Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

M.L. Kersten, F.H. Schippers

A general object-centered database language
a preliminary definition

Computer Science/Department of Algorithmics & Architecture Report CS-R8615 April

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

Godel

A General Object-Centered Database Language

A Preliminary Definition

Martin L. Kersten, Frans H. Schippers
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

1980 Mathematics Subject Classification: 69D42, 69H23, 69K14
1985 CR Categories: D.3.2, H.2.3, I.2.4.
Key Words & Phrases: object-oriented languages, database management, knowledge representation.

Preface

This report describes the programming language *Godel*, intended for the construction of knowledge based applications. *Godel* uses both the object-centered, the rule-oriented, and the procedural programming paradigms. These paradigms are used in an unconventional way, thereby simplifying the maintenance of complex relationships among (static) objects and modelling dynamic behaviour through actor-like objects, called *guardians*.

This report is a working document. It focuses on the syntax and it presents an informal semantic definition. In-depth discussions of topics such as the concurrency philosophy and storage techniques used are presented separately. The language definition given here assumes a teletype-like user interface, which simplifies the language specification and its implementation. A functional prototype has been implemented in C-PROLOG under UNIX BSD4.2.

Acknowledgements

Ideas incorporated in the design of *Godel* have come from many sources. Experience in the design the algorithmic language for interactive information systems, called PLAIN[5] showed the lack of flexibility and self-reference, necessary to design knowledge-based applications. The strongest influences are those of the SMALLTALK language,[2] which showed us that its object-oriented paradigm is far from an object-centered paradigm. In particular, the rigid means to communicate between objects was considered inadequate for a knowledge base environment. The declarative and unification aspects have been influenced by PROLOG.[1] The work closest to ours are languages for actor-systems.[3]

A number of individuals have contributed ideas and criticism throughout its design phase. Many of those have affected the definition of the language reported here. We wish to acknowledge personally; A. Siebes, T. Budd, H. Bal, L. Meertens.

Report CS-R8615
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

1. Introduction

Godel is a programming language designed primarily to support the construction of adaptable information systems and knowledge based applications. Knowledge base applications are typically designed to provide non-experts access to a large set of facts and complex interrelationships among them, to search knowledge using incomplete information, to safeguard a knowledge base content and its users against unwanted situations such as integrity violations, and to aid in the interpretation of the knowledge rules. The following capabilities of *Godel* are of particular interest for these applications.

1. *Data base management*
   The language deals with a database of objects; integrity of the database is guaranteed through a generalized trigger mechanism; object selection is declarative, based on first-order logic; transaction processing primitives are included.
2. *Knowledge base management*
   Both simple facts (objects) and rules to derive facts are stored in the database in a uniform way. Facts can play roles in different interpretation domains concurrently. Both structural inheritance, i.e. using the value of an objects' subcomponent, and behavioural inheritance, i.e. using a value obtained from an objects' controllers, are provided.
3. *Data flow driven computation*
   Operations are triggered when the operational constraints associated with processes become true in a particular state of the database or when an event is recognized. Primitive trigger conditions are the insertion and modification of objects in the database.
4. *Modularity*
   Facts and rules can be logically grouped into knowledge bases through tagging. The level of integration of multiple knowledge bases accessed during a single session can be precisely controlled. The visibility rules for object properties support information hiding.
5. *Input/output*
   Simple I/O statements are provided to communicate with the user at a teletype-like terminal. External formated files, such as those generated by traditional database systems, can be readily included.

Studies in language design for knowledge base applications have identified certain features that provide good support for knowledge engineering and knowledge base design. These features include

1. *Data paradigm*
   The data in a database is not static; rather it evolves over time to meet the changing information requirements of users. A declarative specification of the behavioural properties of data forms a proper basis to cope with database evolution. It allows both data and meta-data to be treated in a uniform fashion.
2. *Object-oriented paradigm*
   Object-oriented computation models provide data independence between applications and the conceptual knowledge base by encapsulating application-oriented behaviour into abstract data types (ADT), which are themselves objects.
3. *Object-centered paradigm*
   The object-centered paradigm takes the object-oriented approach one step further by making class membership a dynamic property of objects. This way evolution of the knowledge base is accommodated without loosing the ability to safeguard its integrity.
4. *Rule-oriented paradigm*
   The rule-oriented paradigm allows for the description and use of procedural knowledge without specifying in advance all allowable control paths.

5 *Polymorphic typing*
The language is basically typeless. Variables need not be declared before use. In essence, types are considered 'first class values' and thus may be the result of expressions. A typing system is nothing more than a restricted symbolic evaluation of the program to reveal processing inconsistencies. A dynamic typing system is easily included using the language primitives provided.

6. *Cooperative problem solving*
Techniques for cooperative problem solving provides the means for distributed knowledge manipulation and forms a basis for contemplating parallel implementation architectures. The integrity of the database is guaranteed by the atomic behaviour of guardians which is implemented through data sharing.

7. *Self-referential*
An essential aspect of knowledge base systems is their ability to explain their behaviour within the same formalism. The computational model and most language features are documented (and implemented) in terms of itself.

8. *Language size*
A limited number of language features simplifies self-reference. It makes implementation feasible.

This report is limited in scope. It focuses on the language syntax and semantics only. The rational which discusses its functional capabilities and its features is given elsewhere. A graphics-oriented development environment, called *Escher*, is anticipated and will include functions such as: a syntax driven editor, a symbolic debugger, a database browser and report facility, and a knowledge base design aid.

2. Summary of the language.

This section presents an informal and intuitive introduction of the features of *Godel* by means of 'a card game'. Seated at the table are three players, called Shorty, Fat Boy, and Sue. A fourth person, the arbiter, shuffles a deck of cards and arrange them in a rectangle of 4 rows and 13 columns. When this is finished the arbiter takes three pieces of paper and writes down a task for person.
The task for Shorty becomes:

> *For every two different cards in the same column exchange them such that the clubs occupy the first row, diamonds the second, hearts the third, and spades the fourth row.*

The task for Fat Boy and Sue becomes:

> *For every card not in the proper column select a card from the proper column (and which is not at the correct column either) and exchange them.*

As soon as the players receive their (private) goal the system is set into motion. All players move until no more actions are possible. The player who moves least recently is the winner.

This trivial game, of course, sorts all the cards by suites and value. The points of interest are that all players (processes) can work in parallel without knowledge of each others' task; the actual sequence of actions is determined by the state of the table (=database); the tasks are described by a high level declarative rule selecting the interested database states and a simple algorithm action; concurrent access is synchronized through an exclusive locking scheme. Below a sketch is given of the *Godel* description for this game.

```
examples

    object card[ x:=1 y:=1 color:=hearts number:=4]
            /* 50 more cards */
    object card[ x:=13 y:=4 color:=diamonds number:=1 /*ace*/]

    function exchange(O,P)
    when O<>undef and P<>undef and P<>O
    [
            V:= O; O:= P; P:= V
    ]

    guardian 'Shorty'
    when C1.x=C2.x and C1<>C2 and C2.color<>C1.color
    [
            when C1.color=clubs and C1.y<>1 [
                    exchange(C1.y,C2.y)
            ]
            when C1.color=diamonds and C1.y<>2 [
                    exchange(C1.y,C2.y)
            ]
            /* the other two cases are similar. */
    ]

    guardian 'Sue'
    when R1.x <> R1.number
    [
            when R2.x = R1.number and R2.number<>R1.number
            [
                    exchange(R1.x,R2.x)
                    exchange(R1.y,R2.y)
    ]]

    guardian 'Fat Boy' := 'Sue'
```

In the programming language *Godel* all information is stored in a database shared by continuously running processes. The static information regarding entities is described in objects, i.e. in the form of a labelled directed graph. The object name is a symbolic reference for use within programs. Each object has a set of attributes which describes its properties. Compound objects can be described through nested attribute definitions or by assigning a new object to an attribute.

Objects can be selected from the database through predicates. The free variables in the predicate are bound with objects in the database such that the predicate becomes true. A set constructor mechanism provides the means to manipulate large collections of objects with a single statement.

The dynamic behaviour of a *Godel* program is described by *guardians*, which describe algorithms to transform the database from one state to another. They become active by observing a particular, declaratively described state of the database. Guardians are objects in their own right. Thus they can be manipulated with the object denotation and modification primitives by a supervisor.

Object components are accessed through the dot (.) operator. Similar to record denotation in conventional high-level languages. The dot-dot (..) operator provides access to substructures of an object without specifying the full attribute path. That is, the dot-dot operator implements structural inheritance. The hat (∧) operator provides access to attributes of guardians which currently govern the object. This way behavioral properties can be described in a modular fashion and dynamically accessed.

Traditional language constructs such as functions, assignment statement, repeat statement, conditional statement and primitive input/output statements are included as well.

The syntax of algorithmic and declarative statements is the prime topic of this report. A more detailed description of the dynamic behaviour of the *Godel* language is given in .[4] For an intuitive understanding of the concepts described here keep the card-game example in mind.

3. Notation, terminology and vocabulary

The vocabulary of the *Godel* interpreter consists of basic symbols classified into letters ([_ a-zA-Z]), digits ([0-9]), brackets, operators, and word symbols. Optional comma and semicolon separators are used at various places to improve readability of the syntax.

syntax

brackets	::= '[' | ']' | '(' | ')' | '{' | '}' .
operator	::= '<' | '>' | '<=' | '>=' | '<>' | '=' | '?' | '+' | '-' | '*' | '/' | ' := ' | ' :+ ' | ' :- ' | '.' | '..' | '∧' | '∧∧'.
wordsymbols	::= **access** | **and** | **div** | **do** | **function** | **guardian** | **if** | **mod** | **not** | **object** | **or** | **read** | **return** | **undo** | **when** | **write.**
comma	::= ',' | /* empty */.
semicolon	::= ';' | /* empty */.

4. Identifiers, built-in objects and comments

Identifiers serve to denote objects, attributes, functions, guardians, and variables. The lexical convention for identifiers is the same as in PROLOG. Identifiers starting with an upper case letter are considered variables. They bind with objects in the database either explicitly, through a modify or definition statement, or implicitly, when used in expressions. The scope of validity of a variable identifier starts with its first occurrence and terminates when the closing bracket of the corresponding lexical scope is reached. The initial value of a variable is the primitive object *undef*.

The lexical conventions for numbers and booleans is identical to those in Pascal. Floating point numbers are not yet supported.

Sequences of characters enclosed by single quote marks are called string literals and are constants of object type *string*. The character set for strings is similar to that of the language C. Thus, the class of allowable characters for string constants exceeds the class of special symbols.

The symbol *undef* represents the unknown object, which is the result of any arithmetic expression in which an unknown value occurs or where an object or an attribute referenced does not exist.

Comments are sequences of characters enclosed by '/*' and '*/'; they may be freely used to make a program easier to read. A comment may be inserted between any two identifiers, literal

constants, special symbols, or word symbols. Nesting of comments is supported.

syntax

string	::= "" { letter | digit | escaped } "" .
escaped	::= '\n' | '\f' | '\' digit digit digit .
number	::= integer | float .

5. Object definitions

The *Godel* database is a set of (named) objects. An object definition adds an object to the database. An object name is either an identifier or a string value. Omission of the name results in a nameless object. In that case, it can only be manipulated indirectly through a selection expression. Multiple objects with the same name can be stored in the database.

The properties of the object are described by attributes. Attributes are objects in their own right. However, they are not directly accessible. Attribute names are only visible in the context of their defining object; like record fields in conventional programming languages. Within the scope of an object definition attribute names hide objects with the same name defined elsewhere. A compound object is obtained when the new attribute is a compound name, i.e. names separated with dots.

An attribute name can be associated with an expression through an assignment statement (Sec. 10.1). If the attribute is described by an assignment statement then its value is determined during object creation. Otherwise its value is the primitive object *undef*. The value can be altered later on by an assignment statement. If the attribute is described by a statement block then this block is executed when the value of the attribute is needed. Then the value is determined by the first explicit return statement encountered or the value of the last statement. A new behaviour is obtained by replacing the definition using an assignment. Such a description can be considered a rule which specifies how a fact is deduced.

Identifiers, numbers, boolean and string literals are considered objects as well. They differ only in the way they are defined; namely implicit by evaluation of an (literal) expression. A new primitive object can be defined by ommitting the attribute list.

Creating copies of an object is provided as a variation on the assignment statement (Sec. 10.1). The access properties (accessrule) are described in Sec 8.4.3.

syntax

object	::= **object** (objectname) attributelist (accessrule) | **object** objectname assignment (accessrule) | **object** objectname (accessrule) .
attributelist	::= '[' { attributedef semicolon } ']'.
attributeref	::= objectname { '.' attributeref } .
attributedef	::= attributeref | attributeref assignment .
objectname	::= identifier | string.

examples

```
object hummingbird [
        isabird                 /* property which tells that it is a bird */
        food := 'honey'         /* hummingbird eats honey */
]
object [                        /* a nameless object */
        father := 'Martin'
        mother := 'Fernande'
        child.son := 'Joost'  /* compound object */
]
object office [
        openinghour := 900
        closinghour := 1700
        /* This expression is evaluated when used in an expression */
        workinghour := [clock.decimal>openinghour and clock.decimal<closinghour]
]
```

5.1. Predefined attributes

As soon as an object is stored in the database it is assigned a set of predefined attributes by the *Godel* interpreter. These primitive attributes can be seen as rules automatically. For example, during the evaluation of an expression the operands are extended with the attribute *value* automatically.

All objects are stored internally as 'balloons' in the database, i.e. a reference to a cell which holds an arbitrary number of references to other 'balloons' and primitive objects. The attribute *self* refers to the internal name of the object instance, i.e. the identity of the 'balloon'.

The *name* of an object is used in programs to access its properties. To continue the metaphor it is a user defined name for the 'balloon'. The *name* can be changed through an assignment in which the source is a string valued expression.

The *value* attribute denotes the current value of an object (or attribute). Note that, the value of an object is the set of used-defined attribute names. The value of an attribute is a set of primitive objects, integers and strings, and names of nested attributes. *Oldvalue* denotes the previously value of the object. That is, the value as it results from a previously committed transaction.

The only predefined attributes for primitive objects are *name* and *value*.

attribute	interpretation
self	internal name of an object or attribute
name	symbolic name of an object or attribute
value	current value of an object or attribute
oldvalue	previous committed value of an object or attribute

examples

```
hummingbird.self          /* denotes the internal representation of the hummingbird */
hummingbird.name          /* denotes the string 'hummingbird' */
hummingbird.value         /* denotes the set {isabird, food} */
hummingbird.food.name     /* denotes the set {'food'} */
hummingbird.food.value    /* denotes the set {'honey}' */
hummingbird.food.oldvalue /* denotes the previous food */
25.value                  /* denotes 25 */
```

6. Function definitions

Functions are parameterized algorithmic descriptions. A function definition consists of a function header, an optional scope, and a statement block. A function is internally represented as a *Godel* object using the same naming convention.

The function header describes the name and the parameter list, i.e. a list of variable names. The parameters are bound with the actual arguments upon function invocation. The actual number of arguments should be equal to those required by the formal definition.

The **when** part of a function, called its scope, is a qualified expression (Sec. 9.1), i.e. a boolean valued expression in which free variables are bound with objects through a selection process. The scope is used to safeguard the accidental use of the function with improper arguments, i.e. it is primarily used to implement a typing scheme for parameters and to select the proper overloaded function.

Assignment within the statement block is restricted to the local variables and the objects referenced by a parameter. The rest of the database can be inspected but can not be altered. Functions used within scope expressions should have no side-effects, i.e. modifications of objects other then local variables are prohibited. A **return** statement terminates the algorithm and it returns the result of the expression. By default the value of the last executable statement in the function body is returned. An ommitted or empty body stands for the body '[true]'.

An overloaded function is a function definition with a non-unique name. If an overloaded function is called then one definition is selected for which the scope expression is true; that function is then evaluated. Overloading built-in operators is allowed. User-defined functions take precedence over the built-in definitions. The overloaded function can be referenced within the function body using the *self* attribute.

syntax

function	::= **function** fcnheader (scope) (body) .
fcnheader	::= fcnname '(' { parameters } ')' .
fcnname	::= identifier | wordsymbols | operator .
parameters	::= variable { comma variable }.
scope	::= **when** qualifiedexpr.
body	::= '[' { functionstmt semicolon } ']'.
functionstmt	::= statement .

examples

```
function isa(Object,Guardian)
when guardian.Guardian <> undef [
        return Guardian.scope(Object)
]

function in(O,Set) [
        return Set.Elm.value=O.value
]

object functionresults [
        uselater; set:= {1,2,3};
        attr1 := in(1,{1,2,3,4})          /* attr1 = true */
        attr2 := [in(1,{1,2,3,4})]        /* attr2 = true when used */
        attr3 := [in(uselater,set)]       /* evaluate when used */
]
```

7. Guardian definition

A guardian is an object which governs the contents of the *Godel* database. A guardian definition consists of a name, an optional scope expression, rules and attribute definitions, and an optional access constraint.

A guardian is represented as a *Godel* object internally; all operations applicable to objects are applicable to guardians as well. A guardian may be assigned attributes and shares the predefined object attributes. The object naming conventions apply.

The database partition governed by a particular guardian is described by a *scope* expression. A scope is a qualified expression (Sec. 9.1), i.e. a boolean valued expression with free variables. All variable bindings such that the scope expression becomes *true*, are subject to control by the guardian. Omission of a scope expression means that all objects in the database are governed.

The guardian rules specify the algorithmic behaviour of the guardian. The rules are preceded by a local scope expression, refining the extent to which the guardian governs the database. The statement block associated with the rule is executed whenever a variable binding is found such that both global and local scope expressions are true. The statement block is considered a database transaction, i.e. an atomic action. The actions of which can be undone with the **undo** statement.

A classical object class can be simulated in *Godel* by tagging the members of a class with the class name. Functions restricted to the members of a class can be defined readily by checking for the appropriate class tag. Methods such as found in Smalltalk can be simulated through guardians watching for message objects. For example, a database of employees can be watched for a negative salary as shown below. If an object is found which violates this rule then a message is displayed.

syntax

guardian	::= **guardian** (objectname) guardianbody (accessrule). | **guardian** objectname assignment (accessrule).
guardianbody	::= (scope) rules .
rules	::= '[' { rule semicolon } ']' .
rule	::= attributedef | **when** qualifiedexpr guardianblock .
guardianblock	::= '[' { guardiansmt semicolon } ']' .
guardianstmt	::= statement .

examples

```
guardian employee_class
when O.employee [

        when O.salary = undef [
                /* salary field does not yet exist */
                O :+ salary
                O.salary := -1
        ]

        when O.salary < 0 [
                write ['What is the salary of ' O.name ' ?\n']
                read O.salary
]       ]

object smith [ employee     /* notifies the guardian */ ]
```

7.1. Guardian behaviour

Guardians are like processes in traditional programming environments. As soon as a guardian is defined, i.e. stored in the database, a process is created. This process will, in due course, execute the statement block for all possible variable bindings provided it satisfies the scope expression. When all bindings have been tried and no qualifying binding has been found then the guardian is put to sleep, awaiting new objects and events, such as changes to the database.

The order in which the variables in expressions are bound to objects in the database is explicitly left undefined. Similar, the evaluation order of an expression is left to the system. This means that the user should be careful not to write (scope) expressions which depend on side-effects of function calls. Moreover, to guarantee that objects are handled once by a guardian a proper tag should be used to represent this state.

Unlike existing programming languages the flow of control is primarily data driven. A local scope expression describes a partial order in which actions should be taken. This means that it is possible to react on specific actions, such as object declarations, to guarantee the integrity of the database.

The examples below illustrate the behaviour of guardians. Assume that the database contains precisely one object. The guardian 'once' is triggered and a single message is generated. This object is invalidated for future printing. Thereafter no binding will be found which satisfies the guardian rules of 'once'.
The second example illustrates an infinitely looping guardian. As soon as the transaction commits the scope of the guardian 'infinite' is satisfied by definition and the process is restarted.

examples

```
guardian once [
        firsttime

        when once.firsttime[
                write 'hello world \ n'
                once:- firsttime
        ]
]

guardian infinite
when O [
        write [O.name, ':hello world \ n']
]

/* as soon as a bird object is found with missing attributes */
/* questions are asked */
guardian [
        when O.isabird and O.name=undef [
                write 'What is the name of the bird ? '
                read O.name

        when O.isabird and O.food=undef [
                write 'What is the food of the bird ? '
                read O.food
]       ]
```

7.2. Predefined guardian attributes

Guardians have also the predefined object attributes as introduced in Section 5.1. The following table lists the attributes specific to guardians. The scope attribute is a reference to a function which represents the scope expression. Functions are also available for local scopes, they imply the use of the global scope function.

An important issue in the definition of guardians is when and under what circumstances they become active and modify the database. After a *Godel* program has been read all guardians are put into dormant mode, unless explicitly put into any of the other modes through the program definition. The dormant mode makes a guardian aware of changes in the database and to react on it, i.e. it becomes active. The sleeping mode turns the guardian into a passive object. The modes can be manipulated by the user. A more detailed presentation of guardian scheduling is given elsewhere.

name	*meaning*
scope	global scope expression
guard.N	scope expression for N-th rule activation
action.N	statement list associated with N-th rule
status	current status of guardian

7.3. Supervisors

The duality of guardians, they are both static and dynamic, makes *Godel* a self-regulating system. For it allows the behaviour of guardians to be statically described by objects. Moreover, a guardian can be defined which selects other guardians and switches their status. This way a scheduler or *supervisor* can be defined.

For example, the following code fragment describes the behaviour of single-thread *Godel* interpreter using the language primitives. When the system starts, this guardian is the sole guardian active. All other guardians are asleep, i.e. are passive objects. This supervisor selects a single guardian, switches its status and waits for its completion. When the action is finished the guardian is put to sleep.

examples

```
object sleeping;    /* primitive object definitions */
object dormant;
object active;

guardian interpreter
when guardian.G
[
        processor:= undef
        when processor=undef and G.status = sleeping [
                processor:= G.name
                G.status:= active
        ]
        when G.name=processor and G.status = dormant [
                processor:= undef
                G.status:= sleeping
        ]
]
```

8. Attribute denotation

An attribute denotation selects attributes of objects for further manipulation. The starting point for the search is the list of object names, i.e. the names introduced by object definitions, or the object referred to by a variable. An attribute expression is a reference to a set of *Godel* objects, i.e. either attributes, objects or guardians, using their symbolic names and the attribute denotation operators dot (.) and hat (∧). An attribute expression is evaluated from left to right. If any of the attribute expressions evaluates to *undef* then the whole expression is replaced by *undef*.

Godel includes two forms of property inheritance; structural property inheritance, which obtains a value of a component of the object, and behavioural property inheritance, which obtains an attribute of one of the guardians responsible for the object.

syntax

attribute	::= attrterm { behaviourprop attribute } .
attrterm	::= attrfactor { structureprop attrterm } .
attrfactor	::= objectname | variable | uniqbinding | setconstructor.
structureprop	::= '.' | '..' .
behaviourprop	::= '∧' | '∧∧' .

examples

```
guardian birds [
        when Somebird.tobeprinted [
                write ['name=', Somebird.name, '\n']
                write ['locomotion=', Somebird.locomotion, '\n']
                Somebird :- tobeprinted  /* print it once */
]       ]

/* projection */
employee.{name,sal}                 /* results in two internal object names */
employee.{V:V.value<>undef}         /* project on all non-null attributes */
```

8.1. Structural property inheritance

Structural inheritance makes attributes of compound objects accessible. The dot operator ('.') looks for attributes directly linked with the object. The dot-dot operator ('..') searches all components of the object for the named attribute. If the attribute does not exist then *undef* is returned.

For example, the predefined object *today* is defined as a compound object *date* with attributes *year*, *month*, and *day*. One may omit the *date* attribute when referring to any of its components using the dot-dot operator.

examples

```
object today [
        date := object [
                /* definition of an compound attribute */
                year  :=  1985
                day   :=  22
                month :=  'November'
]       ]
```

8.2. Behavioural property inheritance

The attributes (and rules) of the guardian are called behavioural properties, because guardians govern objects. The behavioral property operator ('∧') searches a guardian controlling an object, i.e. the scope expression is true with the object being bound. The hat-hat operator ('∧∧') inspects the guardians recursively (supervisors!). The extended example below illustrates the inheritance properties.

examples

```
/* declare the database */
object p [ c := 25; d := 'x'; b :+ 29]
object o [ a := 30; a := 31; b :+ p]
guardian g [
        e :=  'e'
        f := object [ g := 'g'; h := 'h']
]
```

construct	value	construct	value	construct	value	construct	value
o.a	{30 31}	o..a	{30 31}	o∧G.a	*undef*	o∧∧G.a	*undef*
o.b	p	o..b	{p 29}	o∧∧G.b	*undef*	o∧∧G.b	*undef*
o.e	*undef*	o..e	*undef*	o∧G.e	'e'	o∧∧G.e	'e'
p.g	*undef*	p..g	*undef*	o∧G.g	*undef*	o∧G..g	'g'
o.b.c	25	o..c	25	o∧f.h	'h'	o.b∧∧h	'h'

8.3. Property exception handling

One of the difficulties in the description of properties is that their always seems to exist exceptions to a general case. For example, all birds fly except penguins. This can be handled in *Godel* in an effective way through operator overloading and guardians as follows.

For example, the dot operator can be overloaded to handle the default. If the locomotion of a bird is requested and it has not been defined as a structural property then the function returns the default, i.e. the bird flies.

examples

```
function .(O, locomotion)
when O.isabird and not O.locomotion
[
        return 'flying'
]
```

Note that the dot operator used within the scope is automatically interpreted as a different definition; that is its default meaning.

Another approach would be to store the default value in the birds guardian and consider the locomotion a behavioural property with deferred evaluation.

examples

```
/* this example shows use of overloaded names */

function locomotion(O)
when O.isabird    /* parameter typing */
[
        if [ O.locomation=undef    [return 'flying']
                true                    [ return O.locomation]
                ]
]

guardian birds
when O.isabird
[
        locomotion := [ locomotion(O) ]    /* delayed evaluation */
]

object hawk []
object penguin
[
        locomotion:= 'walking'
]

write hawk∧locomotion     /* prints 'flying' */
write hawk.locomotion     /* prints 'undef' */
write penguin∧locomotion  /* prints 'walking' */
write penguin.locomotion  /* prints 'walking' */
```

8.4. Visibility of objects

The construction of safe *Godel* programs requires facilities for information hiding. In principle, all attributes are visible to outer layers. That is, the '..' operator has access to all components of an object. Similar '∧' and '∧∧' inspect all guardians for the definition of the required behavioural property.

Selective dynamic sharing of objects is described by an **access** constraint. The constraint is written as a qualified expression and specifies the guardians (objects) from which it can be accessed directly using the dot operator.

Visibility of attributes can be restricted to the object in which they are defined using the construct **access** *name*. The result is that the corresponding object (attribute) can only be accessed in the lexical context in which it is defined.

syntax

accessrule ::= **access** qualifiedexpr .

examples

```
guardian random [
        seed := object [ value := 0 ] access random
        /* explicit definition of value attribute */
        value := [ seed := (seed * 125) mod 8191 ]
]

object notebook
[
        department := 'Sales'
        name := 'Johny Goodlook'
        address := 'Amsterdam'
        owner := 'Jones Gerard'
]
access G.owner = notebook.owner
```

9. Expressions

Expressions are rules of computation for obtaining values of objects and generating new values by application of operators. Expressions consist of operands, variables, object references, and literals. The rules of composition specify operator precedence.

A factor denoting an attribute is automatically extended with the *value* attribute (unless predefined attributes have been indicated by the programmer already) to obtain its value. The result of extending an operand with the *value* attribute is a set of object references (including primitive objects). If an operand is a reference to a statement block then this block is executed to obtain its value. The result of an arithmetic expression is *undef* if any of the operands is *undef*. An exists test is included to test for the existence of an object. A boolean term consisting of an attribute only is interpreted as an inequality test for *undef*.

In expressions mixing integer and floats evaluation results in a float. Mixing strings and identifiers in expressions results in a string. Integers and floats used in a string expression are coerced to string as well. Identifiers are coerced to strings and vice versa for object selection.

The pattern matching operator (?) tests a string to determine whether it conforms to a pattern. The pattern is specified as a string object. The pattern syntax is similar to that used for search expressions in the editors ex/vi.

syntax

expressionlist	::= expression {comma expressionlist } .
expression	::= conjunction (**or** expression).
conjunction	::= negation (**and** conjunction).
negation	::= (**not**) comparison | (**not**) exists .
exists	::= attribute | attribute <> *undef* | attribute = *undef.*
comparison	::= sum (compop sum) .
compop	::= '<' | '>' | '<=' | '>=' | '<>' | '=' | '?' .
sum	::= (sign) term (addop sum).
sign	::= '+' | '-'.
addop	::= '+' | '-' | '+ +'.
term	::= factor (mulop term).
mulop	::= '*' | '/' | **mod** | **div**.
factor	::= constant | attribute | uniqbinding | setconstructor | functioncall.
functioncall	::= objectname '(' expressionlist)'.
constant	::= number | string .

9.1. Variable definitions

If an expression contains free variables, i.e. newly defined variables, then a unification process is started which binds these variables with objects in the database. All free variables are replaced by object references such that the arithmetic expression in which they are defined do not evaluate to *undef*. The order in which variables are bound with objects in the database is explicitly left undefined. The first such binding fixates the variables for the rest of their scope of definition. If no such binding can be established then the expression becomes *undef*.

A variable can also be bound to an object in the database using an *unique* binding clause. The unique clause consists of a result expression followed by a boolean expression. The boolean expression specifies the scope of variable binding. That is, if precisely one variable binding exists such that the boolean expression becomes true then the result of the construction is the object derived from the result expression. The value *undef* is returned when zero or more then one qualifying variable binding exists.

The set constructor collects all objects specified by the result expression (removing duplicates) for which the boolean expression evaluates to true. The set constructor results in a nameless object in which the attributes represent the set elements. The empty set ({}) and enumerated set are supported by this construct as well.

syntax

qualifiedexpr	::= expression .
uniqbinding	::= '(' expression (':' expression) ')' .
setconstructor	::= '{' (expressionlist) (':' expression) '}'.

examples

```
guardian x                                  /* scope of x  */
when O.isabird<>undef                       /* scope of O  */
[
        when O.food= 'honey'
        [
        write {V:V.name=O.name}             /* scope of V */
        ]                                   /* end scope V */
]                                           /* end scope O */
                                            /* end scope x */
```

B=hawk **and** B.isabird /* is true when B can be bound to a hawk */

{ B : B.isabird } /* set of all birds */

(F.son : F.father = 'Martin') /* references the object 'Joost' */

{ **object**[Z,Y,X]: isa(Z,zoo) **and** isa(Y,zoo) **and** isa(X,bird) **and** Z<>Y **and** X.zoo=Z **and** X.zoo <> Y}

10. Statements

Statements define algorithmic actions to be performed. The object, function, and guardian constructs have been introduced already. A prototypical use of an object statement is the creation of an object whenever a particular message is located in the database. They may be used as statements which result in a deferred evaluation. The modify statements replaces or extends the attributes of an object with new values by expression evaluation. The conditional and generator statements provide flow of control. The activate statement causes the execution of the function or awakening of a guardian. The wait statement causes the sequential execution to be temporarily suspended. The input/output statements are used for communication with the environment.

An expression denoting a function call causes the execution of the designated function. Upon activation the formal function parameters are bound to the actual parameters. The sequential execution in which the function call appears is suspended until the function return value becomes available. The return value is ignored unless the function appears as part of an expression.

A guardian call, an expression which denotes a guardian, allows a user to activate a guardian. The effect is that the guardian will run concurrently with the guardian awakening it. Note that the called guardian effectively starts working after the statement block in which it is used is committed. Thus, waiting for the effects of another guardian should be modelled with an intermediate state in the guardian and a separate rule to continue processing.

The last statement in a statement block behaves as an implicit return expression.

syntax

statement	::= object | function | guardian | modifystmt | conditionalstmt | generatorstmt | functionseq | waitstmt | inputstmt | outputstmt | expression | **undo** | **return** expression .
block	::= statement | stmtblock .
stmtblock	::= '[' { block semicolon } ']' .

10.1. Assignment statements

An assignment statement indicates that the value of an object should be replaced (or extended) by a new value obtained by evaluating an expression. The left-hand side of an assignment statement can not be a primitive object. The assignment statement comes in three forms; the *becomes* (:=), the *insert* (:+), and the *delete* (:-).

The assignment operator (:=) assigns a copy of the object obtained from evaluating the expression. If the expression is a reference to an object in the database then recursively all attributes are copied and assigned to the destination object. The attributes names are introduced

automatically.

If the source is a guardian definition then its scope, attributes and rules are copied. If the destination is a guardian then after transaction commit the new guardian becomes active. This way multiple processes are introduced.

If the source is a function definition then its scope and statement block are copied. Note, that this may lead to an overloaded and ambiguous function.

The insert (:+) and delete (:-) operators provides a means to manipulate the set of object references associated with the object at the left hand sight. The insert operator adds a reference to the object selected through evaluating the expression. A delete operator removes all links to objects specified by the expression. These operators make it possible to manipulate both the set of attributes associated with an object as the value set associated with a single attribute.

syntax

```
modifystmt      ::= attribute assignment .
constructor     ::= expression | object | function | guardian | stmtblock .
assignment      ::= modop constructor .
modop           ::= ':=' | ':+' | ':-'.
```

examples

```
object zoos [
        zoo
        self :+ 'The zoo Artis'
        self :+ 'Berlin zoo'
]
zoos :+ 'Antwerpen zoo'
zoos :- {Z: isa(Z,zoo) and Z.location='Amsterdam'}
/* to change all references to Amsterdam */
{A: O..A='Amsterdam} := 'Mokum'
```

10.2. Conditional statements

The conditional statement provides a general choice mechanism for sequential actions. In this statement the qualified expressions are inspected one at time until one is found which evaluates to true (possibly after variable binding). Then the associated block is executed. Ommission of the qualified expression is interpreted as true and therefore should only be used for the last block. That is, to specify a default.

The scope of variables defined within the qualified expression preceding a block terminates with the closing of the corresponding qualified block. The value of a conditional statement is determined by the last statement executed.

syntax

```
conditionalstmt::= if qualifiedblock
                 | if '[' qualifiedblock { qualifiedblock} ']' .
qualifiedblock  ::= (qualifiedexpr) block.
```

10.3. Generator statements

A generator statement causes the execution of a statement block for as long as any of the qualified expressions is true. This statement simulates a traditional for-statement and repeat-statement. In a generator statement all qualified expressions are repeatedly inspected. If an expression evaluates to true then the associated block is executed where after the whole statement is restarted. The generator statement is terminated when all qualified expressions evaluate to false.

The scope of variables defined within the qualified expression preceding the block terminates with the closing of the corresponding qualified block. The value of a generator statement is determined by the last statement actually executed.

syntax

generatorstmt ::= **do** qualifiedblock
| **do** '[' qualifiedblock { qualifiedblock } ']' .

examples

```
/* the following statements generates integers from 1 to 100 */
O := 1
do O<100 [
                write O.value
                O:=O+1
        ]
do Somebird.tobeprinted [
                write Somebird.name
                Somebird:- tobeprinted
        ]
do O [write O.name]   /* repeatedly writes some object name */
write {O:O}.value.name      /* writes name of all objects */
```

10.4. Function sequence statement

The function sequence statement provides a shorthand for repeatedly calling a function with one argument, which is obtained from the expression list, in the order specified. This construct is the basis for implementing functions such as the read and write statements discussed below.

syntax

functionseq ::= objectname '[' { constructor comma } ']' .

examples

```
printtuple [ {O:O.isabird}.value ]
        /* is equivalent to */
do O.isabird and not O.tobeprinted [ O:+ tobeprinted ]
        /* which marks the objects to be printed */
do O.tobeprinted [ printtuple(O)  O:-tobeprinted]
```

10.5. Wait statement

The **wait** statement blocks further sequential processing until the qualifying expression becomes true. The variables introduced in the expression remain bound until the end of the lexical block in which the wait statement is used. This statement is primarily used for the construction of supervisors and when waiting for a timer to go off.

syntax

waitstmt ::= **wait** qualifiedexpr.

examples

```
wait birds.dormant /* waits until the guardian is put to sleep */
wait isa(V,message) /* waits for the next message */
Currtime := clock
wait clock >Currtime+10  /* wait at least 10 clock ticks */
```

10.6. Input/output statements

The basis for legible input and output is established in *Godel* through **read** and **write** statements. The default file used for input/output is the users' terminal which is designated by built-in objects, called *tty_input* and *tty_output*. Other files can be accessed by redefining the built-in objects *tty_input* and *tty_output*.

The **write** statement prints a list of values. The **read** statement reads values from the terminal. The type of the object to be read is determined by the old value of the object. The default for reading an attribute is a string terminated with a newline.

syntax

inputstmt	::= **read** inputlist .
outputstmt	::= **write** outputlist.
inputlist	::= attribute | '[' { attribute comma } ']'.
outputlist	::= expression | '[' expressionlist ']' .

examples

```
function read(Somebird)
when Somebird.isabird
[
        Somebird := isabird
        Somebird :+ food
        Somebird :+ locomotion
        write 'what is the name ? '
        read Somebird.name
]
```

11. Godel programs and sessions

A *Godel* program is a text file containing a series of object, function, and guardian definitions. In the prototype implementation this file should be created with one of the editors of the host system. A *Godel* session is started by calling the interpreter with the input file. The program is read completely before any of the guardians becomes active. The order in which statements are activated is explicitly left undefined. Any synchronization should be encoded in the application.

syntax

godelprogram	::= { block semicolon } .

References

1. Clocksin and Mellish, *Programming in Prolog,* Springer Verlag (1981).
2. Goldberg, A. and Robson, D., *Smalltalk-80 The language and its implementation,* Addison-Wesley (193).
3. Hewitt, C.E., "Viewing control structures as patterns of passing messages," *Artif. Intell.* 8(3) pp. 323-364 (June 1977).
4. Kersten, M.L. and Schippers, F.H., *The Design of the Godel Interpreter,* In preparation
5. Wasserman, A.I., Sherertz, D.D., Kersten, M.L., Riet, R.P. van de, and Dippe, M. , "Revised Report on the Programming Language PLAIN," *SIGPLAN Notices* , (May 1981).

Appendix A: BNF of *Godel*

syntax

brackets	::= '[' | ']' | '(' | ')' | '{' | '}' .
operator	::= '<' | '>' | '<=' | '>=' | '<>' | '=' | '?' | '+' | '-' | '*' | '/' | ' := ' | ' :+ ' | ' :- ' | '.' | '..' | '^' | '^^'.
wordsymbols	::= **access** | **and** | **div** | **do** | **function** | **guardian** | **if** | **mod** | **not** | **object** | **or** | **read** | **return** | **undo** | **when** | **write.**
comma	::= ',' | /* empty */.
semicolon	::= ';' | /* empty */.
string	::= "" { letter | digit | escaped } "" .
escaped	::= '\ n' | '\ f' | '\ ' digit digit digit .
number	::= integer | float .
object	::= **object** (objectname) attributelist (accessrule) | **object** objectname assignment (accessrule) | **object** objectname (accessrule) .
attributelist	::= '[' { attributedef semicolon } ']'.
attributeref	::= objectname { '.' attributeref } .
attributedef	::= attributeref | attributeref assignment .
objectname	::= identifier | string.
function	::= **function** fcnheader (scope) (body) .
fcnheader	::= fcnname '(' { parameters } ')' .
fcnname	::= identifier | wordsymbols | operator .
parameters	::= variable { comma variable }.
scope	::= **when** qualifiedexpr.
body	::= '[' { functionstmt semicolon } ']'.
functionstmt	::= statement .
guardian	::= **guardian** (objectname) guardianbody (accessrule). | **guardian** objectname assignment (accessrule).
guardianbody	::= (scope) rules .
rules	::= '[' { rule semicolon } ']' .
rule	::= attributedef | **when** qualifiedexpr guardianblock .
guardianblock	::= '[' { guardiansmt semicolon } ']' .
guardianstmt	::= statement .
attribute	::= attrterm { behaviourprop attribute } .
attrterm	::= attrfactor { structureprop attrterm } .
attrfactor	::= objectname | variable | uniqbinding | setconstructor.
structureprop	::= '.' | '..' .
behaviourprop	::= '^' | '^^' .
accessrule	::= **access** qualifiedexpr .
expressionlist	::= expression {comma expressionlist } .

Appendix A: BNF of *Godel*

```
expression      ::= conjunction (or expression).
conjunction     ::= negation (and conjunction).
negation        ::= (not) comparison | (not) exists .
exists          ::= attribute | attribute <> undef | attribute = undef.
comparison      ::= sum (compop sum ) .
compop          ::= '<' | '>' | '<=' | '>=' | '<>' | '=' | '?' .
sum             ::= (sign) term (addop sum).
sign            ::= '+' | '-'.
addop           ::= '+' | '-' | '+ +'.
term            ::= factor (mulop term).
mulop           ::= '*' | '/' | mod | div.
factor          ::= constant | attribute | uniqbinding | setconstructor | functioncall.
functioncall    ::= objectname '(' expressionlist )'.
constant        ::= number | string .
qualifiedexpr   ::= expression .
uniqbinding     ::= '(' expression ( ':' expression ) ')' .
setconstructor  ::= '{' ( expressionlist ) ( ':' expression ) '}'.
statement       ::= object | function | guardian | modifystmt | conditionalstmt
                  | generatorstmt | functionseq | waitstmt | inputstmt | outputstmt
                  | expression | undo | return expression .
block           ::= statement | stmtblock .
stmtblock       ::= '[' { block semicolon } ']' .
modifystmt      ::= attribute assignment .
constructor     ::= expression | object | function | guardian | stmtblock .
assignment      ::= modop constructor .
modop           ::= ':=' | ':+' | ':-'.
conditionalstmt::= if qualifiedblock
                  | if '[' qualifiedblock { qualifiedblock} ']' .
qualifiedblock  ::= (qualifiedexpr) block.
generatorstmt   ::= do qualifiedblock
                  | do '[' qualifiedblock { qualifiedblock } ']' .
functionseq     ::= objectname '[' { constructor comma } ']' .
waitstmt        ::= wait qualifiedexpr.
inputstmt       ::= read inputlist .
outputstmt      ::= write outputlist.
inputlist       ::= attribute | '[' { attribute comma } ']'.
outputlist      ::= expression | '[' expressionlist ']' .
godelprogram    ::= { block semicolon } .
```