PhD thesis, University of Melbourne, 2005.

[22] Anthony M. Sloane. Debugging eli-generated compilers with noosa. In *Proceedings of the 8th International Conference on Compiler Construction, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99*, CC '99, pages 17–31, London, UK, UK, 1999. Springer-Verlag.

[23] Robert W. Gray, Steven P. Levi, Vincent P. Heuring, Anthony M. Sloane, and William M. Waite. Eli: a complete, flexible compiler construction system. *Commun. ACM*, 35(2):121–130, February 1992.

[24] Anthony M. Sloane. Profile-based abstraction and analysis of attribute grammar evaluation. In *SLE*, pages 24–43, 2012.

[25] Anthony M. Sloane, Lennart C. L. Kats, and Eelco Visser. A pure object-oriented embedding of attribute grammars. *Electron. Notes Theor. Comput. Sci.*, 253(7):205–219, September 2010.