1432 Library copy

OBJECT-ORIENTED PARADIGM IN LANGUAGE SPECIFICATION

Dissertation Submitted to
JAWAHARLAL NEHRU UNIVERSITY
in partial fulfilment of requirements for the award of the degree of
Master of Technology
in
Computer Science

by
Bhaskarjya Parashar

42p.

JNU

SCHOOL OF COMPUTER & SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI - 110 067

January 1996

CERTIFICATE

This is to certify that the dissertation entitled

OBJECT-ORIENTED PARADIGM IN LANGUAGE SPECIFICATION

which is being submitted by **Bhaskarjya Parashar** *to the* **School of Computer and Systems Sciences, Jawaharlal Nehru University, New Delhi for** *the award of* **Master of Technology in Computer Science,** *is a record of bonafide project work carried out by him under the supervision and guidance*

of

Prof. R.G. Gupta.

This work is original and has not been submitted in part or full to any University or Institution for the award of any degree.

[illegible]

Prof. G.V. Singh

(Dean, SC & SS)

[illegible]

Prof. R.G. Gupta

(Supervisor)

Dedicated to my parents. . .

ACKNOWLEDGEMENT

I wish to convey my heartfelt gratitude and sincere acknowledgements to my guide ***Prof R.G. Gupta,*** School of Computer & Systems Sciences for his whole hearted, tireless and relentless effort in helping me for the successful completion of the project.

I would like to record my sincere thanks to my Dean, ***Prof. G. V. Singh,*** School of Computer & Systems Sciences for providing the necessary facilities in the centre for the successful completion of the project.

I take this opportunity to thank all my faculty members and friends for their critical comments during the course of the project.

[illegible]

Bhaskarjya Parashar

CONTENTS

1. INTRODUCTION

In making a study of programming languages and their evolution we can also classify them into two main subcategories:

i) Procedural languages or Computational languages.

ii) Object Oriented languages.

Computational languages serve to describe the combination of functions, either by simple combination or by recursive combination or by iterative combination. Most of the programming languages e.g FORTRAN, LISP, APL etc., are of this type. Object oriented languages on the other hand follows a different kind of programming style. They can be seen as having evolved from the procedural type of programming style and which are in a certain sense of higher level than computational languages. In procedural languages we describe algorithms; in object oriented languages we describe objects, to which algorithms determined by their form and yielding function values determined by the parameters of an object are understood to apply.

Computational languages may be classified according to the complexity of the data (objects) and basic functions which they provide.

The simplest computional languages are the machine level assembly languages. A language of this type provides only rudimentary data types (machine words) and a limited fixed repertoire of functions (essentially the wired instructions of a particular machine). Function combination is expressed only by a succession of operations; iterations only by explicitly written transfers; and recursion not at all.

At the next level of sophistication among computational languages or procedural languages we find such languages as FORTRAN. Languages of this kind provide a few data types (integers, reals, dimensional arrays); provide certain basic operations related to these data types (notably indexed access operations for dealing with arrays); and allow the combination of functions in considerably more convenient ways (functions nesting, infix operations) than are allowed by assembley languages. Iteration forms represented by special DO and FOR statements may also be provided. Languages of this kind may quite naturally be extended to include a greater variety of data objects and of corresponding basic operations; such extensions are made by PL/1 and various other languages which provide methods for data structuring more general than array dimensioning.

A still more sophisticated or specialized procedural language will refer directly to compound objects and will provide operations dealing directly with such objects. Examples are LISP (which uses binary lists), APL (which works with arrays as objects) or more explicitly set theoretical languages which may be defined without undue difficulty.

All languages mentioned thus far are computational in that they are concerned with functions and their algorithmic combination. *Object-oriented* languages by contrast serve not for the combination but for the definition of functions. In object-oriented languages, one deals not with procedures but with mechanisms or models. With fully described model one is able to specify these functions in an effective way. A model is described as listing its parts and describing the connections of these parts.To define a part within an object-oriented language, one specifies the manner in which the part reacts to all other parts to which it is connected; these specifications, together with a description of the interconnection pattern characterizing a given mechanism, will determine the action of the mechanism. i.e., will determine certain mechanism related functions.

1.1 The Object-Oriented Paradigm

The object-oriented paradigm is a software design and development technology incorporating several sophisticated and efficient mechanisms that provide an organizational framework for the development of large and complex software projects. In comparision to traditional software techniques, the object-oriented technology improves development of software systems by facilitating better factoring of functionality and related data. To some researchers the object-oriented paradigm *is more* than a better approach to software development; it is a goal that software designers and developers should aim for. In the words of Cox :

To get a grip on object-oriented means coming to the realization that it is an end, not a means --an objective rather than the technologies for achieveing it. It means changing how we view software, shifting our emphasis to the objects we build rather than the processess we use to build them. It means all available tools, from COBOL to Smalltalk and beyond, to make software as tangiblem --and as ameneable to common sense manipulation --as are the everyday objects in a department store. Object-oriented means abandoning the process-centric view of the software universe where the programmer-machine interaction is paramount in favour of a product-centered paradigm driven by the producer-consumer relationship.

The object-oriented programming approach is a *data oriented* approach to software design and development, where the data is encapsulated in objects and messages are used to manipulate the data. An object is the encapsulated data that can be accessed or manipulated by means of a set of interface functions or handles. A message is the mechanism by which a particular operation is performed on the data encapsulated within an object. Thus, an object is defined in terms of the data it encapsulates and the operations on the data that are allowed by the set of interface functions.

Encapsulation of data enables information hiding. The actual method of storage of the encapsulated data is an implementation detail which is independent of how the data is used. The operations which can be performed on the encapsulated data are specified as part of the interface to the object. These operations are also called the *interface functions*. The implementation of the interface functions is internal to the object. The implementation details of the operations that manipulate the stored data can be changed without affecting the interface. Thus, the concept of an object incorporates information hiding and data abstraction.

The only way that an object can be manipulated is by the set of interface functions. The operations which are defined are generic operations that are applicable to other similar objects. All objects that have similar characteristics are said to belong to the same class. The concept of a class is central to the object-oriented paradigm. A class is an abstract definition of the characteristics of objects that have similar appearances and that allow similar operations to be performed on the encapsulated data. Thus, a class is an abstract definition of the data being encapsulated along with the definition of the set of operations that can be performed on the instances of that class. An object is an instance of a particular class and is distinct from other instances of the same class.

The concept of inheritance is also central to the object-oriented paradigm. Inheritance allows the creation of a new class as a model. It is the mechanism that promotes reusability of code. The fundamental idea employed is that of defining new classes in terms of existing classes.

In general, object-oriented programming implies the availlability of *data abstraction*, *inheritance*, and *polymorphism* features. The other important concepts in object-oriented paradigm are *multiple inheritance*, *parameterized types*, and *persistence*. These concepts will later be covered in depth.

In other words, the motivation for using object-oriented paradigm in programming can be summed up in the following points:

**Object-oriented approaches encourage the use of *modern* software engineering technology.

**Object-oriented approaches promote and facilitate code reusability.

**Object-oriented approaches facilitate interoperability.

**Object-oriented approaches result in code which can be easily modified, extended and maintained.

**Object-oriented approaches produce solutions which closely resemble the original problem.

**There is a significant reduction in integration problems.

**The conceptual integrity of both the process and the product improves.

1.2 Basic Object-Oriented Concepts

Object-oriented languages are far more closely related to imperative languages than to functional or even logic programming languages. For example, they (typically) use the same execution model: a version of the von neumann computer in which a complex *structured* state is modified under explicit software control. Object-oriented languages may be viewed as imperative languages that introduce a number of new concepts such as variables, arrays, structures and functions. In the following paragraphs only some of the basic 'new' concepts (in fact, not all these concepts are brand new; some of them were previously used in modern imperative languages or in languages used in the artificial-intelligence area) will be discussed.

1.2.1 Objects

The fundamental concept of object-oriented languages is the *object*. Objects are run-time units that can preserve some local data (the object's internal state) and that can respond to certain requests addressed to them by carrying out an operation. It comprises of an object state, expressed in terms of actual values of a set of *attributes*, together with functions, called *object methods*, operating on this state. The attributes and methods together forms the *features* of an object. Thus an object encapsulates both data and the operations on this data. One of the most important basic operations of object-oriented languages is the activation of a method m for an object o. Here, the object plays the major role; the method, being a component of the object is subordinate to it. This object orientation has given its name to the class of languages. Objects are volatile just like variables are in procedural languages. Some languages have incorporated mechanisms for rmaking objects objects persist beyond the life of the process in which they were created.

1.2.2 Object classes

We know that type information is an important prerequisite for generating efficient and reasonably reliable programs. Object-oriented extend the known type concept of imperative languages such as PASCAL or C (not all object-oriented languages use static typing, as the example of Smalltalk-80 shows). Their types are usually called object classes. An object class specifies attributes and methods together with their types and prototypes (i.e., types of return values and parameters), respectively. In order to belong to the class, objects must contain these features, but they also contain others. Some object-oriented languages such as Eiffel permit further specifications for the methods, such as pre- and post-conditions. This provides a means of restricting the meaning of methods (semantics). Frequently the class also defines the methods; however, under certain conditions, these definitions can be overwritten. The object class forms the modularization unit of object-oriented languages.

In a PASCAL-like language, a rough counterpart of an object is a dynamic variable; then a class corresponds to a record type. A reference value to access an object corresponds to a value of a pointer type, and an operation corresponds to a subroutine whose parameters employ the pointer type.

1.2.3 Class hierarchies and Inheritance

Inheritance is defined as the incorporation of all features of a class A into a new class B. B may also define other features and, under certain conditions, overwrite methods inherited from A. Some languages permit the renaming of the inherited features to avoid name conflicts or simply to allow more meaningful names in the new context.

If A inherits from B then the class A is *derived* from B and B is called a *base class* for A. In other words class A is defined as the sub-class of B: this relation is deonoted with B -> A. If class A is a subclass of class B, B is called the superclass of A. A class system is pair (C,->), where C is the set of classes. As usual, the transitive closure of this relations is denoted by ->+ and the reflexive transitive closure is denoted by ->*. A subclass specializes (or extends) its superclass by introducing properties that are added to those of the superclass. The objects created as instances of the subclass therefore *inherit* the properties of the superclass which they have in addition to the properties defined by the subclass itself. Hence B -> A implies that an object of class A responds to all the requests specified by class B, and it contains all the attributes given by class B. In contrast, an object that is created as an instance of class B does not recognize the requests and attributes specified by class A. Due to the inheritance of properties, we may think that the properties of an object are divided into layers so that the properties originating from a particular class occupy one layer. If an object is created as an instance of class A, it has a layer for each class C such that C ->* A. It is natural to assume a partial order for the class layers of an object: C ->+ A implies that C-layer is above the A-layer.

Since a superclass may in turn be a subclass of another class, the inheritance of properties becomes transitive: if C ->+ A, an instance of A has all the properties specified by class C as well. We say that A is a descendant class of C, and C is an ancestor class of A. We require that the class system is cycle-free, i.e., A ->+ A holds for no A (a class cannot be its own descendant or ancestor).

If a class can have at most one superclass, we say that the class system has single-inheritance. In that case the class system imposed by relation -> is purely hierarchical, tree-like (or actually a forest). Otherwise the class system has multiple inheritance, and the class structure becomes a directed acyclic graph. If a class system has single inheritance, the class layers of an object represent a path in a class tree, ending at the root.

Inheritance is one of the most important concepts of object-oriented languages. It makes extensions and the formation of variants very easy. The resulting inheritance hierarchies also permit a structuring class libraries and the introduction of different levels of abstraction. It provides us with the possibility of reusing parts of an existing implementation in a simple way, extending them and, if need be, adapting them locally to particular requirements or circumstances by overwriting individual methods.

Moreover we retain the facility to define abstract classes. This leads to a flexibility in programming languages similar to that achieved in natural languages by means of abstract concepts: we retain different levels of abstraction. Anything that can be formulated at a high level of abstraction tends to have a wide area of application and thus is to a large extent reusable.

Typed object-oriented languages take account of the inheritance hierarchy in their type system. If a class B inherits from a class A then the type assigned to B is a sub-type of the type assigned to A. Every object of a sub-type is automatically also an element of the subtype; an inheriting class is a subclass of the class from which it inherits. This has the following effect:

Subtype rule. When an object of a certain type is required at an input position (function input parameter, right-hand side of assignments) or as a function return value, objects of any subtype are allowed.

Method selection rule. If class B inherits from class A and overwrites method m then for B objects b the definition of m given by B (or a sub-class) must be used even if b is used as an A object.

Dynamic-binding rule. A method of an object o, which can potentially be overwritten in a sub-class, has to be bound dynamically if the compiler cannot determine the run-time type of o.

1.2.4 Polymorphism

Objects are created by reference values stored in the attributes of other objects or in the local variables or parameters of routines; we call these locations collectively *slots*. If a slot is associated with a statically known class, we say that the language is *statically typed*; the associated class is called the *static class* of the slot (and of the referenced object). In a statically typed language a slot is always known to refer to an object of the static class. However, if C ->+ A, an object of class A can also be viewed as an object of class C; hence a slot with static class C may also refered to an A object. This is called (inclusion) *polymorphism* in object-oriented languages. We note that the class layers of the object above the static class are known statically (*static class layers* while the class layers below the static class are known only at the run time (*dynamic class layers*).

Example 1. Class hierarchies and objects

Suppose that we have the following class hierarchy:

```
Vehicle
        Car
        Bicycle
        Boat

Steering Device
        Rudder
        Wheel
                Sport Wheel
                StdWheel
Engine
        SteamEngine
        DieselEngine
```

where subclass relation is denoted by nesting. Hence, there are three topmost ancestor classes:Vehicle, SteeringDevice, and Engine; each of them has subclasses. The entire class system is a forest. On the other hand, a car object has as its components an Engine object and a Wheel object. This could be expressed as:

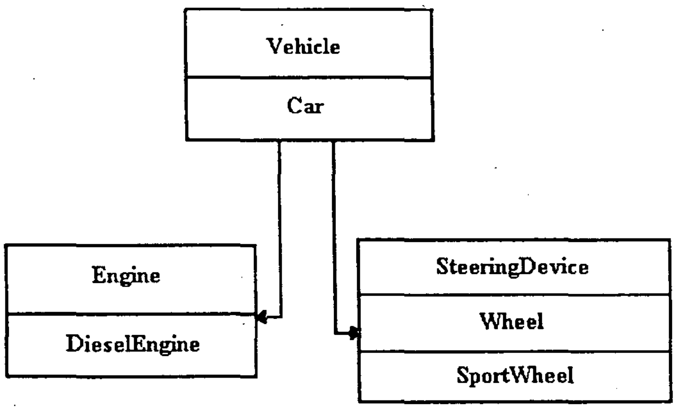


Fig. 1. Examples of objects. A "Car" object (with superclass "Vehicle") has component objects of static classes "Engine" and "Wheel". The arcs show the component objects, the static type of the objects is indicated by the target layer. The component objects have (in this case) dynamicclasses "DieselEngine" and "SportWheel", respectively. In addition the wheel object object has superclass "SteeringDevice".

Car = Engine Wheel

The different layers of a car object and its component objects are shown in Fig. 1.

The properties of an object denoted by a slot can be referenced using the conventional dot notation, as if the properties were fields of a record. However, in a statically typed language only the properties specified by the static class and its ancestor classes can be referenced through the slot.

1.2.5 Dynamic binding

Sometimes it is natural to specify a property for a class, even though the actual meaning of the property depends on the sub-classes. For example, every instance of clas "GeometricShape" recognizes the request "DrawYourself" displaying the shape in question, but the actual drawing procedure depends on the sub-class (say, "Circle") of the object. We say that "DrawYourself" is in that case a *virtual property* of class "GeometricShape": the property is known to the class only in an abstract sense. If a class contains virtual properties, it cannot be instantiated because the meaning of the virtual properties could not be determined for the resulting object. Such a class is called an *abstract class.*

A virtual property is an example of *dynamic binding*: the name of the property is bound to its actual meaning only at run-time. In the case of a virtual property the descendant classes are forced to provide the actual specification of the property. A more liberal form of dynamic binding is the *redefinition* of properties in descendant classes: a descendant class may redefine some property already defined in its ancestor class (e.g. Smalltalk, Eiffel). In that case any object of the descendant class follows the redefined meaning of the property, instead of the original one in the ancestor class. However, an object that is created as an instance of the ancestor class follows the original meaning; this class is not regarded as an abstract class.

Dynamic binding is one of the key features of object-oriented languages, allowing the dynamic adaptation of properties to the current descendant classes.

1.2.6 Information hiding

Often object-oriented languages provide facilities for information hiding: classes may specify that certain properties are not available for the clients (or subclasses) of that class, resembling interface specifications of modules in languages like Modula-2. Such classes support data abstraction in the same sense as modules: a class may export only its abstract properties and hide its implementation details. For example, a considerable part of the object-oriented extensions in C++ pertains to limiting the usage of class properties in various ways. A language may also automatically hide certain kinds of properties; for example, in Smalltalk all attributes of objects (instance variables) are hidden to the clients, while all operations (methods) are visible.

1.2.7 Prototypes

An alternative approach to the conventional object-oriented model is prototype based inheritance. In this approach there are no classes: objects inherit properties from their parent objects rather than from their superclasses. Objects are not created as instances of classes, but they are "cloned" from other objects. This model is conceptually simpler and more flexible, but class-based inheritance is regarded as more reliable.

We can ask ourselves *"Why object-oriented language specification?"*. Although object-orientation is a popular design paradigm for languages and software, it has seldom been used as a language specification/implementation paradigm, that is as a design paradigm for programming languages or implementation softwares.

Nowadays, though a large number of high level programming languages help in producing object-oriented software, the languages themselves are not conceived around an object-oriented base. In other words, the compiler construction methodology is not in tune with the notion of object-oriented paradigm. The current compiler construction methodology follows the *structured programming paradigm.* If the construction procedure is improved upon so that it follows the object-oriented methodology, then the languages subsequently developed will inherently support production of object-oriented software. To make this feasible, the language specification structures (like the context-free grammars, attribute grammars etc.) need to be changed or modified so that they reflect object-oriented characteristics. My work in this paper is concentrated in extending these language specification structures, mainly attribute grammars to incorporate object-oriented characteristics.

2. LANGUAGE SPECIFICATION STRUCTURES

The syntax of a language specifies how programs in a language are built up. Syntactic structure, i.e. the structure imposed by syntax on a language, is the primary tool for working with a language. It has been used to organise language descriptions and translations, as well as rules for reasoning about programs in a language.

The syntax of a programming language is almost always specified using some variant of a notation called context-free grammars, or simply grammars. The most important variants are BNF (Backus Naur Form), EBNF (Extended BNF) and syntax charts.

2.1 Context-Free Grammars

A context-free grammar (CFG) is a four-tuple (T, N, S, P), where T is the set of terminal symbols, N is the set of non-terminal symbols, S in N is the start symbol, and P is a finite set of productions of the form A->w, where A is in N and w is in $(T \cup N)^*$. We assume that N and T are disjoint.

2.1.1 BNF

The concept of a context-free grammar consisting of terminals, non-terminals, productions and a starting non-terminal, is independent of the notation used to write grammars. In BNF, non-terminals are enclosed between the symbols < and > and the empty string is written as <empty>.

Grammars for arithmetic expressions are based on rules like the following:

An expression is a sequence of terms separated by + or -.

A term is a sequence of factors separated by * or **div** .

A factor is a parenthesized expression, a variable or a constant.

The following is a partial grammar for expression without the rules for variables and constants:

```
<expression> ::= <expression> + <term> | <expression> - <term> |
                 <term>
<term> ::= <term> * <factor> | <term> div <factor> | <factor>
<factor> ::= (<expression>) | <variable> | <constant>
```

2.1.2 Extended BNF

In EBNF (extended BNF), non-terminals begin with uppercase letters, terminals consisting of symbols like + and - quoted, and terminals in boldface like **div** appear as it is. Furthermore:

A vertical bar, |, represents a choice.

A parenthesis, (and), are used for grouping.

A braces, { and }, represents Zero or more repetitions.

Brackets, [and], represent an optimal construct.

An EBNF version of the grammar given above is:

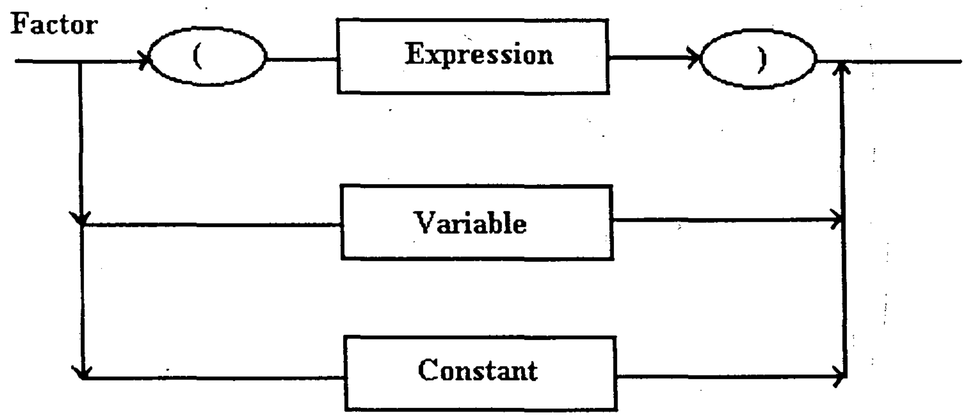
```
Expression ::= Term { ( '+' | '-' ) Term }
Term ::= Factor { ( '*' | div ) Factor }
Factor ::= '(' Expression ')' | variable | constant
```

Here Term { ('+' | '-') Term } represents a sequence of one or more terms separated by either + or - signs.

The quote around + and - , which are known from the context to be terminals, will often be dropped. The quotes around the parentheses in '(' expression ')' will be retained, however, to say that the parentheses are terminals in the language of expressions.

2.1.3 Syntax Charts

A syntax chart or syntax diagram, is another way of writing a grammar. It is constructed as follows - there is a sub-chart for each non-terminal similar to the chart given below for Factor.



Each production for the non-terminal results in a path through the chart. Along the path for a production are the terminals and non-terminals on its right side; the terminals are enclosed in rounded boxes and the non-terminals in rectangular boxes. Braces denoting zero or more repetitions, lead to syntax charts containing cycles.

Context-free grammars gives a specification of the syntax of a programming language. It can also be used to help guide the translation of programs. Many programming language constructs have an inherently recursive structure that can be defined by context-free grammars. For example, we might have a conditional statement defined by a rule such as

If S_1 and S_2 are statements and E is an expression, then
" if E then S_1 else S_2 " is a statement.

This form of conditional statement cannot be expressed using regular expressions (they can specify the lexical structure of tokens). On the other hand, using the syntactic variable *statement* to denote the class of statements and *expression* the class of expressions, we can readily express the above statement as

statement -> if *expression* then *statement* else *statement*

Modern programming languages have evolved around the *context-sensitive* grammars because there is a need of associating semantic rules with productions (obtained from the context-free grammars). To capture the essence of context-sensitivity we have a *syntax-directed definition* in which we generalize the context-free grammar so that each grammar symbol has an associated set of attributes (properties). That syntax-directed definition is a notation for associating semantic rules with productions. Syntax-directed definitions are high-level specifications for translations. They hide many implementation details and free the user from having to state explicitly the order in which translation takes place.

An attribute can represent anything we choose; a string, a number, a type, a memory location or whatever. The value of an attribute at a parse tree node is defined by a semantic rule associated with the production used at that node. The value of a synthesized attribute at a node is computed from the values of attributes at the children of that node in the parse tree; the value of an inherited attributeis computed from the values of attributes at the siblings and parent of that node.

Semantic rules set up dependencies between attributes that will be represented by a graph. From the dependency graph we derive an evaluation order for the semantic rules. Evaluation of the semantic rules defines the values of the attributes at the nodes in the parse tree for the input string. A semantic rule may also have side effects, e.g., printing a value or updating a global variable.

2.2 Form of a Syntax-Directed Definition

In a syntax-directed definition, each grammar production A -> w has associated with it a set of semantic rules of the form $b := f(c_1, c_2, \ldots, c_k)$ where *f* is a function, and either

1. *b* is a synthesized attribute of A and $c_1, c_2, \ldots, c_k$ are attributes belonging to the grammar symbols of the production, or

2. *b* is an inherited attribute of one of the grammar symbols on the right side of the production, and $c_1, c_2, \ldots, c_k$ are grammar symbols of the production.

In either case we say that attribute b *depends* on attribute $c_1, c_2, \ldots, c_k$. An *attribute grammar* is a syntax-directed definition in which the functions in semantic rules cannot have side effects.

2.3 Attribute Grammars

As mentioned earlier, a number of necessary properties of programs cannot be described by a context-free grammar. These properties are described by predicates on context information, so called context conditions. These include the declaration-related properties and the type consistency. Both depend on the scoping and visibility rules of the programming language.

An attribute grammar specifies context-dependent computations on tree structures, which are dsecribed by the underlying context-free grammar. Specifications of computations are associated to its productions. Dependencies between computations are expressed by definition and use of attributes. The context-ferr grammar which is the skeleton of the attribute grammar should be designed such that attribution can be specified as clear as possible and without unnecessary redundancy. Especially context-free grammar terminals without relevant information and certain chain productions can be ommited from the context-free grammar. Hence in case of compiler specification the context-free grammar should be an abstract syntax derived from the complete system[Kasten91].

Attribute grammars are used to described the static semantic analysis in most compiler generating systems.they associate attributes, as carriers of static semantic information, with the symbols of a context-free grammar, the so called **underlying grammar**. In addition they show what the **functional dependencies** between the values of **occurrences of attributes** in the productions of the grammar are. Such a functional dependency can be viewed as a computational prescription, which specifies how the value of the occurrence of an attribute is calculated from the values of other occurrences of attributes of the same production.

Appropriate conditions for the functional dependencies ensure that all **instances of attributes** in every syntax tree for a syntactically and semantically correct program can be **evaluated**, that is, can be assigned a value from their **domain of attribute values.**

Definition (attribute grammar)

Suppose G = (V, T, P, S) is a context-free grammar. We write the pth production in P as p : $X_0 \rightarrow X_1 \ldots Xn_p$, $X_i \in (V \cup T)$, $0 \leqslant i \leqslant n_p$

An **attribute grammar AG** over G consists of

1. an association of two disjoint sets, *Inh*(X), the set of the **inherited** attributes and *Syn*(X), the set of **synthesized** attributes, with each symbol of (V ∪ T

). We let $Attr(X) = Inh(X) \cup Syn(X)$ denote the set of all attributes of X; if $a \in Attr(X_i)$, then a has an **occurrence** in production p at the occurrence of X_i, which we write as a_i. Let V(p) be the set of all attribute occurrences in production p.

$$Inh = \bigcup_{X \in V} Inh(X); \quad Syn = \bigcup_{X \in V \cup T} Syn(X); \quad Attr = Inh \cup Syn$$

2. the specification of a **domain** D_a for each attribute *a* *Attr*, that is, the set of each potential values;

3. a **semantic rule**

$$a_i = f_{p,a,i}(b^1{}_{j1}, \ldots, b^k{}_{jk}) \quad (0 \leq j_l \leq n_p) \; (1 \leq l \leq k)$$

for each attribute *a* *Inh(Xi)* for $1 \leq i \leq np$ and each *a* $syn(X_0)$ in every p, where $b^l{}_{jl}$ attr(Xjl) $(0 \leq j_l \leq n_p)$ $(1 \leq l \leq k)$. Thus, $f_{p,a,i}$ is a function from $D_b{}^1$ * . . . * $D_b{}^k$ to D_a.

We always view an attribute as an attribute of one non-terminal or terminal; that is, the assignments *Inh* and *Syn* can be viewed as injective functions from the set V u T into the set of attributes.

2.4 Why Object-Oriented Attribute Grammars?

Although object-orientation has its indisputable advantages in software development, it is not at all clear that this basically procedural paradigm, can or should be integrated with a declarative specification formalism like attribute grammars. There are several reasons that can motivate this integration.

First, due to their declarativeness attribute grammars are a static specification method. Each attribute has a single value during its lifetime, and the tree structure controlling the evaluation of the attributes remain the same. This is sufficient for specifying various kinds of mappings - like the mapping from a source program to a target program - but in many cases we need a more dynamic view of a program in a language environment. This is the case e.g. in language - oriented editors, interpreters, and debuggers. In a sense incremental evaluation algorithms were the first step towards a more dynamic view of attribute grammars, but when trying to preserve the original form of attribute grammars they also preserve its basic weaknesses as a specification tool. Integrating attribute grammars with object-oriented mechanisms is a possible way to extend this formalism with explicit dynamic capabilities - but of course not the only one.

In a sense object-orientation is the greatest common factor of all applications; this is the basic reason for its wide popularity. A general benefit of object-orientation is that it supports "application-oriented" software, that is, software following closely the concepts and structures of the application domain. In a language specification formalism like attribute grammars this means that the specifications are given in terms of the source language. When combined with appropriate special mechanisms tuned for language implementation, such a specification can to large extent ignore conventional compilation techniques. This makes the specifications more high-level, and therefore easier to construct, understand, and maintain. On the other hand, since object-orientation is the greatest common factor of different aspects of language implementation, too, the implementation descriptions become unified; very different kinds of language implementation problems are solved using the same basic mechanisms.

One could argue that attribute grammars as such also support source language oriented, "high-level" specifications. This is true in principle, but unfortunately in practice the role of attribute grammars in language implementation tends to be a declarative wrapping for a fairly conventional compiler. The reason for this is that attribute grammars are only the skeleton without flesh: the mere fact that you can specify the relations between some data associated with the nodes of a syntax tree is too primitive a notion to be nothing but a basic framework in which more advanced features can be placed.

A particular benefit of object-orientation is its support for reusing existing code: new classes can be specified as extensions on existing ones, without the need to repeat the properties of the ancestor classes. This contributes to the (conceptual) shortness of the code, and to the productivity of the programmer. In object-oriented attribute grammars we have the same effect, at least in principle: common parts of the specifications can be "factored out".

3. OBJECT-ORIENTED LANGUAGE SPECIFICATION

In a context-free grammar (T, N, S, P), where T is the set of terminal symbols, N is the set of non-terminal symbols, S is in N and is the start symbol, and P productions of the form A -> w, where A N and $w \in (T \cup N)^*$. The cardinality of set c is denoted with card(C). as usual, we use symbol => for derivation: vAu => vwu if (A -> w) $\in$ P.

An object-oriented view usually implies that the underlying concept, context-free grammars, is also seen in an object-oriented sense.

`According to the usual interpretation of context-free grammars, a source text is a collection of (nested) instances of non-terminals. Hence, we see that there are obvious candidates for the counterparts for the object-oriented concepts "class" and "object" in context-free grammars. They are as follows:

1. A non-terminal corresponds to a class and a particular language structure generated by a non-terminal (or a node of the syntax tree) in the source text corresponds to an object.
2. A production can be visualised as the structural specification of an object: an occurrence of a non-terminal on the right-handside of a production represents a slot whose static class is the given non-terminal class. For each right-hand side non-terminal occurrence there is a component object for an object of the left-hand class.

To relate the subclass/superclass relations of the classes with features of the context-free grammars we have something very natural, although it appears to be surprisingly simple: each chain production(i.e. a production whose right-hand side consists of a single non-terminal symbol) represents an instance of such a relation.

More precisely, production A -> B implies that A is a superclass of B. The intuitive or logical reasoning behind such an idea is that in typical grammars a bunch of unit productions with the same left-hand side describes the (named) alternative forms of a particular language structure, in other words, the sub-classes of that language structure.

`For example in a PASCAL- like language we might have:

Statement -> assignmentStat | ProcedureCall | WhileStat | . . .

This kind of production pattern is rather usual because language designers tends to give names to the alternative forms of language structures. To be sensible from the object-oriented point of view, we must require that the alternative forms of a language structure are *always* named: we cannot have unnamed classes.This requirement implies the non-terminals will be placed in two separate categories:

1. those non-terminals having only a single production and
2. those having a set of chain productions.

We call these categories *basic non-terminals* and *superclass non-terminals,* respectively.

From the object-oriented point of view, basic non-terminals specify the syntactic compositions of basic language structures and superclass non-terminals specify the ancestor class hierarchies of these language language structures. A basic non-terminal provides a single syntactic pattern for each basic structure. Since these patterns possibly make use of the names of the ancestor classes, such specifications alone would not constitute a complete grammar. Superclass non-terminals, on the other hand, are not syntactic by nature: they simply define the subclass\superclass relations for a set of class names. However, this specification acquires syntactic significance due to the principle of polymorphism: if A is an ancestor class of B, then any pattern requiring an A structure in a particular context must be satisfied with a B structure, too. Consequently, together with this class hierarchy the syntactic specifications of the basic structures specify a complete language. The separation of the class hierarchy and the syntactic definitions of the basic structures is clearly seen in the object-oriented attribute grammar formalisms discussed in the sequel.

We have mentioned before that we cannot have circular class hierarchies, an object-oriented context-free grammar must be cycle-free (that is, A ->+ never holds for a non-terminal A). From a purely syntactic point of view it would be sensible to require that the grammar is reduced (i.e. every non-terminal is needed in the derivation of some sentence of the language); however, from the object-oriented point of view this would be unnccessarily restrictive. We can have two approaches. They are:

1. We require that all the classes (non-terminals) are sometimes used as a static class, or
2. We allow classes that are used only as ancestor classes of the static classes.

Naturally, the latter classes would be "useless" from the syntactic point of view, but they may offer some useful general properties to the objects instantiated using their descendent classes. On the other hand, they do disturb the interpretation of syntax in any way. To make possible these two approaches we present in the following also a "strong" variant of an object-oriented context-free grammar.

Defination 1. A context-free grammar is *pseudo-reduced*, if for all non-terminals A either $S \Rightarrow^{*} uAv \Rightarrow^{*} w$ or ($A \Rightarrow^{+} B$ and $S \Rightarrow^{*} uBv \Rightarrow^{*} w$), where $u,v \in (T \cup N)^{*}$ and $w \in T^{*}$.

Definition 2. A pseudo-reduced, cycle free context-free grammar is *MI(multiple inheritance)*-structured, if for each A N either 1) or 2) holds:

1. card{ p $\in$ P | p = (A -> w) for some w } = 1 and (A -> w) $\in$ P => w $\in$ $(TuN)^*$ \ N
2. (A -> w) $\in$ P => w $\in$ N.

These conditions are clearly non-overlapping; the non-terminals satisfying 1) and 2) are basic non-terminals and superclass non-terminals, respectively. We note that if a non-terminal A has a single chain production, it is regarded as a superclass non-terminal. (Naturally it could be regarded as a basic non-terminal consisting of a single sub-structure as well - and this is possible to express in the real systems - but since we are primarily interested only in the grammatical conditions we make this simplifying decision.) MI stands for multiple inheritance: a class system defined above allows situations in which a class has more than one superclasses. however, multiple inheritance may be undesirable for various reasons. The following definition excludes multiple inheritance (SI stands for single inheritance):

Definition 3. An MI-structured CFG is SI-structured, if ((a -> B)$\in$P and (C -> B) $\in$ P) => A = C.

If only syntactically meaningful non-terminals are allowed, we must have the additional requirement that the grammar is reduced:

Definition 4. An MI-structured context-free grammar is *strongly MI-structured* if it is reduced.

Definition 5. An SI-structured context-free grammar is *strongly SI-structured* if it is reduced.

Finally we associate object-oriented context-free grammar with classes by defining the class system associated with an MI-structured context-free grammar (and hence an SI-structured context-free grammar, too).

Definition 6. The *class system* of an MI-structured context-free grammar is (N, ->) where A -> B if and only if (A -> B) $\in$ P.

In the sequel we shall consider mainly SI-structured context-free grammar, we can determine the objects that correspond to a given input string as follows. Consider the syntax tree of the input string (we note that un-ambiguity is not required for MI- or SI-structured grammars; however, we assume that in case of ambiguity there is some way of choosing the "right" syntax tree). Each instance of a basic non-terminal in the tree corresponds to the lowest class layer of an object. The upper class layers of the objects are determined directly by the ancestor classes of the basic non-terminal (class). This object will be a component object for the object that corresponds to the nearest instance of a basic non-terminal encountered above in the tree.

Example 2. *SI-structured expressions*

Conventionally, expressions are specified syntactically using context-free grammars of the form:

```
Expression -> Expression AddOp Term | Term
Term -> Term MulOp Factor | Factor
Factor -> number | '(' Expression ')'
AddOp -> '+' | '-'
MulOp -> '*' | '/'
```

Here "number" denotes an integer constant. This grammar is not MI-structured, nor SI-structured. A (strongly) SI-structured grammar is obtained by the following transformation:

```
Expression -> Sum | Term
Sum -> Expression AddOp Term
Term -> Multiplication | Factor
Multiplication -> Term MulOp Factor
Factor -> Constant | SubExp
Constant -> number
SubExp -> '(' Expression ')'
AddOp -> Plus | Minus
MulOp -> Times | Div
Plus -> '+'
Minus -> '-'
Times -> '*'
Div -> '/'
```

As can be seen from this example, transforming this grammar into an SI-structured one tends to increase the size of the grammar considerably. On the other hand the need to find names for the alternative right-hand sides also increases the informative contents of the description. The resulting objects are illustrated in the figure 2.

There is a straight forward algorithm for transforming an arbitrary context-free grammar into an SI-structured one, without changing the generated language (elaborated in the next chapter). The resulting grammar may of course grow; it can be shown that the growth rate is quadriatic. However, it is noted that practical grammars are often close to SI- or MI- structured.

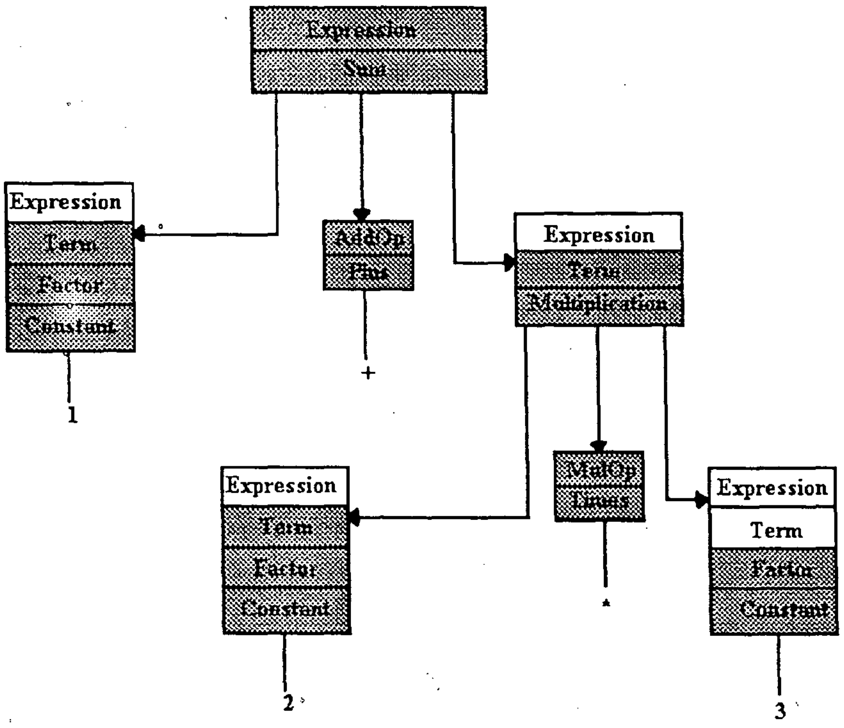


Fig. 2. The objects and their class layers for input string 1+2*3.The components objects are shown by arcs pointing at the static type of the object. The part corresponding to the syntax tree is shaded. We note that the two neighbouring layers in the shaded part correspond to a chain production in the syntactic sense.

As long as we are considering complete sentences of a language, an SI-structured context-free grammar implies that all the superclasses are abstract: a superclass non-terminal cannot appear in a syntx tree without a basic non-terminal below it. However, in some systems it is sensible to consider partial sentences as well (e.g. systems for generating syntax-directed editors); then a missing part may be represented by an instance of a superclass. Hence, in the former approach the syntactic structure is regarded as the virtual property of the superclass. Hence, in the former approach the syntactic structure is regarded as a virtual property of the superclasses, while in the later approach the syntactic structure is an additional property provided by the basic classes.

For better clarification I review two approaches for object-oriented attribute grammars. These approaches are Grosch's Ag system [Gor90], in which object orientation is applied in a relatively simple and clean way and the OOAG [ShK90] which is least object-oriented: the emphasis in this system is in the support of dynamic semantics through a message passing mechanism.

Ag

Ag belongs to a set of compiler construction tools developed at GMD Karlshrue [GrE90]. The purpose of Ag is to generate attribute evaluators that are associated either with concrete or with abstract syntax; in the former case an additional tool is required for parser construction.

Here again a non-terminal is viewed as a class. An attribute grammar is given in the form of nested non-terminal (class) definitions. The nesting implies a class hierarchy: sub-classes are directly within the superclass. A class definition consists of the properties of the class, followed by a list of sub-class definitions enclosed within angled brackets (<>). The properties includes the structural definition or syntactic definition, attribute definitions, and attribution rules. Due to the principle of polymorphism, one may think that there is an implicit chain production A -> B for each superclass-subclass pair (A, B).

Example. *Expressions in Ag*

```
Expr    = [Value: INTEGER]              {Value := 0;}
   < Add    =  Lop: Expr '+' Rop: Expr          {Value := Lop:Value +
Rop:Value;}
         Sub    =  Lop: Expr '-'  Rop: Expr         {Value := Lop:Value -
Rop:Value;}
   Const =  Integer                       {Value := Integer: Value;}
  >.
  Integer    : [Value: INTEGER]
```

Attribute definitions are given in brackets []; here attribute "Value" is associated with all "Expr" structures, with a default rule "Value:=0". The types of the attributes are taken from the target language (Modula-2). The attribute computations rules are given in curly brackets {}; here the rules given for classes "Add", "Sub" and "Const" override the rule given in the supreclass. The structural compositions are given as a sequence of class names, possibly prefixed by a selector (Lop, Rop) allowing unambgious access to the component structures. "Integer" is a terminal class (: instead of =) defined at the end (+ and - are special "terminal" classes not essential here), possessing attribute "Value" as well. It should be noted that this specification assumes that scanning and parsing has been carried out using some other tool (syntax is ambiguous), and that the abstract structure corresponding to this specification has been constructed successfully.

From the user's point of view, the underlying context-free grammar of this example is (the system extracts the productions in a different way depending on whether the specification is for the concrete or for the abstract syntax):

```
Expr = Add | Sub | Const
Add = Expr '+' Expr
Sub = Expr '-' Expr
Const = Integer
```

indicating that "Add", "Sub" and "Const" are subclasses Of "Expr". Since associativity and priorities of operators have been taken care of in the parsing phase the intermediate classes are not necessary.

(end of the example)

In Ag a specification can be split into modules: a module is intended to contain parts that belong logically together. A module is primarily a structuring device for making the specification more manageable and understandable: modules are not processed separately.

The above given example exhibits some essential features of Ag from the object-oriented point of view. First, assuming that no structural (syntactic) specifications are given ot the superclasses, the underlying context-free grammar is strongly SI-structured. The notation guarantees that multiple inheritance is excluded in Ag. Second, within a module, class hierarchies are given in the nested fashion, reflecting the fact that a language is considered to be implemented as a whole: all the sub-classes are assumed to be known beforehand (New sub-classes can be added in a different module, though.). Nested class definitions hamper the reusability if classes, but on the hand specifications become more readable because the class hierarchy is explicitly shown. Third non-terminals can inherit attributes and their computation rules become unnecessary: The whole sequence of nodes connected by chain productions is regarded as a single unit. In other words, descendants can share the attributes of their ancestors.

In a sense an attribute computation rule corresponds to an operation (function) that can be redefined in the descendant classes, introducing thus, dynamic binding. We note that, however, these rules are applied implicitly by the system, and that a rule is activated exactly once for each attribute instance. Therefore the "dynamic binding" is connected to the values of attributes rather than to function calls. For example, we could have

```
Expr    = [Value: INTEGER; X: INTEGER]        {Value := 0; X :=
Value; }
        < . . . as before . . .>
```

The value of X will usually not be 0 but the value determined by the lower class levels.

Grosch also proposes the inheritance of syntactic specifications in the same sprit as the inheritance of attributes: the syntactic specification is "extended" by the sub-class in the sense that the right right-hand side of a sub-class is concatenated with the right-hand side of its superclass. Typical languages rarely allow the direct exploitation of this featurein the concrete syntax, because the variations of a particular language structure do not usually begin in the same way. However, Ag is intended for specifications associated withthe abstract syntax, too. In the abstract, internal representation of a program such structural inheritance is sometimes sensible.

OOAG

The OOAG formalism (object-oriented attribute grammar) proposed in [ShK90] combines attribute grammars with a message passing mechanism borrowed from the object-oriented paradigm. Each node of the syntax tree is treated as an object (implying that every production is regarded as class) communicating with other objects by sending them messages. Since there is no class hierarchy nor inheritance, the object-orientation is relatively weak. On the other hand, the system allows an arbitrary number of named methods for the objects, which is more than the previous systems offer. In OOAG the system guarantees that the attributes have consistent values with regard to the attribute equations by propogating the changes in the tree.The practical feasibility of this somewhat expensive mechanism is still open; in particular, it seems that the mechanism is slow for implementing dynamic semantics (that is, interpretation) of programs. Obviously, the system is primarily intended for implementing other less "semantically" oriented tools in a programming environment.

4. A DISCUSSION ON GRAMMAR TRANSFORMATION

It is clear from the previous chapter that an object-oriented context-free grammar does not reduce the expressive power of the grammar formalism: any context-free language can be described by such a grammar, and any context-free grammar can be transformed into an object-oriented one without changing the language. However, an object-oriented context-free grammar tends to be considerably larger than a corresponding conventional context-free grammar. This drawback might not be seen to be of great significance because the user of those systems is expected to write the object-oriented grammar from scratch, rather than to obtain it from grammar transformation. But in practice it will be sensible to have an automatic transformation tool for the convetional grammar for the language to be specified.

I reproduce below an algorithm [KosMak91] which considers such transformations and also a discussion about their cost in terms of increased grammar size. In general, the results therefore indicate the price of using an object-oriented grammar form.

Let G = (N, T, P, S), be a context-free grammar with the symbols denoting the usual meanings (as mentioned in the earlier chapters). We define a production with a non-terminal *A* on its left-hand side to be an *A-production*. The set of all A-productions is denoted by *prod(A)*. A production A -> α is a chain production if α N. The set of all chain productions with A on the left hand-side is denoted by *chainprod(A)*.

A non-terminal is said to be *mixed* if among A-productions there are both chain productions and non-chain productions and *chain non-terminal* if all A-productions are chain productions, i.e., *prod(A) = chainprod(A).*

A context-free grammar G = (N, T, P, S), is said to be reduced if each symbol X is used in some derivation $S \Rightarrow^{+} aXb \Rightarrow^{*} w$, $w \in T^{*}$. We consider reduced cycle-free context-free grammars only. A context-free grammar is said to be *well-structured* if for each non-terminal A either

1. there is only one A-production
2. all the A-productions are chain productions
3. all the A-productions have only terminal symbols on their right-hand sides

Moreover, we require that each non-terminal appears on the right-hand side of a chain production at most once. It is also necessary that to mention about the size of the transformed grammar, we need a measure for grammar size. A measure by Harrison is *norm* $\| G \| = | G | \log (card(N \cup T))$. Norm counts the number of symbols involved in productions, but only in logarithmic sense. The number of non-terminal symbols is essential in the application we are interested in, but the number of terminal symbols is not changed during the transformations. Hence, the definition of norm is slightly modified as $\| G \| = | G | \log(card(N)$. The context-free grammars are assumed to have at least two non-terminals. The number of non-terminals, card(N), can also be used as a measure of grammar size.

The Algorithm

The algorithm transforming a context-free grammar into a well-structured context-free grammar must pay special attention to chain productions. The form of chain productions is restricted by the requirement that each non-terminal appears on the right-hand side of a chain production.

Evidently, we need a method for eliminating chain productions without effecting the language generated. If A -> B is a chain production to be eliminated, we first define the set *non-chain(A)* = $\{w \in ((N \cup T)^* \setminus N) \mid A \Rightarrow^+ C \Rightarrow w.\ C \in N\}$. The production A -> B can now be replaced by the set {A -> W | w$\in$ non-chain(B)}. We may assume that the chain production whose left-hand side is a mixed non-terminal are more casual than those having a chain non-terminal on the left-hand side.

Let P_C be the set of productions having a chain non-terminal on the left-hand side. If some non-terminal appears on the right-hand side of the productions in P_C more than once, we have to eliminate some productions of P_C.

An ideal situation would be the one in which the sub-classes were changed for as few classes as possible; that is, for as many non-terminals as possible, all productions in prod(A) could be taken to the resulting context-free grammar. However, it is a difficult combinatorial problem to such a set of nonterminals. It is in fact, a NP-complete problem to maximize the cardinality of a set of chain productions such that no non-terminal appears more than once on the right-hand sides and no set prod(A) is split. This problem can be called the CHAIN NON-TERMINALS problem and can be formulated as follows.

> INSTANCE: A context-free grammar G = (N, T, P, S) and an integer k.
>
> PROBLEM: Is it possible to choose a set M of chain non-terminals from N such that the corresponding set of chain productions PM = {A -> B | A$\in$M} has cardinality k and no pair of productions in PM has a common right-hand side?

The following theorem is stated without proof.

Theorem. *CHAIN NON_TERMINALS is NP-complete.*

Although CHAIN NON_TERMINALS is intractable in the worst case, we may assume that in any practical situation it is possible to find the largest set of chain productions to be taken to the resulting context-free grammar or at least find out a heuristic algorithm to perform this task.

In what follows, we withdraw the requirement that, for a chain non-terminal A, all A-productions should be taken to the resulting well-structured context-free grammar. We also note that if A is originally a chain non-terminal and some A-productions are eliminated and some are left unchanged, A becomes a mixed non-terminal.

After eliminating the undesirable chain productions among those having a chain non-terminal in the left hand side, we can still try to find chain productions with a mixed non-terminal on the left-hand side and a right-hand side not yet present in the resulting context-free grammar.

After the chain productions have been handled, we can fix the other requirements for a context-free grammar to be well-structured. Suppose a non-terminal A is not a chain non-terminal and let $\{A \to \alpha_1 \ldots A \to \alpha_n\}$, $N > 1$, be the set of all A-productions which are not chain productions and at least one α_i, $1 \leqslant i \leqslant n$, is not in T^*. We replace these A-productions by the set $\{A \to A_1, A_1 \to \alpha_1, \ldots, A \to A_n, A_n \to \alpha_n\}$ where A_i's are new non-terminals. The whole algirithm is described as follows.

Algorithm WS

Input: A context-free grammar G = (N, T, P, S).

Output: A well-structured context-free grammar G' = (N', T, P', S) such that L(G) = L(G').

Method:

1. Consider the collection {*prod*(A) | A is a chain non-terminal}. Mark the largest possible set of productions from the sub-sets such that no non-terminal appears twice on the right-hand side. Eliminate all other chain productions in *prod*(A). Take the marked chain productions to P'. Update sets *prod*(A).

2. Consider the set P_M of chain productions having a mixed non-terminal on the left-hand side. Mark the largest subset of P_M such that when added to the set choosen in step 3, no pair of of productions have a commom right-hand side. Eliminate all other chain productions in P_M. Take the marked chain productions to P'. Update sets *prod*(A).

3. Take to P' the productions in *prod*(A) if *prod*(A) is a singleton set or if all productions have their right-hand sides in T^*. For all other sets *prod*(A), do the following. If *prod*(a) = {A -> $\alpha_1, \ldots,$ A -> α_n} then take the productions {A -> A_i, A_i -> α_i | $1 \leqslant i \leqslant n$} to P'. The non-terminals A_i are new and are taken to N'. Besides the non-terminals added in this step, N' contains the non-terminals in N.

The correctness of the algorithm follows directly by the definition of well structured context-free grammars. For the increase in grammar size the following theorem is stated.

Theorem. *The algorithm WS can increase the size of the input context-free grammar by a quadriatic factor but not more. Quadratic factor is possible only when chain productions are eliminated.*

To find out exact constant factor in the linear growth of grammar size caused by step 3 of the algorithm WS. Suppose a context-free grammar G is as after steps 1. - 3. of the algorithm; that is, each non-terminal appears at most once on the right-hand sides and if a non-terminal A is on the left-hand side of exactly one chain production, then A is a mixed non-terminal. However, as we are looking for the greatest possible growing factor and we know that chain productions are not replaced in step 3, we may suppose that G has no chain productions at all. If A-productions are replaced in step 3, then at least one of the right-hand sides contains non-terminals. If G has A-productions A -> ε and A -> aB, then the resulting context-free grammar has productions A -> A_1, A_1 -> ε, A -> A_2, A_2 -> aB. The growing factor does not increase if we take more A-productions to G. We may have the correspondings productions for B and so on, but we also need a non-terminal which eventually ends the chain by having terminals on the right-hand side (a single production with ε on the right-hand side is the best choice). This shows that the size of G cannot even increase by factor 2, although we go arbitrarily close to it.

A striking result is obtained if we measure grammar sizes by the cardinality of non-terminal alphabets. As the length of productions is unlimited, there is no upper bound for the number of productions, although we bound the number of non-terminals used. Hence, the number of non-terminals can have an unbounded increase when Algorithm WS is used. We notice that steps 1.-3. have no effect to the number of non-terminals. Naturally the number of productions obtained by using, say 2 non-terminals, the result holds.

5. DISCUSSION

Incorporating object-oriented views in language specification structures is a fairly recent idea and the ultimate motive of developing a compiler in tune with the object-oriented paradigm will require much more effort and planning.

We have discussed the benefits of introducing object-orientation in attribute grammars, but in doing so we can recognize three more problem areas.

First, *conceptual integration*: which object-oriented concepts are sensible in connection with attribute grammars, to what extent can they be applied, and the kinds of problems or conflicts they may cause when interacting with the concepts of language specification structures (attribute grammar). This problem area is particularly emphasized in Ag. We have seen that the concepts of classes and class hierarchies can be introduced in a natural manner by regarding non-terminals as classes, and by deriving the class hierarchies from a structured form of context-free grammars. Inheritance can be applied to attributes and structured specifications. On the other hand, the idea of inherited attributes was found to be in conflict with class inheritance, in particular when multiple class inheritance is allowed. We also have doubts or some confusion on whether multiple inheritance can be introduced in a sensible form in attribute grammars; on whether attribute grammars can be extended so that it would it be a more essential part of the description of semantics, rather than a "factoring-out" technique. We can also think about feasible ways of describing context through class inheritance rather than attribute inheritance.

The second problem area is in the development of more *high-level concepts* for supporting language implementation, and their integration with attribute grammars. We know that the concept of attribute grammar is too primitive and we can think of it as only a basic framework; as the "machine language" of language implementation. For practical development, there must be substantial automation. At present attribute grammars offer automation for the support system (e.g., parallel compilation), rather than for the description of the actual semantics.

Thirdly, there is the problem of describing *dynamic semantics*. We can ask ourselves the questions of whether dynamic semantics can be described using consistent attribute values in an efficient way and whether dynamic semantics, if given in procedural style, can be naturally integrated with a declarative grammar.

We have also seen that the problem of removing multiple inheritance from an object oriented context free grammar in an optimal way is NP-complete. we do not know transformation algorithms that would remove multiple inheritance from an object-oriented program, but also there is no reason why such algorithms could not be developed: they might be useful for translating programs from one object-oriented language (allowing multiple inheritance) to another (requiring single inhertance).

The fact that the size of an object-oriented grammar grow with quadratic factor with respect to the conventional grammar should not be regarded as a drawback as this happens only with unrealistic grammars and in practice the growth seems to be a linear one. It should also be noted that an object-oriented grammar carries more explaining information due to the fact that all the alternative choices must have names. Therefore, object-oriented grammars are easier to read inspite of their larger size. Hence, in fully automatic transformation from a conventional grammar into an object-oriented one is not sensible: the user must at least supply the names of the new subclasses suggested by the transformation algorithm.

BIBLIOGRAPHY

Aho, A. V., R. Sethi and J. D. Ullman [1988]. *Compilers; Principles, Techniques and Tools*, Addison-Wesley, Reading, Mass.

Aksit, M., R. Mostert and B. Haverkort [1990]. "Grammar Inheritance", *Dept. of Computer Science, University of Twente.*

Budd, T. [1991]. *An Introduction to Object-Oriented Programming*, Addison-Wesley, Reading, Mass.

Cardelli, L., and P. Wegner [1985]. "On understanding Types, Data Abstraction and Polymorphism", *ACM Computer Surv.*, 17, 4, pages 471-523.

Fischer, A. E., and F. S. Grodzinsky [1993]. *The Anatomy of Programming Languages*, Prentice-hall, Englewood Cliffs, N.J.

Kastens, U. [1991]. "Attribute Grammars as a Specification Method", *Lecture notes in Computer Science*, Volume 545, pages 16-47, Springer-Verlag.

Koskimies, K. [1991]. "Object-Orientation in Attribute Grammars", *Dept. of Computer Science, University of Tampere,* Report A-!991-1.

Koskimies, K., and E. Makinen [1991]. " On Grammar Transformation Related to Class Hierarchies", *Dept. of Computer Science, University of Tampere*, Report A-1991-2.