Department of CISE, University of Florida, Gainesville, FL 32611-6120, Technical Report TR-03-003, July 2003.

Knowledge Extraction in the SEEK Project Part II: Extracting Meaning from Legacy Application Code through Pattern Matching

Technical Report

S. Shekar, J. Hammer, M. Schmalz, and O. Topsakal
Department of Computer & Information Science & Engineering
University of Florida
Gainesville, FL 32611-6120
jhammer@cise.ufl.edu

Abstract

We describe a methodology for extracting semantic information from legacy application code to enable rapid development of adapters and wrappers, for example, when integrating heterogeneous legacy sources to improve coordination of processes across firms participating in a supply chain. In particular, we describe the algorithm we have developed to extract business rules and application-specific meaning of entities and attributes in code, which is connected to a database system. By understanding the structure and semantics of the code and its components, we are also able to better understand the meaning of the schema and hence the data in the associated database system. We have a fully functional prototype implementation, which we use to illustrate the methodology using application code and a sample database from a subcontractor participating in a large construction project at the University of Florida. We also perform a qualitative evaluation of our approach that demonstrates the extraction capability, accuracy and usefulness of our algorithm. The described knowledge extraction prototype is part of the SEEK (Scalable Extraction of Enterprise Knowledge) project.

1 Introduction

Factors such as increased customizability of products, rapid delivery, and online ordering/purchasing have greatly intensified the competition in the market and have forced enterprises justify the need for a production in a supply chain and extensive enterprise collaboration. An enterprise or business network comprises of several individual enterprises or participants that need to collaborate in order to achieve a common goal (e.g. produce goods or services with small lead times and variable demand). Recent research has led to an increased understanding of the importance of coordination among subcontractors and suppliers in a business network [2, 9]. Hence, there is a need for decision or negotiation support tools to improve the productivity of an enterprise network by improving the user's ability to co-ordinate, plan, and respond to dynamically changing conditions [11].

The usefulness and success of such tools and systems greatly depends on their ability to support interoperability among heterogeneous systems [17]. Currently, the time and investment involved in integrating such heterogeneous systems that make up an enterprise network to achieve a common goal is a significant stumbling block. Data and knowledge integration among systems in a supply chain requires a great deal of programmatic set up and human hours with limited code reusability. There is a need to develop a toolkit that can semi-automatically discover enterprise knowledge from enterprise sources and use this knowledge to configure itself and act as a software glue-ware between the legacy sources. The

SEEK[1] project (Scalable Extraction of Enterprise Knowledge) that is currently underway at the Database Research and Development Center at the University of Florida is directed at developing methodologies to overcome some of the problems of assembling knowledge resident in numerous legacy information systems [3-5, 10].

1.1 Motivation

A legacy source is defined as a complex stand-alone system with poor or out-dated documentation of the data and application code; often the original designer(s) of such a data source are not available to provide information about design and semantics. A typical enterprise network has contractors and sub-contractors that use such legacy sources to manage their data and internal processes. The data present in these legacy sources is an important input to making decisions at the project level. However, a large number of firms collaborating on a project imply higher degree of physical and semantic heterogeneity in their legacy systems due to a number of reasons stated below. Thus enterprise-level decisions support tools are faced with four practical difficulties related to accessing and retrieving data from the underlying legacy source.

The first problem faced by enterprise-level decisions support tools is that the firms can use various internal data storage, retrieval and representation methods. Some firms might use professional database management systems while some others might use simple flat files to store and represent their data. There are many interfaces including SQL or other proprietary languages that a firm may use to manipulate its data. Some firms may even manually access the data at the system level. Due to such high degrees of *physical heterogeneity*, retrieval of similar information from different participating firms amounts to a significant overhead including extensive study about the data stored in each firm, detection of approach used by the firm to retrieve data, and translation of queries to manipulate the data into the corresponding database schema and query language used by the firm.

The second problem is heterogeneity among the terminologies of the participating firms. The fact that a supply chain usually comprises firms working in the same or closely related domain does not rule out variability in the vocabulary or terminology used by them. For example, firms working in a construction supply chain environment might use *Task, Activity, Work-Item* to refer to an individual component of the overall project. Although all these terms mean the same, it is important to be able to recognize that they mean the same thing. In addition, new data fields that have names that provide little insight into what these fields actually represent may have been added over time. This so-called *semantic heterogeneity* manifests itself at various levels of abstraction, including the application code that may have business rules encoded in them, making it very important to establish relationships between the known and unknown terms to help resolve semantic heterogeneities.

Another important problem when accessing enterprise code is that of preventing loss of data and unauthorized access; hence the access mechanism should not compromise on privacy of the participating firm's data and business model. It is logical to assume that a firm will restrict sharing of enterprise data and business rules even among other participating firms. It is therefore important to be able to develop third party tools that have access to the participating firm's data and application code to extract semantic information but at the same time assure the firm of the privacy of any information extracted, from their code and data.

Lastly, the existing solutions require extensive human intervention and input with limited code reusability. This makes the whole knowledge extraction process tedious and cost inefficient. Thus, it is necessary to build scalable *data access* and *extraction technology* that has the following desirable properties:

1. It automates the knowledge extraction process as much as possible.
2. It must be easily configurable through high level specifications.

[1] This project is currently underway at the Database Research and Development Center at the University of Florida and is supported by National Science Foundation under grant numbers CMS-0075407 and CMS-0122193.

3. It reduces the amount of code that must be written by reusing components.

1.2 Solution Approaches

The role of the SEEK system is to act as an intermediary between the legacy data and the decision support tool. Based on the discussion in the previous section, it is crucial to develop methodologies and algorithms to facilitate discovery and extraction of knowledge from legacy sources. SEEK has a build-time component (data reverse engineering) and a run-time component (query translation). In this thesis we focus exclusively on the build-time component, which operates in three distinct phases.

In general, SEEK operates as a three-step process [4]:

1. SEEK generates a detailed description of the legacy source including entities, relationships, application-specific meanings of the entities and relationships, business rules. The Database Reverse Engineering (DRE) algorithm extracts the underlying database conceptual schema while the Semantic Analyzer (SA) extracts the application-specific meanings of the entities and attributes and the business rules used by the firm. We collectively refer to this information as *enterprise knowledge*.
2. The semantically enhanced legacy source schema must be mapped onto the domain model (DM) used by the application(s) that want(s) to access the legacy source. This is done using a schema mapping process that produces the mapping rules between the legacy source schema and the application domain model.
3. The extracted legacy schema and the mapping rules provide the input to the wrapper generator, which produces the source wrapper. The source wrapper at run-time translates queries from the application domain model to the legacy source schema.

This report focuses on the process and related technologies highlighted in phase 1 above. Specifically, the report focuses on developing robust and extendable algorithms to extract semantic information from application code written for a legacy database. We will refer to this process of mining business rules and application-specific meanings of entities and attributes from application code as *semantic analysis*. The application-specific meanings of the entities and attributes and business rules discovered by the Semantic Analyzer (SA) when combined with the underlying schema and constraints generated by the data reverse engineering module, give a comprehensive understanding of the firm's data model.

1.3 Challenges

Formally, *semantic analysis* can be defined as the application of analytical techniques to one or more source code files to elicit semantic information (e.g. application-specific meanings of entities and their attributes and business logic) to provide a complete understanding of the firm's business model. There are numerous challenges in the process of extracting semantic information from source code files with respect to the objectives of SEEK:

- Most of the application code written for databases is written in high-level languages like C, C++, Java, etc. The semantic information to be gathered may be dispersed across one or more files. Thus the analysis is not limited to a single file. Several passes over the source code files and careful integration of the semantic information gathered is necessary.
- The SA may not always have access or permissions to all the source code files. The accuracy and the correctness of the semantic information generated should not be affected by the lack of input. Even partial or incomplete semantic information is still an important input to the schema matcher in phase 3.
- High-level languages, especially object oriented languages like C++ and Java have powerful features such as inheritance and operator overloading, which if not taken into account, would generate incomplete and potentially incorrect semantic information. Thus, the SA has to be able

to recognize overloaded operators, base and derived classes, etc. thereby making the semantic analysis algorithm intricate and complex.

- Due to maintenance operations the source code and the underlying database are often modified to suit the changing business needs. Frequently, attributes may be added to relations with non-descriptive, even misleading names. The associated semantics for this attribute may be split up among many statements that may not be physically contiguous in the source code file. The challenge here is to develop a semantic analysis algorithm that discovers the application-specific meaning of attributes of the underlying relations and captures all the business rules.
- Human intervention in the form of comments by domain experts is typically necessary. See, for example, Huang et al. [8] where the SA merely extracts all those lines of code, which directly represent business rules. The task of presenting the business rule in a language independent format is left to the user. We present all the semantic information gathered about an attribute or entity in a comprehensive fashion with the business logic encoded in a XML document.
- Industrial legacy database applications often have tens of thousands of lines of application code that maintain and manipulate the data. The application code evolves over several generations of developers and the original developers of the code may have even left the project. Documentation for the legacy database application may be poor and out-dated. Therefore, our semantic analysis approach should not rely heavily on the expert human user or documentation to steer the process of knowledge extraction.
- The semantic analysis approach should be general enough to work with any application code with minimal parameter configuration.

The most important contribution of this report is a detailed description of the SA architecture and algorithms for procedural languages such as C, as well as object oriented languages such as Java. Our design has addressed and solved each one of the challenges stated above. This report also highlights the main features of the SA and proves that the design used is scalable and robust.

The report is organized as follows. Section 2 presents an overview of the related research in the field of semantic information extraction from application code and business rules extraction in particular. Section 3 provides a description of the SA architecture and the semantic analysis algorithms used for procedural and object oriented languages. Section 4 is dedicated to describing the implementation details of SA using the Java version as our basis for the explanations, and Section 5 highlights the power of the Java SA in terms of what features of the Java language it captures. Finally, we conclude the report with a summary of our accomplishments and issues to be considered in the future.

2 Related Work

The process of extracting data and knowledge from a legacy application code logically precedes the process of understanding it. The problem of extracting knowledge from application code is an important one. Major research efforts that attempt to answer this problem include *program comprehension*, *control and data flow analysis algorithms*, *program slicing* and *pattern matching*.

An important trend in knowledge discovery research is *program analysis* or *program comprehension*. Program comprehension typically involves reading documentation and scanning the source code to better understand program functionality and impact of proposed program modifications, leading to a close association with reverse engineering. The other objective of program comprehension is design recovery. Program comprehension takes advantage not only of source code but also other sources like inline comments in the code, mnemonic variable names, and domain knowledge. The emphasis is more on the recovery of the design decisions and their rationale. Since a firm's way of doing business is expressed by its software systems, business process re-engineering and program comprehension are also closely linked.

Several major theoretical program comprehension models have been proposed in the literature. Among the more important ones are Shneiderman's model of program comprehension [13] and Soloway's model [15]. Shneiderman's model can be summarized in three steps. The very first step requires the expert user to be able to intelligently guess the program's purpose. In the next step, the model requires the programmer to then identify low-level structures such as familiar algorithms for sorting, searching and other groups of statements. Finally when a clear understanding of the program's purpose is reached, it is represented in some syntax independent form. Soloway's model on the other hand divides the knowledge base and the assimilation process differently. The knowledge base in this model includes programming language semantics, goal knowledge and plan knowledge. While both methods described above were theoretically strong, they suffer from similar drawbacks - both rely heavily on user or human input and both have a low degree of automation of the program comprehension process.

Lexical and syntactic analysis of a program in accordance to the language context free grammar generates an *Abstract Syntax Tree* (AST). The grammars themselves are described in a stylized notation called *Backus Naur Form* [1] in which the program parts are defined by rules and in terms of their constituents. An AST is similar to a parsing diagram, which is used to show how a natural language sentence is broken up into its constituents but without extraneous details like punctuation. Therefore, an AST contains the details that relate to the program's meaning and it forms the basis of several program comprehension techniques. Such techniques can be as simple as a high-level query expressed in terms of the node types in an AST. The tree traversal algorithm then interprets the query, transcends into the tree to the appropriate node and delivers the requested information. More complicated approaches to program comprehension include control flow and data flow analysis.

Once the AST of a program has been constructed, it is possible to perform Control Flow Analysis (CFA) on it [6]. There are two major types of the CFA – *Interprocedural* and *Intraprocedural* analysis. Interprocedural analysis determines the calling relationship among program units while intraprocedural analysis determines the order in which statements are executed within these program units. Together they construct a *Control Flow Graph* (CFG). Interprocedural analysis proceeds by first identifying *basic blocks* in the program. A basic block is a collection of statements such that control can only flow in at the top and leave at the bottom either using a conditional or unconditional branch. These basic blocks are then represented as nodes in the CFG. Forward or backward arcs that represent a branch or a loop respectively indicate the flow of control. In intraprocedural analysis generates a *call graph* in which each routine connected to all the sub-routines it calls with downward arcs. When analyzing programs written in high-level languages like C, C++, Java etc., procedure parameters, pointers, and polymorphism may however prevent us from knowing which routine or method was being invoked until run-time, making CFA a less attractive approach.

Data Flow Analysis (DFA) is concerned with tracing a variable's use in a program by drawing arcs from its point of definition to every node where the variable is used. While interprocedural analysis is straightforward, intraprocedural analysis may pose several problems, for example, when a procedure is called with a pointer argument, which in turn is passed on to another procedure with a different name or alias.

A *Program Dependence Graph* (PDG) is a directed acyclic graph (DAG) whose vertices are assignment statements or predicates of an `if-then-else` or `while` constructs [7]. Different edges represent control and data flow dependencies. Control flow edges are labeled true or false depending on whether they enter a `then` block or an `else` block of the code. In other words, a PDG is a CFG and Data Flow Graph (DFG) integrated in one graph. PDG's can potentially get very complicated as the size of the program increases and unless are accurately constructed, traversal algorithms could run into infinite loops.

Slicing was introduced by Weiser [16] and has served an important basis for various program comprehension techniques. Weiser defines the "*slice* of a program for a particular variable at a particular line in the source code as that part of the code that is responsible for giving a value to the variable at that

point in the code". The idea behind slicing is to retrieve the code segment that has a direct impact on the concerned variables and nothing else. Starting at a given point in the program, program slicing automatically retrieves all relevant code statements containing control and/or data flow dependencies. The program slicer requires three inputs: (1) the *slicing criteria*, (2) *direction of slicing*, and (3) the annotated abstract syntax tree, which contains the control and data flow dependencies on it. Traditionally, the slicing criteria of a program *P* is a pair <*i*, *V*> where *i* is a program statement in *P*, and *V* is a set of variables referred to in statement *i* [16]. The other input to the program slicer is the direction of slicing, which could be either forwards or backwards. *Forward slicing* examines all statements between statement *i* and the end of the program. *Backward slicing*, examines all statements before statement *i* until the first statement of the program.

Situations where the up-to-date business logic is available in the code and through no other source, including the programmer's documentation of the code may very well arise in poorly maintained legacy systems. *Business Rules Extraction* (BRE) from legacy code is therefore an important problem and several solutions have been proposed for the same. Huang et al. [8] define a business rule as a function, constraint or transformation rule of an application's inputs to outputs. Formally, a business rule *R* can be expressed as a program segment *F* that transforms a set of input variables *I* to a set of output variable *O*. Mathematically, this can be represented as $O = F(I)$. The first step in BRE is the identification of important variables in the code that belong to set *O* or *I*. Huang et al. propose a heuristic for identifying these variables. They claim only the overall system input and output variables could be members of these two sets. They call these variables the *domain* variables, which in turn are the slicing variables. The direction of slicing is decided based on the following rule of thumb: If the slicing variable appears in an output (input) statement the direction of slicing is fixed as backwards (forwards) as it is likely that the business rules of interest will be at some point above in the code. The approach by Huang et al. successfully extracts business rules from the code but presents the business rules in language dependant code, that may involve specific and intricate features of the language, that might not easily understood by a managerial level employee.

Sneed and Erdos [14] adopt an entirely different approach for BRE. They argue that business rules are encoded in the form of assignments, results, arguments, and conditions as:

```
<result> ← assignment
(<arguments>)
IF (<conditions>)
```

Their BRE algorithm works as follows: first, the assignment statements are captured along with their location. Next, the conditions that trigger the assignments are captured by representing the decision logic in the code in a tree structure. Therefore the Sneed and Erdos approach reduces the source code to a partial program that only contains statements that affect the values of variables on the left hand side of assignment statements. Their approach suffers from some severely limiting assumptions: they assume that the expert user knows which variables are interesting and that all variables in the code have meaningful names.

Pattern matching identifies interesting patterns code patterns and their dependencies. Paul and Prakash [12] have implemented a pattern matcher by transforming source code and templates constructed from pre-selected patterns into AST's. Their approach has several advantages. Most important is the fact that patterns can be encoded in an extended version of the underlying language and that the pattern matching process is syntax directed rather than character based.

In the next section we describe the SEEK SA architecture and provide a stepwise description of the semantic analysis algorithm used to extract application-specific semantics and business rules from legacy source code.

3 Semantic Analysis Algorithm

A conceptual overview of the SEEK knowledge extraction architecture, which represents the build time component, is shown in Figure 1. SEEK applies Data Reverse Engineering (DRE) and Schema Matching (SM) processes to legacy databases, in order to produce a source wrapper for a legacy source. This source wrapper will be used by another component (not shown in Figure 1) for communication and exchange of information with the legacy source (run-time). It is assumed that the legacy source uses a database management system for storing and managing its enterprise data or knowledge.

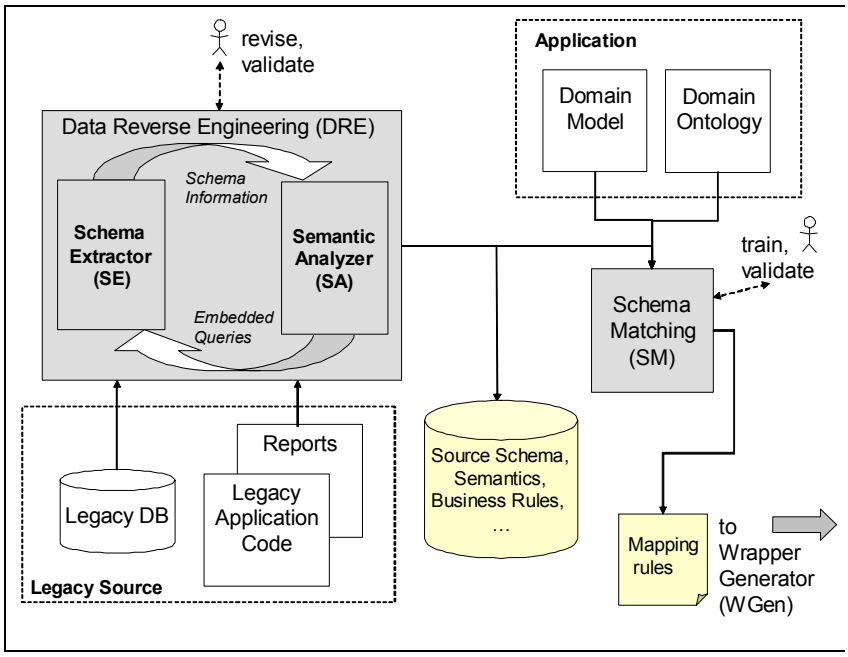

Figure 1: Schematic diagram of the conceptual architecture of SEEK's knowledge extraction algorithm.

First, SEEK generates a detailed description of the legacy source, including entities, relationships, application-specific meanings of the entities and relationships, business rules, data formatting and reporting constraints, etc. We collectively refer to this information as *enterprise knowledge*. The extracted enterprise knowledge forms a knowledgebase that serves as input for the subsequent steps outlined below. In order to extract this enterprise knowledge, the DRE module shown on the left of Figure 1 connects to the underlying DBMS to extract schema information (most data sources support at least some form of Call-Level Interface such as JDBC). The schema information from the database is semantically enhanced using clues extracted by the semantic analyzer from available application code, business reports, and, in the future, perhaps other electronically available information that may encode business data such as e-mail correspondence, corporate memos, etc. It has been our experience (through discussions with representatives from the construction and manufacturing domains) that such application code exists and can be made available electronically.

Second, the semantically enhanced legacy source schema must be mapped into the domain model (DM) used by the application(s) that want(s) to access the legacy source. This is done using a schema matching process that produces the mapping rules between the legacy source schema and the application domain model. In addition to the domain model, the schema matching module also needs access to the domain ontology (DO) describing the model.

Finally, the extracted legacy schema and the mapping rules provide the input to the wrapper generator (not shown), which produces the source wrapper.

The three preceding steps can be formalized as follows. At a high level, let a legacy source L be denoted by the tuple $L = (DB_L\ S_L, D_L, Q_{L,})$, where DB_L denotes the legacy database, S_L denotes its schema, D_L the data and Q_L a set of queries that can be answered by DB_L. Note, the legacy database need not be a relational database, but can include text, flat file databases, and hierarchically formatted information. S_L is expressed by the data model DM_L.

We also define an application via the tuple $A = (S_A, Q_A, D_A)$, where S_A denotes the schema used by the application and Q_A denotes a collection of queries written against that schema. The symbol D_A denotes data that is expressed in the context of the application. We assume that the application schema is described by a domain model and its corresponding ontology (as shown in Figure 1). For simplicity, we further assume that the application query format is specific to a given application domain but invariant across legacy sources for that domain. Let a *legacy source wrapper W* be comprised of a query transformation

$$f_W^Q : Q_A \mapsto Q_L \tag{1}$$

and a data transformation

$$f_W^D : D_L \mapsto D_A, \tag{2}$$

where the *Q's* and *D's* are constrained by the corresponding schemas.

The *SEEK knowledge extraction process* shown in Figure 1 can now be stated as follows. Given S_A and Q_A for an application wishing to access legacy database DB_L whose schema S_L is unknown. Assuming that we have access to the legacy database DB_L as well as to application code C_L accessing DB_L, we first infer S_L by analyzing DB_L and C_L, then use S_L to infer a set of mapping rules M between S_L and S_A, which are used by a wrapper generator *WGen* to produce (f_W^Q, f_W^D). In short:

$$DRE: (DB_L, C_{L,}) \mapsto S_L \tag{3}$$
$$SM : (S_L, S_A) \mapsto M \tag{4}$$
$$WGen: (Q_A, M) \mapsto (f_W^Q, f_W^D) \tag{5}$$

Thus, the DRE algorithm (equation 3) is comprised of schema extraction (SE) and semantic analysis (SA). This report will concentrate on the semantic analysis process by analyzing application code $C_{L,}$ thereby providing vital clues for inferring $S_{L.}$ The implementation and experimental evaluation of the DRE algorithm are described in [4] and will not be dealt with in detail in this report.

3.1 Algorithm Design

The objective of the application code analysis is threefold: (1) Augment entities with domain semantics, (2) extract queries that help validate the existence of relationships among entities in Step 2, and (3) identify business rules and constraints not explicitly stored in the database, but which may be important to the wrapper generator or application program accessing legacy source L. Our approach to code analysis is based on code mining, a combination of program slicing [16] and pattern matching [12]. However our fundamental goal is broader than that described in the literature by Huang et al. [8]. Not only do we want to extract business rules and constraints, we also want to discover application-specific meanings of the underlying entities and attributes in the legacy database. Hence the heuristics used by our algorithms are different from the heuristics proposed by Huang et al. and are tailored to SEEK's objectives. The following section lists the heuristics that form the basis of the SA algorithm.

3.1.1 Heuristics Used

The semantic analysis algorithm is based on several observations based on the general nature of legacy application code. Whether the application code is written for a client side application like an online ordering system or for resource management by an enterprise (e.g., a product re-order system manipulated by the employees), database application code always has queries embedded in it. The data retrieved or

manipulated by queries is displayed to the end user (client or enterprise employee) in a pre-defined format. Both the queries and the output statements contain rich semantic information.

Heuristic 1: Application code typically has report generation modules or statements that display the results of queries executed on the underlying database.

Typically, output statements display one or more variables and/or contain one or more *format strings*. A format string is defined as a sequence of alphanumeric characters and *escape sequences* within quotes. An *escape sequence* is a backslash character and followed by a sequence of alphanumeric characters (e.g., \n, \t etc), which in combination indicate how to align and format the output. For example, in the statement

```
System.out.println("\n Task cost:" + v);
```

the substring `"\n Task cost: "` represents the format string. The *escape sequence* "\n" specifies that the output should begin on a new line.

Heuristic 2: The format string in an input/output statement, if present, describes the displayed variable.

In other words, to discover the semantic meaning of a variable *v* in the source code, we have to look for an output (input) statements in which the variable *v* is displayed (accepted). Sometimes the format string that contains semantic information about the display variable *v* and the output statement that actually displays the variable *v* may be split among two or more statements. Consider following statements:

```
System.out.println("\n Task cost:");
System.out.println("\t" +  v);
```

Let us call the first output statement with the *format string* as *s1* and the second output statement that actually prints the value of the variable *s2*. Notice that, *s1* and *s2* can be separated by an arbitrary number of statements. In such a case, we would have to look backwards in the code from statement *s2* for an output statement that prints no variables but a text string only. The text string contains the *context meaning* or clues about the application specific meaning of the variable. A classic example of this situation in database application code is the set of statements that display the results of a SELECT query in a matrix or tabular format. The matrix title and the column headers contain important clues about the application specific meanings of variables displayed in the individual columns of the matrix.

Heuristic 3: If a diffedrent output statement *s1* displaying variable *v* has no format string (and therefore no semantics for variable *v* that can be extracted from *s*), then the semantic meaning or *context meaning* of *v* may be the format string of another output statement *s2* that only has a format string and displays no variables. Examining statements in the code backwards from *s1* can lead to output statement *s2* that contains the *context meaning* of *v*.

It is logical to assume that variable *v* should have been declared and defined at some point in the code before it is used in an output statement. Therefore if a statement *s* assigns a value to *v* and *s* is a statement that retrieves a particular column value from the result set of a query *q*, then *v*'s semantics can be associated to a particular column in *q* in the database.

Heuristic 4: If a statement *s* assigns a value to variable *v*, and *s* retrieves a value of a column *c* of table *t* from the result set of a query *q*, we can associate *v*'s semantics with column *c* of table *t*.

As Sneed and Erdos [14] rightly observed, business logic is encoded either as assignment statements or conditional statements like `if..then..else`, `switch..case`, etc. or a combination of them. Mathematical formulae translate into assignment statements while decision logic translates into conditional statements.

Heuristic 5a: If variable *v* is part of an assignment statement *s* (i.e. appears either on the left hand side or is used in the right hand side of the assignment statement), then statement *s* represents an expression involving variable *v*.

Heuristic 5b: If variable *v* appears in the condition expression of an `if..then..else` or `switch..case` or any other conditional statement *s*, then *s* represents a business rule involving variable *v*.

Typically, in legacy application code the statements that are of interest to us are distributed throughout the application code. Hence, extracting semantic information for a variable *v* may amount to making one full pass over the legacy application code. Additionally, a large subset of variables declared in the source code appear either in input, or output, or in *database* statements. Let us denote this subset of variables using the set *V*. We refer to the statements that extract individual column values from the result set of a query and those statements that execute the queries on the database as *database* statements.

If we attempted to mine semantic information for all the variables in set *V* in parallel, and in one single pass over the code, we face the risk of extracting either incomplete or potentially incorrect information due to the complexity of the process of extracting semantic knowledge. Hence, by limiting the number of passes of the source code to one, although the run-time complexity of the algorithm decreases, the correctness of the result may be jeopardized, which is not desirable.

Since the emphasis in SEEK is not so much on run-time efficiency, but rather on completeness and correctness, we adapt Weiser's *program slicing* approach [16] to mine semantic information from application code. The SEEK SA aims at augmenting entities and attributes in the database schema with their application-specific meanings. As already discussed, output (input) statements provide us with the semantic meaning of the displayed variable. Variables that appear on the left hand side of *database* statements can be mapped to a particular column and table accessed in the query. Hence, it is reasonable to state that the variables that appear in input/output or on *database* statements should be traced throughout the application code. We will call these variables *slicing variables*.

As we described in section 2, *program slicing* generates a reduced source code that only contains statements that use or modify the slicing variable. *Slicing* is performed by making a single pass over the source code and examining every statement in the code. Only those statements that contain the slicing variables are retained in the reduced source code.

Heuristic 6: The set of *slicing variables* includes variables that appear in input, output or *database* statements. This is the set of variables that will provide the maximum semantic knowledge about the underlying legacy database.

Heuristic 7: *Slicing* is performed once for each *slicing variable* to generate a reduced source code that only contains statements that modify or use the *slicing variable*.

The *program slicing* routine takes three inputs in addition to the source code itself: (1) slicing variable, (2) direction of slicing, and (3) constraint of termination condition for slicing. So far, we have discussed how to compose the set of slicing variables. The direction of slicing for a given slicing variable can be decided based on whether the slicing variable appears in an input, output or *database* statement. If the slicing variable appears in an input statement, it is logical to surmise that the value of the variable being accepted from the user will be used in statements below the current input statement. Hence, the statements of interest are below the input statement and the direction of slicing can be fixed as forward. On the other hand, if the slicing variable appears in an output statement, then the statements that define and assign values to that variable will appear above the current output statement in the code. Hence the direction of slicing is fixed as backward. The third kind of slicing variables are those that appear in *database* statements. Since these statements assign a value to the slicing variable, it is reasonable to assume that all statements that modify or manipulate this slicing variable or related statement will be below the current *database* statement in the code, with the exception of the statement that stores SQL query itself. Hence in this case neither forward nor backward slicing will suffice. Therefore, we adopt a combination of forward and backward slicing techniques, which we call *recursive slicing* to generate the reduced code. *Recursive slicing* proceeds as follows:

1. Perform backward slicing from the current *database* statement retaining all statements that use or modify the slicing variable, stopping only when a statement that defines the SQL SELECT query has been encountered in the code.
2. Append all statements below the current *database* statement in the code, to the program slice generated in step 1.
3. Finally, perform forward slicing from current *database* statement retaining only those statements that alter or use the slicing variable. This generates the final program slice.

The default termination condition for slicing whether forward, backward or recursive is the function or class scope. In other words, slicing is automatically terminated at the point when the slicing variable goes out of scope. We summarize these insights in the final four heuristics.

Heuristic 8a: The direction of slicing is fixed as forward if the slicing variable appears in an input statement and therefore only statements below this input statement in the source code, that contain the slicing variable will be part of the program slice generated.

Heuristic 8b: The direction of slicing is fixed as backward if the slicing variable appears in an output statement and therefore only statements above this output statement in the source code, that contain the slicing variable will be part of the program slice generated.

Heuristic 8c: If the slicing variable appears in a *database* related statement slicing must be performed recursively. The search for statements in the forward direction, that are part of the program slice, is bounded by the occurrence of an SQL SELECT query.

Heuristic 9: The termination criterion for slicing is determined by the scope of the variable. In other words slicing terminated at the point where the slicing variable goes out of scope.

The following section describes the steps of the semantic analyzer algorithm in detail.

3.1.2 Semantic Analysis Algorithm Steps

Application code for legacy database systems is typically written in high-level languages like C, C++, Java etc. In this report we discuss the implementation of a C and Java semantic analyzer. Not only does the C semantic analyzer serve as a good example of how to implement an SA for a procedural language like C, it also serves as a learning experience before proceeding to design and implement a semantic analyzer for object oriented languages like Java. The lessons learned from implementing the C semantic analyzer are useful in building the Java semantic analyzer for the following reasons:

1. The language grammar for statements like the `if..then..else`, `switch..case`, and assignment statements are similar in C and Java. Thus the business rule extraction strategy used in the C semantic analyzer can be reused in the Java semantic analyzer.
2. Queries that are embedded in legacy application code are written in SQL both in C and Java. Hence the module that analyzes queries need not be re-designed for the Java SA.

We now describe the six-step semantic analysis algorithm pictured in Figure 2. Semantic analysis begins by invoking the AST generator that uses the source code as input and generates and AST as output. Next, the pre-slicer module identifies the slicing variables by traversing the AST. Since the identification of the slicing variables logically precedes the actual program-slicing step, we call this module the 'pre-slicer' module. The code-slicer module, as the name suggests, generates the program slice corresponding to that slicing variable by retaining only those statements that contain the slicing variable. The primary objective of the analyzer module is to extract all the semantic information including data type, column and variable name, business rules, etc. corresponding to the slicing variable from the reduced AST. It stores the semantic information extracted into appropriate data structures used to generate semantic analysis result reports. Once semantic analysis has been performed on all slicing variables, the semantic analysis results data structure is examined to see if there is any slicing variable for which the analyzer was not able to clearly ascertain the semantic meaning of the slicing variable. Therefore, if an ambiguity in the meaning of a slicing variable is detected, the ambiguity resolver module

is invoked. The ambiguity resolver presents all the semantic information extracted for the slicing variable to the user and accepts the semantic meaning of the slicing variable from the expert user. Finally the result generator module compiles the semantic analysis results, generates a report that serves as an input to the knowledge encoder. We describe each of these six steps in detail.

Step 1: AST generation for the application code.

The SA process begins with the generation of an abstract syntax tree (AST) for the legacy application code. The following discussion references Figure 3, which is an expansion of the AST Generator representation shown in Figure 2. In Figure 3, the process flow on the left side is for building ASTs for C code, the one on the right side is for developing ASTs for Java code.

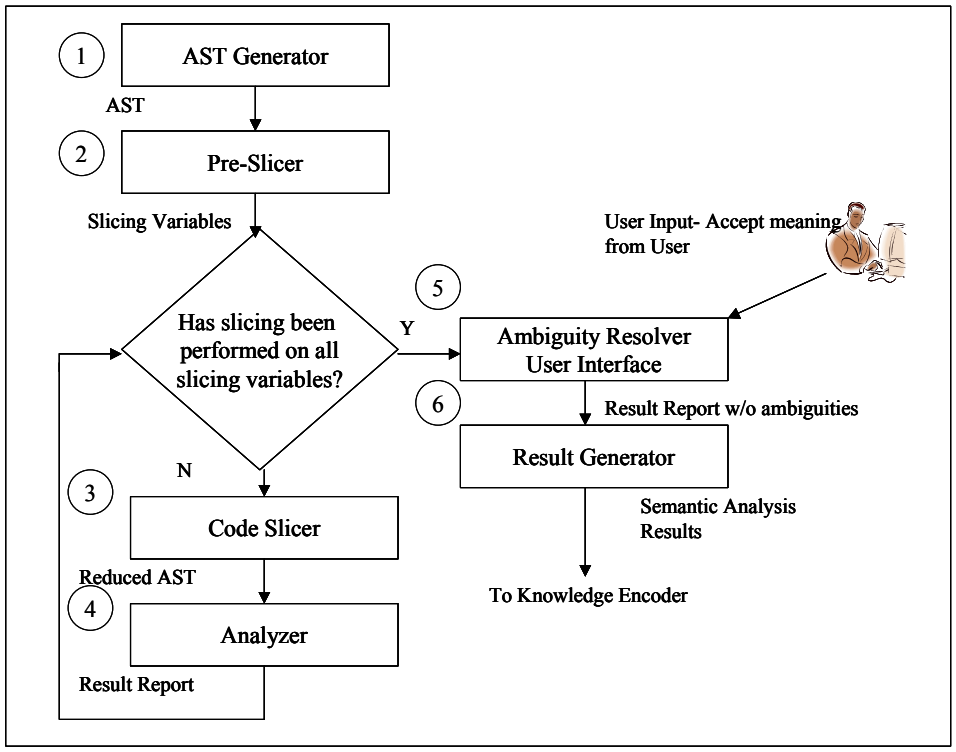

Figure 2: Semantic analysis implementation steps.

The AST generator for C code consists of two major components: the lexical analyzer and the parser. The lexical analyzer for application code written in C reads the source code line-by-line and breaks it up into tokens. The C parser reads in these tokens and builds an AST for the source code in accordance with language grammar (see Appendix A for a listing of the grammar for the C code that is accepted by the semantic analyzer). The above approach works well for procedural languages such as C. However, when applied directly to object oriented languages (e.g., Java), it greatly increases the complexity of the problem due to issues such as ambiguity induced by multiple inheritance, diversity resulting from specialization of classes and objects, etc.

As more and more application code is written in Java, it becomes necessary to develop an algorithm to infer semantic information from Java code. As previously implied, the grammar of an object-oriented language is complex when compared with procedural languages like the C language. Building a Java lexical analyzer and parser would require the parser to look ahead multiple tokens before applying the appropriate production rule. Thus, building a Java parser from scratch does not seem like a feasible solution. Instead, tools like *lex* or *yacc* can be employed to do the parsing. These tools generate N-ary AST's. N-ary trees, unlike binary trees, are difficult to navigate using standard tree traversal algorithms. Our objective in the AST generation is to be able to extract and associate the meaning of selected

partitions of application code with program variables. For example, format strings in input/output statements contain semantic information that can be associated with the variables in the input/output statement. This program variable in turn may be associated with a column of a table in the underlying legacy database. Standard Java language grammar does not put the format string information on the AST, which defeats the purpose of generating AST's for the application code.

The above reasons justify the need for an alternate approach for analyzing Java code. Our Java AST builder (depicted on the right-hand side of Figure 3) has four major components, the first of which is a code decomposer. In Java its possible that more than one class has been defined in the same source code file. The semantic analysis algorithm, which is based on the heuristics described above, takes a source code file that has just one class or file scope. Therefore, the objective of the Java source code decomposer is to decompose the source code into as many files as there are classes defined in it. It splits the original source code into a number of files, one per class and then passes these files one by one to the pattern matcher. The objective of the pattern matcher module is two-fold. First, it reduces the size of the application code being analyzed. Second, while generating the reduced application code file, it performs selected text replacements that facilitate easier parsing of the reduced source code.

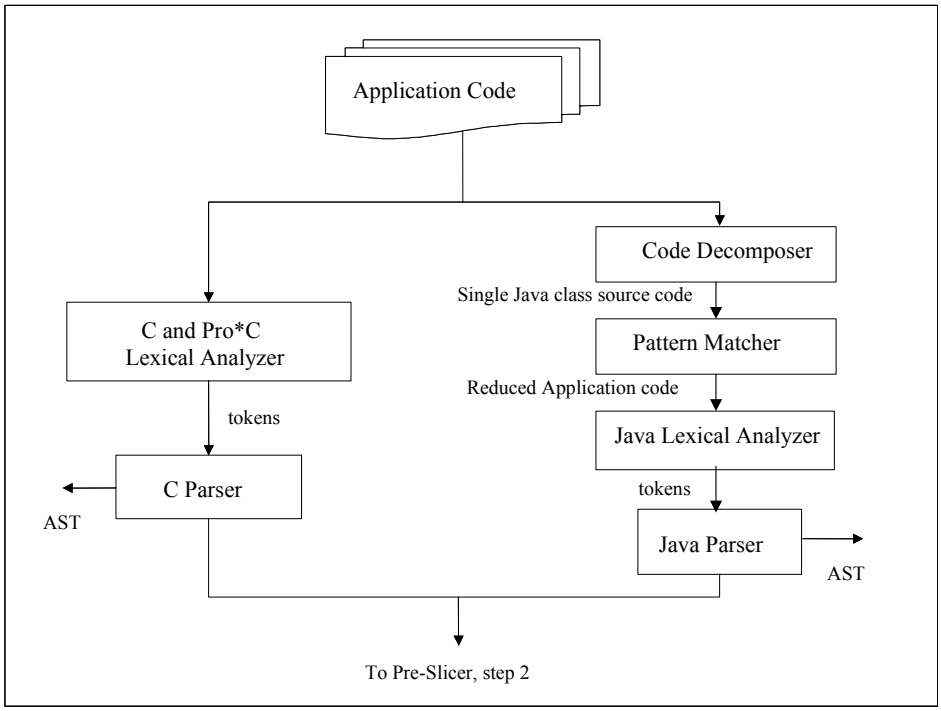

Figure 3: Generation of an AST for either C or Java code.

The pattern matcher works as follows: It scans the source code line by line looking for patterns such as `System.out.println` that indicate output statements or `ResultSet` that indicate JDBC statements. Upon finding such a pattern, it replaces the pattern with an appropriate pre-designated string. After this text replacement has been performed, the statement is closer in syntax to that of a procedural language. The replacement string is chosen based on the grammar of this Java like procedural language. For example, in the following line of code:

```
System.out.println("Task Start Date" + aValue);
```

the pattern `System.out.println` is replaced with `printf`, and following line is generated in a reduced source code file:

```
printf("Task Start Date" + aValue);
```

After one pass of the application code, the pattern matcher generates a reduced source code file that contains only JDBC and output statements, which more closely resemble a procedural language. Appendix B provides a listing of the grammar production rules for this C like language. In writing a lexical analyzer and parser for this reduced source code, we can re-use most of our C lexical analyzer and parser. The lexical analyzer reads the reduced source code line by line and supplies tokens to the parser that builds an AST in accordance with the Java language grammar.

Step 2: Pre-Slicer

The pre-slicer identifies the set of *slicing variables* i.e., the set of variables that appear in input, output and *database* statements as described in heuristic 7. The pre-slicer performs one pre-order traversal of the AST and examines every node corresponding to an input, output and *database* statement. The pre-slicer searches the sub-tree of these nodes and adds all the variables in the sub-tree to the set of slicing variables. The pre-slicer extracts the signature (name of function, return type, number of parameter, and data types of all the parameters) of all functions defined in the source code file.

Steps 3 through 5 are performed for every variable in the set of slicing variables. After analysis has been performed on all the slicing variables, step 6 is invoked.

Step 3: Code Slicer

The code slicer traverses the AST in pre-order and retains only those nodes that contain the slicing variable in their sub-tree. Each time the code slicer encounters a statement node, it searches the sub-tree of the statement node for the occurrence of the slicing variable. If the slicing variable is present, the code-slicer pushes the statement node (and therefore its sub-tree) onto a stack. After traversing all the nodes in the AST, the code-slicer pops out the nodes in the stack two at a time, connects them using the left child-right sibling notation of N-ary trees, and pushes the resulting binary tree back on to stack. Finally, the code slicer is left with just one binary tree in the stack that corresponds to the reduced AST or the program slice for the given slicing variable. The reduced AST is sent as an input to the Analyzer.

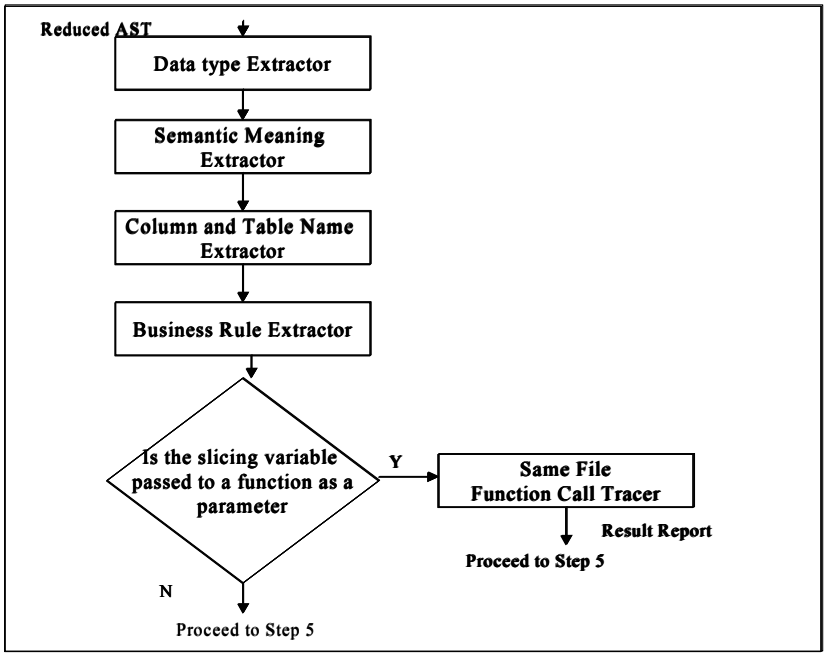

Figure 4: Sub-steps executed inside the analyzer module.

Step 4: Analyzer

Figure 4 shows a flowchart containing the sub-steps executed by the analyzer module. The analyzer traverses the reduced AST and extracts semantic knowledge for the slicing variable. The data type

extractor, searches the reduced AST for a *dcln* node to learn the data type of the slicing variable. The semantic meaning extractor searches the reduced AST for *printf/scanf* nodes. These nodes contain the mapping information from the text string to the identifier. In other words we can extract the contextual meaning of the identifier from the text string. The column and table name extractor searches the reduced AST for an *embSQL* node to discover the mapping between the slicing variable and a corresponding column name and table name in the database. The business rules extractor scan the reduced AST looking for *if*, *switch*, *assign* nodes that correspond to business rules involving the slicing variable.

Besides extracting the data type, meaning, business rules and database association of the slicing variable, the analyzer also checks to see if the slicing variable is passed to a function as a parameter. If the slicing variable is passed to a function as a parameter, then the analyzer invokes the *function call tracer*. The *function call tracer* executes the following steps:

1. The *function call tracer* records the name of function to which the variable is passed and the parameter position.
2. It sets a flag indicating that the a merge of the semantic knowledge discovered for the formal and actual parameter would be required after semantic analysis has been performed on all slicing variables for this file.

Finally, it adds the formal parameter corresponding to this slicing variable to the set of slicing variables gathered by the pre-slicer for this file.

It is important to note that unless the formal and actual parameter result records are merged, the knowledge discovered about a single semantic entity will exist in two separate semantic analysis result records. The three steps executed by the function call tracer are necessary for the following reason: The formal parameter may not be in the set of slicing variables identified by the pre-slicer. In that case, if the *function call tracer* did not add the formal parameter to the set of slicing variables, the business logic may never be discovered. Therefore, the semantic information extracted for the actual parameter may be incomplete, potentially incorrect. Situations where the business rules are abstracted into individual functions are common both in procedural and object-oriented languages.

Step 5: Ambiguity Resolver

The ambiguity resolver's primary function is to check the semantic information discovered for every slicing variable to see there is any ambiguity for the knowledge extracted. The ambiguity resolver detects an ambiguity if the meaning of the slicing variable is unknown, but the analyzer has been able to extract a *possible* or *context* meaning of the slicing variable as described in heuristic 3. The ambiguity resolver displays all the semantic knowledge discovered for the slicing variable including the *possible* or *context* meaning in a user interface and asks the user to enter the meaning of the slicing variable given all this information. This is the only step in the entire semantic analysis algorithm that is dependant on user input.

Step 6: Result Generator

The result generator has the following dual functionality. First, it merges the semantic knowledge extracted for the formal and actual parameters in a function call. Second, it replaces the slicing variables in the business rules with their application specific meanings thereby converting the business rules extracted into a source code-independent format. The merge algorithm executed by the result generator is an $O(N^2)$ algorithm, that iterates through all the *N* semantic analysis result records checking every record with the other *N-1* records to see if they represent a pair of formal and actual parameter records that need to be merged. Finally, the result generator writes all the discovered semantic knowledge to a file.

At the end of this six-step semantic analysis algorithm, control is returned to the schema extractor in the DRE algorithm. In the next section, we describe the Java semantic analyzer and justify the need for a more elaborate analyzer and result generator.

3.2 Java Semantic Analyzer

Most application code for databases written today is written in Java, making it important to verify that the semantic analysis algorithm is able to mine semantic information from application code written both in procedural languages like C and object-oriented languages like Java. Java is an object-oriented language with powerful features like inheritance, operator overloading and polymorphism. This means methods can now be invoked on objects either defined in Java's Application Program Interface (API) or on objects that may be user defined. Alternatively, the function call may be defined by a base class higher above in the inheritance hierarchy. The semantic analysis algorithm presented in previous section cannot handle such cases.

In order to take in account all the above-mentioned features of Java, we re-designed the analyzer and the result generator module of the semantic analyzer. Figure 5 depicts an enlarged view of the analyzer module and outlines the sub-steps executed inside the analyzer module.

The sequence of sub-steps executed inside the analyzer module remains unchanged in most cases. However, if the slicing variable is passed to a function as a parameter, then the steps executed in the Java SA result generator module are different. It becomes important to determine whether the method was invoked on an object or is simply a call to a function defined in the same file or in the base class. If the method is invoked on an object, the definition of the method is not present in the same source code file, unless it is an object of the object that we are examining. In that case, the source code decomposer ensures that the input to the Java SA is a file that has only one class scope. If the method was not invoked on an object, one of three cases can occur:

1. The definition of the method is present in the same file; or
2. The definition of the method is present in the base class; or
3. It is a call to a method in the Java library.

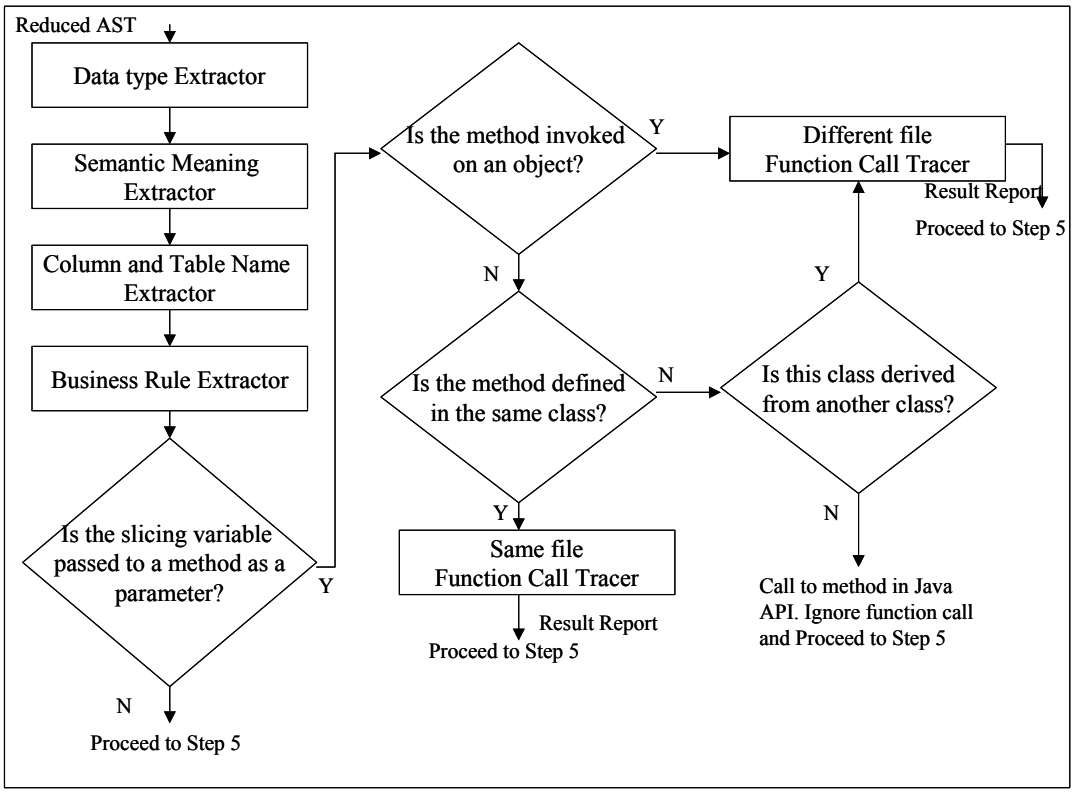

Figure 5: Sub-steps executed inside the Java SA analyzer module.

We will now analyze each of the three cases above with respect to their implications on the semantic analysis algorithm.

Case one generates two possibilities. If the method invoked is defined in the same file, the *same file function call tracer* is invoked. The same file function call tracer is identical to the function tracer in step 5 of the semantic analysis algorithm described in the previous section. However, if the method is not invoked on an object and the method name is not present in the list of methods defined in this file, then we can determine if it is a call to a method defined in the base class as follows: we check to see if the class we are analyzing is derived from any other class. If the class is not derived from any class, we can conclusively state that the method being invoked is a call to method in the Java API. If the class is indeed derived from another class, the possibility of the method being defined in the base class exists. Hence, we invoke the *different file function call tracer*. The *different file function call tracer* executes the following steps:

1. The *different file function call tracer* records the name of function to which the variable is passed and the parameter position.
2. It sets a flag indicating that the a merge of the semantic knowledge discovered for the formal and actual parameter would be required after semantic analysis has been performed on all source code files.
3. Finally, it adds the name of the function, and the parameter position of this slicing variable, and the name of the object on which this method is invoked (in this case the base class name) to the global set of slicing variables. The set of slicing variables for every source code file except the first source code file is the union of the set off slicing variables discovered by the pre-slicer for that individual file and the global set of slicing variables.

The case when a method is invoked on an object boils down to the case when the definition of the method is not present in the same file and can be handled in the exact same fashion by invoking the *different file function call tracer*.

The SA result generator has to be modified to support integration of semantic knowledge extracted in the analysis of multiple source code files. If for the result record of a particular slicing variable the flag that indicates that a merge is required across different semantic analysis result files has been set, additional semantic knowledge about the same physical entity is present in another results file that was generated by analyzing a different source code file. The object name tells us which result file to examine. The method name and the parameter position point to a particular result record in that file, whose results should be integrated with the current result record under consideration.

With the above described changes and additions to the semantic analysis algorithm, the SA will now be able to extract semantic information from source code written in Java. In the following sections we describe the implementation details of Java SA prototype and illustrate the major steps of semantic analysis using an example.

4 Implementation of the Java Semantic Analyzer

The current version of the Java SA prototype aims at extracting semantic information from application code and at tracing function calls with the same source code file. The SA prototype is implemented using the Java SDK 1.3 from Sun Microsystems. The prototype was tested with application code written in Java. In this section we use italics to introduce new concepts and slicing variable names. Nodes in the AST are represented by the node name in italics and are placed within single quotes. Class names, method and data members of classes, and built-in data types are highlighted using italicized courier font. Code statements and fragments are represented using the courier font.

4.1 Implementation Details

The driver program accepts the name of the source code file to be analyzed as a command line argument. The main method then invokes the Java Pattern Matcher and passes the name of the source code file to it as a parameter. The Pattern Matcher module generates a new reduced source code file by replacing pre-defined patterns with suitable text. The Pattern Matcher invokes the lexical analyzer and parser on the reduced code file. The parser generates the AST on which a series of pre-order traversal methods that perform the major steps of semantic analysis are invoked. The pre-slicer method returns a set of slicing variables. The code slicer and analyzer methods are invoked on the AST for each slicing variable which is passed as a parameter to both methods. Finally, the result generator method saves the extracted semantic knowledge to the semantic analysis results data structure. We now outline the implementation details of each step in the semantic analysis algorithm.

SA-1 AST Generator: The Java Pattern Matcher scans the source code file looking for *pre-defined patterns* or *pre-defined pattern generators*. *Pre-defined patterns* include output, declaration, and *JDBC patterns*. For example, the text string '`System.out.println`' is a *pre-defined output pattern*. *JDBC patterns* include database connectivity statements and query execution statements, methods and objects. If the Pattern Matcher encounters a *pre-defined pattern*, it performs appropriate text substitutions and writes out the new reduced source code file it generates. In object-oriented languages like Java, objects can be instantiated and methods invoked on these objects. A method invocation on an object may have the same functionality as one of the pre-defined patterns. Hence it is important to be able to trace such method invocations on objects and replace them with appropriate text. The object, on which the method is invoked, is referred to as a *pre-defined pattern generator*. The Pattern Matcher adds the object instance and method combination to the list of *pre-defined patterns*. For example, consider the following statements:

```
PrintWriter p = new PrintWriter(System.out);
p.println("Task End Date");
```

are functionally equivalent to the statement:

```
System.out.println("Task End Date");
```

Here, `p` is an instance of the object of type `PrintWriter`. The object `PrintWriter` is the *pattern generator* and `p.println` is an output pattern we will henceforth look for in the code. We append `p.println` to array of output patterns in the output patterns data structure. Therefore, when the Java Pattern Matcher reads the line:

```
p.println("Task End Date");
```

it recognizes that `p.println` is a *pre-defined output pattern* and re-writes the line to the modified file as:

```
printf("Task End Date");
```

The goal of the Pattern Matcher is to generate a reduced source code file that is closer to a procedural language like C. Hence all declaration statements involving the *new* operator have to be re-written like a C style declaration statement without the *new* operator. The Pattern Matcher stores every new user-defined class into a data structure thereby making the identification of lines in the source code that declare objects of pre-defined or built-in data types a simple lookup operation on the data structure maintained. The Pattern Matcher also maintains the value of the string variables at every point in the code in a simple data structure that tracks the value of a string at any given point in the code. The Pattern Matcher uses this data structure to regenerate queries that have been composed in several stages as a combination of string variables and text strings using the overloaded addition (+) operator for strings.

The Java Lexical Analyzer reads the reduced source code file generated by the Java Pattern Matcher and tokenizes it. The tokens are sent to the Java parser that applies the appropriate production rule from the language grammar and generates a sub-tree that corresponds to that statement. Therefore,

the root node of the sub-tree corresponds to a statement in the code and has additional information like the actual starting and ending lines and column numbers of the statements in the source code. The parser pushes these sub-trees onto a stack as it generates them. After the Parser has parsed the last line in the reduced source code, it begins to construct the AST. The sub-trees are popped two at a time from the stack and connected using the left child-right sibling representation of a N-ary tree as a binary tree. The resulting binary tree is pushed back onto stack and this operation is repeated till there is only one tree left in the stack. This binary tree represents the AST of the modified source code.

SA-2 Pre-Slicer: The step is implemented as a method that performs a pre-order traversal of the AST, marking nodes it has visited while trying to identify a list of slicing variables. When it encounters a '*printf*', '*embSQL*', or '*scanf*' node that corresponds to an output, SQL or input statement respectively in the code, it performs a pre-order traversal of this statement node. If it finds an '*identifier*' node in the sub-tree which corresponds to the occurrence of a variable in that statement, it appends the '*identifier*' node's left child, which has the actual variable name, to the list of slicing variables. The list of slicing variables is maintained as an array of `String` in memory. Lastly, the pre-slicer marks the identifier node as visited.

The other task that the pre-slicer accomplishes is to compose a list of methods defined in the source file. If the pre-slicer encounters a '*function*' node, it traverses the sub-tree of the '*function*' node and it appends the name of the method, number of parameters, return type of the function, and parameter list to the data structure.

SA-3 Code Slicer: The step is implemented by performing a pre-order traversal of the AST. The code slicer examines every node in the tree that corresponds to a statement. If the slicing variable is one of the nodes in the sub-tree of the statement node, then the code slicer takes the statement node and disconnects it from its parent and sibling nodes in the tree and pushes it into a stack. At the end of the pre-order walk of the entire AST, the stack contains only those statements nodes that have the slicing variable present in them. A reduced AST is constructed using the same approach as the Java Parser in step 1 uses to construct the AST.

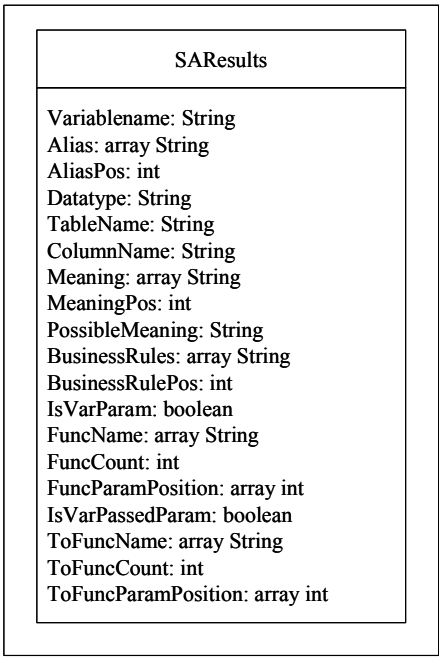

Figure 6: Semantic analysis results data structure.

SA-4 Analyzer: While traversing the reduced AST in pre-order, if the analyzer encounters a '*dcln*' node, which corresponds to a declaration of a variable in the source code, it extracts the data type of the variable

and saves it to the semantic analysis results data structure. If the analyzer encounters either an '*assign*', '*if*', or '*switch*' node on the reduced AST, which correspond to either a assignment statement, `if..then..else` statement, or a `switch` statement respectively, it executes the two steps described below to extract the corresponding business rule. First, using the line and column numbers stored in the statement node, it retrieves the statements corresponding to this node in the reduced AST from the source code file, and assigns it to the data member that stores the business rules in the semantic analysis results data structure. Second, every occurrence of the variable name in the business rule is replaced by its meaning. The step transforms the business rule extracted into a code-independent format. '*embSQL*' nodes contain the mapping information from an identifier name to corresponding column and table name in the database.

The semantic analysis results data structure or *SAResults* data structure shown in Figure 6 stores semantic knowledge extracted for each slicing variable. The meaning and business rules are defined as an array of String as there may be more than one meaning or business rule that can be associated with a slicing variable. If the slicing variable is passed to a method as a parameter, then the name of the function and parameter position is saved in the *SAResults* data structure in *ToFuncName* and *ToFuncParamPosition* data members respectively. A slicing variable may be passed as a parameter to more than one function. Hence both *ToFuncName* and *ToFuncParamPosition* are defined as arrays. If the slicing variable itself is defined in the formal parameter list of a function definition, then the name of the function and parameter position are stored in *FuncName* and *FuncParamPosition* data members of the *SAResults* data structure. *Alias* is an array of String, used to the store the formal parameter variable names corresponding to a variable, if there are. The rest of the members of the semantic analysis results data structure are self- explanatory.

SA-5 Ambiguity Resolver: If the meaning of a variable is not known at the end of step 5, we present the information gathered about the slicing variable including the data type, column and table name in the data base, business rules, and the context or possible meaning of the variable in a Java swing interface. The user is prompted to enter the meaning of the variable given this information. The meaning entered by the user is saved to the *SAResults* data structure.

SA-6 Result Generator: The primary objective of the result generator is to iterate through all the records of the *SAResults* data structure and merge the records corresponding to the formal and actual parameter. Two records *i* and *j* in the array of *SAResults* result records are merged only if the *ToFuncName* field of *i* is identical to the *FuncName* field of *j*, the *ToFuncParamPosition* field of *i* is identical to the *FuncParamPosition* of *j*, and both *isVaramParam* of *j* and *isVarPassedParam* of *i* are both true. This condition verifies that record *i* corresponds to the actual parameter and record *j* corresponds to the formal parameter. The variable name corresponding to entry *j* is saved as an alias of the variable corresponding to entry *i*. In the next section, we illustrate the SA process using an example.

4.2 Illustrative Example

In this section we will use the source code listed in Appendix C to simulate the Java SA prototype stepwise. The test code has been written in Java SDK version 1.3 from Sun Microsystems. The test code has been written for a manufacturing domain database. It contains queries and business rules that would typically be embedded in application code written for manufacturing domain databases. The test code first establishes a JDBC connection to the underlying Oracle database. After the connection has been established, a query is executed on the underlying database to extract the project start date, project finish date and cost for a certain project with name 'Avalon'. The code also contains a business rule which checks to see if the total project cost is over a certain threshold, and if so offers a 10% discount for such projects. The project cost in the underlying database is updated to reflect the discount. The task start date, finish date and unit cost for all the tasks of this project that have the name 'Tiles' are extracted. For each

task, the task unit cost is raised by 20% if the number of days between the start and end of the task is less than ten. Also the code ensures that the start and end of the individual tasks are well within the project start and end dates. We now simulate the various steps in semantic analysis for this given test code.

Step 1: AST generation

The Java Pattern Matcher generates the reduced source code as listed in Appendix D. The Java lexical analyzer and parser construct the AST for this reduced source code file. The AST of the reduced source code is as listed in Appendix E. Each line in the AST represents a node in the AST and the number of periods in the beginning of each line of the AST, denotes the level of that node in the AST. The N-ary tree corresponding to this AST can be visualized by taking a mirror image of the tree printed in this format and inverting it.

Step 2: Pre-Slicer

As described in the previous section, the pre-slicer's task is two-fold. First, it generates a list of slicing variables. Second, it maintains a list of all methods defined in the source file and their signatures. Table 1 shows the information maintained by the pre-slicer for slicing variables. Table 2 highlights the information maintained by the pre-slicer for methods defined in the same source file.

Table 1: Information maintained by the pre-slicer for slicing variables.

Slicing Variable	Type of Statement	Direction of Slicing	Text String (only for print nodes)
pfinish	Output	Backwards	"Project Finish Date for Avalon "
pcost	database	Recursive	-----
tfinish	output	Backwards	-----

Table 2: Signatures of methods defined in the source file maintained by the pre-slicer.

Method Name	Return Type	Number of Parameters	Parameter List
CheckDuration	float	3	Date, date, float
checkifValidDate	void	1	Date

Steps 3 through 6 are executed for each slicing variable. We will illustrate steps 3 through 6 for slicing variable *tfinish*.

Step 3: Code Slicer

The code slicer generates the reduced AST as shown in Figure 7. The reduced AST is constructed by retaining only those statement nodes in the original AST in which the slicing variable *tfinish* occurs some where in the sub-tree of that statement node.

```
--------REDUCED AST--------
program
. dcln(2)
. . <identifier>(1)
. . . Date(0)
. . =(2)
. . . <identifier>(1)
. . . . tfinish(0)
. . . rhscall(2)
. . . . <identifier>(1)
. . . . . getDate(0)
. . . . <string>(1)
. . . . . "Task_Finish_Date"(0)
. assign(2)
. . <identifier>(1)
. . . tcost(0)
. . rhscall(4)
. . . <identifier>(1)
. . . . checkDuration(0)
. . . <identifier>(1)
. . . . tstart(0)
. . . <identifier>(1)
. . . tfinish(0)
```

```
. if(2)
. . or(2)
. . . <(2)
. . . . rhscall(1)
. . . . . <identifier>(1)
. . . . . . tstart.getDate(0)
. . . . rhscall(1)
. . . . . <identifier>(1)
. . . . . . pstart.getDate(0)
. . . >(2)
. . . . rhscall(1)
. . . . . <identifier>(1)
. . . . . . tfinish.getDate(0)
. . . . rhscall(1)
. . . . . <identifier>(1)
. . . . . . pfinish.getDate(0)
. . block(1)
. . . emptyprintf(1)
. . . . <string>(1)
. . . . . "The task start and
finish dates have to be within
the project start and finish
dates"(0)
---------------------------
```

Figure 7: Reduced AST generated by the code slicer for slicing variable *tfinish*.

Step 4: Analyzer

The analyzer traverses the reduced AST and extracts semantic information for the slicing variable *tfinish*. The information extracted by the analyzer is shown in Table 3. The analyzer stores the semantic knowledge extracted in the `SAResults` data structure.

Table 3: Semantic knowledge extracted for slicing variable *tfinish*.

Variable Name	Tfinish
Data type	Date
Alias	----
Table Name	MSP_Tasks
Column Name	Task_Finish_Date
Meaning	----
Possible Meaning	Finish Date of Task Start Date of Task Unit Cost for Task
Is variable defined as a function parameter	No
Function Name	-----
Function Parameter Position	-----
Is variable passed as parameter	Yes
To Function Name	CheckDuration
To Function Parameter Position	2
Business Rules	if ((tstart.getDate() < pstart.getDate()) || (tfinish.getDate() > pfinish.getDate())) System.out.println("The task start and finish dates have to be within the project start and finish dates");

Step 5: Ambiguity Resolver

If the meaning of the slicing variable is not known at the end of step 5, the ambiguity resolver is invoked. The ambiguity resolver presents the semantic information extracted for the slicing variable, along with any possible or context meaning to the expert user, and accepts the meaning of the slicing variable *tfinish* from the user. Figure 8 shows a screen snapshot of the ambiguity resolver user interface.

Figure 8: Screen snapshot of the ambiguity resolver user interface.

Step 6: Result Generator

The result generator detects that a merge will be required to integrate the semantic knowledge discovered for the slicing variable *tfinish* as it has been passed to another method in the source code[2]. The `SAResults` record corresponding to the formal parameter is found by searching for a *SAResults* record

[2] The 'Is variable passed as parameter' field is set to 'yes'.

that has the exact same value in the fields corresponding to the function name and function parameter position as the slicing variable has in the its `ToFuncName` and `ToFuncParamPosition` fields. Table 4 shows the semantic information extracted for the formal parameter *t*. Table 5 shows the semantic information for the variable *tfinish* after the semantic knowledge of the formal and actual parameter have been merged.

Table 4: Semantic information gathered slicing variable *t*.

Variable Name	T
Data type	Date
Alias	----
Table Name	----
Column Name	----
Meaning	----
Possible Meaning	----
Is variable defined as a function parameter	Yes
Function Name	CheckDuration
Function Parameter Position	2
Is variable passed as parameter	No
To Function Name	----
To Function Parameter Position	----

Business Rules	if (s.getDate() - t.getDate() < 10) {
	revisedcost = f + f * 20/100;
	System.out.println("Estimated New Task Unit Cost : " + revisedcost); }
	else {
	revisedcost = f; }

Table 5: Semantic information for variable *tfinish* after the merge operation.

Variable Name	Tfinish
Data type	Date
Alias	T
Table Name	MSP_Tasks
Column Name	Task_Finish_Date
Meaning	Task End Date
Business Rules	if ((tstart.getDate() < pstart.getDate()) || (tfinish.getDate() > pfinish.getDate())) System.out.println("The task start and finish dates have to be within the project start and finish dates");

	if (s.getDate() - t.getDate() < 10) {
	revisedcost = f + f * 20/100;
	System.out.println("Estimated New Task Unit Cost : " + revisedcost); }
	else {
	revisedcost = f; }

5 Qualitative Evaluation of the Java Semantic Analyzer Prototype

In this section, we will use code fragments from the source code listed in Appendix C to highlight and demonstrate important features of the Java programming language that the Java SA prototype can accurately capture. In Java, the tuples that satisfy the selection criteria of an SQL SELECT query are returned in a `resultSet` object. The Java Database Connectivity (JDBC) Application Program Interface (API) [33] provides several `get` methods for `resultSet` objects to extract individual column values from a tuple in the `resultSet`. The parameter of a `resultSet get` method can either be a string or an integer. The string parameter has to a column name from the SELECT query column list while the integer parameter has to be an integer between zero and the number of columns in the SELECT query minus one. The two scenarios in Figure 9 highlight the types of parameters that can be passed to a `resultSet get` method.

SA Feature 1: The Java SA can accurately extract the table name and column name from a SQL SELECT query that corresponds to the slicing variable even if the column number instead of the column name was specified as the parameter in the `resultSet get` method.

Scenario A:

```
String query = "SELECT Task_Start_Date, Task_Finish_Date, Task_UnitCost FROM
MSP_Tasks WHERE Task_Name = 'Tiles'";

ResultSet rset = stmt.executeQuery(query);

Date tstart = rset.getDate("Task_Start_Date");
```

Scenario B:

```
String query = "SELECT Proj_Start_Date + 1, Project_Finish_Date -1,
Project_Cost FROM MSP_Projects WHERE Proj_Name = 'Avalon'";

ResultSet rset = stmt.executeQuery(query);

Date pstart = rset.getDate(0);
```

Figure 9: Code fragment depicting the types of parameters that can be passed to a `resultSet get` method.

In Scenario A, in Figure 9, the Java SA extracts the column name that the slicing variable *tstart* corresponds to by extracting the string parameter sent to the `resultSet get` method. If the `resultSet get` method parameter is an integer, the Java SA extracts the corresponding column name by moving *n* levels down to the right in the sub-tree corresponding to the column list of the SQL SELECT query.

An SQL SELECT query's column list is defined as list of comma separated mathematical expressions in the language grammar. Scenario B in Figure 9 is an example of a SELECT query where the column names are used in mathematical expressions instead of being specified directly.

SA Feature 2: The Java SA can map the slicing variable to the corresponding column name in the SQL SELECT query even if the column name is embedded in a complex mathematical expression.

The Java SA determines the column name corresponding to the variable *pstart* in two steps. First, it locates the first child of the SELECT query '*columnlist*' node. This node represents the sub-tree corresponding to the mathematical expression `Proj_Start_Date + 1`. In the second step, the Java SA accurately identifies the column name by searching for a previous undeclared identifier in the sub-tree of the mathematical expression. This strategy ensures that the Java SA can always extract the column

name without getting confused by the presence of other variables, integers and operands in the expression.

One of the most powerful features of object-oriented languages like Java is operator overloading. A classic example of operator overloading in Java is the overloaded '+' operator for strings. In Java, queries are executed by passing the SQL query as a parameter of type of string to either the *execute* or *executeQuery* methods, which are defined for *Statement* and *PreparedStatement* objects. The query string itself can be composed in several stages using the string concatenation (+) operator as shown in the code fragment in Figure 10.

```
stmt.executeUpdate("UPDATE MSP_Tasks SET Task_UnitCost = " + tcost + "
WHERE Task_Start_Date = '" + tstart + "' AND Task_Finish_Date = '" +
tfinish + "' ");
```

Figure 10: SQL query composed using the string concatenation operator (+).

SA Feature 3: The Java SA can capture the semantics of the string concatenation (+) operator.

The Java SA enables this feature by monitoring the value of string variables at every point in the code. Therefore, the Java SA regenerates an SQL query composed in stages using the string concatenation operator by simply substituting the string variables with its value at that point in the code.

In Java, output methods like *print* and *println* accept a string parameter and display the string content. This makes it possible to have a situation where an output statement displays only string variables and no format or text strings in the same statement. The string variables in turn may have been assigned values in a series of one or more assignment statements prior to their use in the output statement. We define such output statements *indirect output* statements.

```
String displayString;
displayString = "Project Start Date " + pstart;
System.out.println(displayString);
```

Figure 11: Code fragment demonstrating *indirect output* statements.

SA Feature 4: The Java SA can capture semantics hidden in *indirect output* statements.

Figure 5-3 depicts an example of an *indirect output* statement. The Java SA discovers the meaning of the variable *pstart*, which might not have been extracted if this powerful feature was not built into the Java SA. Semantic information hidden behind *indirect output* statements are extracted by parsing the right hand side of all assignment statements whose left hand side is a string variable.

The format string is an output statement in Java is a combination of text that contains the semantic meaning of the output variable and escape sequences used to position or align the output. In some situations however, it is necessary to split the format string between two or more output statements. One output statement has the semantic meaning of the output variable and the other has the escape sequences for alignment of the output variables on the standard output. A common example of such a situation in code occurs when displaying data stored in an object or array in a tabular format. Rich semantic clues are embedded in the output statements that display the title or heading of each column or the table itself. These format strings of such output statements contain clues to the meaning of the output variables in the given context. Hence we define it as the context meaning of the output variable. This is especially

important when the format string corresponding to the output variable is made up only of escape sequences that shed little light on the meaning of the variable.

SA Feature 5: The Java SA can extract *context* meanings (if any) for variables.

When the Java SA encounters an output statement with no format string, the Java SA examines statements before the output statement until it encounters an output statement that only has a format string and displays no variable. The Java SA extracts this as the possible meaning of the variable and presents the information as guideline to the expert user to resolve any ambiguities. The result of this search for possible meaning is not affected by the presence of any number of statements in between. For example, consider the code fragment shown in Figure 12. The Java SA cannot extract any meaning for the variables *tfinish*, *tstart* and *tcost*. However, the semantic clues embedded in the output statement that serves as a title for the tabular display of data is captured by the Java SA as the *context* meaning of these variables (notice, the Java SA intelligently disregards output statements that have a format string made up of non-alphanumeric characters only). The Java SA extracts the string '`Finish Date of Task Start Date of Task Unit Cost for Task`' as the context or possible meaning for the variables *tfinish*, *tstart* and *tcost*.

```
System.out.println("Finish Date of Task Start Date of Task Unit Cost for Task");
System.out.println("------------------------------------------------------------");
while (rset.next())
{
   Date tstart = rset.getDate("Task_Start_Date");
   Date tfinish = rset.getDate("Task_Finish_Date");
   float tcost = rset.getFloat("Task_UnitCost");
   tcost = checkDuration(tstart, tfinish, tcost);
   stmt.executeUpdate("UPDATE MSP_Tasks SET Task_UnitCost = " + tcost + "
   WHERE Task_Start_Date = '" + tstart + "' AND Task_Finish_Date = '" +
   tfinish + "' ");
   System.out.print(tfinish);
   System.out.print("\t" + tstart);
   System.out.println("\t" + tcost);
}
```

Figure 12: Code fragment demonstrating *context* meaning of variables.

SA Feature 6: The Java SA can capture business rules involving method invocations on variables.

Since the Java parser treats all method invocations on objects as a simple function call (it ignores the fact that the method was invoked on an object), the Java SA parses the method name to learn if the method was in fact invoked on a pre-defined variable or object. If this approach were not adopted, then the business rule in Figure 13 would not have been discovered for slicing variable *tstart*. Note that code segment in Figure 13 comes from a method.

```
if ((tstart.getDate() < pstart.getDate()) ||
(tfinish.getDate() > pfinish.getDate()))
{
System.out.println("The task start and finish dates have
to be within the project start and finish dates");
}
```

Figure 13: Business rules involving method invocations on slicing variables.

The an important aspect of object-oriented languages like Java is to encapsulate all the data manipulation statements into individual function such that each function has a specific functionality. Consequently, the application code written in these languages will contain a sequence of function calls with variables being passed to these functions as parameters. Therefore, potentially the semantic knowledge (business rules and meaning) for a single physical entity or variable may be distributed among several functions. Tracing each of these function calls would generate a comprehensive report of the semantics of the slicing variable.

SA Feature 7a: Function calls are traced i.e. if a slicing variable is passed as a parameter to a method defined within the same file, then the semantic information gathered for the formal and actual parameter is integrated.

The Java SA captures parameter passing and traces function calls by recording the name of each function that a variable is passed to along with the rest of the semantic knowledge discovered for that variable.

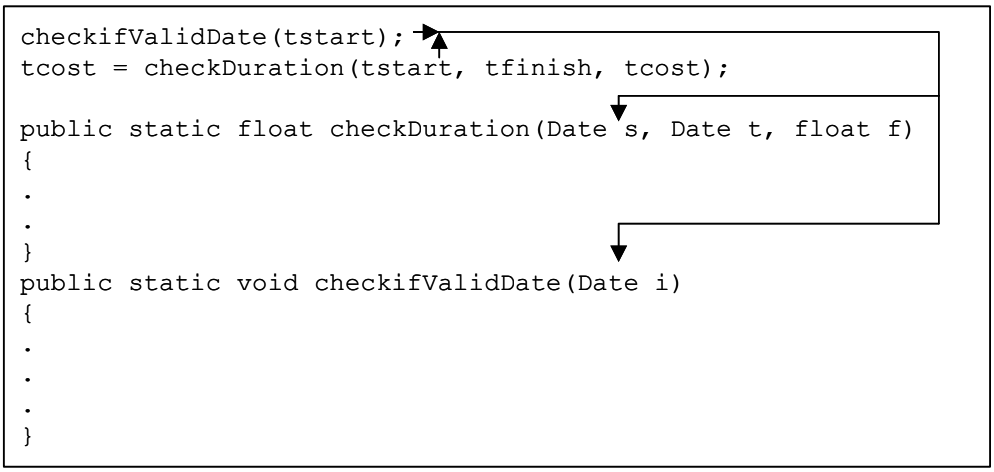

Figure 14: Code fragment illustrating an example where the slicing variable tstart is passed to two functions.

SA Feature 7b: The same variable may be passed to more than one function as a parameter. The Java SA can capture and integrate the semantic knowledge extracted for the actual parameter and all its associated formal parameters. In the code fragment shown in Figure 14, the Java SA traces the slicing variable *tstart* to two different methods `checkDuration` and `checkIfValidDate` and merges the semantic knowledge extracted for the actual parameter *tstart* and both its associated formal parameters *s* and *i*.

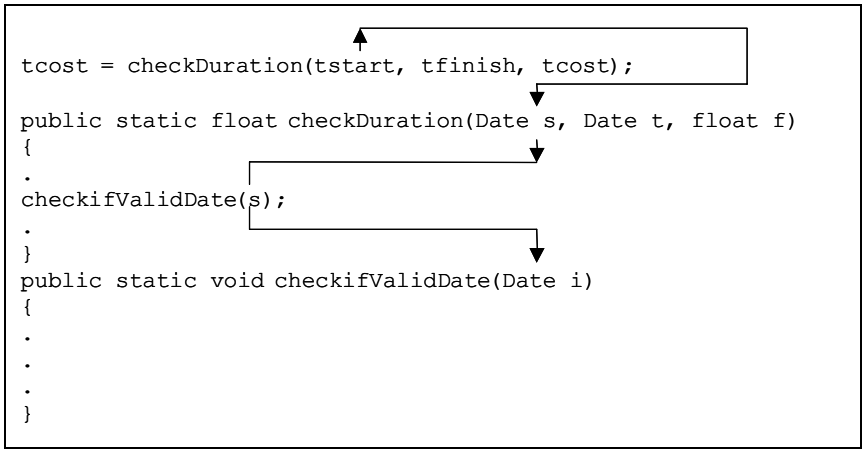

Figure 15: Code fragment illustrating parameter chaining.

Figure 15 demonstrates another interesting scenario where the slicing variable *tstart* is passed to function `checkDuration` and its value is received in formal parameter *s*. The variable *s* is in turn passed to another function `checkifValidDate` and its value received in variable *i*. The same variable is passed from one function to another in a chain of function calls, a situation we term *parameter chaining*. *Parameter chaining* occurs when a variable passed as a parameter to function *f1* is passed again from function *f1* to another function *f2* as a parameter. The Java SA can recognize and integrate semantic information extracted in such situations.

SA Feature 7c: The Java SA can capture *parameter chaining*.

Parameter chaining is captured using a sophisticated merge algorithm in the result generator module of the Java SA. The current Java SA prototype can extract semantic knowledge from application code that directs its output to the standard output, which is one of the many ways to display data in Java. However, if we wanted our Java SA to be able to extract semantic information from Java Servlets, we would have to do the following:

1. Add a new pattern to the Java Pattern Matcher to identify and modify output statements in Java Servlets. Output statements in Java Servlets have the format string and output variables embedded inside HTML source code.
2. Plug-in HTML parsers into the Pattern Matcher to extract the format string embedded in HTML source code and re-write the output statement like a regular output statement.

The rest of the semantic analyzer modules need not be modified to capture the semantic information from Java Servlets.

SA Feature 8: The Java SA prototype design is extensible and can capture semantics from new Java technologies by plugging in appropriate patterns and parsers into the Pattern Matcher with minimal modification to the actual semantic analyzer modules. The approach used to extract semantic information does not have to be re-engineered each time a different kind of input source code has to be analyzed.

We have highlighted some of the important features of the Java SA prototype that clearly demonstrate that the Java SA can extract semantic information from application code written in Java with minimal user input. Not only can the Java SA capture application-specific meanings of entities and attributes, it can also extract business rules dispersed in the application code. As demonstrated the Java SA is able to capture the semantics of overload operators and parameter chaining. The strength of the Java SA prototype lies in its extensible and modular design, making it a useful and easily maintainable toolkit for knowledge discovery and extraction.

6 Conclusion and Future Work

In this report we have provided a general solution for the semantic analysis problem. Our algorithm examines the application code using a combination of several program comprehension techniques and extracts semantic information that is explicitly or implicitly present in the application code. The semantic knowledge extracted is documented and can be used for various purposes such as schema matching and wrapper generation, code improvement, code documentation effort etc. We have manually tested our approach with application code written in ANSI C and Java to validate our semantic analysis algorithm and to estimate how much user input is required. The following section lists the contribution of this work and the last section discusses possible future enhancements.

6.1 Contributions

The most important contributions of this work are the following. First, a broad survey of existing program comprehension and semantic knowledge extraction techniques was presented in Chapter 2. This overview not only updates us with the knowledge of different approaches, but also provides a significant guidance while developing the SA algorithm.

The second major contribution is the design and implementation of a semantic analysis algorithm, which puts minimum restrictions on the input (application code), is as general as possible in design, and extracts the maximum possible knowledge as possible from all the code files, with minimal external intervention.

Third, a different and new approach is presented for mining the *context meaning* of variables that appear in the application code. Fourth, an approach is presented on how to map a particular column of a table in the underlying database to its application specific meaning that is extracted from the source code. The fifth and major contribution is the approach used to extract business rules from application code and present them in a code-independent format.

The most significant contribution of the semantic analysis algorithm is its easily extensible design. The algorithm can be easily configured and extended to mine semantic information from a new Java technology by simply plugging the corresponding modules to the pattern matcher, which is a preliminary step in the semantic analysis algorithm. Only minimal changes to the core semantic analysis algorithm and modules are required. It is also important to note that the semantic analysis algorithm proposed can be used to mine application code written in procedural and object-oriented languages. If a source code in a language different from Java or ANSI C is presented to the SA, only a new pattern matcher module will have to be plugged in. Also, the complexity of the semantic analysis algorithm does not increase exponentially with the features of the language it was built for. For example, the Java SA algorithm complexity both in terms of run-time and algorithm design does not increase significantly with the features like polymorphism, inheritance and operator-overloading etc. that it has to capture.

One of the more significant aspects of the prototype we have built is that is highly automatic and does not require human intervention except in on phase when the user might be asked to resolve any ambiguity in the semantic knowledge extracted. The system is also easy to use and the results are well-documented. Another vital feature is the choice of tools. The implementation is in Java, due to the popularity and portability.

6.2 Future Work

We plan to extend our semantic analyzer prototype to accomplish the following:

- To extract the semantic meaning of functions: In both procedural and object-oriented languages, developers are encouraged to write individual functions that implement a specific functionality or feature. Application code for databases usually follows the above-described design closely. Therefore, it is possible to encounter an assignment statement in the application code, where the right hand side of the assignment is a call to a function, and the left hand side of the assignment statement is the slicing variable. Little is learned from extracting the assignment statement as a business rule, as the functionality of the function being invoked is not known. Hence it is important to be able to extract the semantic meaning of functions.
- Enhance the extraction power of the algorithm: Currently our semantic analysis algorithm puts a restriction on the format of the output statements. The semantic analyzer can only analyze output statements that direct their output to the standard output. However, output can be directed to a file or displayed in HTML format as is frequently done in application code. It is important therefore to extend the semantic analysis algorithm to capture semantic knowledge from such statements as well.
- Representation of the extracted business rules. It is important to leverage existing technology or to develop our own model to represent business rules extracted from application code in a completely code independent format. One of the goals of SEEK is to be able to represent business rules extracted in a code independent format that can be easily understood by people outside of the code development community, and such that it can be easily exchanged in the form of e-mails and memos.
- Conduct further performance analysis experiments especially for large application code files and make the prototype more efficient.

Acknowledgements

This material is based upon work supported by the National Science Foundation under grant numbers CMS-0075407 and CMS-0122193.

References

[1] J. W. Backus, "The syntax and semantics of the proposed international algebraic language of information processing," International Conference on Information Processing, Paris, France, 1959.

[2] G. Ballard and G. Howell, "Shielding production: An essential step in production control," *Journal of Construction Engineering and Management, ASCE*, vol. 124, pp. 11-17, 1997.

[3] J. Hammer and W. O'Brien, "Enabling supply-chain coordination: Leveraging legacy sources for rich decision support," in *Research frontiers in supply chain management and e-commerce*, e. a. E. Akçalý, Ed.: Kluwer Science Series in Applied Optimization, 2003, pp. 20.

[4] J. Hammer, M. Schmalz, W. O'Brien, S. Shekar, and N. Haldavnekar, "Knowledge extraction in the seek project part I: Data reverse engineering," University of Florida, Gainesville, FL, Technical Report TR02-008, September 2002 2002.

[5] J. Hammer, M. Schmalz, W. O'Brien, S. Shekar, and N. Haldavnekar, "Seeking knowledge in legacy information systems to support interoperability," ECAI-02 International Workshop on Ontologies and Semantic Interoperability, Lyon, France, 2002.

[6] M. S. Hecht, *Flow analysis of computer programs*. Amsterdam, Holland: Elsevier North-Holland Publishing Co., 1977.

[7] S. Horwitz and T. Reps, "The use of program dependence graphs in software engineering," Fourteenth International Conference on Software Engineering, Melbourne, Australia, 1992.

[8] H. Huang, W. T. Tsai, S. Bhattacharya, X. P. Chen, Y. Wang, and J. Sun, "Business rule extraction from legacy code," presented at 20th International Computer Software and Applications Conference (COMPSAC '96), 1996.

[9] L. Koskela and R. Vrijhoef, "Roles of supply chain management in construction," 7th Annual International Conference Group for Lean Construction, 1999.

[10] W. O'Brien, R. R. Issa, J. Hammer, M. S. Schmalz, J. Geunes, and S. X. Bai, "Seek: Accomplishing enterprise information integration across heterogeneous sources," *ITCON - Journal of Information Technology in Construction*, vol. 7, pp. 101-124, 2002.

[11] W. J. O'Brien, M. A. Fischer, and J. V. Jucker, "An economic view of project coordination," *Construction Management and Economics*, vol. 13, pp. 393-400, 1995.

[12] S. Paul and A. Prakash, "A framework for source code search using program patterns," *Software Engineering*, vol. 20, pp. 463-475, 1994.

[13] B. Shneiderman and R. Mayer, "Syntactic/semantic interactions in programmer behavior: A model and experimental results," *International Journal of Computer and Information Services*, vol. 7, pp. 219-239, 1979.

[14] H. M. Sneed and K. Erdos, "Extracting business rules from source code," Fourth Workshop on Program Comprehension, 1996.

[15] E. Soloway and K. Ehrlich, "Empirical studies of programming knowledge," *IEEE Transactions on Software Engineering*, vol. 10, pp. 595-609, 1984.

[16] M. Weiser, "Program slicing," 5th International Conference on Software Engineering, San Diego, CA, 1981.

[17] G. Wiederhold, "Mediators in the architecture of future information systems," *IEEE Computer*, vol. 25, pp. 38-49, 1992.

Appendix A. Grammar for C code used in Semantic Analysis Algorithm

CProgram	->	Consts Forwards Dclns Function+	=> "program";
Includes	->	('#include' '"' <filename> '"' ';')*	=> "include";
Consts	->	(Const ';')+	=> "consts"
	->		=> "consts";
Const	->	‘#define' Name	=> "const";
Forwards	->	(Forward ';')+	=> "forwards"
	->		=> "forwards";
Forward	->	^' Type Name Params	=> "forward";
Dclns	->	(DclnList ';')+	=> "dclns"
	->		=> "dclns";
Type	->	Id;	
DclnList	->	Type Dcln list ','	=> "dcln";
	->	‘struct’ Type Dcln list ‘,’	=> "structdcln";
Dcln	->	Id '=' Expression	=> "="
	->	Id;	
Function	->	Type Name Params ‘{' Dclns Statement+ '}'	=> "function";
Params	->	'(' DclnList ? ')'	=> "params";
Block	->	'{' Statement* '}'	=> "block";
Statement	->	Assignment ';'	
	->	Name '(' (Expression list ',')? ')' ';'	=> "call"
	->	'printf' '(' String? Expression list ',' ')' ';'	=> "print"
	->	'printf' '(' String? ')' ';'	=> "emptyprint"
	->	'scanf' '(' String ? Id list ',' ')' ';'	=> "scanf"
	->	'if' '(' Expression ')' Statement ('else' Statement)?	=> "if"
	->	'while' '(' Expression ')' Statement	=> "while"
	->	'for' '(' Assignment ';' Expression ';' Assignment ')' Statement	=> "for"
	->	'for' '(' ';' ';' ')' Statement	=> "for"
	->	'do' Statement 'while' Expression ';'	=> "do"
	->	'switch' '(' Term ')' '{' Case+ 'default' ':' Block '}'	=> "switch"
	->	Block	
	->	SQLprefix SQLstatement SQLterminator?	=>"embSQL"
	->	(DclnList ';')+	=> "dclns"
	->	Primary ‘++’	=> "++"
	->	Primary ‘--’	=> "--"
	->	;	

SQLprefix	->	EXEC SQL DBclause?	=> "beginSQL"
SQLterminator	->	END-EXEC	=> "endSQL"
	->	;	
SQLstatement	->	'SELECT' columnlist 'INTO' hostvariablelist ‘FROM’ tablelist	=> 'SQLselectone'
		'SELECT' columnlist 'INTO' hostvariablelist ‘FROM’ tablelist 'WHERE' SQLExpression	=> 'SQLselectone'
	->	'SELECT' columnlist 'INTO' hostvariablelist 'FROM' tablelist 'WHERE' ‘EXISTS’ SQLExpression	=> ‘SQLselectone’
	->	'SELECT' columnlist 'INTO' hostvariablelist 'FROM' tablelist 'WHERE' ‘NOT EXISTS’ SQLExpression	=> ‘SQLselectone’
	->	'SELECT' ‘COUNT’ ‘(’ ‘*’ ‘)’ columnlist 'INTO' hostvariablelist 'FROM' tablelist 'WHERE' SQLExpression	=> ‘SQLselectonecount’
	->	'SELECT' ‘DISTINCT’ columnlist 'INTO' hostvariablelist 'FROM' tablelist 'WHERE' SQLExpression	=> ‘SQLselectonedistinct’
	->	'SELECT' columnlist 'INTO' hostvariablelist 'FROM' tablelist 'WHERE' SQLExpression ‘GROUP’ ‘BY’ columnlistgroupby	=>’SQLselectonegroupby’
	->	'SELECT' columnlist 'INTO' hostvariablelist 'FROM' tablelist 'WHERE' SQLExpression ‘ORDER’ ‘BY’ columnlistgroupby	=>’SQLselectonegroupby’
	->	'SELECT' columnlist ‘FROM’ tablelistmod	=> 'SQLselecttwo'
		'SELECT' columnlist ‘FROM’ tablelistmod 'WHERE' SQLExpression	=> 'SQLselecttwo'
	->	'SELECT' columnlist 'FROM' tablelistmod 'WHERE' ‘EXISTS’ SQLExpression	=> ‘SQLselecttwo’
	->	'SELECT' columnlist 'FROM' tablelistmod 'WHERE' ‘NOT EXISTS’ SQLExpression	=> ‘SQLselecttwo’
	->	'SELECT' ‘COUNT’ ‘(’ ‘*’ ‘)’ columnlist ‘FROM’ tablelistmod 'WHERE' SQLExpression	=> ‘SQLselecttwocount’
	->	'SELECT' ‘DISTINCT’ columnlist ‘FROM’ tablelistmod 'WHERE' SQLExpression	=> ‘SQLselecttwodistinct’
	->	'SELECT' columnlist ‘FROM’ tablelistmod 'WHERE' SQLExpression ‘GROUP’ ‘BY’ columnlistgroupby	=>’SQLselecttwogroupby’
	->	'SELECT' columnlist ‘FROM’ tablelist 'WHERE'	=>’SQLselecttwogroupby’

		tablelistmod 'WHERE' SQLExpression ‘ORDER’ ‘BY’ columnlistgroupby	
	->	'INSERT' ‘INTO’ tablelist ‘VALUES’ ‘(‘hostvariablelist ‘)’	=>’SQLinsert’
	->	'DELETE' Id ‘FROM’ tablelist 'WHERE' SQLExpression	=>’SQLdelete’
	->	‘UPDATE’ tablelist ‘SET’ (SQLAssignment ’,’) list 'WHERE' SQLExpression	=>’SQLupdate’
	->	;	=> ‘SQLselect’
tablelist	->	(Name list ',')	=> ‘tablelist’
tablelistmod	->	(tablename list ',')	=>‘tablelist’
tablename	->	Id Id	=>’tablename’
columnlist	->	(Term list ',')	=>'columnlist"
columnlistgroupby ->		(Name list ',')	=>'columnlistgroupby"
Hostvariablelist ->		(Variable list ',')	=> 'hostvariablelist"
	->		=> 'hostvariablelist"
Variable	->	':' Name ;	
SQLExpression	->	SQLExpression ‘AND’ SQLAssignment	=> "SQLExpression"
		SQLExpression ‘OR’ SQLAssignment	=> "SQLExpression"
		SQLAssignment;	
SQLAssignment	->	Id '=' Name	=> "SQLAssignment="
	->	Id '>' Name	=> "SQLAssignment>"
	->	Id '<' Name	=> "SQLAssignment<"
	->	Id '>=' Name	=> "SQLAssignment>="
	->	Id '<=' Name	=> "SQLAssignment<="
	->	Id '<>' Name	=> "SQLAssignment<>"
	->	Id '=' Name '(' (Expression list ',')? ')' ';'	=> "SQLAssignment="
	->	Id '>' Name '(' (Expression list ',')? ')' ';'	=> "SQLAssignment>"
	->	Id '<' Name '(' (Expression list ',')? ')' ';'	=> "SQLAssignment<"
	->	Id '>=' Name '(' (Expression list ',')? ')' ';'	=> "SQLAssignment>="
	->	Id '<=' Name '(' (Expression list ',')? ')' ';'	=> "SQLAssignment<="
	->	Id '<>' Name '(' (Expression list ',')? ')' ';'	=> "SQLAssignment<>"
	->	Id 'LIKE' String	=> "SQLAssignmentLIKE"
	->	Id '=' SQLStatement	=> "SQLAssignment="
	->	Id '=' ‘ANY’ SQLStatement	=> "SQLAssignment="
	->	Id '>' ‘ANY’ SQLStatement	=> "SQLAssignmen>"
	->	Id '<' ‘ANY’ SQLStatement	=> "SQLAssignment<"
	->	Id '<=' ‘ANY’ SQLStatement	=> "SQLAssignment<="
	->	Id '>=' ‘ANY’ SQLStatement	=> "SQLAssignment>="
	->	Id '<>' ‘ANY’ SQLStatement	=> "SQLAssignment<>"

	->	Id '=' ‘ALL’ SQLStatement	=> "SQLAssignment="
	->	Id '>' ‘ALL’ SQLStatement	=> "SQLAssignment>"
	->	Id '<' ‘ALL’ SQLStatement	=> "SQLAssignment<"
	->	Id '<=' ‘ALL’ SQLStatement	=> "SQLAssignment<="
	->	Id '>=' ‘ALL’ SQLStatement	=> "SQLAssignment>="
	->	Id '<>' ‘ALL’ SQLStatement	=> "SQLAssignment<>"
	->	Id '=' ‘IN’ SQLStatement	=> "SQLAssignment="
	->	Id '>' ‘IN’ SQLStatement	=> "SQLAssignment>"
	->	Id '<' ‘IN’ SQLStatement	=> "SQLAssignment<"
	->	Id '<=' ‘IN’ SQLStatement	=> "SQLAssignment<="
	->	Id '>=' ‘IN’ SQLStatement	=> "SQLAssignment>="
	->	Id '<>' ‘IN’ SQLStatement	=> "SQLAssignment<>"
DB clause	->	‘BEGIN’ ‘DECLARE’ ‘SECTION’	=> "DBclause"
	->	‘END’ ‘DECLARE’ ‘SECTION’	=> "DBclause"
	->	‘WHENEVER’ ‘SQL’ ‘WARNING’ ‘CALL’ Name '(' (Expression list ',')? ')'	=> "DBclause"
	->	‘WHENEVER’ ‘SQL’ ‘NOT’ ‘FOUND’ ‘CALL’ Name '(' (Expression list ',')? ')'	=> "DBclause"
	->	‘WHENEVER’ ‘SQL’ ‘NOT’ ‘FOUND’ ‘DO’ ‘BREAK’	=> "DBclause"
	->	‘WHENEVER’ ‘SQL’ ‘NOT’ ‘FOUND’ ‘DO’ ‘CONTINUE’	=> "DBclause"
	->	‘COMMIT’ ‘WORK’	=> "DBclause"
	->	‘WHENEVER’ ‘SQL’ ‘ERROR’ ‘CALL’ Name '(' (Expression list ',')? ')'	=> "DBclause"
	->	‘DISCONNECT’ ‘ALL’	=> "DBclause"
	->	‘USE’ Id	=> "DBclause"
	->	‘CONNECT’ Name ‘IDENTIFIED’ ‘BY’ Name	=> "DBclause"
	->	‘COMMIT’	=> "DBclause"
	->	‘COMMIT’ ‘WORK’ ‘RELEASE’	=> "DBclause"
	->	‘COMMIT’ ‘WORK’	=> "DBclause"
	->	‘OPEN’ Name	=> "DBclause"
	->	‘CLOSE’ Name	=> "DBclause"
	->	‘DECLARE’ Name ‘FOR’	=> "DBclause"
	->	‘FETCH’ Name ‘INTO’ hostvariablelist	=> "DBclause"
Case	->	'case' '<integer>' ':' Block	=> "case";
Assignment	->	Id '=' Expression	=> "assign";
	->	Id '+''=' Expression	=> "assign";
	->	Id '-''=' Expression	=> "assign";
Expression	->	LExpression '?' LExpression ':' LExpression	=> "?"
	->	LExpression;	
LExpression	->	LExpression '&&' Comparison	=> "and"
	->	LExpression '||' Comparison	=> "or"
	->	LExpression '~' Comparison	=> "xor"

	->	Comparison;	
Comparison	->	Term '<=' Term	=> "<="
	->	Term '==' Term	=> "=="
	->	Term '>=' Term	=> ">="
	->	Term '!=' Term	=> "!="
	->	Term '<' Term	=> "<"
	->	Term '>' Term	=> ">"
	->	Term;	
Term ->		Term '+' Factor	=> "+"
	->	Term '-' Factor	=> "-"
	->	Factor;	
Factor	->	Exp '*' Factor	=> "*"
	->	Exp '/' Factor	=> "/"
	->	Exp '%' Factor	=> "%"
	->	Exp ;	
Exp	->	Primary '**' Exp	=> "**"
	->	Primary;	
Primary	->	'-' Primary	=> "-"
	->	'+' Primary	
	->	'!' Primary	=> "!"
	->	'++' Primary	=> "++"
	->	'--' Primary	=> "--"
	->	Primary ‘++’	=> "++"
	->	Primary ‘--’	=> "--"
	->	Atom;	
Atom	->	'eof'	=> "eof"
	->	'<integer>'	
	->	Id	
	->	'(' Expression ')';	
	->	Name '(' (Expression list ',')? ')' ';'	=> "rhscall"
Initializer	->	'<integer>'	
	->	'&' Name	=> "&";
Id	->	'*' Name	=> "*"
	->	'&' Name	=> "&"
	->	Name;	
Name	->	‘<identifier>';	
String	->	‘<text>’ ;	

Appendix B. Grammar for Java Semantic Analyzer

JProgram	->	‘{’ Consts Forwards Dclns Function+ ‘}’	=> "program"
Includes	->	('#include' '"' <filename> '"' ';')*	=> "include"
Consts	->	(Const ';')+	=> "consts"
	->		=> "consts"
Const	->	‘#define' Name	=> "const"
Forwards	->	(Forward ';')+	=> "forwards"
	->		=> "forwards"
Forward	->	‘^' Type Name Params	=> "forward"
Dclns	->	(DclnList ';')+	=> "dclns"
	->		=> "dclns"
Type	->	Id;	
DclnList	->	AccessLevel ‘static’? ‘final’? ‘transient’? ‘volatile’? Type Dcln list ','	=> "dcln"
	->	‘struct’ Type Dcln list ‘,’	=> "structdcln"
Dcln	->	Id '=' Expression	=> "="
	->	Id;	
Function	->	Type Type Type Name Params ‘{' Dclns Statement+ '}'	=> "function"
Params	->	'(' DclnList ? ')'	=> "params"
Block	->	'{' Statement* '}'	=> "block"
Statement	->	Assignment ';'	
	->	Name '(' (Expression list ',')? ')' ';'	=> "call"
	->	'printf' '(' (String)* (Expression)* list '+' ')' ';'	=> "print"
	->	'printf' '(' String List ‘+’ ')' ';'	=> "emptyprint"
	->	'printf' '('Expression list '+' ')' ';'	=> "onlyvarprint"
	->	'if' '(' Expression ')' Statement ('else' Statement)?	=> "if"
	->	'while' '(' Expression ')' Statement	=> "while"
	->	'for' '(' Assignment ';' Expression ';' Assignment ')' Statement	=> "for"
	->	'for' '(' ';' ';’ ‘)’ Statement	=> "for"
	->	‘do’ Statement ‘while’ Expression ‘;’	=> "do"
	->	‘switch’ ‘(’ Term ‘)’ ‘{‘ Case+ ‘default’ ‘:’ Block ‘}’	=> "switch"
	->	Block	

	->	SQLprefix SQLstatement SQLterminator?	=>"embSQL"
	->	(DclnList ‘;’)+	=> "dclns"
	->	‘try’ ‘{‘ Statement* ‘}’ ‘catch’ ‘(‘ Type Id ‘)’ ‘{‘ Statement* ‘}’ ‘;’	=> "try"
	->	;	
SQLprefix	->	SQL? Dbclause?	=> "beginSQL"
SQLterminator	->	END-EXEC	=> "endSQL"
	->	;	
SQLstatement	->	‘SELECT’ columnlist ‘FROM’ tablelistmod	=> "SQLselecttwo"
	->	‘SELECT’ columnlist ‘FROM’ tablelistmod ‘WHERE’ SQLExpression	=> "SQLselecttwo"
	->	‘SELECT’ columnlist ‘FROM’ tablelistmod ‘WHERE’ ‘EXISTS’ SQLExpression	=> "SQLselecttwo"
	->	‘SELECT’ columnlist ‘FROM’ tablelistmod ‘WHERE’ ‘NOT EXISTS’ SQLExpression	=> "SQLselecttwo"
	->	‘SELECT’ ‘COUNT’ ‘(‘ ‘*’ ‘)’ columnlist ‘FROM’ tablelistmod ‘WHERE’ SQLExpression	=> "SQLselecttwocount"
	->	‘SELECT’ ‘DISTINCT’ columnlist ‘FROM’ tablelistmod ‘WHERE’ SQLExpression	=> "SQLselecttwodistinct"
	->	‘SELECT’ columnlist ‘FROM’ tablelistmod ‘WHERE’ SQLExpression ‘GROUP’ ‘BY’ columnlistgroupby	=> "SQLselecttwogroupby"
	->	‘SELECT’ columnlist ‘FROM’ tablelistmod ‘WHERE’ SQLExpression ‘ORDER’ ‘BY’ columnlistgroupby	=>"SQLselecttwogroupby"
	->	‘INSERT’ ‘INTO’ tablelist ‘VALUES’ ‘(‘hostvariablelist ‘)’	=>"SQLinsert"
	->	‘DELETE’ Id ‘FROM’ tablelist ‘WHERE’ SQLExpression	=> "SQLdelete"
	->	‘UPDATE’ tablelist ‘SET’ (SQLAssignment ‘,’) list ‘WHERE’ SQLExpression	=> "SQLupdate"
	->	;	=> "SQLselect"
tablelist	->	(Name list ‘,’)	=> "tablelist"
tablelistmod	->	(tablename list ‘,’)	=> "tablelist"
tablename	->	Id Id	=> "tablename"
columnlist	->	(Term list ‘,’)	=> "columnlist"
	->	‘*’ ;	=> "columnlist"
columnlistgroupby ->		(Name list ‘,’)	=> "columnlistgroupby"
Hostvariablelist ->		(Variable list ‘,’)	=> "hostvariablelist"
	->		=> "hostvariablelist"

Variable	->	‘:’ Name ;	
SQLExpression	->	SQLExpression ‘AND’ SQLAssignment	=> "SQLExpression"
		SQLExpression ‘OR’ SQLAssignment	=> "SQLExpression"
		SQLAssignment;	
SQLAssignment	->	Id ‘=’ Name	=> "SQLAssignment="
	->	Id ‘>’ Name	=> "SQLAssignment>"
	->	Id ‘<’ Name	=> "SQLAssignment<"
	->	Id ‘>=’ Name	=> "SQLAssignment>="
	->	Id ‘<=’ Name	=> "SQLAssignment<="
	->	Id ‘<>’ Name	=> "SQLAssignment<>"
	->	Id ‘=’ Name ‘(‘ (Expression list ‘,’)? ‘)’ ‘;’	=> "SQLAssignment="
	->	Id ‘>’ Name ‘(‘ (Expression list ‘,’)? ‘)’ ‘;’	=> "SQLAssignment>"
	->	Id ‘<’ Name ‘(‘ (Expression list ‘,’)? ‘)’ ‘;’	=> "SQLAssignment<"
	->	Id ‘>=’ Name ‘(‘ (Expression list ‘,’)? ‘)’ ‘;’	=> "SQLAssignment>="
	->	Id ‘<=’ Name ‘(‘ (Expression list ‘,’)? ‘)’ ‘;’	=> "SQLAssignment<="
	->	Id ‘<>’ Name ‘(‘ (Expression list ‘,’)? ‘)’ ‘;’	=> "SQLAssignment<>"
	->	Id ‘LIKE’ String	=> "SQLAssignmentLIKE"
	->	Id ‘=’ SQLStatement	=> "SQLAssignment="
	->	Id ‘=’ ‘ANY’ SQLStatement	=> "SQLAssignment="
	->	Id ‘>’ ‘ANY’ SQLStatement	=> "SQLAssignmen>"
	->	Id ‘<’ ‘ANY’ SQLStatement	=> "SQLAssignment<"
	->	Id ‘<=’ ‘ANY’ SQLStatement	=> "SQLAssignment<="
	->	Id ‘>=’ ‘ANY’ SQLStatement	=> "SQLAssignment>="
	->	Id ‘<>’ ‘ANY’ SQLStatement	=> "SQLAssignment<>"
	->	Id ‘=’ ‘ALL’ SQLStatement	=> "SQLAssignment="
	->	Id ‘>’ ‘ALL’ SQLStatement	=> "SQLAssignment>"
	->	Id ‘<’ ‘ALL’ SQLStatement	=> "SQLAssignment<"
	->	Id ‘<=’ ‘ALL’ SQLStatement	=> "SQLAssignment<="
	->	Id ‘>=’ ‘ALL’ SQLStatement	=> "SQLAssignment>="
	->	Id ‘<>’ ‘ALL’ SQLStatement	=> "SQLAssignment<>"
	->	Id ‘=’ ‘IN’ SQLStatement	=> "SQLAssignment="
	->	Id ‘>’ ‘IN’ SQLStatement	=> "SQLAssignment>"
	->	Id ‘<’ ‘IN’ SQLStatement	=> "SQLAssignment<"
	->	Id ‘<=’ ‘IN’ SQLStatement	=> "SQLAssignment<="
	->	Id ‘>=’ ‘IN’ SQLStatement	=> "SQLAssignment>="
	->	Id ‘<>’ ‘IN’ SQLStatement	=> "SQLAssignment<>"
DB clause	->	‘BEGIN’ ‘DECLARE’ ‘SECTION’	=> "Dbclause"
	->	‘END’ ‘DECLARE’ ‘SECTION’	=> "Dbclause"
	->	‘WHENEVER’ ‘SQL’ ‘WARNING’ ‘CALL’ Name ‘(‘ (Expression list ‘,’)? ‘)’	=> "Dbclause"
	->	‘WHENEVER’ ‘SQL’ ‘NOT’ ‘FOUND’ ‘CALL’ Name ‘(‘	=> "Dbclause"

		(Expression list ‘,’)? ‘)’	
	->	‘WHENEVER’ ‘SQL’ ‘NOT’ ‘FOUND’ ‘DO’ ‘BREAK’	=> "Dbclause"
	->	‘WHENEVER’ ‘SQL’ ‘NOT’ ‘FOUND’ ‘DO’ ‘CONTINUE’	=> "Dbclause"
	->	‘COMMIT’ ‘WORK’	=> "Dbclause"
	->	‘WHENEVER’ ‘SQL’ ‘ERROR’ ‘CALL’ Name ‘(’ (Expression list ‘,’)? ‘)’	=> "Dbclause"
	->	‘DISCONNECT’ ‘ALL’	=> "Dbclause"
	->	‘USE’ Id	=> "Dbclause"
	->	‘CONNECT’ Name ‘IDENTIFIED’ ‘BY’ Name	=> "Dbclause"
	->	‘COMMIT’	=> "Dbclause"
	->	‘COMMIT’ ‘WORK’ ‘RELEASE’	=> "Dbclause"
	->	‘COMMIT’ ‘WORK’	=> "Dbclause"
	->	‘OPEN’ Name	=> "Dbclause"
	->	‘CLOSE’ Name	=> "Dbclause"
	->	‘DECLARE’ Name ‘FOR’	=> "Dbclause"
	->	‘FETCH’ Name ‘INTO’ hostvariablelist	=> "Dbclause"
Case	->	‘case’ ‘<integer>’ ‘:’ Block	=> "case"
Assignment	->	Id ‘=’ Expression	=> "assign"
Assignment	->	Id ‘+’’=’ Expression	=> "assign"
Assignment	->	Id ‘-’’=’ Expression	=> "assign"
Expression	->	Lexpression ‘?’ Lexpression ‘:’ Lexpression	=> "?"
	->	Lexpression;	
Lexpression	->	Lexpression ‘&&’ Comparison	=> "and"
	->	Lexpression ‘||’ Comparison	=> "or"
	->	Lexpression ‘~’ Comparison	=> "xor"
	->	Comparison;	
Comparison	->	Term ‘<=’ Term	=> "<="
	->	Term ‘==’ Term	=> "=="
	->	Term ‘>=’ Term	=> ">="
	->	Term ‘!=’ Term	=> "!="
	->	Term ‘<’ Term	=> "<"
	->	Term ‘>’ Term	=> ">"
	->	Term;	
Term ->		Term ‘+’ Factor	=> "+"
	->	Term ‘-’ Factor	=> "-"
	->	Factor;	
Factor	->	Exp ‘*’ Factor	=> "*"
	->	Exp ‘/’ Factor	=> "/"
	->	Exp ‘%’ Factor	=> "%"
	->	Exp ;	
Exp	->	Primary ‘**’ Exp	=> "**"
	->	Primary;	

Primary	->	‘-‘ Primary	=> "-"
	->	‘+’ Primary	
	->	‘!’ Primary	=> "!"
	->	‘++’ Primary	=> "++"
	->	‘—‘ Primary	=> "—"
	->	Primary ‘++’	=> "++"
	->	Primary ‘—’	=> "—"
	->	Atom;	
Atom	->	‘eof’	=> "eof"
	->	‘<integer>’	
	->	Id	
	->	‘(‘ Expression ‘)’;	
	->	Name ‘(‘ (Expression list ‘,’)? ‘)’ ‘;’	=> "rhscall"
Initializer	->	‘<integer>’	
	->	‘&’ Name	=> "&"
Id	->	‘*’ Name	=> "*"
	->	‘&’ Name	=> "&"
	->	Name;	
Name	->	‘<identifier>’;	
	->	‘<text>’;	
String	->	‘<text>’ ;	
AccessLevel	->	‘public’	
	->	‘private’	
	->	‘protected'	

Appendix C. Test Code Listing

```
import java.sql.*;
import java.math.*;

public class TestCode1
{

public static void main(String[] args)
{
    try
    {

      DriverManager.registerDriver(new
      oracle.jdbc.driver.OracleDriver());

      Connection conn = DriverManager.getConnection
      ("jdbc:oracle:thin:@titan:1521:orcl","hamish","tiger");

    Statement stmt = conn.createStatement();

      String query = "SELECT Proj_Start_Date + 1, Project_Finish_Date -
      1, Project_Cost FROM MSP_Projects WHERE Proj_Name = 'Avalon'";

      ResultSet rset = stmt.executeQuery(query);

      Date pstart = rset.getDate(0);
      Date pfinish = rset.getDate(1);
      float pcost = rset.getFloat(2);

      if (checkCost(pcost) > 1000000)
      {
            //Give 10% discount for big budget projects
            pcost = pcost - pcost * 10/100;
            stmt.executeUpdate("UPDATE MSP_Projects SET Project_Cost =
            " + pcost + " WHERE Proj_Name = 'Avalon'");

      }
```

```
String displayString;
displayString = "Project Start Date " + pstart;
System.out.println(displayString);
System.out.println("Project Finish Date for Avalon " + pfinish);

String query = "SELECT Task_Start_Date, Task_Finish_Date,
Task_UnitCost FROM MSP_Tasks WHERE Task_Name = 'Tiles'";
//This query extracts the start and finish date Task Name 'Tiles'
ResultSet rset = stmt.executeQuery(query);

System.out.println("Finish Date of Task Start Date of Task Unit
Cost for Task");
System.out.println("-----------------------------------------
------------");

while (rset.next())
{
      Date tstart = rset.getDate("Task_Start_Date");
      Date tfinish = rset.getDate("Task_Finish_Date");
      float tcost = rset.getFloat("Task_UnitCost");

      checkifValidDate(tstart);
      tcost = checkDuration(tstart, tfinish, tcost);

      stmt.executeUpdate("UPDATE MSP_Tasks SET Task_UnitCost = "
      + tcost + " WHERE Task_Start_Date = '" + tstart + "' AND
      Task_Finish_Date = '" + tfinish + "' ");

      System.out.print(tfinish);
      System.out.print("\t" + tstart);
      System.out.println("\t" + tcost);

      if     ((tstart.getDate()     <     pstart.getDate())     ||
      (tfinish.getDate() > pfinish.getDate()))
      {
            System.out.println("The task start and finish dates
            have to be within the project start and finish
            dates");
      }


}
rset.close();
```

```
      stmt.close();
      conn.close();
   }
   catch (Exception e)
   {
       System.out.println("ERROR : " + e);
       e.printStackTrace(System.out);
   }
}

public static float checkDuration(Date s1, Date t1, float f1)
{
      float revisedcost;
      if (s1.getDate() - t1.getDate() < 10)
      {
            // 20 % raise in cost for rush orders
            revisedcost = f1 + f1 * 20/100;
            System.out.println("Estimated  New  Task  Unit  Cost  :  "  +
            revisedcost);
      }
      else
      {
            revisedcost = f1;
      }

      return revisedcost;

}

public static void checkifValidDate(Date i1)
{
      Date d = new Date();
      d.setYear(1970);
      d.setMonth(1);
      d.setDate(1);

      if (i1.getDate() > d.getDate())
      {
            System.out.println("Invalid Date !");
      }
```

```
}
```

Appendix D. Reduced Source Code Generated by the Java Pattern Matcher

```
{
public static void main(String[] args)
{
   try
   {
           DriverManager.registerDriver(
           oracle.jdbc.driver.OracleDriver());

           Connection conn  = DriverManager.getConnection
           ("jdbc:oracle:thin:@titan:1521:orcl","hamish",
           "tiger");

           String   query   =   "SELECT   Proj_Start_Date   +   1,
           Project_Finish_Date  -1,  Project_Cost  FROM  MSP_Projects
           WHERE Proj_Name = 'Avalon'";

           SELECT  Proj_Start_Date  +  1,  Project_Finish_Date  -1,
           Project_Cost FROM MSP_Projects ;

         Date pstart = getDate(0);
         Date pfinish = getDate(1);
         float pcost = getFloat(2);

           if (checkCost(pcost) > 1000000)
           {
                 //Give 10% discount for big budget projects
                 pcost = pcost - pcost * 10/100;
           }

         String displayString;
           displayString = "Project Start Date " + pstart;
         printf(displayString);
           printf("Project Finish Date for Avalon " + pfinish);

           String  query  =  "SELECT  Task_Start_Date,  Task_Finish_Date,
           Task_UnitCost FROM MSP_Tasks WHERE Task_Name = 'Tiles'";
           //This  query  extracts  the  start  and  finish  date  Task  Name
           'Tiles'
```

```
            SELECT   Task_Start_Date,   Task_Finish_Date,   Task_UnitCost
            FROM MSP_Tasks ;

            printf("Finish Date of Task Start Date of Task Unit Cost
            for Task");
            printf("-----------------------------------------------------
            ------");

           while (rset.next())
            {
                Date tstart = getDate("Task_Start_Date");
                Date tfinish = getDate("Task_Finish_Date");
                float tcost = getFloat("Task_UnitCost");
                 checkifValidDate(tstart);
                 tcost = checkDuration(tstart, tfinish, tcost);

                printf(tfinish);
                printf("\t" + tstart);
                printf("\t" + tcost);

                 if    ((tstart.getDate()    <    pstart.getDate())    ||
                 (tfinish.getDate() > pfinish.getDate()))
                 {
                       printf("The task start and finish dates have to
                       be within the project start and finish dates");
                 }
            }
           rset.close();
            stmt.close();
            conn.close();
        }
      catch (Exception e)
      {
          printf("ERROR : " + e);
            e.printStackTrace(System.out);
      }
}

public static float checkDuration(Date s1, Date t1, float f1)
{
      float revisedcost;
      if (s1.getDate() - t1.getDate() < 10)
      {
```

```
            // 20 % raise in cost for rush orders
            revisedcost = f1 + f1 * 20/100;

            printf("Estimated New Task Unit Cost : " + revisedcost);
      }
      else
      {
            revisedcost = f1;
      }
      return revisedcost;
}

public static void checkifValidDate(Date i1)
{
    Date d;
      d.setYear(1970);
      d.setMonth(1);
      d.setDate(1);
      if (i1.getDate() > d.getDate())
      {
          printf("Invalid Date !");
      }
}
    }
```

Appendix E. AST for the Test Code

```
---------- AST -----------
program(7)
. consts(0)
. forwards(0)
. dclns(0)
. <identifier>(1)
. . void(0)
. function(5)
. . <identifier>(1)
. . . main(0)
. . params(1)
. . . dcln(2)
. . . . <identifier>(1)
. . . . . String[](0)
. . . . <identifier>(1)
. . . . . args(0)
. . dclns(0)
. . try(19)
. . . call(2)
. . . . <identifier>(1)
. . . . . DriverManager.registerDriver(0)
. . . . rhscall(1)
. . . . . <identifier>(1)
. . . . . . oracle.jdbc.driver.OracleDriver(0)
. . . <identifier>(1)
. . . . Connection(0)
. . . assign(2)
. . . . <identifier>(1)
. . . . . conn(0)
. . . . rhscall(4)
. . . . . <identifier>(1)
. . . . . . DriverManager.getConnection(0)
. . . . . <string>(1)
. . . . . . "jdbc:oracle:thin:@titan:1521:orcl"(0)
. . . . . <string>(1)
. . . . . . "hamish"(0)
. . . . . <string>(1)
. . . . . . "tiger"(0)
```

```
. . . dclns(1)
. . . . dcln(2)
. . . . . <identifier>(1)
. . . . . . String(0)
. . . . . =(2)
. . . . . . <identifier>(1)
. . . . . . . query(0)
. . . . . . <string>(1)
. . . . . . . "SELECT Proj_Start_Date + 1, Project_Finish_Date -1,
Project_Cost FROM MSP_Projects WHERE Proj_Name = 'Avalon'"(0)
. . . embSQL(3)
. . . . beginSQL(0)
. . . . SQLselecttwo(2)
. . . . . columnlist(3)
. . . . . +(2)
. . . . . <identifier>(1)
. . . . . . Proj_Start_Date(0)
. . . . . <integer>(1)
. . . . . . 1(0)
. . . . . -(2)
. . . . . <identifier>(1)
. . . . . . Project_Finish_Date(0)
. . . . . <integer>(1)
. . . . . . 1(0)
. . . . . <identifier>(1)
. . . . . . Project_Cost(0)
. . . . . tablelist(1)
. . . . . <identifier>(1)
. . . . . . MSP_Projects(0)
. . . . endSQL(0)
. . . dclns(3)
. . . . dcln(2)
. . . . . <identifier>(1)
. . . . . . Date(0)
. . . . . =(2)
. . . . . <identifier>(1)
. . . . . . pstart(0)
. . . . . rhscall(2)
. . . . . <identifier>(1)
. . . . . . getDate(0)
. . . . . <integer>(1)
. . . . . . 0(0)
```

```
. . . . dcln(2)
. . . . . <identifier>(1)
. . . . . . Date(0)
. . . . . =(2)
. . . . . <identifier>(1)
. . . . . . pfinish(0)
. . . . . rhscall(2)
. . . . . <identifier>(1)
. . . . . . getDate(0)
. . . . . <integer>(1)
. . . . . . 1(0)
. . . . dcln(2)
. . . . . <identifier>(1)
. . . . . float(0)
. . . . . =(2)
. . . . . <identifier>(1)
. . . . . pcost(0)
. . . . . rhscall(2)
. . . . . <identifier>(1)
. . . . . getFloat(0)
. . . . . <integer>(1)
. . . . . 2(0)
. . . if(2)
. . . . >(2)
. . . . . rhscall(2)
. . . . . <identifier>(1)
. . . . . checkCost(0)
. . . . . <identifier>(1)
. . . . . pcost(0)
. . . . . <integer>(1)
. . . . . 1000000(0)
. . . . block(1)
. . . . . assign(2)
. . . . . <identifier>(1)
. . . . . pcost(0)
. . . . . -(2)
. . . . . <identifier>(1)
. . . . . pcost(0)
. . . . . *(2)
. . . . . <identifier>(1)
. . . . . pcost(0)
. . . . . /(2)
```

```
. . . . . . . . . <integer>(1)
. . . . . . . . . . 10(0)
. . . . . . . . . <integer>(1)
. . . . . . . . . . 100(0)
. . . dclns(1)
. . . . dcln(2)
. . . . . <identifier>(1)
. . . . . String(0)
. . . . . <identifier>(1)
. . . . . displayString(0)
. . . assign(2)
. . . . <identifier>(1)
. . . . . displayString(0)
. . . . +(2)
. . . . . <string>(1)
. . . . . "Project Start Date "(0)
. . . . . <identifier>(1)
. . . . . pstart(0)
. . . onlyvarprintf(1)
. . . . <identifier>(1)
. . . . . displayString(0)
. . . printf(2)
. . . . <string>(1)
. . . . . "Project Finish Date for Avalon "(0)
. . . . <identifier>(1)
. . . . . pfinish(0)
. . . dclns(1)
. . . . dcln(2)
. . . . . <identifier>(1)
. . . . . String(0)
. . . . . =(2)
. . . . . <identifier>(1)
. . . . . query(0)
. . . . . <string>(1)
. . . . . "SELECT Task_Start_Date, Task_Finish_Date, Task_UnitCost
FROM MSP_Tasks WHERE Task_Name = 'Tiles'"(0)
. . . embSQL(3)
. . . . beginSQL(0)
. . . . SQLselecttwo(2)
. . . . . columnlist(3)
. . . . . <identifier>(1)
. . . . . Task_Start_Date(0)
```

```
. . . . . <identifier>(1)
. . . . . . Task_Finish_Date(0)
. . . . . <identifier>(1)
. . . . . . Task_UnitCost(0)
. . . . . tablelist(1)
. . . . . <identifier>(1)
. . . . . . MSP_Tasks(0)
. . . . endSQL(0)
. . . emptyprintf(1)
. . . . <string>(1)
. . . . . "Finish Date of Task Start Date of Task Unit Cost for
Task"(0)
. . . emptyprintf(1)
. . . . <string>(1)
. . . . . "------------------------------------------------------
"(0)
. . . while(2)
. . . . rhscall(1)
. . . . . <identifier>(1)
. . . . . rset.next(0)
. . . . block(7)
. . . . . dclns(3)
. . . . . dcln(2)
. . . . . <identifier>(1)
. . . . . Date(0)
. . . . . =(2)
. . . . . <identifier>(1)
. . . . . tstart(0)
. . . . . rhscall(2)
. . . . . <identifier>(1)
. . . . . getDate(0)
. . . . . <string>(1)
. . . . . "Task_Start_Date"(0)
. . . . . dcln(2)
. . . . . <identifier>(1)
. . . . . Date(0)
. . . . . =(2)
. . . . . <identifier>(1)
. . . . . tfinish(0)
. . . . . rhscall(2)
. . . . . <identifier>(1)
. . . . . getDate(0)
```

```
. . . . . <string>(1)
. . . . . . "Task_Finish_Date"(0)
. . . . . dcln(2)
. . . . . <identifier>(1)
. . . . . float(0)
. . . . . =(2)
. . . . . <identifier>(1)
. . . . . tcost(0)
. . . . . rhscall(2)
. . . . . <identifier>(1)
. . . . . getFloat(0)
. . . . . <string>(1)
. . . . . "Task_UnitCost"(0)
. . . . . call(2)
. . . . . <identifier>(1)
. . . . . checkifValidDate(0)
. . . . . <identifier>(1)
. . . . . tstart(0)
. . . . . assign(2)
. . . . . <identifier>(1)
. . . . . tcost(0)
. . . . . rhscall(4)
. . . . . <identifier>(1)
. . . . . checkDuration(0)
. . . . . <identifier>(1)
. . . . . tstart(0)
. . . . . <identifier>(1)
. . . . . tfinish(0)
. . . . . <identifier>(1)
. . . . . tcost(0)
. . . . . onlyvarprintf(1)
. . . . . <identifier>(1)
. . . . . tfinish(0)
. . . . . printf(2)
. . . . . <string>(1)
. . . . . "\t"(0)
. . . . . <identifier>(1)
. . . . . tstart(0)
. . . . . printf(2)
. . . . . <string>(1)
. . . . . "\t"(0)
. . . . . <identifier>(1)
```

```
. . . . . . . tcost(0)
. . . . . if(2)
. . . . . or(2)
. . . . . <(2)
. . . . . rhscall(1)
. . . . . <identifier>(1)
. . . . . tstart.getDate(0)
. . . . . rhscall(1)
. . . . . <identifier>(1)
. . . . . pstart.getDate(0)
. . . . . >(2)
. . . . . rhscall(1)
. . . . . <identifier>(1)
. . . . . tfinish.getDate(0)
. . . . . rhscall(1)
. . . . . <identifier>(1)
. . . . . pfinish.getDate(0)
. . . . . block(1)
. . . . . emptyprintf(1)
. . . . . <string>(1)
. . . . . "The task start and finish dates have to be within
the project start and finish dates"(0)
. . . call(1)
. . . . <identifier>(1)
. . . . . rset.close(0)
. . . call(1)
. . . . <identifier>(1)
. . . . . stmt.close(0)
. . . call(1)
. . . . <identifier>(1)
. . . . . conn.close(0)
. . catch(4)
. . . <identifier>(1)
. . . . Exception(0)
. . . <identifier>(1)
. . . . e(0)
. . . printf(2)
. . . . <string>(1)
. . . . . "ERROR : "(0)
. . . . <identifier>(1)
. . . . . e(0)
. . . call(2)
```

```
. . . . <identifier>(1)
. . . . . e.printStackTrace(0)
. . . . <identifier>(1)
. . . . . System.out(0)
. function(8)
. . <identifier>(1)
. . . float(0)
. . <identifier>(1)
. . . checkDuration(0)
. . params(1)
. . . dcln(6)
. . . . <identifier>(1)
. . . . . Date(0)
. . . . <identifier>(1)
. . . . . s1(0)
. . . . <identifier>(1)
. . . . . Date(0)
. . . . <identifier>(1)
. . . . . t1(0)
. . . . <identifier>(1)
. . . . . float(0)
. . . . <identifier>(1)
. . . . . f1(0)
. . dclns(1)
. . . dcln(2)
. . . . <identifier>(1)
. . . . . float(0)
. . . . <identifier>(1)
. . . . . revisedcost(0)
. . if(3)
. . . <(2)
. . . . -(2)
. . . . . rhscall(1)
. . . . . . <identifier>(1)
. . . . . . . s1.getDate(0)
. . . . . rhscall(1)
. . . . . . <identifier>(1)
. . . . . . . t1.getDate(0)
. . . . <integer>(1)
. . . . . 10(0)
. . . block(2)
. . . . assign(2)
```

```
. . . . . <identifier>(1)
. . . . . . revisedcost(0)
. . . . . +(2)
. . . . . <identifier>(1)
. . . . . . f1(0)
. . . . . *(2)
. . . . . <identifier>(1)
. . . . . . f1(0)
. . . . . /(2)
. . . . . <integer>(1)
. . . . . . 20(0)
. . . . . <integer>(1)
. . . . . . 100(0)
. . . . printf(2)
. . . . . <string>(1)
. . . . . "Estimated New Task Unit Cost : "(0)
. . . . . <identifier>(1)
. . . . . revisedcost(0)
. . . block(1)
. . . . assign(2)
. . . . . <identifier>(1)
. . . . . revisedcost(0)
. . . . . <identifier>(1)
. . . . . f1(0)
. . <identifier>(1)
. . . return(0)
. . <identifier>(1)
. . . revisedcost(0)
. . <null>(0)
. function(8)
. . <identifier>(1)
. . . void(0)
. . <identifier>(1)
. . . checkifValidDate(0)
. . params(1)
. . . dcln(2)
. . . . <identifier>(1)
. . . . . Date(0)
. . . . <identifier>(1)
. . . . . i1(0)
. . dclns(1)
. . . dcln(2)
```

```
. . . . <identifier>(1)
. . . . . Date(0)
. . . . <identifier>(1)
. . . . . d(0)
. . call(2)
. . . <identifier>(1)
. . . . d.setYear(0)
. . . <integer>(1)
. . . . 1970(0)
. . call(2)
. . . <identifier>(1)
. . . . d.setMonth(0)
. . . <integer>(1)
. . . . 1(0)
. . call(2)
. . . <identifier>(1)
. . . . d.setDate(0)
. . . <integer>(1)
. . . . 1(0)
. . if(2)
. . . >(2)
. . . . rhscall(1)
. . . . . <identifier>(1)
. . . . . . i1.getDate(0)
. . . . rhscall(1)
. . . . . <identifier>(1)
. . . . . . d.getDate(0)
. . . block(1)
. . . . emptyprintf(1)
. . . . . <string>(1)
. . . . . . "Invalid Date !"(0)
----------------------------
```

Appendix F. Semantic Analysis Results Output

```
Variable Name     :pfinish
Alias             :
Table Name        :MSP_Projects
Column Name       :Project_Finish_Date
Data Type   :Date
Meaning           :Project Finish Date for Avalon
Business Rules  :
      if   ((b.getDate()   <   Project   Start   Date   .getDate())   ||
      (a.getDate() > Project Finish Date for Avalon .getDate()))
      {
            printf("The task start and finish dates have to be within
            the project start and finish dates");
      }


Variable Name     :revisedcost
Alias             :
Table Name        :
Column Name       :
Data Type   :float
Meaning           :Estimated New Task Unit Cost :
Business Rules  :
            Estimated New Task Unit Cost :  = c + c * 20/100;

            Estimated New Task Unit Cost :  = c;

Variable Name     :tfinish
Alias             :t1
Table Name        :MSP_Tasks
Column Name       :Task_Finish_Date
Data Type   :Date
Meaning           :Task Ending Date
Business Rules  :
      c = checkDuration(b, a, c);

      if   ((b.getDate()   <   Project   Start   Date   .getDate())   ||
      (a.getDate() > Project Finish Date for Avalon .getDate()))
      {
```

```
        printf("The task start and finish dates have to be within
        the project start and finish dates");
    }


    if (b.getDate() - a.getDate() < 10)
    {
        // 20 % raise in cost for rush orders
        Estimated New Task Unit Cost :  = c + c * 20/100;
        printf("Estimated New Task Unit Cost : " + Estimated New
        Task Unit Cost : );
    }
    else
    {
        Estimated New Task Unit Cost :  = c;
    }


Variable Name     :tstart
Alias             :i1
                  :s1
Table Name        :MSP_Tasks
Column Name       :Task_Start_Date
Data Type   :Date
Meaning           :Task Beginning Date
Business Rules  :
    c = checkDuration(b, a, c);

    if   ((b.getDate()   <   Project   Start   Date   .getDate())   ||
    (a.getDate() > Project Finish Date for Avalon .getDate()))
    {
        printf("The task start and finish dates have to be within
        the project start and finish dates");
    }


    if (b.getDate() > d.getDate())
    {
        printf("Invalid Date !");
    }

    if (b.getDate() - a.getDate() < 10)
```

```
{
      // 20 % raise in cost for rush orders
      Estimated New Task Unit Cost :  = c + c * 20/100;
      printf("Estimated New Task Unit Cost : " + Estimated New
      Task Unit Cost : );
}
else
{
      Estimated New Task Unit Cost :  = c;
}
```

```
Variable Name     :tcost
Alias             :f1
Table Name        :MSP_Tasks
Column Name       :Task_UnitCost
Data Type   :float
Meaning           :Task Cost
Business Rules  :
      c = checkDuration(b, a, c);

      Estimated New Task Unit Cost :  = c + c * 20/100;

      Estimated New Task Unit Cost :  = c;
```

```
Variable Name     :pstart
Alias             :
Table Name        :MSP_Projects
Column Name       :Proj_Start_Date
Data Type   :Date
Meaning           :Project Start Date
Business Rules  :
      displayString = "Project Start Date " + Project Start Date ;

      if   ((b.getDate()   <   Project   Start   Date   .getDate())   ||
      (a.getDate() > Project Finish Date for Avalon .getDate()))
      {
            printf("The task start and finish dates have to be within
            the project start and finish dates");
      }
```

```
Variable Name     :pcost
Alias             :
Table Name        :MSP_Projects
Column Name       :Project_Cost
Data Type   :float
Meaning           :
Business Rules  :
        if (checkCost(pcost) > 1000000)
        {
            //Give 10% discount for big budget projects
            pcost = pcost - pcost * 10/100;
        }
```