Implementing Semantics of Object Oriented Languages Using Attribute Grammars

KAREN A. LEMONE
Dept of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609

JEFFREY J. MCCONNELL
Dept of Computer Science
Canisius College
Buffalo, NY 14208

MARY ANN O'CONNOR
Digital Equipment Corporation
Nashua, NH 03062

JOE WISNEWSKI
Compass, Inc
Wakefield, MA 01880

Abstract

Object-oriented solutions are becoming an established paradigm for writing robust, reusable software. Many object-oriented languages have begun to appear. This paper examines how the special object-oriented concepts might be implemented in a compiler using the also well-established tool of attribute grammars.

KEYWORDS: Attribute grammar, object-oriented, inheritance, polymorphism, late binding

1 Introduction

Attribute Grammars, invented by Knuth in the 60's (Knu68, Knu71), are used for semantic analysis by compiler writers in many of today's compilers and compiler tools (Far82, Lor84, Aho86, Fis88). Static type checking, storage allocation, expressions for array offsets, even symbol table creation and dataflow analysis (Bab78, Boc78) are described using attributes and attribute grammars.

Object-oriented programming languages present special problems to the compiler designer. Encapsulation, information hiding, message passing and methods, inheritance, overloading and polymorphism, late binding, are concepts associated with object oriented programming (Wil87). The syntax and semantics for expressing these concepts is an issue for compiler designers, as well.

In this paper, an attribute grammar is presented describing some object-oriented programming language concepts. Every effort has been made to keep the syntax and semantics simple and yet broad enough to be applied to actual object-oriented languages. A pseudocode example (a Banking System) and its BNF are used for illustration.

Section 2 reviews object-oriented concepts in order to show the philosophy and simplifications made to arrive at the pseudocode; Section 3 contains the BNF and the bank example; Section 4 reviews attribute grammars and their use in creating language processors; Section 5 shows the attributes and semantic functions for implementing the semantics of object-oriented programming concepts. Section 6 draws conclusions concerning the applicability of this to object-oriented languages.

2 Review of Object-Oriented Concepts

Object-oriented concepts evolved from encapsulation and information hiding of abstract data types. To encapsulation and information hiding are added messages, methods, inheritance, overloading, polymorphism, and late binding (Cox86, Sch87).

In this paper, we adopt the definitions of object-oriented as defined in [Weg87]. Thus, an object-oriented language supports objects which, themselves, belong to classes. The classes are hierarchically connected by an inheritance mechanism.

In addition, we consider polymorphism and late-binding.

2.1 Objects, Classes and Data Abstraction

Objects are dynamic data on which dynamic actions are performed, and which have a state. A class is a static grouping to which similar objects belong and under whose auspices new objects may be created. The data is described by variables and behavior is characterized by permitted operations.

Data abstraction hides the internal representation of the data and the implementation of the operations. The user specifies what operation is to be performed, not how the operation is to be performed.

2.2 Messages and Methods

Data abstraction prevents an object's data from being directly manipulated. To perform an operation on an object, a message is sent. The message contains a selector which tells the object what to do. The message may also contain parameters to aid in the operation or through which the object may return values.

A class groups actions called methods and variables called class variables. When an object of a particular class receives a message to perform an action, it finds the method(s) corresponding to the message selector and performs the action. For a banking system, a message may be sent to a Savings Account object telling it to generate an account number for a new Savings Account. *GenerateAccountNumber* is the selector in the message. The sender of the message does not know how this number is generated. The object Savings Account invokes its method, *GenerateAccountNumber* to generate the account number.

Two types of variables may be identified - class variables whose values vary for each object instance of that class and shared variables whose values are common and accessible to, and changeable by, all object instances of a class.

2.3 Subclasses, Superclasses and Inheritance

Reusability of software is enhanced via inheritance, a technique for defining new classes called subclasses in terms of existing classes. These new classes are defined by describing how they are similar and how they differ from their preexisting super classes. They may share the same memory layout and respond to messages with the same or slightly altered methods.

A subclass is said to inherit characteristics of its superclass, enabling the creation of classes which are very similar to other classes, thus saving "programmer" time. Multiple inheritance allows a class to inherit from more than one superclass. Altering a method from that of a superclass is referred to as specialization. In our model, inheritance of data and methods is stated explicitly. This does not require the programmer to write (much) more code, and is worth the tradeoff in safety.

A Bank Account class may have two subclasses, Checking Account and Savings Account. These subclasses may inherit many, but not all, of the methods of Bank Account. A class called NOWAccount may be defined and be an example of multiple inheritance if it is a subclass of both Checking Account and Savings Account and inherits from both. For NOWAccount, one of the inherited methods, GetInterestRate, may be specialized by appending a method, CheckForMinimumBalance, to the beginning.

2.4 Overloading and Polymorphism

Overloading allows an operator to operate on more than one data type. For example, virtually all languages allow "+" to operate on both integer and real type data. Polymorphism extends this capability to allow procedures to operate on different types and functions to return values of different types on different calls (Har84). For example, a procedure Sort would be polymorphic if it could be used as the selector in a message to both an integer array object and a character array object. The objects themselves would invoke methods corresponding to the selector to perform the appropriate actions. In the bank example for (simple) polymorphism, GetInterestRate is the selector in a message to both a Savings Account object and a NOWAccount object.

2.5 Late Binding

Binding is the process whereby object operations and operands are combined. Early binding combines them at compile time (static binding) or before (program writing time) and is implemented with case statements which must be changed if a new data type is to be included for the operation. Late binding combines them as late as when the program is running (dynamic binding). Messages often require dynamic binding - in our example, the message to tell an account to add interest to the balance involves dynamic binding since the type of account is not known at compile time (see Late Binding example in section 5.5)

3. Pseudocode BNF and an Example

Briefly, a program consists of a number of class descriptions followed by a number of actions.

The syntax for each class consists of the class name, the names of any super classes, the class and shared variables, methods, both inherited and those introduced in the class, and the definitions of the introduced variables and methods.

Class variables have their own values for each object instance of the class while shared variables share the value and make it accessible to each object instance.

Actions are the usual statements found in algorithmic programming languages with the addition of the message statement whose syntax is the ObjectName to receive the message, followed by a selector which tells the object what to do, followed by any parameters if necessary.

NEW creates a new instance of a class, and GET, given appropriate information, searches a data base for a previously created object.

Tokens are bolded. The metalanguage is extended BNF with {What}+ indicating 1 or more What's and [What] indicating and optional What. It is these productions to which attribute functions will be added in section 4. Rather than repeat all the words that end in Name, the notation ___Name is used. Delimiters such as semicolons are omitted. BNF for expressions and data types is also omitted for simplicity.

```
Program                  -->
  {ClassDescription}+ {Action}+

ClassDescription         -->
 Class              ClassName
 SuperClass
            SuperClassNames|None
 ClassVariables
     {Inherit     InheritClassNames
     From ClassName}+     | None
     Introduce   IntroClassNames
                     | None
 SharedVariables
    {Inherit      SharedVarNames}
     From ClassName}+     | None
     Introduce   SharedVarNames
                  | None
 Methods          Methods|None
 EndClass         ClassName

Methods                     -->
  {Inherit     InheritMethodNames
      From ClassName}+     | None
  Introduce   IntroMethodNames
                | None
  {SpecializeSpecializeMethodName
      With MethodName Before}+
  {SpecializeSpecializeMethodName
      With MethodName After}+
                | None
  Definitions        Definitions
                | None

 Definitions              -->
     {Method   MethodName
               | MethodName
            (ParameterNames)
      LocalNames
                 LocalNames | None
      BeginMethod
           {Statement}+
      EndMethod  MethodName}+

Action                 -->
   Action  ActionName
   LocalNames    LocalNames | None
   BeginAction
        {Statement;}+
   EndAction  ActionName

Statement              -->
    Assignment
   | Selection
   | WhileLoop
   | Instantiation
   | Message
   | Return (Expression)
   | Read (Names)
   | Write (Expression)

Assignment                      -->
   Name := Expression

Selection                       -->
   If  (Expression) Then
        {Statement;}+
```

```
    {ElseIf (Expression)
          Then{Statement;}+}
    [Else {Statement;}+
   EndIf

WhileLoop                         -->
     While (Condition)
         {Statement;}+
      EndWhile

Instantiation                    -->
    ClassName :=  New(ClassName)
   |ClassName :=   Get  (ClassName)

Message                           -->
     ObjectName : Selector[
                 (ParameterNames)]

Selector                         -->
    MethodName


___Names                        -->
     ____Name
     |  ____Name , ____Names
     |  ____Name : TypeName

___Name                          -->
       ____Name [:= InitialValue]
```

Figure 1 BNF For OOPL Concepts

The pseudocode program in the Appendix consists of five classes: a Utilities class providing input and output procedures, a Bank Account class with two subclasses Checking Account and Savings Account; the fifth class, NOW Account is a subclass of both Checking Account and Savings Account and is provided as an example of multiple inheritance.

Although the example is somewhat lengthy, a true banking system would be far longer. Only the class declarations are shown here. Actions that operate on object instances from these classes are shown in section 5.

Types are omitted; for example, owner, address, phonenumber etc are all (likely) of type string.

4. Review of Attribute Grammar Concepts

Attribute grammars consist of BNF describing the syntax with semantic functions describing the semantics. In the BNF for the pseudocode example described here, a program consists of class descriptions describing both the data and operations associated with objects of each class.

Attribute grammars are used in compiler environments to specify semantics in the same sense that BNF specifies syntax. Just as compiler tools allow BNF to be input to create the syntax analyzer, there are also tools which create a semantic analyzer from attribute grammar input (Far82). Many of these syntax/semantic analyzer generator tools work together :

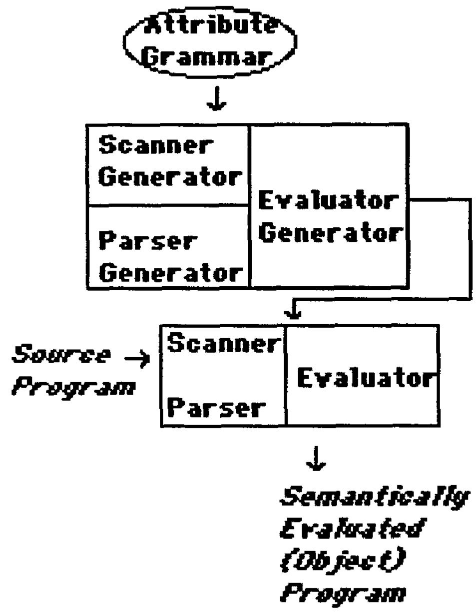


Figure 2 Syntax/Semantic Analyzer Tool

In Figure two, an attribute grammar, consisting of BNF plus token descriptions and semantic functions is input. From the token description, a lexical analyzer is built; from the BNF a syntax analyzer is built; from the attributes and semantic functions, the evaluator is built. Alternatively, for some attribute grammars for which it is too difficult at compiler generation time to generate the evaluator (Ken76, Ken77), handcoded evaluators can be written with varying ease and efficiency. Although it is possible to evaluate some attributes at parse time, this paper presumes that all attributes are evaluated after the program has been parsed.

4.1 Attributes

Attributes are variables to which values are assigned. Each attribute variable is associated with a nonterminal or terminal of the grammar. Thus, in the BNF pseudocode for our example, the variable *MethodList* (italics will be used for attributes) is associated with the nonterminal

ClassDescription (as well as with various other nonterminals). The notation is :

```
ClassDescription.MethodList
```

ClassDescription is the nonterminal representing a syntactic class and *MethodList* is its associated attribute which will be assigned a semantic value.

Values are assigned to attributes by equations called semantic functions. These equations are evaluated locally, that is, within the scope of a production. Consider, for example, the production defining the syntax of ClassDescription. To it has been added a semantic equation defining a semantic value for a variable *MethodList* to be associated with the nonterminal ClassDescription.

At compile time, the attribute *MethodList* will be evaluated at the node(s) ClassDescription :

```
ClassDescription       -->
  Class              ClassName
 SuperClass
              SuperClassNames|None
  ClassVariables
     Inherit      {InheritClassNames}
     From ClassName}+    | None
     Introduce   IntroClassNames
                   | None
  SharedVariables
    Inherit      {SharedClassNames}
     From ClassName}+    | None
     Introduce   SharedClassNames
                  | None
  Methods              Methods|None
  EndClass            ClassName


ClassDescription.MethodList :=
                 Methods.MethodList
```

The attribute, *MethodList*, is associated with both the nonterminals ClassDescription and Methods. At compile time, the value of *MethodList* is passed up the parse tree since the nonterminal Methods appears on the right-hand side of the equation and the nonterminal ClassDescription appears on the left-hand side. Such attributes are termed synthesized. Attributes whose values are passed down the parse tree are termed inherited.

Terminals may have only synthesized attributes, and their values are assigned by the lexical analyzer. Inherited attributes of the start symbol are given values via parameters when attribute evaluation begins.

5 Implementing Semantics of Object-Oriented Languages Using Attributes

In this section, we show how inheritance, multiple inheritance, message passing, polymorphism, and even late binding can be facilitated at compile time by attribute grammars. In what follows, we show separate parse trees for separate class declarations. In addition, it can be assumed that the parser can create a topological ordering in which to evaluate attributes. Thus, for our example, the attributes for BankAccount are evaluated before those for SavingsAccount and before those for CheckingAccount. The attributes for SavingsAccount and CheckingAccount are evaluated before those for NOWAccount and the attributes for all the class declarations are evaluated before those of the actions. This ordering is shown in Figure three.

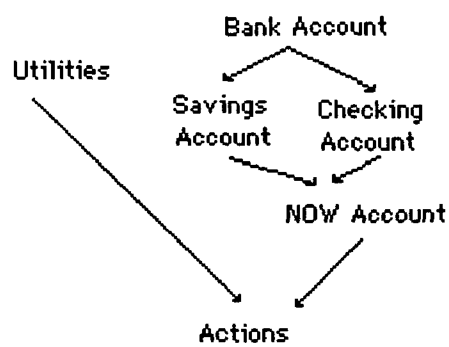
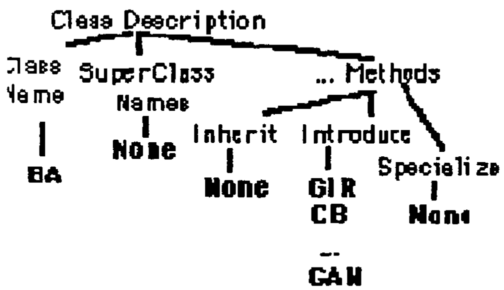


Figure 3 Attribute Evaluation Order

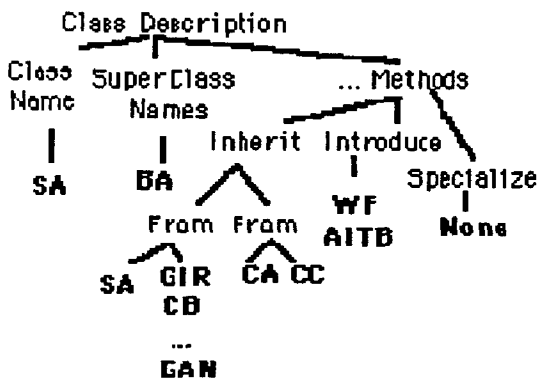
5.1 The Unevaluated Parse Trees

For the four classes of interest here, Bank Account (BA), Savings Account (SA), Checking Account (CA) and NOWAccount (NOW), the parse trees before attribute evaluation are shown in Figure four. Nodes that are not of interest here (e.g., Shared Variables) are not shown for simplicity. Method Names are abbreviated, e.g., GIR for GetInterestRate.

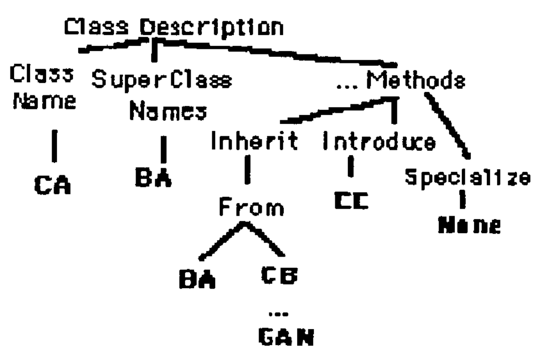
Bank Account :



Savings Account :



Checking Account :



NOW Account

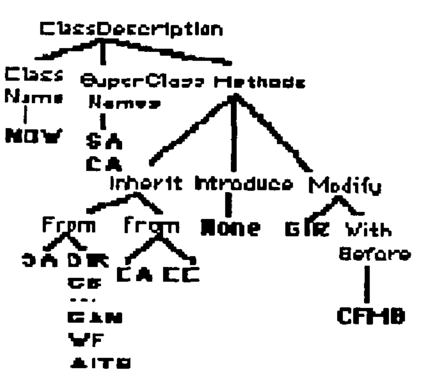


Figure 4 The Class Trees

5.2 The Attributes

There are three attributes :

(1) *ClassName*, abbreviated *CN*, an inherited attribute, passed down the class declaration tree to the Methods subtree. As the mnemonics imply, it carries the name of the current class down the tree.

(2) *SuperClassNames*, abbreviated *SN*, an inherited attribute which descends to the Methods subtree and is used to make available the names of superclasses to the class.

(3) *MethodList*, abbreviated *ML*, a synthesized attribute which, when it finally reaches the top of the tree, contains a list of methods available to the current class. Its value is a list of pairs : ClassName & MethodName for each MethodName available to the ClassName, and for each class name. They are ordered so that the methods belonging to the current class are listed first, with superclass methods listed second.

5.3 The Semantic Functions

The two productions of interest here are those for ClassDescription and Methods : The semantic functions follow each production.

```
ClassDescription         -->
  Class                  ClassName
 SuperClass
               SuperClassNames|None
  ClassVariables
      {Inherit       InheritClassNames}
      From ClassName}+      | None
      Introduce    IntroClassNames
                        | None
  SharedVariables
     {Inherit      SharedClassNames}
      From ClassName}+      | None
      Introduce    SharedClassNames
                       | None
  Methods                Methods|None
  EndClass              ClassName

 (i) Methods.CN   = LexValue
                       (ClassName)
 (ii) Methods.SN ={LexValue
                   (SuperClassNames)}
 (iii) ClassDescription.ML =
                          Methods.ML

Methods                   -->
   {Inherit InheritMethodNames
      From ClassName}+   |None
   Introduce  IntroduceMethodNames
                   | None
   {Specialize  SpecializeMethodName
```

```
      With MethodName Before}+
   {Specialize  SpecializeMethodName
      With MethodName After}+
                   | None
   Definitions    Definitions|None

(iv) Methods.ML = Methods.CN
& {LexValue(IntroduceMethodNames)}
        + Methods.CN
& {LexValue
        (SpecializeMethodName '
        NewMethodName) } + Methods.SN  &
({LexValue(InheritMethodNames)}
  -  {LexValue
          (SpecializeMethodNames)})
```

The notation is explained as follows :

{ } indicates an ordered list.

& is defined for both elements and lists :

a & {B, C, D, ...} is {a & B, a & C, a & D, ...}

{a, b, ...} & {B, C, D, ...} is {a &B, a & C, a & D, ..., b & B, b & C, b & D, ...}

where lowercase represents elements and uppercase represents sets.

+ appends two lists, that is,

{a, b, c, ...} + {d, e, ...} is {a, b, c, d, e}

- is set difference, that is,

{a, b, c} - {b} is {a, c}

' appends two method names, that is,

a ' b appends the new method name b before or after a, according to the syntax.

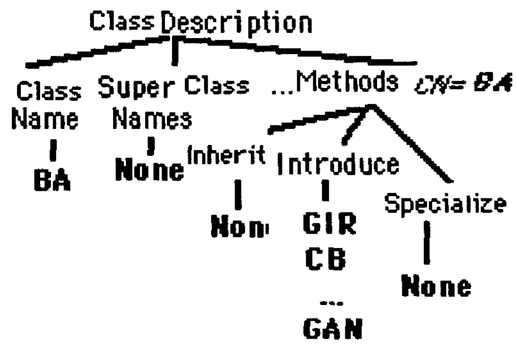
5.4 Attribute Evaluation

Using a topological ordering, the attributes for Bank Account will be evaluated first, then those for Savings Account, Checking Account and NOWAccount.

Bank Account

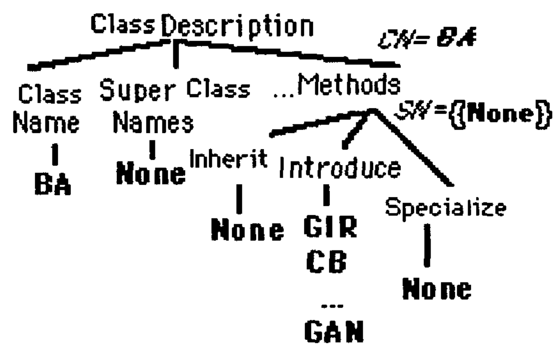
Attribute *CN* can be evaluated using function (i):

Bank Account :



Attribute *SN* can be evaluated using function (ii) :

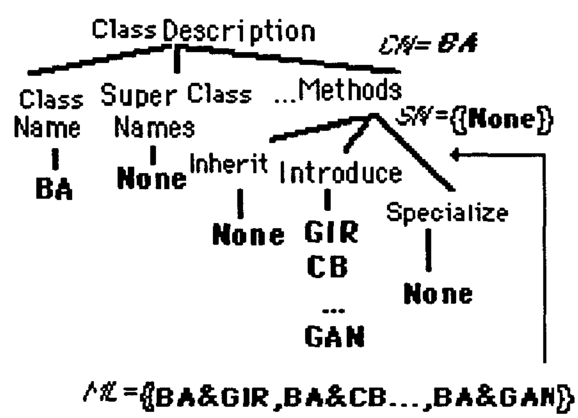
Bank Account :



Attribute *ML* can be calculated using functions (iii) and (iv). For function (iv),

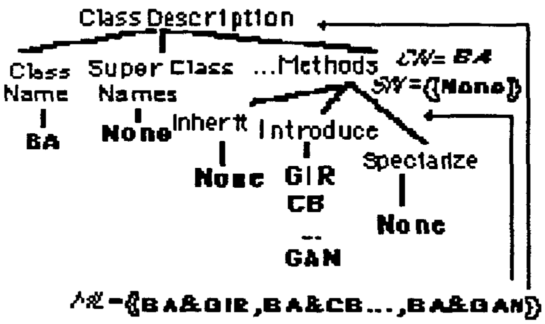
```
Methods.ML  = BA & { GIR, CB, ..., GAN }  +
None & ({ None } - {None} ) + BA & {None} =
{ BA & GIR, BA & CB, ..., BA & GAN } :
```

Bank Account :



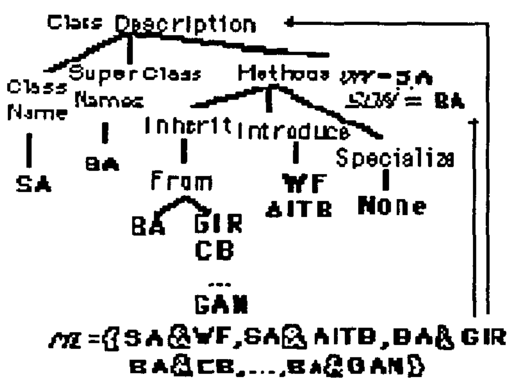
Function (iii) is simply a copy rule and the completely attributed tree is :

Bank Account :

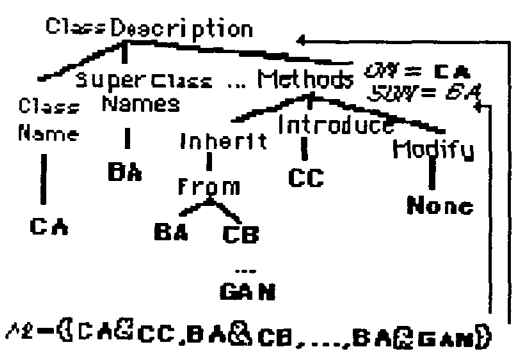


The rest of the trees are shown after complete attribute evaluation :

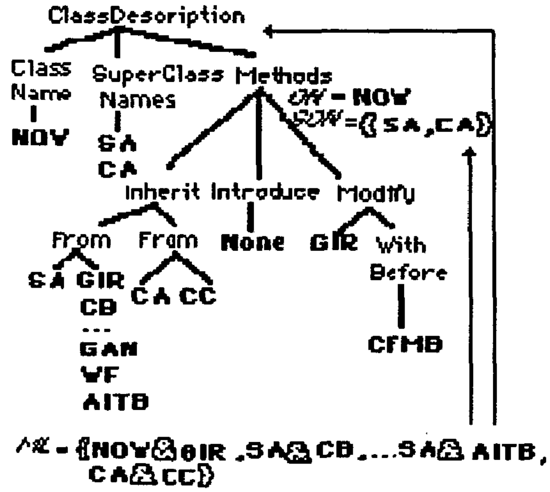
Savings Account:



Checking Account:



NOWAccount:



5.5 An Application

Inheritance, polymorphism, and late binding issues can now be facilitated using these attributes. We address each of these three issues by defining actions for the bank example.

(1) Inheritance

Since the code which defines a method may be passed down from class to subclass to subclass, and perhaps altered along the way, it is helpful and efficient to maintain the lists shown above which indicate the methods which a class has available to it. This information need not be used until run-time when a message is sent to an object to perform the selector operation. This technique is shown in Figure five.

```
 Action CreateNewSavingsAccount

{Example of inheritance, object
instantiation and messaging}
     LocalNames
           CompoundInterestRate := 6,
{ThisAccount used for instantiation of
savings account object}
           ThisAccount : SavingsAccount,
{IO used for instantiation of a utilities
object}
            IO         : Utilities
BeginAction  {CreateNewSavingsAccount}
{Instantiate a savings account object}
    ThisAccount := New (SavingsAccount);
{Instantiate a Utilities  object }
```

```
    IO := New (Utilities);
    { Send messages to object utilities and
to object savings account to set fields }
    IO : PutPrompt ("Owner?");
    IO : Input (Owner) ;
    ThisAccount : SetOwner(Owner);
    IO : PutPrompt ("Address?")
    IO : Input (Address);
    ThisAccount : SetAddress (Address);
    IO : PutPrompt ("PhoneNumber?");
    IO : Input (PhoneNumber);
    ThisAccount :
           SetPhoneNumber(PhoneNumber);
    ThisAccount : GenerateAccountNumber;
    IO : PutPrompt ("InitialDeposit");
    IO : Input (InitialDeposit);
    ThisAccount : DepositFunds
                      (InitialDeposit);
    ThisAccount : AddInterestToBalance;

EndAction CreateNewSavingsAccount
```

Figure 5 Inheritance

Here, the object Savings Account is created and messages are sent to it. The names of the corresponding methods are easily found from the attribute list computed for Savings Account.

Multiple inheritance is easily facilitated with the computed attribute lists, also.

(2) Polymorphism

The action in Figure six illustrates polymorphism by sending a message with the same selector (AddInterestToBalance) to two different objects. The attributes computed for NOWAccount and Savings Account again facilitate the selection of the correct method.

```
Action AddInterest
{Example shows simple polymorphism - can be
resolved at compile time}
Local Names
 Account1 used for instantiation of a
 NOWaccount object}
 Account1 : NOWAccount,
{Account2 used for instantiation of a
Savings account object}
Account2 : SavingsAccount,
{IO used for instantiation of a Utilities
object}
IO : Utilities
BeginAction {AddInterest}
  IO := NEW(Utilities);
  IO : Input(AccountName);
  Account1 :=
    GET(NOWAccount,AccountName);
{Presumably, AccountName has
2 different accounts}
 Account2 :=
   GET(SavingsAccountAccountName);
   Account1 :  AddInterestToBalance;
   Account2 :  AddInterestToBalance;

EndAction AddInterest
```

Figure 6 Simple Polymorphism

(3) Late Binding

Dynamic binding is a philosophy basic to many object-oriented programming environments. And yet, it is often admitted that dynamic binding is a flexibility that is paid for in terms of efficiency . The solution may be to do as much bookkeeping at compile time as possible to facilitate dynamic binding. Figure seven shows a message that is sent to an object whose identity is not known until run-time (since the object type itself is an input value) :

```
Action AddInterest

{Example shows late binding}

  Local Names
{Used for instantiation of an account
object}
Account : BankAccount,
 {Used for instantiation of a Utilities
object}
IO           : Utilities

BeginAction {AddInterest}

IO := NEW(Utilities);
IO : Input(AccountType);
IO : Input(AccountName);
IO : Input(Deposit);
Account := NEW(AccountType);
Account : DepositFunds(Deposit);
Account : AddInterestToBalance

EndAction AddInterest
```

Figure 7 Late Binding

Here, the specific object, whether it be a Savings Account, a Checking Account or a NOWAccount is not known until run-time. Nevertheless, once the proper object is identified, the bookkeeping done by the compiler facilitates accessing the proper method.

6. Conclusions

The use of attributes in compilers for semantic analysis is well established. Object-oriented languages present unusual semantic analysis needs in addition to the usual ones found in all languages. Using a very high-level syntax and BNF, three simple attributes, and four functions, we computed information that facilitates inheritance (including multiple inheritance), polymorphism, even bookkeeping for late binding.

In a real language, there would, of course, be many more nodes to the parse tree, and many more ⁰functions to compute. In addition, restrictions were made to the concepts here to ensure simplicity, and in some cases, just to ensure a decision. For example, not all object-oriented languages would declare inheritance in the ways done here, e.g., it is common to assume methods are inherited and to explicitly remove those not desired rather than mentioning explicitly those to be inherited (Gol84). Thus, the attribute functions need to be specialized for a particular syntax.

Appendix

A Simple Banking System Using OOPL Constructs

```
{ Class Descriptions }

Class     Utilities
     SuperClass            None
     ClassVariables
          Inherit          None
          Introduce        None
     SharedVariables
           Inherit          None
          Introduce        None
     Methods
           Inherit       None
           Introduce
                  Input,
                  PutPrompt
           Specialize    None
     Definitions
            Method Input (Info)
               LocalNames  None
               BeginMethod
                  Read(Info);
            EndMethod Input
            Method  PutPrompt (Info)
               LocalNames None
               BeginMethod
                  Write (Info);
            EndMethod  PutPrompt
EndClass   Utilities

Class      BankAccount
     SuperClass        None
     ClassVariables
           Inherit    None
           Introduce
                  Owner,
                  Address,
                  PhoneNumber,
                  AccountNumber
                  Balance
     SharedVariables
          Inherit    None
          Introduce


          HighInterestRate := 0.07,
             LowInterestRate := 0.045,
               NextAccountNumber:= 0
     Methods
          Inherit    None
          Introduce
               GetInterestRate,
               DepositFunds,
               SetOwner,
               SetAddress,
               SetPhoneNumber,
               GenerateAccountNumber
           Specialize     None
     Definitions
        Method GetInterestRate
          LocalNames None
          BeginMethod
          If (Balance > 10000) Then
          Return (HighInterestRate);
          Else
            Return (LowInterestRate);
          EndIf;
        EndMethod  GetInterestRate
        Method DepositFunds
                  (AmountDeposited)
          LocalNames None
          BeginMethod
              Balance := Balance +
              AmountDeposited
        EndMethod  Deposit Funds
        Method SetOwner (Name)
           LocalNames None
           BeginMethod
              Owner := Name;
        EndMethod  SetOwner
        Method  SetAddress(Name)
           LocalNames None
           BeginMethod
             Address := Name;
        EndMethod  SetAddress
        Method  SetPhoneNumber
                       (Number)
```

```
        LocalNames None
        BeginMethod
          PhoneNumber := Number;
    EndMethod  SetPhoneNumber
    Method GenerateAccountNumber
        LocalNames None
        BeginMethod
          NextAccountNumber :=
          NextAccountNumber + 1;
     AccountNumber:=
          NextAccountNumber;
    EndMethod
         GenerateAccountNumber
EndClass     BankAccount

Class SavingsAccount
      SuperClass
             BankAccount
      ClassVariables
           Inherit
                Owner,
                Address,
                PhoneNumber,
                AccountNumber,
                Balance
            From BankAccount
            Introduce   None
      SharedVariables
          Inherit
           HighInterestRate := 0.07,
           LowInterestRate  := 0.045,
              NextAccountNumber:= 0
          Introduce   None
      Methods
           Inherit
               GetInterestRate,
               DepositFunds,
               SetOwner,
               SetAddress,
               SetPhoneNumber,
               GenerateAccountNumber,
            From BankAccount
            Introduce
               WithDrawFunds,
               AddInterestToBalance
            Specialize    None
       Definitions
            Method WithDrawFunds
                (AmountRequested)
              LocalNames
                    IO,PromptString
              BeginMethod
               If (AmountRequested >
                       Balance) Then
               IO := New(Utilities);
               IO := PutPrompt
                      (PromptString);
                Else  Balance :=
                    Balance -
                    AmountRequested;
               EndIf
           EndMethod  WithdrawFunds
```

```
          Method AddInterestToBalance
           LocalNames InterestRate
           BeginMethod
            InterestRate := self :
                  GetInterestRate;
            Balance := Balance +
              Balance *InterestRate;
           EndMethod
               AddInterestToBalance
EndClass      SavingsAccount

Class   CheckingAccount
     SuperClass
               BankAccount
     ClassVariables
         Inherit
              Owner,
              Address,
              PhoneNumber,
              AccountNumber
              Balance
      From BankAccount
      Introduce
            JointAccount,
            SecondAccountHolder
      SharedVariables
          Inherit
           HighInterestRate := 0.07;
           LowInterestRate  := 0.045;
              NextAccountNumber:= 0
          From BankAccount
          Introduce
              BounceFee :=  12.50
       Methods
         Inherit
              DepositFunds,
            {GetInterestRate removed}
              SetOwner,
              SetAddress,
              SetPhoneNumber,
              GenerateAccountNumber,
         From BankAccount
         Introduce
              ClearCheck
         Specialize  None
         Definitions
         Method
            ClearCheck(CheckAmount)
            LocalNames IO, SomePrompt
            BeginMethod

If (CheckAmount > Balance Then
             IO := New(Utilities);
            IO:PutPrompt(SomePrompt);
                 Balance := Balance -
                   BounceFee;
             Else Balance :=
                Balance -CheckAmount;
             EndIf
          EndMethod  ClearCheck
EndClass  CheckingAccount
```

```
Class    NOWAccount
      SuperClass
             SavingsAccount,
             CheckingAccount
     ClassVariables
         Inherit
               Owner,
               Address,
               PhoneNumber,
               Balance,
               AccountNumber
               JointAccount,
               SecondAccountHolder
         From CheckingAccount
         Introduce     None
     SharedVariables
         Inherit
            HighInterestRate := 0.07;
            LowInterestRate := 0.045;
              NextAccountNumber:= 0
         From SavingsAccount
          Introduce   None
     Methods
         Inherit
              GetInterestRate,
              DepositFunds,
              SetOwner,
              SetAddress,
              SetPhoneNumber,
              GenerateAccountNumber,
              WithDrawFunds,
              AddInterestToBalance
         From  Savings Account
         Inherit
               ClearCheck
         From     CheckingAccount
         Introduce     None
         Specialize
               GetInterestRate
         With CheckForMinimumBalance
         Before
Definitions
         Method
             CheckForMinimumBalance
            LocalNames None
             BeginMethod
              If (Balance <
                     1000) Then
                    Return (0);
                  EndIf;
         EndMethod
              CheckForMinimumBalance

                  CheckForMinimumBalance
EndClass     NOWAccount

      { Actions }
       ...
```

Bibliography

Aho, A.V., Sethi, R., & Ullman, J. D. , Compilers, Principles, Techniques, and Tools, (1986) Addison Wesley, Reading MA.

Babich, W.A. & Jazayeri, M., "The Methods of Attributes for Data Flow Analysis, Part I. Exhaustive Analysis", Acta Informatica 10, (1978) 245 - 264.

Babich, W.A. & Jazayeri, M., "The Methods of Attributes for Data Flow Analysis, Part II. Demand Analysis", Acta Informatica 10, (1978) 265 - 272.

Bochmann, G.V., & Ward, P., "Compiler Writing System for Attribute Grammars", Computer Journal 21, 2 (1978) pp. 144 - 148.

Briot, J., & Cointe, P., "A Uniform Model for Object-Oriented Languages Using the Class Abstraction", (1987), Architectures and Languages, pp 40 - 43.

Cox, B.J., Object Oriented Programming, An Evolutionary Approach, (1986), Addison Wesley, Reading MA.

Demers, A., Reps, T., & Teitelbaum, T. , "Attribute Propagation by Message Passing", ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments.

Farrow, R., "Experience with an Attribute Grammar - Based Compiler", Ninth ACM Symposium on Principles of Programming Languages (1982) pp. 95 - 107.

Fischer, C.N. & LeBlanc, R.J., Crafting a Compiler, (1988) Benjamin/Cummings, Menlo Park, CA.

Goldberg, A., & Robson, D., Smalltalk-80: The Language and its Implementation, (1984) Addison-Wesley, Reading, MA

Harland, D.M., Polymorphic Programming Languages, Design & Implementation (1984), Harland/Ellis Horwood Limited.

Kennedy, K. & Warren, S.K., "Automatic Generation of Efficient Evaluators for Attribute Grammars", Third ACM Symposium on Principles of Programming Languages (1976), pp. 32 - 49.

Kennedy, K. & Ramanathan, J., "A Deterministic Attribute Grammar Evaluator Based on Dynamic

Sequencing", Fourth ACM Symposium on Principles of Programming Languages , (1977), pp. 72 - 85.

Knuth, D.E., "Semantics of Context-free languages", Mathematical Systems Theory, Vol. 2, No. 2, (1968) pp. 127 - 145.

Knuth, D.E., "Semantics of Context-free languages : correction", Mathematical Systems Theory, Vol. 5, No. 1, (1971) pp. 95.

Liskov, B., "Data Abstraction and Hierarchy", Addendum to OOPSLA '87, pp 17 - 34.

Lorho, B., (ed.), Methods and Tools for Compiler Construction, (1984), Cambridge University Press.

Schriver, B., & Wegner, P., (ed) Research Directions in Object-Oriented Programming, (1987), MIT Press.

Snyder, A., "Encapsulation and Inheritance in Object-Oriented Programming Languages", (1987), Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages and Applications, SIGPLAN Notices 21, 11

Wegner, P., "Dimensions of Object-Based Language Design", (1987), OOPSLA '87, pp 168 - 182.

Wilson, R., "Object Oriented Languages Reorient Programming Techniques", (1987), Computer Design pp. 52 - 62.

Work by Professor Lemone performed while on sabbatical leave at Ecole Polytechnique Fédérale de Lausanne, Switzerland.
Work by Professor McConnell performed under a WPI Goddard Fellowship & a Canisius faculty development grant.