NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

A Semantic Analyzer for an Object-Oriented Language

Wai Ming Wong

A Thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements for
the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

July. 1993

ISBN 0-315-87294-2

Canada

Abstract

A Semantic Analyzer for an Object-Oriented Language

Wai Ming Wong

The object-oriented paradigm has been a subject of much research and discussion in recent years. The paradigm encourages meaningful and well-defined data abstractions by introducing the concepts of classes, instances and inheritance. The intense interest in the paradigm has motivated the design of many new object-oriented languages as well as extensions to existing ones. The new language features required to support object-orientation present new and unique challenges for compiler developers.

We view semantic analysis as the task of ensuring that the source program is semantically correct. It is not to be confused with the task of code generation and optimization.

In this thesis we present the design of a semantic analyzer for an object-oriented language called Dee. The design includes the data structures, algorithms as well as the interface with other components of the compiler.

Given that the interactions between different language features of Dee can be very complex, we demonstrate that a highly modular and consistent semantic analyzer can be achieved with appropriate language and compiler design decisions.

The semantic analyzer has been implemented using the C language and has been incorporated into the Dee compiler. The Dee compiler and its development environment are currently available for use on Unix-based workstations.

Acknowledgments

The author wishes to express the deepest gratitude to his supervisor, Dr. Peter Grogono, without whose careful supervision and patience this work would not have been possible. The author also wishes to thank Benjamin Yik-Chi Cheung and Lawrence A. Hegarty for their valuable help and discussions.

List of Figures

Contents

Chapter 1

Introduction

Software maintenance is undoubtedly the most costly operation during the life cycle of a software product. There has been a continuous search for a suitable programming paradigm or methodology to reach the goal of maintainability. As indicated by [4], the recently advocated object-oriented paradigm satisfies the deep requirement of an earlier approach of structured programming. Both approaches promote higher level of abstractions rather than the underlying implementation machinery. One of the reasons for the attention on object-oriented paradigm is the promise or belief that it surpasses the ability of the structured programming approach in producing highly maintainable software. The Dee research project is an attempt to achieve high maintainability by adopting the object-oriented paradigm.

Dee is an integrated development environment which includes the Dee language compiler as well as the facilities for developing and maintaining Dee programs. It is intended to provide both the efficiency found in compiled languages like C++ [7] and the rich development found in interpreted languages like Smalltalk [2].

The Dee semantic browser provide rapid access to accurate and up-to-date views of the classes [1] [9]. The environment is further enriched by a set of well-defined system classes.

The Dee language (simply called Dee) is a strongly-typed, class-based, object-oriented language. It supports concepts of classes, instances, single and multiple

inheritance and, genericity. The design of Dee has been intentionally kept simple to facilitate implementation as well as to improve readability and conciseness. Simplicity does not imply incompleteness. Dee is also designed to completely and consistently support the object-oriented paradigm. If a language provides features that deviate from the chosen programming paradigm, it cannot achieve the full benefits of the paradigm. A program written in C can be compiled by a C++ compiler but this does not make the program object-oriented! A language cannot rely on the programmers to follow the rules of the paradigm and select the appropriate language features to use. If that is the case, one could argue that assembly languages are object-oriented! One can always write an object-oriented program in assembler. In fact, a language cannot be classified as object-oriented if it does not have explicit features supporting inheritance [8]. In essence, Dee is a full object-oriented language without being polluted by odd language features that are counter productive. Dee is a tool which naturally guides the programmer to think in terms of object-orientation.

Since Dee is intended for the development of production-quality software, Dee source programs are compiled rather than interpreted. The task of developing the compiler for Dee is both unique and challenging due to its full support of object orientation, genericity and views. The design and implementation of the Dee parser and code generator can be found in the thesis by Lawrence Hegarty [5]. The thesis by Benjamin Cheung and the report by Joe Yau illustrate the semantic browser [1] [9]. In this thesis, we describe the semantic analyzer for the Dee compiler. Contrarily to the common practice of interleaving code generation with semantic analysis, we consider semantic analysis as strictly checking the static semantic correctness of the source program. This strict definition of semantic analysis allows us to dedicate effort for its development making sure that the semantic rules are properly applied.

In Chapter 2, we give a brief description of the general organization of Dee classes which serves as a base for discussions in subsequent sections. In Chapter 3, we describe the overall architecture of the Dee compiler. In Chapter 4, we give detailed explanations on the functions of the semantic analyzer. In Chapter 5, we describe

the implementation. In Chapter 6, we conclude our discussion with an assessment of the work completed.

Chapter 2

Constituents of a class in Dee

A Dee program consists of a root class and the classes needed by the root class. The root class in Dee is application specific. A class in Dee is a static class from which objects are instantiated. The class itself is not an object. Instead, it defines the behaviour of its instances. The components of each of its instances are determined by its instance variables. Its methods dictate the actions each of its instances can perform.

A class consists of a header and a list of attributes. The header contains the name of the class, class parameters and, a list of ancestor classes. Attributes are either instance variables or methods. The following is an example class in Dee:

Example 2.1

```
class Point
   public var x:  Float
   public var y:  Float
   var colour :  String
   public cons MakePoint ( ix :  Float iy :  Float )
   begin
      x := ix
      y := iy
```

```
    colour := "Red"
end
public method MovePoint ( nx :  Float ny :  Float )
begin
    x := nx
    y := ny
end
```

The class name uniquely identifies the class. In the above example, the class Point has three instance variables; x, y and, colour. The instance variables are by default private having a scope of the class in which they are declared; called the host class. Public instance variables are visible by clients of the class whereas private ones are not. We could declare an instance variable of the type Point in a client as follows:

Example 2.2

```
class ABC -- ABC is a client of Point.
    var dot :  Point -- instance variable dot.
    ...
    ...
    public method MoveDot ( nx :  Float ny :  Float )
    var oldx :  Float
    var old_colour :  String
    begin
        ...
        oldx := dot.x -- legal access
        old_colour := dot.colour -- illegal access
        ...
        ...
    end
```

The statement, old_colour := dot.colour is not allowed since colour is a private variable of Point. On the other hand, read access to x is allowed. Dee does not allow a client to write to public variables of its suppliers. Therefore, the expression dot.x is not allowed to be on the left-hand-side of an assignment statement.

A method has a signature and an optional body. The MakePoint method of Point has the following signature:

```
public cons MakePoint ( ix :  Float iy :  Float )
```

Similarly, the MovePoint method has the signature of:

```
public method MovePoint ( nx :  Float ny :  Float )
```

Visibility of methods can be public or private. Again, the default is private. A method can be a regular method; like MovePoint or a constructor; like MakePoint. A constructor is used to create objects. The method name is a unique name within the host class identifying the method. A method can have any number of formal arguments and an optional result type. A method may not return any object and the result type is omitted as in MovePoint of class Point. A method may have local variables and a body. The local variables have local scope and they are never visible to clients nor to descendants. Examples of local variables are oldx and old_colour in the method MoveDot in class ABC. A method body consists of a sequence of Dee statements. A method without a body is considered to be abstract. Its implementation is to be defined in the descendants of the host class.

Dee has statements similar to imperative languages except procedure or function calls. A call is a method invocation which involves a receiver object, an attribute name, and an argument list if required. The attribute name may be an instance variable name or a method name. It is referred to as the message. Method invocation as a whole is called an application. Applications are shown in the following example:

Example 2.3

```
class Line
   public var a :  Point
   public var b :  Point
   public cons
      MakeLine ( iax :  Float iay :  Float
      ibx :  Float iby :  Float)
   begin
      a.MakePoint ( iax, iay )
      b.MakePoint ( ibx, iby )
   end
   public method HorizontalMove ( step :  Float )
   begin
      a.MovePoint ( a.x + step, a.y )
      b.MovePoint ( b.x + step, b.y )
   end
```

In the above example, all statements or expressions involving the dot operator are applications; for example "a.MakePoint" and "a.MovePoint". In Dee, an application involving a constructor implicitly creates the object. Since MakePoint is a constructor, the compiler also generates the code to create the object wherever MakePoint is invoked. Therefore, the application "a.MakePoint (iax, iay)" creates the object a.

An expression in Dee is actually a series of applications which yields an object. In the application "a.MovePoint (a.x + step, a.y)" above, the actual arguments are expressions. The expression "a.x + step" is internally represented as "a.x._plus (step)". Since the dot operator associates to the left the expression is interpreted as "(a.x)._plus(step)". The application a.x yields an object and this object becomes the receiver of the _plus message.

Chapter 3

The Dee compiler

A class is a compilation unit in Dee. Programmers write classes using regular text editors and the source files are stored as conventional text files. Figure 3.1 shows the inputs and outputs of the Dee compiler.

The code generated by the compiler is in the form of C source files. The Maker and Linker utilities read the C sources and produce the executable code file. Although this approach to code generation is not unique to Dee, it is considered to be highly practical. It allows us to capitalize on the efforts already invested in machine code generation and optimization for the C compiler. Since C is a widely used and relatively standardized language, the Dee compiler becomes portable across various machines.

Each Dee class is defined by a single document called the canonical document. It is the Dee source file that the owner develops. In addition to regular functionalities, the compiler also creates a class interface for the class it currently compiles. An interface is a machine-readable file describing all attributes of the class. Interfaces of all compiled classes are saved in a data base managed by the class interface manager [1] [9]. The compiler often searches the data base during compilation to obtain information about other classes. If it does not detect any error, it will write the interface of the compiled class to the data base. The same interfaces data base is also used by the semantic browser to create human-readable views of classes for the

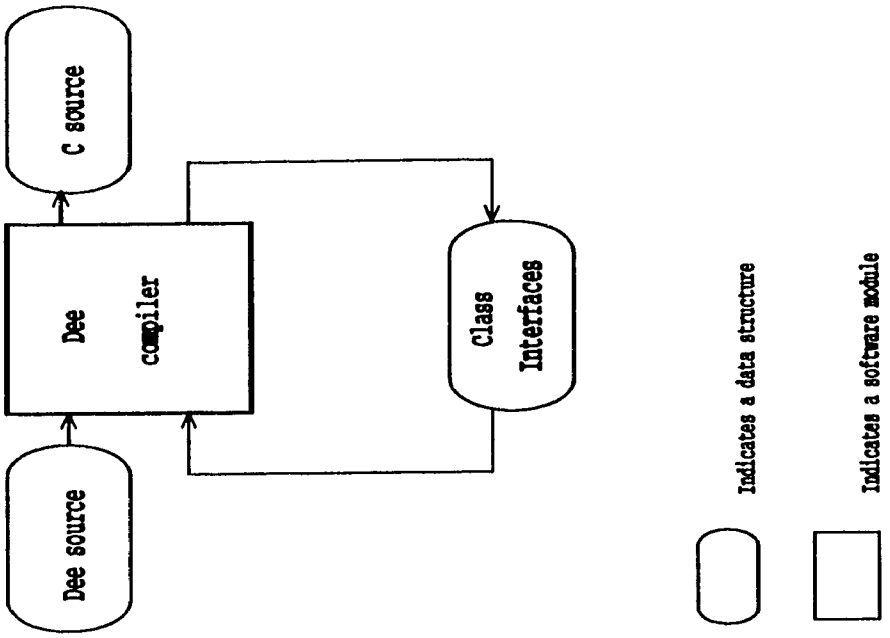


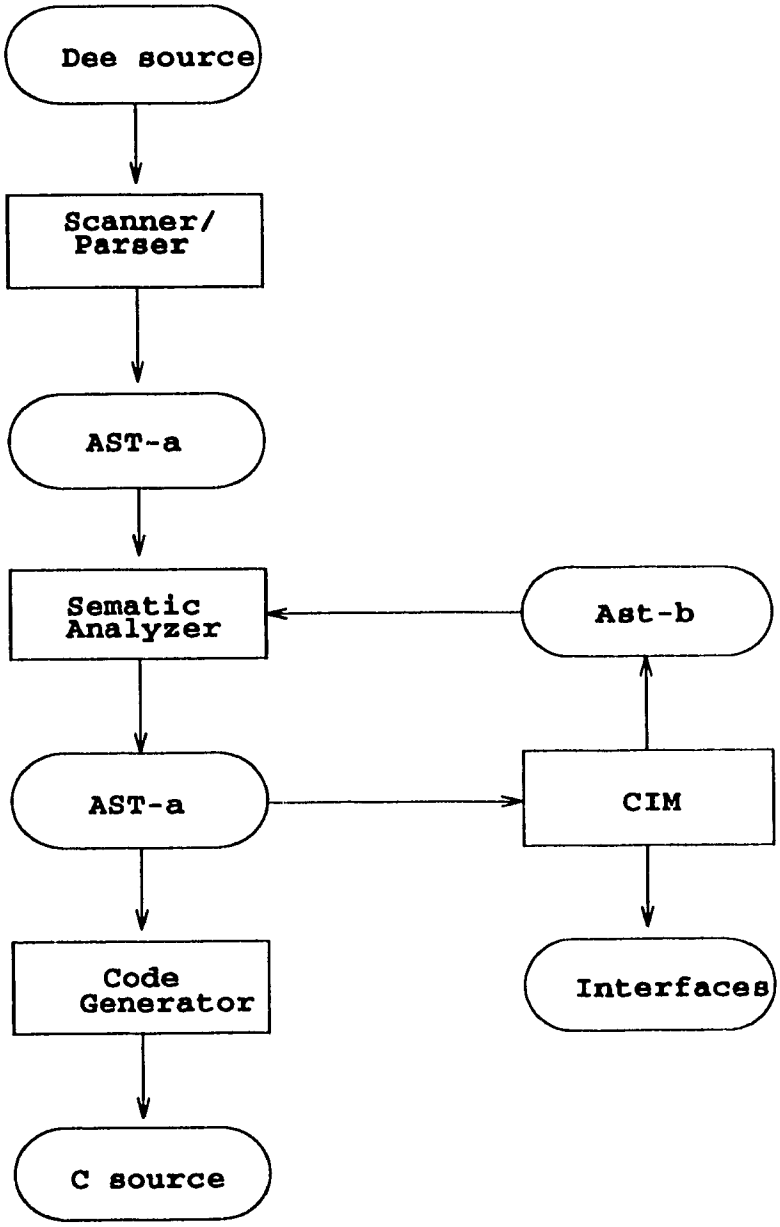
Figure 3.1:

programmers.

The compiler itself is made up of four modules; the parser, the code generator, the class interface manager and, the semantic analyzer. Their relationship is shown in Figure 3.2.

The scanner and parser convert the source file into an abstract syntax tree; the AST-a in Figure 3.2. The abstract syntax tree is a data structure created in memory to be used throughout the compilation process as a common communication medium for all the modules. It is a transient structure which exists only during compilation. The semantic analyzer analyses the AST-a and adds semantic information to it. The code generator, in turn makes use of all information generated by other modules to emit C codes.

The semantic analyzer is the only module interacting directly with the class interface manager module. During semantic analysis, it requires the interfaces of other already compiled classes. The class interface manager performs the necessary accesses to the class interface data base. The requested interface is then converted into an



Ast-a is the Abstract Syntax Tree of the Class being Compiled.

Ast-b is the Abstract Syntax Tree of the Classes it needs.

Figure 3.2:

abstract syntax tree structure which the semantic analyzer is capable of processing. If no error is found, the abstract syntax tree of the currently compiled class is passed to the class interface manager for data base update. At this point the class is free from any syntactic and semantic errors. The code generator can safely perform its tasks based on the resulted abstract syntax tree.

Chapter 4

Functions of the Semantic Analyzer

The semantic analyzer enforces the static semantic rules that are not detected by the Dee language grammer. In this thesis, we demonstrate the semantic rules by examples. The complete description of the Dee semantics is found in chapter 3 and 4 of [3].

We have identified the following functions that the semantic analyzer has to per form:

- Processing of inheritance and extension.
- Processing of genericity.
- Type substitutions.
- Resolution of names.
- Detection of type errors.
- Detection of structural errors.

4.1 Processing inheritance and extension

Inheritance is a powerful mechanism for software reuse in the object-oriented paradigm. It allows classes to be defined by modifying or extending existing ones. Extendibility is possible through constrained polymorphism and dynamic binding.

Dee supports multiple inheritance and extension. Extension is the same as inheritance except that all extended attributes are private unless explicitly redeclared as public in a child class. Public attributes are visible by client classes whereas private ones are not. An example of extension would be a stack. A stack may be implemented by an array but the clients of stack should not be able to perform random accesses to the array. When a stack extends an array, the array operations become private and are only visible to the class stack but not the clients of stack. Consider the following example inheritance hierarchy:

Example 4.1

```
class P
   public var pa :  Int
   public var pb :  Int
   private var pc :  Boolean
   private var oldp :  P
   public method pma ( ax :  Int ay :  Int ) :  Int
   begin
      .....
   end
   private method pmb
   begin
      .....
   end
```

```
class Q
   public var qa :  Float
   public var qb :  Float
   private var qc :  String
   public method qma ( ax :  Float ay :  Float ) :  Float
   begin
      .....
   end
   private method qmb ( ax :  Float )
   begin
      .....
   end

class C
   inherits P
   extends Q

   public var ca :  Float
   private var cb :  Point
   public method cma ( loc :  Point )
   begin
      .....
   end
```

In this example, both P and Q are parents of C. Conversely, C is a child of both P and Q. The ancestor relation is the reflexive, transitive closure of parent relation. The descendant relation is the reflexive, transitive closure of child relation[3]. Since the closures are reflexive, C is considered as both the ancestor and the descendant of itself. The class C gets all the attributes of P and Q. C may declare additional attributes as well as overloading (redeclaring) the inherited ones. Let Ap, Aq and

Ac be a set of attribute names declared in class P, Q and C respectively. As a result of inheritance, the set of actual attribute names that C has access to is the union of Ap, Aq and Ac. We call this set AC. In general, AC is the union of all attribute names of the child class and all its parents. AI is a set of names that appear in at least two classes in the inheritance hierarchy. A substantial portion of the semantic rules regarding inheritance deal with the resolution of name conflicts when AI is not an empty set. The classes in Example 4.1 can be described as:

Example 4.2

```
Ap = { pa, pb, pc, oldp, pma, pmb }

Aq = { qa, qb, qc, qma, qmb }

Ac = { ca, cb, cma }

AC = Ap U Aq U Ac
   = { pa, pb, pc, oldp, pma, pmb, qa, qb, qc,
     qma, qmb, ca, cb, cma }

AI = {}
```

In this simple example, AI is an empty set and we do not have any name conflicts.

4.1.1 Effects of inheritance

In general, for each attribute e in AC and e is not in AI, the semantic analyzer sets the attribute visibility to private if e is inherited by extension. Otherwise, the visibility is unchanged. The only exception is that an inherited or extended constructor is always set to private. The rationale for this exception is that the set of attributes of the parent is a subset of the attributes of the child class. The constructor for the

parent is therefore very likely to be insufficient for the child class and it should not be visible to clients of the child. The child should declare its own constructor. However, the parent's constructor is useful for the child class. It can be used to initialize the inherited attributes in the child's constructor.

Some parent attributes may contain the parent class name as the type. This is demonstrated in Example 4.1 by the variable oldp in class P.

```
private var oldp :  P
```

The semantic analyzer changes the parent class name to the child class name when inheriting oldp. The attribute oldp for class C becomes:

```
private var oldp :  C
```

The result of inheritance can be seen in the following descendant's view of the class C:

Example 4.3

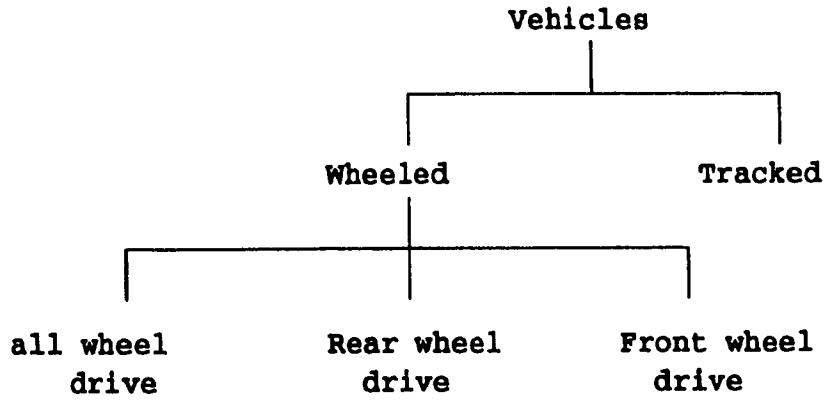
```
class C
   inherits P
   extends Q

   -- Attributes from P
   public var pa :  Int
   public var pb :  Int
   private var pc :  Boolean
   private var oldp :  C
   public method pma ( ax :  Int ay :  Int ) :  Int
```

```
private method pmb

-- Attributes from Q
private var qa :  Float
private var qb :  Float
private var qc :  String
private method qma ( ax :  Float ay :  Float ) :  Float
private method qmb ( ax :  Float )

-- Attributes from C itself
public var ca :  Float
private var cb :  Point
public method cma ( loc :  Point )
```

4.1.2 Constrained polymorphism and the concept of class conformance

Polymorphism is defined as the ability to take several forms [6]. In object-oriented programming, this refers to the ability of an object to be treated as an instance of various classes. Given that unconstrained polymorphism is undesirable, polymorphism in object-oriented programming is constrained by inheritance. The inheritance hierarchy depicts the possible classes to which an object may belong. An instance of a child class can be considered as an instance of all its parent classes at all levels of inheritance. Consider the following inheritance hierarchy:

Example 4.4



In the above example, let the object "my family car" be an instance of "front wheel drive vehicles". The object "my family car" can also be an instance of "wheeled vehicles" as well as "Vehicles" in general. However, "my family car" cannot be an instance of "Tracked vehicles". The class "front wheel drive vehicles" is said to conform to the class "Wheeled vehicles" as well as to the class "Vehicles".

In general, a class X conforms to a class Y when Y is a parent of X. If X conforms to Y, every instance of X is also an instance of Y and every attribute of Y is also an attribute of X. The class X has fewer instances than Y but it has more attributes. Based on our definition of the ancestor and descendant relation, X conforms to itself.

4.1.3 Resolution of attribute name conflicts

A name conflict occurs when an attribute of a certain name is declared in more than one parent of a child class. The semantic analyzer detects name conflicts automatically and produce errors accordingly. In fact, name conflicts have to be resolved before codes can be generated. An example of name conflict is shown below:

Example 4.5

```
class P
   ....
   public var tmp :  Int
   ....
```

```
    ....

class Q
    ...
    public var tmp :  Float
    ....
    ....

class C
    inherits P
    extends Q
    ....
    ....
    method abc ( a :  C ) : C
    begin
        ....
        tmp := 1;
        ....
    end
```

In this example, the class C is supposed to inherit the instance variables tmp from both P and Q. The statement; "tmp := 1" is ambiguous since it is not clear which tmp variable the programmer is intended. The semantic analyzer cannot arbitrarily choose one or the other.

Resolving instance variable name conflicts

The only situation where a variable name conflict is not ambiguous is that all variables with the conflicting name actually denote the same variable. Another way of describing it is that the variables come from the same source class. Consider the following example:

Example 4.6

```
class GP
   public var tmp :  Int
   ....
   ....

class P
   inherits GP
   ....
   ....

class Q
   inherits GP
   ....
   ....

class C
   inherits P
   extends Q
   ....
   ....
```

Since class P and Q inherits from the class GP, they both inherit the variable tmp. The tmp in P and the tmp in Q actually are the same variable inherited from the class GP. In this case, the name conflict is permitted. Given that a class can have class parameters, the variable tmp may have a type more complex than Int. The semantic analyzer also has to ensure that the type of the conflicting variables in all parent classes have to be the same. Detail discussion on genericity and class parameters can be found in Section 4.2.

Resolving method name conflicts

Method name conflicts are also resolved using unambiguous cases provided that the signatures of all methods with the same name in all parents are equal. Two method signatures are equal if the number of arguments are the same and the types of all formal arguments and result type are the same.

The semantic analyzer tries to resolve the conflicts automatically where it is possible to do so. In the cases where conflicts cannot be resolved automatically, Dee allows the programmer to associate a "from clause" to a method in the child class. A method with a from clause cannot have a body. Instead, it specifies the parent class in which the method body to be used is found. An example use of the from clause is as follows:

Example 4.7

```
class P
   ....
   public method abc ( a :  Int ) :  Int
   begin
      ....
   end

class Q
   ....
   public method abc ( a :  Int ) :  Int      .
   begin
      ....
   end

class C
   inherits P Q
```

```
....
public method abc ( a :  Int ) :  Int from P
```

In this example, the method body of the method abc in class P is used.

The semantic analyzer has to enforce the following rules regarding the use of the from clause:

1. The class specified in the from clause must be one of the immediate parent classes. In our example, it has to be either P or Q but it cannot be one of the parents of P and Q.

2. The signature of method in the child class has to conform to the signature of the one in the specified parent class. Signature conformance means that all the formal argument types and the return type conform to the ones in the parent. The number of formal arguments must also be the same.

3. The method body in the class specified by the from clause has to be concrete. Since a from clause can only refer to a concrete method, a method body with a from clause is considered as concrete.

4. The use of the from clause in a method found in only one of the parents is redundant but it is not an error.

The following table depicts the rules used by the semantic analyzer to automatically resolve method name conflicts. Whenever the semantic analyzer cannot resolve a name conflict, the table indicates that the programmer has to specify a from clause for the current class to be compiled without errors. The table also describes a special case of inheritance where inheritance takes place even when a method is redeclared in a child class. If a method is redeclared with an abstract body and one of its parents has a method with the same name having a concrete body. The child class inherits this concrete body.

Only two parent classes are used in the table for illustration purposes. The same rules apply to any number of parent classes.

Parent0	:	A parent class having a method of the same name
Parent1	:	A parent class having a method of the same name
Child	:	A child class either inheriting or redefining the method
Actual Body	:	The body actually used after inheritance
Requires From:		The From clause has to be used to direct inheritance
abs	:	Abstract method body
con	:	Concrete method body
Nil	:	No method of the same name defined in the class

Parent0	Parent1	Child	Actual Body	Requires From	Comments
abs	nil	abs	abs	no	remains abstract
con	nil	abs	con	no	child gets the body parent 0
abs	nil	con	con	no	use the body of the child
con	nil	con	con	no	use the body of the child
abs	nil	nil	abs	no	remains abstract
con	nil	nil	con	no	child gets the body parent 0
abs	abs	abs	abs	no	remains abstract
abs	con	abs	con	no	child gets the body parent 1
con	abs	abs	con	no	child gets the body parent 0
con	con	-a-	con	yes	child gets the body from chosen parent
abs	abs	con	con	no	use the body of the child
abs	con	con	con	no	use the body of the child
con	abs	con	con	no	use the body of the child
con	con	con	con	no	use the body of the child
abs	abs	nil	abs	no	remains abstract
abs	con	nil	con	no	child gets the body parent 1
con	abs	nil	con	no	child gets the body parent 0
con	con	-a-	con	yes	child gets the body from chosen parent

Table (1)

–a– The child has to define the method with either a concrete body or a from clause to guide inheritance. In cases where the method is not defined in the child at all or it is defined with an abstract body in the child, the semantic analyzer produces an error.

4.1.4 Redeclaration of attributes

In Dee, The semantic analyzer automatically inherits all attributes of the parents if they do not present any conflicts or the conflicts can be resolved. The programmer

may also redeclare the inherited attributes based on certain rules. The semantic analyzer is responsible for enforcing these redeclaration rules. An example of instance variable redeclaration is shown below:

Example 4.8

```
class P
   public var pa :  Int
   public var pb :  Int
   private var pc :  Boolean
   private var oldp :  P
   public method pma ( ax :  Int ay :  Int ) :  Int
   begin
      .....
   end
   private method pmb
   begin
      .....
   end

class Q
   public var qa :  Float
   public var qb :  Float
   private var qc :  String
   public method qma ( ax :  Float ay :  Float ) :  Float
   begin
      .....
   end
   private method qmb ( ax :  Float )
   begin
      .....
```

```
    end

class C
    inherits P
    extends Q

    public var pc :  Float -- Redeclaring pc of P to public.
    public var ca :  Float
    private var cb :  Point
    public method cma ( loc :  Point )
    begin
        .....
    end
```

Redeclaration of instance variables

If the programmer is redeclaring an instance variable in a child class, the instance variable must be of the same type as the one in its parent. Only the visibility of the variable can be redeclared. The possible visibility redeclarations are shown below:

	Extended variable			Inherited variable		
	Parent	Child	Result	Parent	Child	Result
a)	Public	Public	Accepted	Public	Public	Accepted
b)	Public	Private	Error	Public	Private	Error
c)	Private	Public	Accepted	Private	Public	Accepted
d)	Private	Private	Accepted	Private	Private	Accepted

In case a and d, a warning message is produced.

The above tables shows that the rules for both extended and inherited variables are the same. A private variable is allowed to be redeclared either public or private. In case a and d, the visibility is the same. It is not an error. Instead, there are warning messages saying that the redeclaration is exactly the same and is useless.

The only error case is case b where a public variable can never be redeclared as private. Examples of visibility redeclaration are shown in the following example where the parent classes P and Q are the same as in Example 4.8:

Example 4.9

```
class C
   inherits P
   extends Q

   private var pa :  Int -- disallowed
   public var pc :  Boolean -- allowed
   public var ca :  Float
   private var cb :  Point
   public method cma ( loc :  Point )
   begin
      .....
   end
```

Redeclaration methods

If the programmer is redeclaring a method in a child class, both the visibility and the method signature may be redeclared. The redeclared method must have a signature conforming to the signature of the method in all the parents. The following table illustrates the visibility redeclaration rules:

	Extended variable			Inherited variable		
	Parent	Child	Result	Parent	Child	Result
a)	Public	Public	Accepted	Public	Public	Accepted
b)	Public	Private	Error	Public	Private	Error
c)	Private	Public	Accepted	Private	Public	Accepted
d)	Private	Private	Accepted	Private	Private	Accepted

The above rules are similar to the ones for redeclaring visibility of instance variables. The only exceptions are the warning messages. Even though the visibility is the same, the method signature and the method body may vary. In that case, we cannot report a warning that the redeclaration is exactly the same and is useless. An example of method redeclaration is shown in the following example where the parent classes P and Q are the same as in Example 4.8:

Example 4.10

```
class C
   inherits P
   extends Q

   public var ca :  Float
   private var cb :  Point
   public method cma ( loc :  Point )
   begin
      .....
   end
   public method qmb ( ax :  Float ) -- qmb from Q is now public
   begin
      .....  -- May have a different body.
   end
```

A method may be redeclared to have a concrete body, an abstract body, or a from clause. When the redeclared method has a concrete body, the body in the child class is used. As indicated in Section 4.1.3, a method with a from clause has a concrete body. The body referenced by the from clause is used.

When the redeclared method has an abstract body, the net result of the redeclaration depends on its parent as illustrated below :

Assuming the following inheritance hierarchy:

```
class P
   method abc ( a :  Int ) :  Int
   begin -- A concrete body.
      ....
   end

class Q
   method abc ( a :  Int ) :  Int
   begin -- An abstract body.
   end

class C
   inherits P Q
   method abc ( a :  Int ) :  Int
   begin -- An abstract body.
   end
```

In the above example, method abc in class C has an abstract body and it has a concrete body in P, C inherits the method body from P. If the method abc in both P and Q have abstract bodies, the abc in C remains abstract. However, if abc in both P and Q have concrete bodies, the programmer has to specify which body to use by a from clause.

4.2 Processing genericity

Genericity is also a technique for software extendibility and reuse. Although it offers similar benefits as inheritance, genericity and inheritance are considered to be complementary [6]. It is the ability to define type parameters for modules. It allows the same code module to be used to manipulate data of different types. This concept of

type parameters is adopted by statically typed object-oriented languages like Eiffel and Dee where classes may have class parameters. In Dee, programmers can define generic classes for general use. For example:

Example 4.11

```
class Table ( key :  Comparable info :  Any )
....
....
```

In a client class of Table, an attribute may be declared as the following:

```
address_book :  table ( String, Address )
```

In the above example, String corresponds to the class parameter key and Address corresponds to the parameter Any.

In Dee, class parameters must be qualified. In the above example, key is qualified by the class Comparable and info is qualified by Any. The classes Comparable and Any are called constraining classes. The semantic analyzer verifies that the constraining classes are valid classes and that the actual arguments conforms to the constraining classes. It also has to ensure that the number of actual arguments are the same as the number of parameters declared in the class Table. In the above example, String conforms to Comparable and Address conforms to Any. The class Any is provided to be a sensible default for parameter qualification. Any is an ancestor of some, but not all, classes.

4.2.1 Mixing inheritance and genericity

Dee allows the mixed use of inheritance and genericity. In other words, parameterized classes can inherit attributes from other parent classes. They can also be parents of

other classes. However, the semantic analyzer has to ensure that all constraining classes of the child class conform to the constraining classes of all the parents. The number of parameters also has to be the same. An example of mixed inheritance and genericity is the following:

Example 4.12

```
class P ( t :  Any )
   var
      a :  t
   ....
   ....

class Q ( u :  Any )
   var
      b :  u
   ....
   ....

class C ( s :  Comparable )
   inherit P Q
   ....  ....
```

In the above example, all classes have one class parameter and the constraining class Comparable conforms to Any. The class P and Q are allowed to be the parents of C and C is allowed to be the child of P and Q.

4.2.2 Complicated class parameter declarations

Since a class may have any number of parameters and the constraining classes themselves may also have parameters, the parameter list of a class may become very complex. An example of a complex class parameter list is demonstrated below:

```
class C ( A : List ( String Table ( String Employee ) ) )
```

The type of any attribute declaration may also be very complex. For example:

Example 4.13

```
var
   temp :  C ( List ( String Table ( String Employee ) ) )
```

The semantic analyzer should be able to process as many levels of nesting as the system resources permit.

4.3 Type substitution

Class parameters are used as types in attribute declarations. The concept of type in Dee is clarified in Section 4.5. The following example shows the use of class parameters as types for class attributes:

Example 4.14

```
class List ( K : Comparable T : Any )
   public var
      temp :  T

   public method insert ( key :  K node :  T )
   var
      work_node :  T
   begin
      ....
   end
```

```
public method delete ( key :  K )
begin
   ....
end
public method find ( key :  K ) : T
begin
   ....
end
....
....
```

K and T are class parameters that can be used as types in the class List. When List is compiled, the variable "temp" is treated as an object of class Any. The method parameter "key" is treated as an object of class Comparable.

Type substitution refers to the replacement of class parameter names by their corresponding constraining class names. The semantic analyzer determines that T is a class parameter name by T's presence in the parameter list of the current class List. It then replaces T by the constraining class Any. Since class attributes and local variables may be referenced many times, the semantic analyzer performs type substitution only once for all the attributes of the current class before type checking is performed. Therefore, type substitution overhead is reduced during type checking. After the semantic analyzer has performed type substitution, the class List becomes:

```
class List ( K : Comparable T : Any )
   public var
      temp :  Any
   public method insert ( key :  Comparable node :  Any )
   var
      work_node :  Any
   begin
      ....
```

```
end
public method delete ( key :  Comparable )
begin
   ....
end
public method find ( key :  Comparable ) :  Any
begin
   ....
end
....
```

If List is a parent of another class, the constraining classes of the child are used in type substitution. Consider the following example class:

Example 4.15

```
class MyList ( I : Int B : Any )
   inherits List
   public var
      mykey :  I
   ....
   ....
```

Type substitution is shown in two steps for clarity. First, the class parameter names of List are replaced by corresponding class parameter names of MyList. The result is shown below:

```
class MyList ( I : Int B : Any )
   inherits List

   -- Attributes from List.
```

```
public var temp :  B
public method insert ( key :  I node :  B )
public method delete ( key :  I )
public method find ( key :  I ) : B

-- Attribute of MyList

public var mykey :  I
....
....
```

Then, the class parameter names are replaced by their corresponding constraining classes. The result is shown below:

```
class MyList ( I : Int B : Any )
   inherits List

   -- Attributes from List.

   public var temp :  Any
   public method insert ( key :  Int node :  Any )
   public method delete ( key :  Int )
   public method find ( key :  Int ) :  Any

   -- Attribute of MyList

   public var mykey :  Int
   ....
   ....
```

Type substitution is also required when List is a supplier of a client class. When compiling the client, the semantic analyzer fetches the signature of the required attribute of List through the class interface manager. Since parameterized classes can have different arguments, the semantic analyzer has to perform type substitution on the supplier signatures every time they are retrieved. Consider the following example:

Example 4.16

```
class C
   var
      waiting_list :  List ( String Person )
   public method add_person ( person :  Person )
   begin
      ....
      waiting_list.insert ( person.name, person ) -- Line 7
      ....
   end
   ....
   ....
```

In Line 7, the signature of insert is retrieved:

```
public method insert ( key :  K node :  T )
```

Since waiting_list is declared as an object of List (String Person), K is replaced by String and T is replaced by Person. The signature after type substitution is then used in type checking. It is shown below:

```
public method insert ( key :  String node :  Person )
```

4.4 Resolution of names

Names in a Dee class can either be class names, supplier attribute names, variable (or object) names, or class parameter names. The following example shows the different categories of names in a Dee class:

Example 4.17

```
class C ( T : Comparable )
   inherits P
   var abc :  T
   public method calculate ( a :  T ) : T
   var
      def :  MyType ( String )
   begin
      ....
      def.show ( a );
      self.abc := a;
      ....
      result := a;
   end
   ....
   ....
```

In this example, class names are C, P, Comparable, MyType and String. Variable names are abc, def, and a. The supplier attribute name is the method "show" of the supplier class MyType. Supplier attribute names are only found immediately to the right of a dot in an application. T is the class parameter name. The method "calculate" of C is declared but not used. If it is used, it will be treated as a supplier attribute name.

Dee also supports two special variables "self" and "result". Every class has a variable "self". Every method that returns a result has a variable "result". The variable "self" is a convenient way of referring to the current class. The variable "result" is used to return result of a method. Result is returned by assigning to the variable "result". In the above example, "self" has the type C(T) whereas "result" has the type T.

4.4.1 Using the class interface manager

The information on supplier classes and their attributes are contained in class interfaces managed by the class interface manager. The semantic analyzer invokes the appropriate routines of the class interface manager to obtain the information it needs at the time the needs arise. If the name refers to the current class or to an attribute of the current class, the information is already contained in the abstract syntax tree currently in memory and class interface manager routines will not be invoked.

4.4.2 Resolution of class names

Before the semantic analyzer can process inheritance, it has to obtain the interfaces of all the parent classes indicated in the inherit and extend lists of the current class. The interfaces describe all attributes of the parents which are to be included in the current class based on the inheritance rules described in Section 4.1. In Example 4.17, there is only one parent; P. If there is no class interface for P the class interface manager will report an error.

When a class name is used in a method or variable signature, the semantic analyzer also retrieves the required information from the class interface through the class interface manager so that class conformance can be verified. If the class MyType in Example 4.17 exists and is declared as the following:

```
class MyType ( T : Comparable )
....
```

....

The class interfaces of the classes MyType and String have to exist and String must conform to Comparable.

4.4.3 Resolution of supplier attribute names

A class being compiled often needs the attributes of other classes. The need arises when an attribute is actually used; not declared. The following example demonstrates the need for an attribute of a supplier:

Example 4.18

```
class Employee_Data_Base
   var
      data_base :  Table ( String Employee )
   public method Show_Employee ( key :  String )
   var
      one_employee :  Employee
   begin
      one_employee := data_base.search ( key ) -- Line 8
      ....
   end
   ....
   ....
```

The interface containing the attribute "search" is required at line 8 where the message "search" is sent to the object data_base. The semantic analyzer is responsible for obtaining the interface for the attribute "search" in the class Table through class interface manager. It also ensures the correct usage of the attribute in the expression. In the above example, "search" should be a method with one parameter (or formal

argument) and a return type. If the attribute is an instance variable of the class Table, the semantic analyzer will give an error. Type checking is described in details in Section 4.5.

4.4.4 Resolution of variables names

Dee does not support implicit declaration of variables. Every variable used in a class has to be declared. Variables can be instance variables of the class, method local variables, or exception handler variables. Instance variables have a scope of the current class. Local variables have a scope of the method in which they are declared. Method parameters and return result are also considered as method locals. Exception handler variables have a scope of the exception handler itself. Examples of exception handler variables are shown below:

Example 4.19

```
class C
   var
      abc :  T1
   method do_something ( v :  Comparable ) :  C
   var
      def :  Bool
      ghi :  T2
   begin
      def := ....;
      ....
      attempt abc := ghi.conversion(v);
         handle x0 :  C0
            x0 := ....;
            abc := ....; -- Line 14
            ....
```

```
        handle x1 :  C1
           x1 := ....;
           ....
        handle x2 :  C2
           x2 := ....;
           ....
           end
           ....
    end
```

In this example, the variables x0, x1 and x2 are exception handler variables.

When a variable name appears in a statement, the semantic analyzer first searches for the name in the enclosing handler. If the name is not found, it then continues the search in the enclosing method and finally the current class. If the name is not found in any of these three scoping levels, an error will be produced. In general, a variable declared in an outer scope is accessible in the inner scope. In Line 14 of the above example, the instance variable "abc" is accessible in a handler.

4.4.5 Resolution of class parameter names

Class parameters can be used in declaring objects. Consider the following class:

```
class C ( T : Comparable )
   var abc :  T
   var def :  MyType
   ....
   ....
```

The semantic analyzer detects that T is a class parameter name instead of a class name. It replaces T by Comparable. Type substitution has already been discused in Section 4.3.

4.5 Detection of type errors

Dee is a strongly typed object-oriented language with a type system and a conformance relation linked to the inheritance hierarchy. The main task of the semantic analyzer is to check the type correctness of each statement in each method in the current class.

4.5.1 Classes and types

A basic class without class parameters is a type in Dee. The declaration "i : Int" can be either described as "i is an object of class Int" or as "i has type Int". On the other hand, a parameterized class denotes many types. Consider the class Set defined below:

```
class Set ( T : Comparable )
```

Variables can be declared as the following:

```
a :  Set ( Int )
b :  Set ( String )
```

Both the types Set (Int) and Set (String) are valid types according to the rules of genericity (Section 4.2) but they denote two different types. Therefore, a type is defined by a class name and its arguments.

4.5.2 Type conformance

In Section 4.1.2, we gave a simple definition of conformance where a class X conforms to a class Y when Y is a parent of X. A more general definition must consider the class arguments. Let X and Y be types. The type X is in the form of C (A1,...,An).

Y is in the form of P (B1,...,Bm). P and C are the parameterized class names. A1 to An are arguments of C and B1 to Bm are arguments of P. X conforms to Y if all of the following are true:

1. P is a parent of C.
2. The number of arguments of C is the same as the number of arguments of P, or n = m.
3. For all x where $1 \leq x \leq n$, Ax conforms to Bx. In other words, each argument of C conforms to the corresponding argument of P.

4.5.3 Type checking

With the completion of certain preparation works like processing inheritance and type substitution, type checking becomes a relatively simple and uniform operation. The semantic analyzer checks type correctness by applying the type conformance rules described in Section 4.5.2. repeatedly on various parts of each statement in the current class.

4.5.4 Type checking statements

An example method is shown below. It is a method of the class Set used to determine if an object is a subset of another:

Example 4.20

```
class Set ( T : Comparable )
   ....
   ....
   public method <= (other:  Set(T)): Bool
       Subset relation
```

```
    var x:  T i:  Iterator(T)
begin
    new i.makeiterator(self)
    result := true -- Line 6, an assignment statement
    from i.init until i.finished do -- Line 7, a loop statement
        x := i.current
    if other.member(x) -- Line 9, an if block.
        then
    else result := false
        break
    fi
    i.next -- Line 14, an application as a statement
    od
end
....
```

An assignment statement in Dee has the general form of v := E where v is a variable and must not be the special variable self. E is an expression. The semantic analyzer determines the type of both v and E and the type of E has to conform to the type of v. An example of an assignment can be found in Line 6 of the example above.

A Dee statement can be an application in the form of x.m(a1,...,an). The receiver object is x and the message is m. The arguments for the method are a1 to an. The application should have the type void when used in this context. However, the semantic analyzer only gives a warning message if the application has any other types. An example of an application as a statement can be found in Line 14 of the example above. The type void is not a class. It is an artificial type used within the semantic analyzer only. The result type of a method that does not return result is void.

The expressions appearing after the keywords if, elsif, while, and until must have the type Bool. The class Bool does not need to be explicitly declared in the current class. An example of an if block can be found in Line 9 of the example above.

4.5.5 Type checking expressions

A Dee expression is either a literal constant, a variable or an application. The type of a literal constant is determined by the parser and it cannot be in error. A variable must be declared in Dee and the semantic analyzer looks for its declaration in order to determine its type. The search is in the order described in Section 4.4.4. Another type of expression is an application. The semantic analyzer spends most of its time type checking applications. An application is in the general form of:

```
x.m(a1,...,an)
```

Where:

```
x is the receiver.
m is the attribute name; the message.
a1 to an are the meth d arguments.
```

The semantic analyzer must find the type of an application. The following example is used to demonstrate the entire process of finding a type of an application. The supplier class Set is the one declared in Example 4.20.

Example 4.21

```
class MyClientClass
....
public method method_a () :  Bool
var
   seta :  Set ( Int )
   setb :  Set ( Int )
begin
   ....
```

```
....
if seta <= setb then -- Line 10
   ....
else
   ....
fi
....
....
```

The if statement in Line 10 is to test if the set seta is a subset of setb. For the if statement to be type correct, the expression must have the type Bool. The expression is internally represented as an application show below:

```
seta._lessequal ( setb )
```

The types of the local variables are verified when the semantic analyzer is processing the declaration section of method_a. It makes sure that class Set exists and that the argument Int conforms to the class parameter of Set. Since the class Set is declared to be "class Set (T : Comparable)" and Int conforms to Comparable, the type "Set (Int)" is a valid type.

When processing the expression, the semantic analyzer needs to find the type of the receiver object (variable) seta. It searches for its declaration according to the rules described in Section 4.4.4. It finds out that it is a local variable of method_a and has a type Set (Int).

Next, the semantic analyzer has to obtain the signature of the attribute _lessequal of the class Set. This is done by calling a routine in class interface manager. If the attribute does not exist, the class interface manager routine returns an error. In our example, the method _lessequal does exist and has the following signature:

```
public method _lessequal ( other :  Set ( T ) ) : Bool
```

Since the name T in the signature is a class parameter, the semantic analyzer needs to substitute it by the corresponding class argument of the receiver type. In our example, T is replaced by Int.

```
public method _lessequal ( other :  Set ( Int ) ) :  Bool
```

The method _lessequal has one parameter and there is one argument; setb provided in the expression. The semantic analyzer proceeds to find the type of setb also according to the rules in Section 4.4.4. It finds out the setb is a local variable and has the type "Set (Int)". For the method invocation to be type correct, each method argument must conform to its corresponding method parameter. In our example, the variable setb and the method parameter have the same type; "Set (Int)". Since a type conforms to itself, the method invocation is correct. As indicated in the signature of _lessequal, the result type is Bool. Therefore, the entire application has the type Bool and the if statement in our example is type correct.

4.6 Detection of structural errors

The semantic analyzer is also responsible for checking the structural consistency of the methods as well as the class. It has to check the following:

- A constructor must have a result type of its host class.
- If the result type of a method is not void, there must be an assignment to the variable result in every path through the method.
- If the result type is void, there must not be any assignments to the variable result.
- The keywords break and continue cannot be used outside of a loop construct.

- There must not be an assignment to the variable self.

- A class must not contain both abstract methods and constructors.

Chapter 5

The Implementation

In the first part of this chapter, we describe the key algorithms used by the semantic analyzer. We present the algorithms in pseudo code notation rather than in the implementation language (C) in order to highlight the essential points without a mass of low-level detail. The pseudo code given here, in addition to supporting explanation of the semantic analyzer, should provide a useful guide to the actual code for anyone who chooses to extend it.

Although we do not show complete C code, we have included a number of function prototypes to clarify the explanations. The Appendix B contains complete listings of a few central functions.

5.1 The pseudo codes

The pseudo codes are designed to depict the programming constructs usually found in procedural programming languages. There are loops and conditional statements which are self explanatory. The comments in the pseudo codes are enclosed in /* and */. The equal sign "=" has an assignment semantics as in the C programming language. The "==" denotes equality in a comparison; for example:

```
if ( a <in> A_set == TRUE ) then .....
```

We hope to achieve a clear and concise description by generalizing all data entities in the semantic analyzer by a collection of simple abstract data entities and a collection of operations that can be performed on these entities.

5.1.1 Sets

A set is an unordered collection of elements containing no duplicated elements. A set is defined in the following way:

```
Let Parent_Attributes = { Add, Subtract, Multiple, Divide }
```

This defines a set Parent_Attributes which has 4 elements. The operations allowed to be performed on a set are demonstrated in the following examples:

```
Let Child_Attributes  = {  Absolute, Modulo,
                           Remainder, Add, Subtract }

Parent_Attributes <union> Child_Attributes
                  yields { Add, Subtract, Multiple, Divide,
                      Absolute, Modulo, Remainder }

Parent_Attributes <intersect> Child_Attributes
                  yields { Add, Subtract }

Parent_Attributes - Child_Attributes
                  yields { Multiple, Divide }

/* Difference removes elements of the second set from the first. */

Add <in> Child_Attributes yields TRUE.
```

```
Multiply <in> Child_Attributes yields FALSE.

Child_Attributes <equals> Parent_Attributes yields FALSE.

Child_Attributes <not equals> Parent_Attributes yields TRUE.

<Card> Child_Attributes yields 5.

The <Card> operator returns the cardinality of a set.
```

5.1.2 Lists

A list is an ordered collection of elements. There may be duplicated elements in a list. Lists are polymophic. They can contain elements of any type including list type. Therefore, we can have a list of lists. Lists are defined in the following form:

```
Type_list = list[ Int, Float, Bool ]
```

Allowed operations on lists are:

```
The <head> operator returns the first element of the list;
for example:

   <head> Type_list yields Int

The <tail> operator returns the list with the first element
removed; for example:
```

```
<tail> Type_list yields list[ Float, Bool ]
```

```
The <len> operator returns the length of the list; for example:
```

```
<len>  Type_list yields 3
```

The semantic analyzer represents the type of a variable in a Dec program as a list; for example:

```
Let Type(t) denotes the type list of the type t.
```

If,

```
Var Payroll : Table ( String, PayInfo )
```

Then,

```
Type(Table) = list[ Table, Type(String), Type(PayInfo) ]
```

In general,

```
Type(t)= list[type 1, ..., type n]
```

where,

type 1 is the class name of the type and type 2 to n are the type arguments. Note that the type arguments can also have arguments. Each type argument is also a type list.

Another example of a list is the signature of an attribute; for example:

```
Let Signature(a) denotes the signature of the attribute a.
```

If,

```
method convert( operand : Int ) : Float
```

Then,

```
Signature(convert) = list[ Type(Int), Type(Float) ]
```

In general,

```
Signature(a)= list[ type 1, ..., type n ]
```

where:

In the case of a method:

type 1 to n-1 are the types of each corresponding parameter. Type n is the return type if exists.

In the case of a variable:

There is only one type in the list.

Each type in the list is itself a list as defined above (Type(t)).

5.1.3 Maps

A map maps elements of one set (called the domain) to elements of another set(called the range). Maps can be explicitly declared in the following form:

```
Variable_map_of_c = map[ Index -> Int, Response -> Bool,
                         Salary -> Float, Vacation -> Int ]
```

```
The <Dom> operator returns the domain of the map; for example:
```

```
   <Dom> Variable_map_of_c yields { Index,
                                    Response, Salary, Vacation }
```

```
The <Rng> operator returns the Range of the map; for example:
```

```
   <Rng> Variable_map_of_c yields { Int, Bool, Float }
```

A mapping may be applied to an element of its domain to yield the corresponding element from the range. This is similar to function application. In the above example, Variable_map_of_c(Index) yields Int whereas Variable_map_of_c(Salary) yields Float.

Maps are used in the semantic analyzer to map attributes to their signatures. This is similar to a table of signatures with the attribute name as the key. For example:

```
Let Attribute_map_c = map[ a1 -> Signature(a1), ...,
                           an -> signature(an) ]
```

Therefore, we can retrieve the signature of an attribute by:

```
Attribute_map_c(a1) yields Signature(a1).
```

5.2 The algorithms

```
Let:
    current class be c.

    Inherit_set(c)  = { x : class c inherits class x }

    Extends_set(c)  = { x : class c extends class x }

    Var_set(c)      = { v : v is an instance variable of class c }

    Method_set(c)   = { m : m is a method of class c }

    Parent_set(c)   = { A set of all parents of class c }
            .
                    = Inherit_set(c) <union> Extends_set(c)

    Attribute_set(c)= { A set of all attributes of class c }

                    = Var_set(c) <union> Method_set(c)
```

```
Child_map  = [ map of all attributes declared in c to
               their signatures ]
           = [ a0 -> Signature(a0), ..., an -> Signature(an) ]

Total_Inherit_set(c)= { x : x is an attribute of at least one
                        of the parents of c and is not declared
                        in the class c }

Total_Attri_set(c)= { A set of all attributes of the class c
                      after the effect of inheritance }

Ancestor_set(c) = { A set of all ancestors of c in all levels of
                    the inheritance tree. }
                  /*
                     For example, C inherits from B and B inherits
                     from A.  The Ancestor_set(C) is { B, A }.
                   */
```

The distinction between a set and a list is important. All attributes of a class is collected in a set since an attribute cannot be declared more than once. The parent classes are collected in a set since the same parent class name cannot appear more than once in the inherits and extends section of the source file . The implementation must detect duplication and produce appropriate error messages.

Many of the algorithms in this section check a relationship that must exist between corresponding components of two lists. Clearly, the relationship cannot exist if the lists do not have the same length. Accordingly, the algorithms check that the two lists have the same length.

5.2.1 The type conformance algorithm

```
/*
   Input  : Type lists Type_list0 and Type_list1.
   Output : TRUE if Type_list0 conforms to Type_list1.
            FALSE otherwise.
   For Type_list0 to conform to Type_list1, the type of
   Type_list0 and its type arguments have to conform to
   that of Type_list1.
 */

Function Conform( Type_list0, Type_list1 ) : Bool
Begin

   /* Type_list0 and Type_list1 are the type list of two types */

   if ( <len> Type_list0 <> <len> Type_list1 ) then
      return ( FALSE )

   if ( <head> Type_list1
      <in> Ancestor_set(<head> Type_list0) ) then {

      Type_list0 = <tail> Type_list0
      Type_list1 = <tail> Type_list1
      /* Now <len> Type_list1 should be one less. */
      loop <len> Type_list1 times {
         if ( Conform( <head> Type_list0,
                       <head> Type_list1 ) == FALSE ) then
            return ( FALSE )
         Type_list0 = <tail> Type_list0
         Type_list1 = <tail> Type_list1
```

```
      }
   }
   else
      return ( FALSE )

return ( TRUE )
End Conform
```

5.2.2 The type equality algorithm

```
/*
   Input  : Type lists Type_list0 and Type_list1.
   Output : TRUE if Type_list0 equals to Type_list1.
            FALSE otherwise.
   For Type_list0 to equal to Type_list1, the type of Type_list0
   and its type arguments have to equal to that of Type_list1.
 */

Function EqualType( Type_list0, Type_list1 ) : Bool
Begin

   /* Type_list0 and Type_list1 are the type list of two types */

   if ( <len> Type_list0 <> <len> Type_list1 ) then
      return ( FALSE )

   if ( <head> Type_list0 == <head> Type_list1 ) then {
      Type_list0 = <tail> Type_list0
      Type_list1 = <tail> Type_list1
      /* Now <len> Type_list1 should be one less. */
```

```
      loop <len> Type_list1 times {
         if ( EqualType( <head> Type_list0,
                         <head> Type_list1 ) == FALSE ) then
            return ( FALSE )
         Type_list0 = <tail> Type_list0
         Type_list1 = <tail> Type_list1
      }
   }
   else
      return ( FALSE )

return ( TRUE )
End EqualType
```

5.2.3 The signature conformance algorithm

```
/*
   Input    : Child signature list SigChild and parent
              signature list SigParent.
   Output   : TRUE if SigChild conforms to SigParent.
   SigChild conforms to SigParent only if all argument types
   and the return type of SigChild conforms to that of SigParent.
 */
       .
Function SignatureConform( SigChild, SigParent ) : Bool
Begin
   /* SigChild and SigParent are lists representing two
    * signatures */
```

```
   if ( <len> SigChild <> <len> SigParent ) then
      return ( FALSE )
   else {
      loop <len> SigChild times {
         if ( Conform( <head> SigChild,
                       <head> SigParent )== FALSE ) then
            return ( FALSE )
         else {
            SigChild = <tail> SigChild
            SigParent = <tail> SigParent
         }
      }
   }
return ( TRUE )
End SignatureConform
```

5.2.4 The signature equality algorithm

```
/*
   Input   : Child signature list SigChild and parent
             signature list SigParent.
   Output  : TRUE if SigChild conforms to SigParent.
   SigChild equals to SigParent only if all argument types and
   the return type of SigChild equals to that of SigParent.
 */

Function EqualSignature( SigChild, SigParent ) : Bool
Begin
   /* SigChild and SigParent are lists representing
    * two signatures */
```

```
   if ( <len> SigChild <> <len> SigParent ) then
      return ( FALSE )
   else {
      loop <len> SigChild times {
         if ( EqualType( <head> SigChild,
                         <head> SigParent ) == FALSE ) then
            return ( FALSE )
         else {
            SigChild = <tail> SigChild
            SigParent = <tail> SigParent
         }
      }
   }
return ( TRUE )
End EqualSignature
```

5.2.5 The inheritance algorithm

```
/*
   Input   : current/child class c
   Output  : Total_Attri_set(c)
 */

Procedure Inheritance
Begin

/*
   Before inheritance, there are only attributes declared in
   the current class; c.
```

```
 */

Total_Attri_set(c) = Attribute_set(c)

/* Start of loop handling inheritance and redeclarations. */

for each p <in> Parent_set(c) {

   Let Redeclare_set(c) =
                  Attribute_set(c) <intersect> Attribute_set(p)
   Let Inherit_set(c) = Attribute_set(p) - Redeclar_set(c)
   Let Parent_map = [ map of all attributes of p to their signatures ]

   /* Handles inheritance */

   for each a <in> (Inherit_set(c)) {
      if ( a in Total_Inherit_set(c) ) then {

         /* The attribute is inherited from multiple parents */
         sc = Total_Inherit_Set( a )
              /* Signature(a) in a previously processed
               * parent class. */

         sp = Parent_map( a ) /* Signature(a) in the parent class */
         if ( EqualSignature ( sc, sp ) == FALSE ) {
            ERROR : signature a in c is not the same as
                    a in parent p.  A From clause is required.
         }
      }
      else {
```

```
            /* Inherits the attribute by copying the signature. */
            Total_Attri_set(c) <union> { a }
         }
      }

      Total_Inherit_set(c) = Total_Inherit_set(c) <union> Inherit_set(c)

      /* Handles redeclaration */

      for each a <in> Redeclare_set(c) {
         sc = Child_map( a )      /* Signature(a) in the child class */
         sp = Parent_map( a )     /* Signature(a) in the parent class */
         if ( SignatureConform( sc, sp ) == FALSE ) {
            ERROR : signature a in c does not conform to
                    a in parent p.
         }
         if ( a <in> Method_set(c) and it has a from clause ) {
            if ( the class specified in the from clause == p ) {
               The method body of a in p has to be concrete.
            }
         }
      } /* for a */
   } /* for p */

   End Inheritance
```

5.3 The program design

5.3.1 The Abstract Syntax Tree

The central data structure of the Dee compiler is the abstract syntax tree. Although the abstract syntax tree is built by the parser, it is not a parse tree. It is abstract in the sense that most terminal symbols are discarded and only the essential structure of the source program is retained. An abstract syntax tree of a Dee program is shared by all the three processing modules; the parser, the semantic analyzer and, the code generator. The parser builds the abstract syntax tree based on the source program. The semantic analyzer decorates the abstract syntax tree with required information. Finally, the code generator emits codes based on the abstract syntax tree. The abstract syntax tree is a means of communicating the processing results of one module to another.

An abstract syntax tree is made up of nodes connected together to represent all elements in the source program. Abstract syntax tree nodes are implemented in C as a structure. The structure is assigned with the appropriate information depending on which Dee program construct the node represents. The definition of the abstract syntax tree node structure can be found in Appendix A.

5.3.2 Interfaces with other modules

In developing the Dee compiler, we found that the existence of a well-defined, central data structure such as the Abstract Syntax Tree greatly simplified the task of specifying interfaces between compiler components, and therefore to the concurrent implementation of these components by different memebers of the team. We have defined only one entry point to the semantic analyzer as the following:

```
void SA_CheckClassSemantics ( AST );
```

The parameter is an abstract syntax tree representing the current class. The semantic analyzer module is only called once for each compilation unit after the parser has completed its work.

When an error is encountered during semantic analysis, the semantic analyzer calls a common error handling routine to record the error. The calling convention of the error routine is as the following:

```
void Error ( StringPtr, int );
```

The first parameter is a pointer to a string containing an error message. The second parameter is the line number where the error occurs.

As described in Chapter 3 and Chapter 4, the semantic analyzer accesses class interfaces through class interface manager [1] [9]. We have identified seven class interface manager functions to be used in the semantic analyzer. The retrieval functions allocate and return an abstract syntax sub-tree representing the information requested. The semantic analyzer is responsible for freeing the sub-tree if it is no longer required.

```
int CIM_Init();
```

This function prepares the class interface manager for further processing. It is only called once at the start of the semantic analyzer module.

```
int CIM_Close();
```

This function terminates all class interface manager accesses and it should be called after all semantic processing is done and before exiting the semantic analyzer module.

```
int CIM_Write_Class( AST );
```

This function converts an abstract syntax tree to a class interface and the new interface is added to the data base. It should only be called with an abstract syntax tree representing a valid Dee program without any errors.

```
AST CIM_Read_Class( StringPtr );
```

This function retrieves the interface document for a class and converts the interface to an abstract syntax tree. The class name is passed as an argument. The function returns an abstract syntax tree if the class is found. If not, it returns a NULL pointer.

```
int CIM_Get_Class_Params( StringPtr, AST * );
```

This function retrieves the interface of a class and returns the class parameter list in the form of an abstract syntax sub-tree. The first parameter is the class name. The second parameter is an address of a pointer variable which is used to contain the address of the abstract syntax sub-tree returned by the function. This function returns 0 if the class is found and returns 1 if the class is not found.

```
int CIM_Get_Ancestor_List( StringPtr, AST * );
```

This function retrieves the interface of a class and returns the ancestor class list in the form of an abstract syntax sub-tree. The first parameter is the class name. The second parameter is an address of a pointer variable which is used to contain the address of the abstract syntax sub-tree returned by the function. This function returns zero if the class is found and returns one if the class is not found.

```
int CIM_Get_Attribute_Item( StringPtr, char *, AST * );
```

This function retrieves the interface of a class and returns the attribute signature in the form of an abstract syntax sub-tree. The first parameter is the class name. The second parameter is the name of the attribute. The third parameter is an address of a pointer variable which is used to contain the address of the abstract syntax sub-tree returned by the function. This function returns zero if the class is found and returns one if the class is not found.

5.3.3 Organization of the semantic analyzer module

The task of semantic analysis is divided into a series of steps described in the flow chart in Figure 5.1.

The design of the semantic analyzer is modular enough that the entire process of semantic analysis is clearly depicted by the main routine of the semantic analyzer(SA_CheckClassSemantics). The routine is shown in Appendix B.

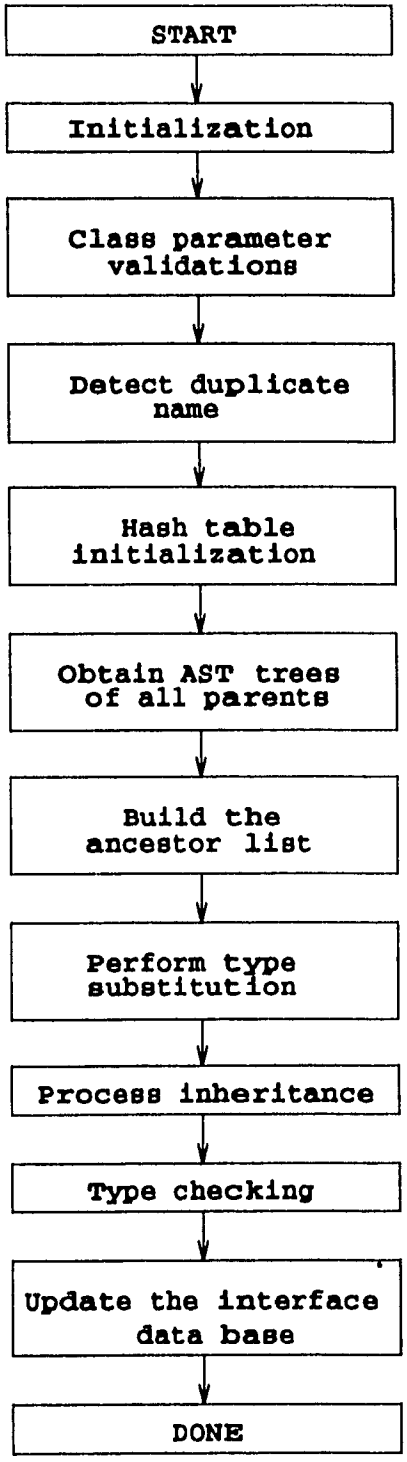

START
Initialization
Class parameter validations
Detect duplicate name
Hash table initialization
Obtain AST trees of all parents
Build the ancestor list
Perform type substitution
Process inheritance
Type checking
Update the interface data base
DONE

Figure 5.1:

Chapter 6

Conclusion

6.1 Full implementation

Although the exercise in designing and implementing an Object-Oriented language is a valuable experience by itself, the success of the Dee research project is judged by the quality of its end products; the Dee language, the compiler and its development environment. Dee is a working system available for students to conduct further research. It can also be made available for production-quality software development. It is crucial for the Dee compiler to work correctly; ensuring that only valid Dee programs are compiled into executable form. The semantic analyzer plays an important role in verifying source program correctness. Not only do we need to establish a set of consistent and easy to implement semantic rules, the semantic analyzer also has to enforce all the semantic rules described in Chapter 4.

Although the functional specification described in Chapter 4 is informal, we spent much effort in exhaustively enumerating all possible situations. This set of concise and relatively simple semantic rules helps to ensure a full implementation.

I left the project before the Dee compiler was completed. After my departure, several errors were found in the code for the semantic analyzer. Most of these errors were minor, concerning memory management, AST traversal, and error reporting. There were two major errors.

1. The "ancestor list" constructed by the semantic analyzer originally contained only the classes mentioned in the "inherits" and "extends" declarations of the source program. This list was later extended to include all of the ancestors of the current class, as its name suggests.

2. If a class might inherit the same attribute from two parents, the user is required to include a "from" clause indicating the desired parent. The type checker must nevertheless check the signatures of the attribute in both parents, not just the parent from which the attribute is inherited.

In retrospect, it is clear that both of these errors are actually design errors. The Dee compiler is a complex artefact, and it was not possible for us to realize all of the implications of the design at the beginning of the project. We learned two lessons from errors of this kind. First, they were relatively easy to correct, which illustrates the importance of the simple and robust architecture that we chose for the compiler. Second, these errors would have been discovered much earlier if we have used formal techniques to specify the compiler components.

6.2 Maintainability

We view the Dee project as an on-going research project as well as a production-quality software product which has a life span far beyond the initial development period. Maintainability is the key to extended software life span. The program design of the semantic analyzer has to be well disciplined. This is particularly true for the Dee semantic analyzer where there is no standardized or automated software tools to help its development.

For a program module to be maintainable, it has to be readable. We define readability as the overall ease of understanding the program structure as well as the algorithms involved rather than the superficial coding style. However, a proper coding style is also necessary to ensure readability. In the design of the Dee semantic

analyzer module, we strive to derive simple and concise algorithms that are consistent with the overall approach of the entire module. The module as a whole represents a systematic approach to semantic checking rather than a mixture of odd semantic checking routines. The major algorithms as presented in Chapter 5 are conceptualized as numerous loops iterating through elements of various lists or sets. Simple concepts like these allow us to perform the best implementation possible. They can also be easily understood by those who will be maintaining the semantic analyzer module.

Another attribute of maintainability is modularity. The semantic analyzer module is designed to interact with the other components of the compiler in a well defined fashion. Module independence is achieved by loose coupling with the rest of the compiler as well as information hiding. Module independence allows modifications to the module with minimum effects on other modules. The only common link to the three compiler modules; the parser, the semantic analyzer and, the code generation, is the abstract syntax tree. The abstract syntax tree structure is well defined and the access to the tree by each module is also well defined. The semantic analyzer does not need any other information from the parser other than the abstract syntax tree. This is demonstrated by the only entry point to the semantic analyzer where the only function parameter is the abstract syntax tree of the current class.

The interactions with the class interface manager are also well defined. In fact, they were defined well before the implementation stage. The early definition allowed development efforts for the semantic analyzer and the class interface manager to proceed in parallel. The seven functions identified in Section 5.3.2 represent the only links in the semantic analyzer to the class interface manager. The entire process of class data base management and class information retrieval are completely encapsulated.

After the original developers of the Dee project have left the project, there were various enhancement made to the semantic analyzer. The semantic analyzer is readable enough that the new developers can incorporate the new features with minimal learning effort.

We believe that we have established a set of concise and consistent semantic rules

for the Dee language. We have also been successful in implementing the rules in the Dee semantic analyzer in a modular fashion and within a relatively short time frame. We are confident that given the current design and code quality, we have provided a solid foundation for future works in the Dee project.

Bibliography

[1] B. Cheung. A semantic browser for dee. Master's thesis, Department of Computer Science, Concordia University, April 1992.

[2] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[3] Peter Grogono. The dee report. Technical Report OOP-91-2, Department of Computer Science, Concordia University, January 1991.

[4] Peter Grogono. Issues in the design of an object oriented programming language. *Structured Programming*, 12(1):1-15, January 1991.

[5] L. Hegarty. Implementing the dee system: Issues and experiences. Master's thesis, Department of Computer Science, Concordia University, April 1992.

[6] B. Meyer. *Object-oriented Software Construction.* Prentice-Hall, 1988.

[7] Bjarne Stroustrup. *The C++ Program ning Language.* Addison-Wesley, 1986.

[8] Peter Wegner. Dimensions of object-based language design. *OOPSLA '87 Proceedings, ACM SIGPLAN Notices*, 22(12):168–182, December 1987.

[9] J. Yau. The design and implementation of the class interface manager of unix dee. Master's thesis, Department of Computer Science, Concordia University, September 1992.

Appendix A

The Abstract Syntax Tree Structure

```
/*
 * FILE : deedefs.h
 * Dee AST struct definitions.
 */

#ifndef _DEEDEFS_
#define _DEEDEFS_

#define         NIL             NULL

typedef enum { FALSE, TRUE } Boolean;
typedef int HashIndex;
typedef char *StringPtr;

/* Loop types */

typedef enum { Infin_l, While_l, Until_l } LoopType;
```

```
/* Literal Number types */

typedef enum { Int, Float, Byte } NumberType;

/* Mode for local variable */

typedef enum { Param, Result, LocalVar } LocalModeType;

typedef enum { IGNOREATTR, FROMSELF,
               FROMPARENT, FROMCLAUSE } AttriSrc;

typedef enum { MethodM, ConsM } MethodType;

/* A method body is either a ''from'', abstract,
 * concrete or an Instr */

typedef enum { BodyUnknown, BodyFrom, BodyAbs, BodyConcrete,
               BodySpecial } BodyType;

/* An ident node can be one of local, inst var, handler
 * local or method */

typedef enum { IdenLocal, IdenInstVar, IdenMethod,
               IdenCons, IdenHandlerLocal } IdenType;

/* Abstract syntax tree definitions. */

typedef enum {
    List, Class, Type, Var, Method,
    Local, Assign, If, IfPair, DoLoop, Loop,
```

```
    Apply, Iden, Break, Continue, Nil,
    Attempt, Handler, Signal, Bool, Number,
    String, Null, Undef, Signature, SymTab, CTemp
    } ASTNodeType;


typedef struct ASTNode *AST;

struct ASTNode {
   ASTNodeType NType;
   int Column, Line;

   union {

      /* Lists of nodes are reprsented with List nodes. The
       * empty list is represented by NIL. */

      struct {
         AST Node;
         AST Next;
      } List;

      /* A Class node describes an entire class. */

      struct {
         HashIndex ClassName;      /* Name of the class */
         StringPtr ClassComment;   /* Comment following class
                                    * header */
         AST ClassParamList;       /* List of Signature nodes */
```

```
    /* List of the actual classes corrspnding to formal
     * class params */

    AST InheritList;          /* List of Class for
                               * inherited classes */
    AST ExtendList;           /* List of Class for
                               * extended classes */
    AST InvarList;
    AST AttributeList;
    AST Ancestors;            /* list of all ancestors
                               * of this class */
    AST Uses;                 /* The classes of all
                               * variables used in */
                              /* stmts of all methods */
    Boolean ClassHasSpecial;  /* true if the class has any
                               * special methods */
} Class;

/* A type has a name and a list of arguments, which
 * are themselves types. E.g. Array[Table[Int String]]. */

struct {
    HashIndex TypeName;       /* Type name */
    AST TypeArgList;          /* List of Type containing
                               * arguments */
} Type;

struct {
    HashIndex SigId;
    AST SigType;
```

```
    AST SigOriginalType;    /* Never altered by type
                             * substitution */
    int StackOffset;
} Signature;

/* Instance variable descriptor. */

struct {
   StringPtr VarComment;    /* Comment following variable */
   AST VarType;             /* Type node giving the type of
                             * the variable */
   Boolean VarPublic;       /* True if this is a
                             * public variable */
   AttriSrc AttributeSource;
   AST SourceClass;
} Var;

/* Method descriptor. */

struct {
   Boolean MethPublic;      /* True if this is a public method*/
   MethodType MethKind;     /* One of method or cons */
   HashIndex MethName;      /* Method name */
   StringPtr MethComment;   /* Comment following header */
   AST Result;
   AST MethOriginalResult; /* never altered by SA */
   AST MethLocalList;       /* List of Local local
                             * var descriptors */
   AST MethParamList;       /* This is a pointer into the
                             * MethLocalList where the
```

```
                                * parameters start
                                * (not sep. list)*/
   AST Require;                /* Require part of a method  */
   AST Ensure;                 /* Ensure part of a method   */
   AST Body;                   /* List of statement nodes   */
   BodyType MethBodyType;      /* What kind of body does this
                                * method have */
   AST DefinedBy;              /* Set by the SA */
   AST ImplementedBy;          /* Set by the SA */
   int LocalCount;             /* Number of local variables */
   int ParamCount;             /* Number of parameters      */
   AttriSrc AttributeSource;
   BodyType FromBodyType;      /* The true body type of a
                                * from body */
} Method;

/* Local variable descriptor. Local variables include
 * parameters, result, self, and declared local
 * variables. */

struct {
   HashIndex LocName;          /* Local variable name */
   AST LocType;                /* Type node giving type
                                * of variable */
} Local;

struct {
   HashIndex Id;
   IdenType IdenKind;
   int LocDisp;                /* Stack displacement if
```

```
                        * a local*/
   AST IdenType;        /* type of this id filled
                         * in by the AST */
 } Iden;

/* The next group of nodes represent statements. */

/* Assignment statement: LHS := RHS. LHS is always a
 * local variable or self instance var */

struct {
   AST AssignVar;       /* LHS Identifier node    */
   AST AssignExpr;      /* RHS expression subtree */
} Assign;

/* If statement */
struct {
   AST IfPairList;      /* List of IfPair nodes */
   AST IfElse;          /* List of statements in
                         * the else part */
} If;

/* A pair consisting of an expression E and a list
 * of statements S, corresponding to "if E then S"
 * or "elsif E then S". */
struct {
   AST PairExpr;        /* Bool expression    */
   AST PairStmts;       /* List of statements */
} IfPair;
```

```
/* A controlled loop: "from S until E while E do S od". */
struct {
   AST FromStmts;          /* List of initialization
                            * statements */
   AST UntilCond;          /* Bool expression */
   AST WhileCond;          /* Boolean expression */
   AST LoopStmts;          /* List of loop statements */
} Loop;

/* Attempt statement: attempt S handlers end */
struct {
   AST AttStmtList;        /* List of statements to
                            * be attempted */
   AST AttHandlerList;     /* List of Handler
                            *  exception handlers */
} Attempt;

/* An exception handler:  var:type statements. */
struct {
   AST HandlerVar;         /* Local node for handler
                            * variable */
   AST HandlerStmtList;    /* List of statements for
                            * handler  */
} Handler;

/* Signal statement */
struct {
  AST SignalExpr;          /* Expression node for
                            * exception object */
} Signal;
```

```
/* An application node can occur either as a
 * statement or an expr. */
struct {
   AST Receiver;           /* Either an Apply node or
                            * an Iden node  */
   HashIndex AttrName;     /* Name of the method in
                            * the application */
   IdenType AttrKind;      /* can only be InstVar,
                            * Method or Cons   */
   AST AttrType;           /* static class of the
                            * attribute        */
   AST ApplyList;          /* List of expressions:
                            * the arguments    */
} Apply;

/* The following nodes represent expressions. */

/* The expression "undefined Expr". */
struct {
   AST UndefExpr;         /* Expression node */
} Undef;

/* A boolean literal: either TRUE or FALSE. */
struct {
   Boolean BoolVal;
} Bool;

/* A numeric literal which may be an Int or a Float. */
struct {
```

```
            NumberType NumKind;     /* an int or a float*/
            int IntVal;             /* if int, here's the
                                     * real value  */
            double DoubleVal;       /* if float */
            unsigned char ByteVal;
            StringPtr NumVal;       /* String representation
                                     * of value */
         } Num;

         /* A string literal */
         struct {
            StringPtr StrVal;
         } String;

      } Tag;

};  /* ASTNode */
```

Appendix B

Listings of the key routines in the semantic analyzer

```
void    SA_CheckClassSemantics ( class )
AST class;
{
   extern Boolean SA_ForceCIMWrite;

   if ((SA_HashTable =
      (AST *) calloc(1, sizeof (AST) * HASHMAX)) == NULL)
   {
      SA_FatalError(
         "Cannot allocate hash table for semantic analysis");
   }
   else
   {
      CIM_Init ();
      current_class = class;
      SA_InitHashCodes ();
      SA_ValidateClassParamType ( class );
```

```
        SA_DetectDuplicates ( class );
        SA_InitHashTable ( class );
        SA_LoadParentAST ( class, INHERITED ).
        SA_LoadParentAST ( class, EXTENDED );
        SA_BuildAncestorList ( class, INHERITED );
        SA_BuildAncestorList ( class, EXTENDED );
        SA_AddParentAncestors( class );
        SA_TypeSubstitution ( class );
        SA_InheritAttributes ( class, INHERITED );
        SA_InheritAttributes ( class, EXTENDED );
        SA_WrapupInheritance ( class );
        SA_BuildSelfType( class );
        SA_InitUndefinedSig();
        SA_CheckMethodBodies ();
        if ( IsError () == FALSE
            || SA_ForceCIMWrite == TRUE )
        {
            CIM_Write_Class ( class );
            CIM_Close ();
        }
        free ( SA_HashTable );
    }
}

/*
 * Determines if type0 conforms to type1.  Accepts NULL as
 * argument.  If both type0 and type1 are NULL returns TRUE.
 * If only one of them is NULL, returns FALSE.  It is
 * possible type0 and type1 has the same name, if all class
 * parameters conform, return TRUE.  type0 is the child.
```

```
 * type1 is the parent.
 */

static Boolean  SA_Conform ( type0, type1 )
AST type0; AST type1;
{
   register AST type0_arg_list;
   register AST type0_arg;
   register AST type1_arg_list;
   register AST type1_arg;
   Boolean status = FALSE;

   if ( type0 == NilType )
     return TRUE;
   else if ( type1 != NULL && SA_IsAncestor( type0, type1 ) )
   {
      type0_arg_list = type0->Tag.Type.TypeArgList;
      type1_arg_list = type1->Tag.Type.TypeArgList;
      while ( type0_arg_list && type1_arg_list ) {
         type0_arg = type0_arg_list->Tag.List.Node;
         type1_arg = type1_arg_list->Tag.List.Node;
         if ( ! SA_Conform ( type0_arg, type1_arg ) )
            return FALSE;
         type0_arg_list = type0_arg_list->Tag.List.Next;
         type1_arg_list = type1_arg_list->Tag.List.Next;
      } /* while */
      if ( ! type0_arg_list && type1_arg_list )
         return FALSE;
      else return TRUE;
   }
```

```
   else
      return FALSE;
}

/*
 * Given two type ast sub-tree containing their arguments.
 * This function recursively checks to see if they denote
 * the same type.
 */

static Boolean  SA_EqualType ( type0, type1 )
AST type0; AST type1;
{
   register AST type0_arg_list;
   register AST type0_arg;
   register AST type1_arg_list;
   register AST type1_arg;
   Boolean status = FALSE;

   if ( type0 == NULL && type1 == NULL ) {
      status = TRUE;
   }
   else {
      if ( type0 != NULL
        && type1 != NULL
        && type0->Tag.Type.TypeName == type1->Tag.Type.TypeName ) {

         status = TRUE;
         type0_arg_list = type0->Tag.Type.TypeArgList;
         type1_arg_list = type1->Tag.Type.TypeArgList;
```

```
            while ( type0_arg_list && type0_arg_list ) {
               type0_arg = type0_arg_list->Tag.List.Node;
               type1_arg = type1_arg_list->Tag.List.Node;
               if ( (status =
                     SA_EqualType ( type0_arg, type1_arg )) == FALSE ) {
                  break;
               }
               type0_arg_list = type0_arg_list->Tag.List.Next;
               type1_arg_list = type1_arg_list->Tag.List.Next;
            }
            if ( type0_arg_list || type1_arg_list ) {
               status = FALSE;
            }
         }
      }
      return ( status );
}

static void SA_MethSigConform ( parent_method, child_method )
AST parent_method; AST child_method;
{
   register AST parent_params;
   register AST child_params;
   register int i;
   char * method_name
            = HashItem(parent_method->Tag.Method.MethName);
   int parent_i;
   int child_i;

   if ( SA_Conform (
```

```
      child_method->Tag.Method.Result,
      parent_method->Tag.Method.Result ) == FALSE ) {
      sprintf ( wrkstr,
         "Parent and child have incompatible result types \
in method %s.",
         method_name );
      SA_Error ( wrkstr, child_method->Line );
   }

   parent_i = parent_method->Tag.Method.ParamCount;
   child_i = child_method->Tag.Method.ParamCount;
   if ( parent_i != child_i ) {
      sprintf ( wrkstr,
         "Parent and child have different number of \
parameters in method %s.",
         method_name );
      SA_Error ( wrkstr, child_method->Line );
   }

   /*
    * Even if there is an error we still checks the params.
    * Use the smaller of the two param. count
    */

   i = ( parent_i < child_i ? parent_i : child_i );
   child_params = child_method->Tag.Method.MethParamList;
   parent_params = parent_method->Tag.Method.MethParamList;
   for ( ; i; i-- ) {
      if ( SA_Conform ( child_params->Tag.List.Node->
                        Tag.Signature.SigType,
```

```
                        parent_params-> Tag.List.Node->
                        Tag.Signature.SigType) == FALSE ) {
            sprintf ( wrkstr,
             "Incompatible parameter type redeclaration; \
parameter %s, method %s.",
              HashItem (child_params->
                        Tag.List.Node->Tag.Signature.SigId),
              method_name );
            SA_Error ( wrkstr, child_method->Line );
         }
         child_params = child_params->Tag.List.Next;
         parent_params = parent_params->Tag.List.Next;
      }
}

/*
 * This function is similar to SA_MethSigConform except that
 * it checks * if the signatures are the same.
 */

static void SA_MethSigEqual ( parent, method0, method1 )
AST parent; AST method0; AST method1;
{
   register AST method0_params;
   register AST method1_params;
   register int i;
   char * method_name = HashItem (method0->Tag.Method.MethName);
   char * parent_name = HashItem (parent->Tag.Class.ClassName);
   int method0_i;
   int method1_i;
```

```
   if ( SA_EqualType ( method1->Tag.Method.Result,
                       method0->Tag.Method.Result ) == FALSE ) {

      sprintf ( wrkstr,
         "Multi-parent method %s in parent %s has \
incompatible result type.", method_name, parent_name );
      SA_Error ( wrkstr, 0 );
   }

   method0_i = method0->Tag.Method.ParamCount;
   method1_i = method1->Tag.Method.ParamCount;
   if ( method0_i != method1_i ) {
      sprintf ( wrkstr,
         "Multi-parent method %s in parent %s has \
incompatible number of parameters.", method_name, parent_name );
      SA_Error ( wrkstr, 0 );
   }

   /*
    * Even if there is an error we still checks the params.
    * Use the smaller of the two param. count
    */

   i = ( method0_i < method1_i ? method0_i : method1_i );
   method1_params = method1->Tag.Method.MethParamList;
   method0_params = method0->Tag.Method.MethParamList;
   for ( ; i; i-- ) {
      if ( SA_EqualType ( method1_params->Tag.List.Node->
                          Tag.Signature.SigType,
```

```
                        method0_params->Tag.List.Node->
                        Tag.Signature.SigType) == FALSE ) {
            sprintf ( wrkstr,
             "Parameter %s of multi-parent method %s in \
parent %s has incompatible type.",
              HashItem (method1_params->
                        Tag.List.Node->Tag.Signature.SigId),
              method_name, parent_name );
            SA_Error ( wrkstr, 0 );
         }
         method1_params = method1_params->Tag.List.Next;
         method0_params = method0_params->Tag.List.Next;
      }
}

static void SA_InheritAttributes ( class, inherit_type )
AST class; InheritType inherit_type;
{
   register AST parents_list;

   if ( inherit_type == INHERITED )
      parents_list = class->Tag.Class.InheritList;
   else
      parents_list = class->Tag.Class.ExtendList;

   /* Inherit attributes from each parent specified. */

   while ( parents_list ) {
      if ( parents_list->Tag.List.Node->NType == Class )
         SA_InheritFromOneParent
```

```
            ( parents_list->Tag.List.Node, inherit_type );
      parents_list = parents_list->Tag.List.Next;
   } /* end while */
}

static void SA_InheritFromOneParent ( parent, inherit_type )
AST parent; InheritType inherit_type;
{
   register AST attribute_list =
                        parent->Tag.Class.AttributeList;
   register AST attribute;

   while ( attribute_list ) {
      attribute = attribute_list->Tag.List.Node;
      attribute_list = attribute_list->Tag.List.Next;

      switch ( attribute->NType ) {

      case Var : {
         HashIndex var_name =
            attribute->Tag.Var.VarType->Tag.Signature.SigId;

         if ( SA_HashTable [var_name] == NULL ) {
            SA_InheritOneVar ( parent,
                              attribute, inherit_type );
         }
         else {
            switch ( SA_HashTable[var_name]->
                     Tag.Var.AttributeSource ) {
            case FROMSELF :
```

```
            SA_RedeclareVar ( attribute, inherit_type );
            break;

         case FROMPARENT :
            SA_VarWithMultiParents ( attribute );
            break;
         }
      }
      break;
}

case Method : {
      HashIndex method_name =
                     attribute->Tag.Method.MethName;

      attribute->Tag.Method.FromBodyType = BodyUnknown;
      if ( SA_HashTable [method_name] == NULL ) {
         SA_InheritOneMethod ( attribute, inherit_type );
      }
      else {
         switch ( SA_HashTable[method_name]->
                     Tag.Method.AttributeSource ) {
         case FROMSELF :
            SA_RedeclareMethod ( parent,
                              attribute, inherit_type );
            break;

         case FROMPARENT :
            SA_CheckMethInherit ( parent,
                                  attribute, inherit_type );
```

```
                break;

            case FROMCLAUSE :
                /*
                 * Don't mind having another parent with the
                 * same method.  The from clause has already
                 * been resolved. However, the signatures have
                 * to conform.
                 */
                SA_MethSigConform ( attribute,
                                    SA_HashTable[method_name]);
                break;
            }
        }
        break;
    }

    default :
        SA_FatalError ("Non attribute in attribute list.");
        break;

    } /* end switch */
  } /* end while */
}
```