D^b

A Novel Database Language

Final Year Project
BEng Software Engineering

Peter Nicol 2002

Supervisor: Dr. Chris Harrison
Moderator: Dr. Alessandro Artale

Abstract

The relational model of data was first presented in 1970 by E.F. Codd and has since become the standard basis for the storage and retrieval of data. Recent work on the relational model has focused on adding object-oriented concepts such as inheritance, the aim of which is to enrich the relational model with the benefits of object-oriented techniques. The standard language of relational databases, SQL, has evolved in such a manner that it now contains many logical mistakes. The authors of [Date & Darwen, 2000] propose a new language model ('D'), with which to build a relational language which avoids such pitfalls and adds type inheritance to the relational model.

A language (D^b) is specified which attempts to conform to the 'D' model, including a type system which features "specialisation by constraint". A sample implementation of the language is produced which compiles a subset of D^b and targets the .NET platform. The implementation features rollback transactions and data persistence through basic XML files.

Acknowledgements

I would like to thank my project supervisor, Chris Harrison, for his help and encouragement during the past seven months.

Contents

List of Figures

1. INTRODUCTION

Relational databases have become the most popular way for data to be stored. Almost all the common commercial database packages use the relational model and its standard query language, SQL. However SQL contains a number of logical mistakes, as well as being rather poorly designed. The authors of [Date & Darwen, 2000] propose a new language which can be succeed SQL, following the relational model much more closely and avoiding the errors present in SQL. This language has not been specified: however a model for that language, 'D', is specified including a novel type system.

This project entails the development of a language which fits the 'D' language model. This language must be well-designed, and support the notion of operators, transactions and databases. The language must also feature the type system proposed by the 'D' model, which includes "specialisation by constraint" of values.

The background introduces the concepts behind the relational model and SQL, and gives a brief overview of the 'D' language model and its type system. Microsoft's .NET Framework is also examined in view of its suitability as a target platform for such a language.

The nature of the work falls into two main steps. The first real step is the development of the language. The language requirements are largely taken from analysis of the description of the 'D' model, and are stated in Chapter 3. The language is then designed according to several language design principles, in order to arrive at the grammar, which is presented in EBNF. Several type constraints are placed over language constructs, in order that type-safety be preserved.

The next step is the realisation of the language. This takes the form of a compiler which creates output compatible with the Microsoft .NET platform. The design and implementation of the compiler are described in Chapter 4, using UML diagrams to communicate the compiler's conceptual design. The resultant .NET classes are also described, as well as the techniques used to implement transactions and allow databases to be persistent.

An evaluation of the language, the implementation and the project methods is given in Chapter 5, identifying aspects of the current language and implementation which could be improved.

Conclusions which have been arrived at during the course of the project are described in Chapter 6, including a small analysis of the type system proposed by the 'D' model. Chapter 7 contains further work, listing the many aspects of this project which could be expanded on in the future.

The appendices contain an LL1 description of the grammar of D^{b}, and a description and samples of the methods used to test the implementation.

2. BACKGROUND

Relational Model of Data

Data calculated by a computer program is volatile, in that it only exists for the lifetime of the program which creates it. For some data this may be sufficient, however for large classes of application it can be highly beneficial to allow data to persist, i.e. exist after program termination. A database system is a computer program with the sole purpose of facilitating storage and retrieval of persistent data.

Prior to the relational model, two main models of database storage existed: the hierarchical and the network model. The hierarchical model allowed data to be structured as a tree, with the most complex data type as the root, being recursively refined into smaller structures until the simple data types at the leaves. The network model enforced fewer restrictions on the connections between data, even allowing self-reference. Programs which used such databases were very tightly coupled to the internal structure of the database, traversing over network nodes or through the hierarchy. Changing the data structure in the database required similar changes to the applications.

The relational model was first presented in [Codd, 1970] as a fundamentally new way of storing data. By extending mathematical set theory, the relational model allowed data to be stored in terms of 'relations'. A relation consists of a set of attributes and a set of tuples. Each attribute consists of a name (which must be unique to the relation attribute set) and a type. A tuple consists of a set of values which correspond to the set of attributes of the relation. The power of the relational model lies in the relational operators, which allows data to be combined and manipulated without the need for the application to know about the internal structure of the data.

The advantages of the relational database over the hierarchical and network models proved sufficiently convincing that almost all commercial database systems now implement some form of relational model. Almost all of these support a variant of Structured Query Language, or SQL.

SQL

SQL functions as a data sub-language, designed to be utilised from more powerful languages and facilitate interaction with a database system. It is the primary language for relational databases, and is both an ANSI [URL 1] and ISO [URL 2] standard.

SQL has the interesting property of being the standard query language for relational databases, without fully implementing the relational model. In several areas SQL appears non-relational, most obviously by implementing bag semantics, but more importantly by assigning significance to the order of attributes within relations. Entirely aside from SQL's relational implementation, it also suffers from a lack of adherence to basic language design principles, which can lead to confusion for those initially learning the language.

'D' Model

The authors of [Date & Darwen, 2000] propose that SQL be replaced by a new language, which is based in a restatement of the relational model. The new language would implement the true relational model, be well-designed and generally learn from the mistakes which are now a 'feature' of SQL. While this new language is not defined, a model which the language must adhere to, 'D', is presented as a series of prescriptions and proscriptions.

The 'D' model also enriches the basic relational model with a new form of type inheritance. This form of inheritance is radically different from existing models, embracing the concept of "specialisation by constraint", whereby values can be considered to be of multiple types.

The ‘D’ Type System

Languages generally support some form of type system. A type system seeks to assign a type to each piece of data, in order to restrict actions involving that data to well-defined operations. Subtyping is the mechanism by which instances of a type are deemed suitable to be substituted for instances of another type. Inheritance is a mechanism which allows a type to be defined incrementally in terms of other types.

The ‘D’ type system makes a distinction between values and variables. Values are immutable (constant) whilst variables can be assigned different values over time. Types define a set of values, and subtypes define subsets of values. A value which is a member of a subtype can be freely substituted for a value of a supertype. Variables however have no such subtyping relationship. Operations are not contained within types, but are defined externally, and feature multiple dispatch, whereby the operation to be invoked depends on the exact type of all value arguments. ‘D’ specifies that read-only arguments are matched by the most-specific type of the argument, while update arguments must be a precise match for the parameter type. The exact form of inheritance supported by the ‘D’ model will be subject to investigation.

Microsoft .NET Framework

The .NET Framework is Microsoft’s vision of a uniform software platform. It consists of two major components: the common language runtime and the common class library. Provided suitable compilers exist, programs can be written that will be able to access existing .NET components, regardless of the language those classes were written in. This is possible through the advanced code management of the common language runtime (CLR), which acts similarly to a virtual machine.

A .NET assembly contains types and/or methods, and can function as a library resource for other components, and optionally as a stand-alone application. Executable components of an assembly are stored in Common Intermediate Language (CIL), which is a hardware-independent set of instructions. When the assembly is referenced by an executing application, the CLR will verify the executable instructions in the assembly to ensure that the assembly is type safe. If the assembly is unverifiable (and ‘trust’ levels are not suitably set), the assembly will be blocked from executing. Otherwise the CIL which defines the assembly can be converted into native processor instructions by a ‘just in time’ compiler. This approach ensures the type safety and security of applications.

The ‘D’ model states that it should be possible for the language to express full applications as well as database queries. The ability of .NET assemblies to be used as either a stand-alone application or as a library for other components makes the platform well suited to such a language. The ability of other programs to use .NET components regardless of the language they were written in makes targeting the .NET platform an obvious choice for the implementation of D^b.

General Requirements

The basis of this project is to create a novel database language. The language should serve as a suitable example of the ‘D’ model and its type system, while adhering to good language design principles. A sample compiler should implement the main components of the language. The implementation must target the .NET platform, with the generated assemblies capable of forming part of a larger software product.

This project will encounter many difficulties. Designing a programming language is a difficult task in itself, as no exact process exists which can be followed to create a ‘good’ language. The ‘D’ model describes languages which have several related concepts, all of which must be supported. Creating an implementation will also present a technical challenge, as the ‘D’ type system is rather unique, and shall have to be implemented in a non-compatible type system. Data persistence must also be implemented in some form.

The project requires investigation into a series of different areas: language design principles, type systems, the relational database model, compiler theory and also the .NET Framework.

2 DEVELOPING D^b

In [Date & Darwen, 2000], the authors present their model for future database languages, named “D”. “D” is firmly based in the relational model, but also supports user-defined types and a form of type inheritance. The model is presented as a series of prescriptions and proscriptions, which any language which is said to conform to the model must satisfy. As D^b is to be an implementation of “D”, the book acts as a textbook for the project and most requirements can be taken from there. Further requirements are added in order to restrict ambiguity, and to identify specific language design principles which are to be taken into account.

The language is then designed, taking the requirements into account and based on the language design principles identified. This involves primarily specifying the grammar and the type rules associated with each construct or statement.

2.1 Language Requirements

Values

The language must allow values of any well-defined type to be specified. To facilitate strong type-checking, it must be possible for an implementation to statically associate a type to every value. The language must treat values as immutable, and not provide any means by which a value may be altered. All values of type *t* are substitutable for values of any type which is a supertype of *t*.

Variables

The language must facilitate the declaration of variables of any well-defined type. Variables can be both assigned to and substituted for values, in which case the value of a variable is the last value assigned to it. The language must not allow a variable to hold a value which is a supertype of the associated type of the variable. All variables must be declared before they are used and assigned to before they are used in place of a value.

General Type Rules

A type defines a set of values. A type with a supertype defines a set of values which is a subset of the set of values of the supertype. All types are supertypes and subtypes of themselves. Types are distinct if the intersection of their value sets evaluates to the empty set. The language must not allow constructs which act to *coerce* a value from one distinct type to another. The language must ensure that the creation of a value *v* of type *t* for which any constraint over *t* evaluates to false is an error. Facilities to create values of a supertype from a value of a subtype and test if a value is of a subtype must be supplied. The language will also allow the explicit conversion of a value of a supertype to a value of its subtype, at the risk of an error if the value is not of the correct type.

Built-In Types

The language must supply a set of pre-defined atomic types, and operators for the manipulation of values of such types. Built-in types will be treated as simple types which are implicitly defined, and must include:

- A **boolean** type, for modelling the truth values **true** and **false**. Logical operators $\wedge$, $\vee$, $\neg$ ad the equality operation must all be provided over boolean values.
- An **integer** type, which models whole numbers, positive and negative. Addition, subtraction, multiplication and division operators must be provided for integer values, each of which returns an integer value. Operators for comparing values of type **integer** will also be provided, with at least ‘greater than’, ‘less than’ and equality tests being supported, each of which returns a value of the type **boolean**.
- A **real** type, which models the mathematical set of real numbers. Addition, subtraction, multiplication and division operators must be provided for **real** values, each of which returns a **real** value. Operators for comparing values of type **real** will also be provided, with at least ‘greater than’, ‘less than’ and equality tests being supported, each of which returns a value of the type **boolean**.

- A **character** type, which models a Unicode character. The equality operator must be provided over two values of type **character**.
- A **string** type, used to model a list of values of type **character**. Operators to retrieve and set the value of each element in a given **string** value will be provided. An operator which returns the length of a **string** as an **integer** must also be provided, as must an operator which concatenates two **string** values together.

Simple Types

The language must allow the creation of new types which have no component attributes. Such types can be considered base types (implemented in terms of another simple type) or a subtype of one or more existing simple types.

Complex Types

The language must allow the creation of types which consist of one or more components. Each complex type must have an associated identifier which uniquely identifies the type. Each component of the type will have an associated well-defined type and an identifier which uniquely identifies the particular component. The language must allow for the actual implementation of the type to be supplied by an external software component. A complex type may be defined as a subtype of zero or more well-defined complex types.

The language will supply operators which access a particular component value of a complex value. The equality operator must be well-defined for all complex types, such that values of a complex type are equal if and only if all the values of corresponding components are equal.

Explicit Subtyping

The subtyping relationship is explicit in the case of simple and complex types, and will be specified by the user. If a user-defined type has one supertype, a constraint must be specified such that value set of the subtype is the set of values of the supertype for which the constraint evaluates to true. If the type is defined as a subtype of two or more supertypes, the value set is given by the n-ary intersection of all supertype value sets. All supertypes of a type must be non-distinct.

All pairs of non-distinct types which share a common supertype must have a common subtype.

Relation Headers

The language must allow the notion of relational headers. A relational header is a set of attributes. An attribute is the combination of an identifier and an associated type. An attribute's identifier must uniquely identify it within the header.

One header x conforms to another y if all attributes identifiers in x are present in y, and the type of each attribute in x is a subtype of the corresponding attribute type in y.

(To ease the notation of relation and tuple valued operators, TUPLE_x and RELATION_x will be used to denote respective types which are of relational header x (i.e. have the set of attributes x))

Tuple Types

The language must allow the specification of tuple types. Tuple types conform to a specific relational header. A value of a tuple type consists of the set of attribute values, such that each attribute value consists of a name which uniquely identifies the attribute value, an associated type and a value of that type. The identifier of each attribute value must correspond to an attribute identifier in the tuple type's associated relational header.

A tuple type t is considered a subtype of tuple type s if and only if the header of t conforms to the header of s.

The language will support the equality operator over values of tuple types for which there exists a relational header that both headers of the arguments conform to. Such that two tuple types are equal if and only if the values of corresponding attributes in each argument are equal. The language will also supply an operation which extracts the value of a particular attribute from a tuple value.

The language will supply a *join* operator, which performs the relational join of two tuple values *a* and *b*. The arguments to the operator must be such that all attributes with equal identifiers have a common supertype. The result is a value of type $\text{TUPLE}_{a \cup b}$, with the types of attributes with common identifiers being cast-up to the common supertype, and values of attributes corresponding to their values in the source arguments. Invoking this operator over two tuple values where the common attributes do not agree in value is an error.

The language will support a *project* operator over tuple values, such that the result is a tuple with a subset of the attributes. The value of each attribute in the result is equal to the value of the corresponding attribute in the argument.

The language will support a *rename* operator over tuple values, such that the result is a tuple with the same attribute values as the argument, but with the identifiers of specific attributes being altered. The types and values of attributes in the result with otherwise remain unchanged.

The language will support a *wrap* operator over tuple values, such that the result is a tuple with a subset of the attributes of the argument, and one additional attribute *a* of a tuple-type. Attribute *a* will contain all the attributes of the argument which are not present in the result. All attributes types and values in the argument remain unchanged in the result.

The language will support an *unwrap* operator which takes an argument of type *t* and an attribute name *n* such that *t* contains an attribute of name *n* and tuple-type *s*. The result is a value of tuple type $\text{TUPLE}_{(t-\{n\}) \cup s}$. The values and types of attributes in the result are equal to their corresponding source attributes in *t* or *s*.

The language will support a *membership* operator, which takes a tuple value *t* and a relation value *r* as arguments. The header of the tuple *t* must conform to the header of *r*. The operator returns a boolean truth value to indicate whether *t* is in the body of *r*.

Relation Types

The language must allow the specification of relation types. Relation types are associated with a specific relational header. The value of a particular relational type is a set of tuple values, such that each tuple value conforms to the header of the relational type.

Relation types also have a set of keys. Each key consists of a set of attribute identifiers, such that all tuples in values of the type can be uniquely identified by the combination of all attributes in the key. The set of all attributes in the corresponding relational header is always a member of the set of keys. Attempting to add a tuple value *v* to relation *r* where *r* contains a tuple value *v'* such that $v \neq v'$ and *r* contains a key *k* such that for each attribute in *k*, *v* and *v'* agree in value, is an error. This ensures that all tuples are uniquely identified by the combination of attributes in each key.

A relation type *t* is a subtype of relation type *s* if and only if the header of *t* conforms to the header of *s*, and every key *k* in *s* is a superset of a key in *t*.

The language will also support an operator to test if a relation value v_1 is a subset of v_2. The result is true if and only if every tuple value in v_1 is a member of v_2. The language will also support the equality operator over relation values, whereby the result is true if and only if v_1 is a subset of v_2 and v_2 is a subset of v_1.

The language will supply a *join* operator over two relation values *a* and *b*. The result *c* has the set of attributes which is the union of the attributes of *a* and *b*. Where *a* and *b* have attributes of the same name, the attributes must have a common supertype, and the corresponding attribute in *c* will be of that supertype. The tuples in *c* are the result of the tuple-join operation on all tuples from *a* and *b* where the common attributes agree in value.

The language will support a *union* operator over two relation values a and b. The result c is a relation type of the most-specific header of a and b. c contains all tuples which are in a, and all tuples which are in b.

The language will support an *intersection* operator over two relation values a and b. The result c is a relation type of the most-specific header of a and b, where a tuple is a member of c if and only if it is a member of a and a member of b.

The language will support a *difference* operator over two relation values a and b. The result c is a relation type of the most-specific header of a and b. All tuples in the result c shall be members of a such that the tuple is not a member of value b, and all tuples in a not in b are members of c.

The language will support a *project* operator over a relation value v. The result p is a relation with a subset of the attributes in relation value v. p will have the set of keys of v, minus any keys that contained attributes not in p, union the set of attributes in p. The value of p is the result of applying the equivalent *project* operator over all tuples in v and adding them to p.

The language will support a *restrict* operator over a relation value v, which also takes as argument an expression e. The result r is of the same type as v. Each tuple in r is a member of v, and each tuple t in v is a member of r if and only if $e(t)$ evaluates to true.

The language will support a *rename* operator over a relation value v, which takes as argument a set of pairs of attributes s and t. The result r is a relation of the type of v but all attributes named corresponding to s identifiers now being identified by the corresponding t identifier. All tuple members of r correspond to members of s after the tuple-equivalent renaming operation has been performed.

The language will support an *extend* operation over a relation value v. Additional arguments are a new type t, an identifier n and an expression e. The type of e when applied to tuples of v is required to be a subtype of t. The result type r has the attributes of v and an additional attribute n of type t. The tuples in r are the results of applying the equivalent *extend* operation over each tuple value in v.

The language will support a *wrap* operation over a relation value v. A new attribute identifier n and a set of attributes s will also be specified such that the set of attributes in v (v') are a superset of s. The result r is a relation type of type t, such that the set of attributes of t is given by $(v'-s) \cup \{n\}$. Attribute n in t is of a tuple type with the set of attributes s. The set of tuples in r is the result of applying the corresponding *wrap* operation over all tuples in v.

The language will support an *unwrap* operation over a relation value v of type RELATIONx. An attribute in v, n, will also be specified such that attribute n is of a tuple type TUPLEt in each tuple of v. t and x-$\{n\}$ must have no common attributes. The resultant value is a relation value of type RELATION$_{(x-\{n\})\cup t}$ and has the set of tuples given by applying the corresponding *unwrap* operator over each tuple in v.

The language will support a *group* operation over a relation value v. A new attribute identifier n and a set of attributes s will also be specified such that v has attributes which are a superset of s. The result r is a relation of type t whereby r contains all attributes in v not in s (s'), and an extra attribute named n of a relation type which has attribute set s. For each tuple x in the result, the value of the n attribute is given by the projection of s onto the value v after it has been restricted to where attributes s' agree in value to the corresponding attributes in x.

The language will support an *ungroup* operation over a relation value v. The attribute to ungroup (a) will also be specified, such that the type of a in v is of a relation type t. The result r has all attributes in v minus a, and addition attributes corresponding to each attribute in t. The set of tuples in r is given by the n-ary union of the extension of the a attribute of each tuple x in v, with the other attributes of x.

The language will support an *update* operation over a relation value v. The operation takes as argument the identifier n of an attribute in v and an expression e. The result r of the operation is of the same relation type as v, and the tuple values in r correspond to $e(t)$ where t is each tuple in v.

The language will support an *extract* operation over a relation value *v* of cardinality one. The operation returns a value of the corresponding tuple type of *v*. Invoking *extract(v)* where v is not of cardinality one is an error.

The language will support a *count* operation over a relation value *v*, which returns the cardinality of *v* as an integer value.

<u>Operators</u>

The language must support the notion of read-only and update operators. Each operator has an associated identifier.

Read-only operators take a well-defined number of value parameters, and return a value of a well-defined type. Update operators take a well-defined number of parameters which may be values or variables, and do not return a value. The bodies of update operators must treat variable parameters in the same way as local variables. Any changes to the value of variable parameters by an update operator will be visible to the caller. The value of arguments passed to value parameters will remain constant over operator execution.

Read-only operators may share their identifier with other read-only operators, as long as the two operators are distinguishable by either number of parameters or the types of corresponding parameters (by ordinal position). Update operators may share their identifier with other update operators, as long as the operators are distinguishable by either number of parameters, update/value attribute of corresponding parameters, or types of parameters (by ordinal position).

Operator invocations consist of the name of the operator and an argument list. Operators invocations will be resolved in the following order:

1. By identifier
2. By number of arguments
3. By variable/value status of arguments (in the case of update operators)
4. By the declared type of arguments
5. By the most-specific type of the arguments (at run-time)

Update operator invocations must specify which arguments are to be treated as variables and which as values. The most-specific type of a value is the type which the value is a member of and which defines the smallest set of values.

Operator *o* is 'more-specific' than operator *p* if all read-only parameters in *o* are of a subtype to the type of corresponding parameter in *p*. Operator *o* is 'compatible' with operator *p* if all read-only parameters in *o* are non-distinct (ie share at least one value) with the type of the corresponding parameter in *p*. Operator *o* is ambiguous with *p* if *o* is compatible with *p* but neither *o* is more-specific than *p* or *p* more specific than *o*. All operators where *o* is ambiguous with *p* must have a corresponding operator *q* with the same identifier such that each parameter in *q* is the more-specific type of corresponding parameters in *o* and *p*.

A read-only operator *o* which is more-specific than *p* must always return a value which is a subtype of the return type of *p*.

<u>Type Tests</u>

The language will supply an operator of boolean type which returns true if a value of a particular type *t* is a member of the set of types of some other specified type *s*, such that *s* is a subtype of *t*.

The language will also supply an operator which converts a value of type *t* to a value of type *s*, where *s* is a subtype of *t*. If during execution the value is found not to be of type *s*, a run-time error shall occur.

Databases

The language must allow the declaration of databases. A database consists of a set of relational variables. A relational variable in a database can be either real or virtual. A real relational variable exists in space, and must persist over the course of a database relation. A virtual relational variable has a value only through the evaluation of a corresponding expression. The language must allow all relational variables in a database to be accessed and assigned to in a uniform manner. Databases must have an associated unique identifier.

The language must also allow for relationships between database relations to be expressed, such as a *foreign key* relationship, where an attribute of one relation uniquely identifies another.

Transactions

The language shall support the notion of transactions. Transactions must have an associated unique identifier, and exactly one associated database which it can interact with. The language will allow transaction bodies to access the specific database in the same manner as a local variable.

The language shall allow transactions to terminate through two distinct methods. They may either commit (indicating successful termination) or rollback, indicating failure. A transaction which successfully terminates marks all alterations to the database as permanent before terminating. A transaction which fails will undo all alterations it has made to the database before terminating – i.e. will have no lasting effect on the database.

The language shall allow a construct which starts a transaction and executes particular instructions on the event of that transaction failing. Transactions may only start other transactions which interact with the same database. In the event of a transaction failing, all other transactions which it started also fail, regardless of whether they made changes to the database and completed successfully.

Application Entry-Point

The language shall allow the specification of an application starting point. This is capable of starting any transaction which has been defined.

Language Aims

The language should attempt to conform to general language design principles, in particular:

Principle of Locality
: The language should support the capability to limit the effects of actions to a well-defined scope.

Principle of Type Completeness
: All types should be treated equally, with no areas in the language where specific types are allowed and others not.

Principle of Abstraction
: The language shall supply mechanisms which abstract over semantically-meaningful constructs. These mechanisms can then be used in place of the original construct.

Also, the language should strive to:

… be defined by a small set of rules with the minimum number of special cases [*Simple*]
… use the same construct as often as possible [*General*]
… express constructs which are semantically similar in a similar form [*Consistent*]
… consist of independent features which can be combined or used in isolation [*Orthogonal*]

The language must allow the expression of small components in a clear and straightforward way, while supporting constructs for the division of larger components into smaller subsystems. The primary purpose of the language is to act as an example of the 'D' model.

2.2 Language Design

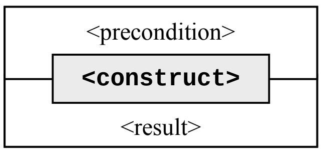
All languages subscribe to at least one language paradigm. The imperative paradigm uses variables to specify storage locations, with storage being denoted through the assignment statement. The majority of programmers are familiar with this form of language, and so D^b shall also primarily support this paradigm. Indeed, while the 'D' model is not explicitly tied to any particular paradigm, the definition of the model explicitly gives rules governing the notion of variables, so an imperative language can be evaluated against the model much more easily.

With the language design principles defined in the requirements to aim for, each major component of the language can be designed accordingly. While the original model specification refers to 'read-only' and 'update' operators, D^b shall use the traditional equivalent of function and procedure abstractions respectively.

The syntax of the language is presented in EBNF (Extended Backus Naur Form), a form of BNF which also allows regular-expressions. For readability, not all of the grammar rules presented here are in LL1 form (which eases the load on the implementation). However the language can be expressed as an LL1 grammar, and has been in Appendix A. Where source-code examples have been supplied, keywords are highlighted in bold text.

Type Checking

It is very important that D^b be type-safe, in that it does not allow any operator to be invoked over arguments of unsuitable types. Where such type-checking must be applied, type rules are expressed in the form:



<precondition> is the rule which must be satisfied in order for the rule to be followed. If the construct is found in a source description where no type rule is found of the correct form such that the pre-condition is satisfied, the source description contains a type error. The pre-condition is expressed using (informal) mathematical set notation. The following extra notation has been used:

$T \preccurlyeq S$	- T is a subtype of S, or T and S are the same type
$T \prec S$	- T is a subtype of S
RelationTypes	- The set of all relation types
TupleTypes	- The set of all tuple types
Types	- The set of all types
tuple[X]	- The tuple type with set of attributes X
relation[X]	- The relation type with set of attributes X
< T, id >	- an attribute of type 'T' and with identifier 'id'
attributes(T)	- the set of attributes of T, where T is a relation or tuple type
components(C)	- the components of complex type C
id(a)	- the identifier of attribute/component/parameter a
type(a)	- the type of attribute/component/parameter a
id(f)	- the identifier of function/procedure/transaction f
type(f)	- the return type of function f
parameter(f, i)	- the parameter in position *i* in function/procedure/transaction f
type(p)	- the type of the parameter p

The <construct> section contains the basic structure of the syntactic rule, with placeholders being used where the user would insert expressions or identifiers, etc. Placeholders used in the pre-condition are underlined. Placeholders in capitals refer to types, while those in small letters refer to identifiers.

Finally the <result> section contains the type that can be deduced from the construct being found in the source description such that the pre-condition is met. This is omitted in the case of the type rules for statements, as statements do not have such a type. An alternative would be to invent some form of 'void' type.

Only those areas of the language which have been implemented are presented here. The full D^b language syntax is described in Appendix A.

Base Lexemes

A lexeme is the atomic unit of the language, defining basic 'words'. They form the building-blocks of the other syntax rules and so are presented here.

```
<integer_const>     ::=   <number>+
<float_const>       ::=   <integer_const> . <integer_const>
<bool_const>        ::=   true | false
<char_const>        ::=   ' <non_apostrophe> '
<string_const>      ::=   " (<non_quote>)* "
<identifier>        ::=   <letter> (<letter> | <number>)*

<number>            ::=   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter>            ::=   a | b | c | d | e | f | g | h | i | j |
                          k | l | m | n | o | p | q | r | s | t |
                          u | v | w | x | y | z | A | B | C | D |
                          E | F | G | H | I | J | K | L | M | N |
                          O | P | Q | R | S | T | U | V | W | X |
                          Y | Z
<char>              ::=   [any unicode character value]
<non_apostrophe>    ::=   [any <char> not apostrophe character]
<non_quote>         ::=   [any <char> not quote character]

<commentchar>       ::=   [any <char> not carriage return]
<comment>           ::=   # (<commentchar>)* <carriagereturn>
```

Pre-defined Types

Semantics:

The following simple types will be pre-defined by the system and available for users to create larger more complex types from:

integer	defines set of values from -2,147,483,648 to +2,147,483,647
float	defines set of values from -3.402823×10^{38} to $+3.402823 \times 10^{38}$
string	defines a sequence of zero or more characters (zero-based)
char	defines the set of Unicode characters
boolean	defines truth values **true** and **false**.

By supporting these pre-defined types and their operations, D^b meets the requirements of the 'D' model that a number of basic simple types be system defined.

Type Expressions

Syntax:

```
<type_expr>         ::=   <identifier> | <rel_type>
<rel_type>          ::=   <tuple_type> | <relation_type>
<tuple_type>        ::=   tuple <rel_header>
<relation_type>     ::=   relation <rel_header>
<rel_header>        ::=   { [<rel_attr> (, <rel_attr>)*] }
<rel_attr>          ::=   <type> <identifier>
```

Semantics:

Each type in the system has an associated unique name. In the case of the built-in and structured types, this takes the form of an identifier. The 'names' of specific relational types (tuples and relations) are rather more complex to describe, as they contain the specification of a relational header. A relational header is the set of attributes and their corresponding types.

Examples:

```
Circle
tuple{ integer id, integer age }
relation{ integer age, string jobtitle }
```

Note:

Functions are not considered to be values of a type in D^b. The 'D' model does not define function to have types - in order to conform with the principle of type completeness, there would have to be the introduction of such strange notions as function values having a most specific type and the storage of functions in a relation. Such theoretical complications are difficult to remedy and for this reason D^b will not treat functions as values of types. This however comes at the cost of generality, and leads to slightly awkward syntax for relational operators. The author firmly believes a solution to this problem would be greatly beneficial.

The same argument applies to types themselves: the set of all types in the system is in fact a type itself, defining a set of values, each of which are types. But what is the most-specific type of a type? This is again not defined in the 'D' model, and so types are not treated as values either.

Local Declarations

Syntax:

```
<local_decl> ::= <type_expr> (<var_decl> | <value_decl>);
<var_decl>   ::= variable <identifier> = <expression>
<val_decl>   ::= value <identifier> = <expression>
```

Semantics:

Local declarations may appear in statement lists. They allow the introduction of names which can either be values or variables. Values may not alter their value after declaration, whereas variables hold the value which was last assigned to them.

The syntax forces definite assignment on declaration. Another possibility would be to burden the implementation with enforcing the assignment to variables before allowing them be used as values. However, forcing definite assignment on declaration does not do any real harm (in certain situations it can be slightly inefficient) and allows similar syntax for the two variations.

Examples:

```
float value pi = 3.141592654;
integer variable age = 5;
```

Scope Rules

Syntax:

```
<stmnt_block>::= {  (<statement>)* }
```

Semantics:

To support the principle of locality, D^b allows a statement to consist of a sequence of statements, inside a *statement-block*. The statement block introduces a new scope, and any identifiers declared inside that block may 'override' identifiers declared elsewhere. Identifiers declared in a *statement-block* are local to that block and inaccessible from an external block.

Identifiers which can be declared inside a statement block are variables and values only. Allowing the declaration of functions or procedures inside statement block would allow a 'more-specific' function to be visible only inside a particular block, and would prove difficult to implement. Unfortunately therefore D^b does not truly support the principle of locality.

Type Definitions

Syntax:

```
<complex_def>        ::=   complex type <identifier> [<stype_list>]
                                  <complex_body> [<complex_extras>];
<stype_list>         ::=   : <stype_entry> (, <stype_entry>)*
<stype_entry>        ::=   <identifier> [when <expression>]
<complex_body>       ::=   { <complex_member>* }
<complex_member>     ::=   <type_expr> <identifier> [<constraint>]
<complex_extras>     ::=   <generalisation>+ <specialisation>
<generalisation>     ::=   generalisation <identifier>( <arglist> )
<specialisation>     ::=   specialisation ( <arglist> )
```

Type Rules:

$$\exists T{:}\mathrm{ComplexTypes}\ \forall i{:}1\ldots n \cdot S_i \preccurlyeq T$$

$$\boxed{\textbf{complex type } \underline{c} : \underline{S}_1, \ldots\ \underline{S}_n\ \{\ \ldots}$$

Semantics:

Where a type is a subtype of a single type, the constraint must be specified after the supertype identifier. Constraints are specified by expressions, in which the keyword *value* is a placeholder for the current value being tested. This is necessary to differentiate between identifiers of components and function identifiers.

Examples:

```
complex type Ellipse { float a; float b; };

complex type Circle : Ellipse when value.a = value.b
{ float r; }
generalisation Ellipse(value.r, value.r)
specialisation Circle(Ellipse.a);

complex type Square : Para, Rhombus
{ Point a; Point b; Point c; Point d; }
generalisation Para(value.a, value.b, value.c, value.d)
generalisation Rhombus(value.a, value.b, value.c, value.d)
specialisation Square(Para.a, Para.b, Para.c, Para.d);
```

Notes:

Generalisations and specialisations for each derived type are expressed as selector invocations. These are required so that values of a subtype are substitutable for values of the supertype. As subtypes are not required to inherit the structure of their supertypes, there must be an expression which allows values of the supertype to be expressed in terms of the subtype, and vice versa, i.e. generalisation and specialisation statements. An alternative would be to express the values of components in the subtype in terms of the supertype, and have the compiler deduce the equivalent generalisation statements. However this cannot always be calculated by the compiler, and so would lead to additional syntax *and* the user adding the equivalent of generalisation and specialisation expressions in certain cases anyway. It is therefore rejected in favour of the user having to explicitly state the relationship.

Values

Syntax:

```
<value>              ::=   <sel_invoc> | <base_literal>
<sel_invoc>          ::=   <type_expr> ( <sel_arglist> )
<sel_arglist>        ::=   [<expression> (, <expression>)*]
<base_literal>       ::=   <integer_const> | <float_const> |
                           <bool_const>|<char_const>|<string_const>
```

Semantics:

Values may be introduced directly through a selector invocation, which consists of the type name and values from which to initialise the value. Selectors are automatically generated for all defined types. This includes the built-in types, however these types can also be specified in their shorthand form (e.g "`integer(3)`" is equivalent to just "`3`").

Variable identifiers can also be used in place of names, in which case the value is the last value to be assigned to the variable. In the case of the value term being of a complex or tuple type, the components of the value can be extracted using the dot syntax.

Relation values are slightly different in that they accept zero or more arguments, rather than a fixed number. This is necessary as at present D^b does not support collections other than relations. In future relation selectors may instead take a single argument which is of type *set*.

Examples:

```
3.14
tuple{integer age}(20);
relation{integer age}( tuple{integer age}(1), tuple{integer age}(4) );
```

<u>Assignment</u>

Syntax:

```
<assignment>        ::=    <assignable> := <expression> ;
<assignable>        ::=    <identifier> (. <identifier>)*
```

Type Rule:

$$\frac{\underline{X} \preccurlyeq \underline{V} \wedge \mathrm{isvar}(\underline{V})}{\boxed{\underline{V} := \underline{X}}}$$

Semantics:

The language allows variables to be assigned values. The result of assigning a value *v* to a variable *var* is that (until the next assignment to *var*) expressions which use *var* in a value position, the result shall be the same as if *v* was used instead.

Notes:

Assignment is a statement and so there is no resultant type. The decision to disallow assignment inside expressions was taken in the interests of generality, as functions abstract over expressions and arguments to functions must remain constant during their execution – so clearly functions cannot abstract over assignment. It follows that assignment must be a statement, which goes against other common languages such as C++ [Stroustrup, 2000] and Java [URL 3].

<u>Databases</u>

Syntax:

```
<database_def>      ::= database <identifier> { <database_rel>* }
<database_rel>      ::= <real_rel> ;
<real_rel>          ::= <relation_type> <identifier>
```

Semantics:

Databases provide persistent storage, in that all relational variables in the database will persist, i.e. maintain their value after the application has terminated.

Note:

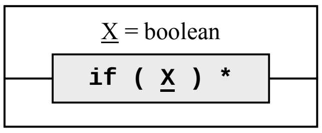
The 'D' model insists that <u>only</u> relational variables may be stored in databases – the authors explicitly reject the notion of persistence as "a property of data orthogonal to its type" as promoted in [Atkinson & Buneman, 1988]. This seems so fundamental to the 'D' model that D^b shall adhere to this requirement and break the principle of type completeness with respect to persistence.

Conditional Statement Execution

Syntax:

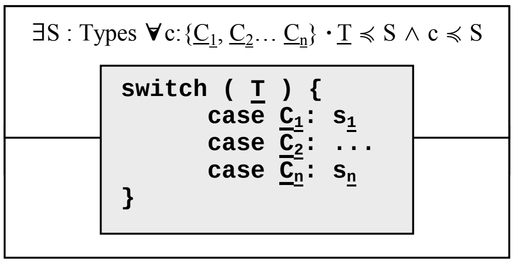
```
<switch>      ::= switch ( <expression> ) { <switch_body> }
<switch_body> ::= <case_entry>* [<other>]
<case_entry>  ::= <case_value> : <statement>
<case_value>  ::= case <when_entry> : <statement>
<when_entry>  ::= <expression> (or <expression>)*
<other>       ::= other : <statement>

<if_stmnt>    ::= if ( <expression> ) <statement> [<else>]
<else>        ::= else <statement>
```

Type Rules:



$\underline{X}$ = boolean

```
if ( X ) *
```



$\exists S : \text{Types}\ \forall c{:}\{\underline{C_1}, \underline{C_2} \ldots \underline{C_n}\} \cdot \underline{T} \preccurlyeq S \wedge c \preccurlyeq S$

```
switch ( T ) {
      case C1: s1
      case C2: ...
      case Cn: sn
}
```

Semantics:

The *if* statement allows a particular statement to be executed based on the run-time value of a particular expression. If the expression evaluates to **true** then the statement is executed, else it is skipped. The extended form of the statement allows specification of an additional statement to be executed when the expression evaluates to **false**.

The *switch* statement allows a more convenient and readable syntax for conditional execution based on the value of a particular expression. The *other* clause allows the user to specify a statement to execute when the value is 'any other not specified'. Once the expression has been matched to a particular case clause, the corresponding statement is executed, and then control passes to statement immediately following the *switch* construct. Execution does not 'fall through' to the next *case* statement.

Note:

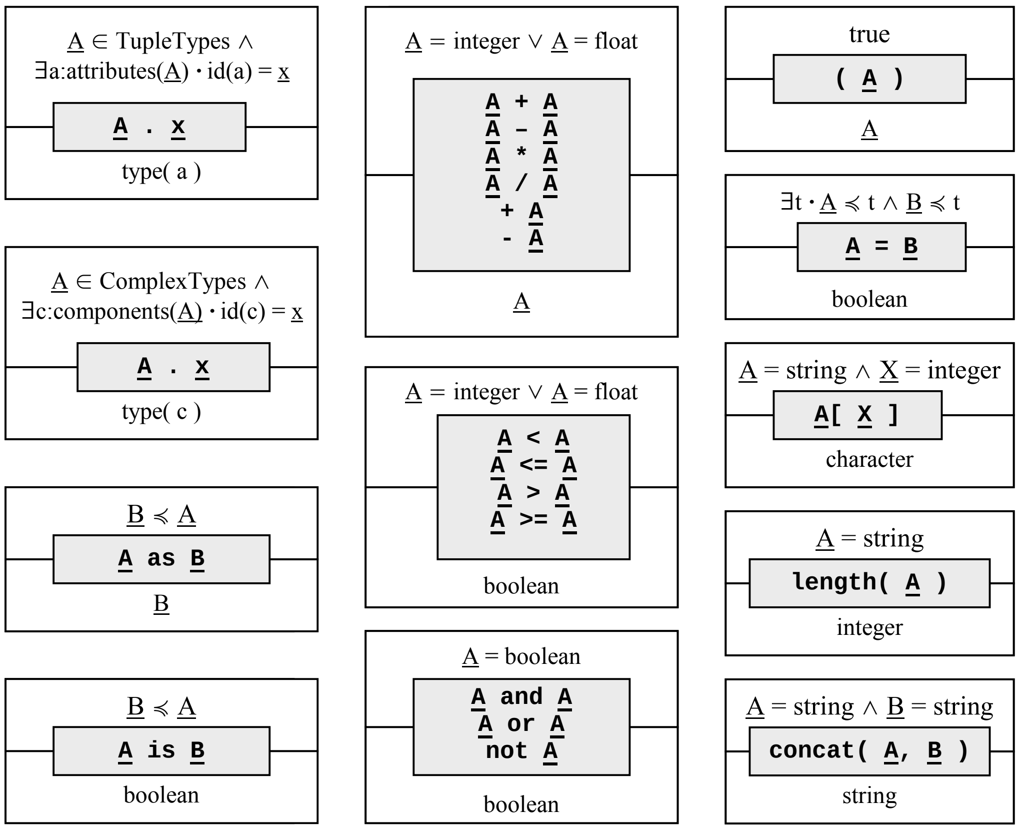
In some languages, the *if* statement can also appear in expressions. This doesn't seem unreasonable (the <*statement*> clauses become <*expression*> clauses, all having some common supertype, which would be considered the type of the *if* statement) but doesn't seem particularly essential either. In the interests of keeping the language simple, such variations have been omitted.

General Expressions

Syntax:

```
<expression>        ::=   <unaryexpr> | <mathexpr> | <relexpr> |
                          <value> | <compterm> | <func_invocation> |
                          <array_ref> | <typetest> | ( <expression> )
<unaryexpr>         ::=   + <expression> | - <expression> |
                          not <expression>
<mathexpr>          ::=   <expression> <mathop> <expression>
<compexpr>          ::=   <expression> <comparisonop> <expression>
<comparisonop>      ::=   and | or | < | > | = | >= | <=
<mathop>            ::=   + | - | * | /
<array_ref>         ::=   <expression> [ <term> ]
<typetest>          ::=   <expression> (is | as) <type_expr>
```

Type Rules



Semantics:

Expressions can be used to calculate values in the language. Basic expressions take the form of terms and operators. In more complex forms, they can also contain sub-expressions and function calls. Operators over the basic types are defined as the mathematical infix operators, with the asterisk (*) being used for multiplication and the forward-slash (/) being used for division.

Notes:

Mathematical versions of the logical operators (∧,∨,¬) could have been used instead of their textual equivalent. This would have required the user to enter text using a Unicode-enabled font. As not all systems are Unicode-enabled (and not all fonts support such operators), the decision was made to stick with the textual versions. Future additions to the language may allow operators to also be expressed in their Unicode forms.

The above grammar is very non-LL1. The real rules are given in Appendix A.

Statement Repetition

Syntax:

```
<for>                 ::=    for (<init>; <test>; <next>) <statement>
<init>                ::=    <type_expr> <identifier> := <expression>
<test>                ::=    <expression>
<next>                ::=    <assignment>
<do>                  ::=    do <statement> while ( <expression> );
<while>               ::=    while ( <expression> ) <statement>
```

Type Rules:

$$\text{type}(\underline{V}) \preccurlyeq \underline{T} \wedge \underline{E} = \text{boolean}$$

$$\boxed{\mathtt{for\ (}\underline{\mathtt{T}}\mathtt{\ i\ :=\ }\underline{\mathtt{V}}\mathtt{;\ }\underline{\mathtt{E}}\mathtt{;\ N)}}$$

$$\underline{E} = \text{boolean}$$

$$\boxed{\mathtt{do\ s\ while\ (\ }\underline{\mathtt{E}}\mathtt{\ );}}$$

$$\underline{E} = \text{boolean}$$

$$\boxed{\mathtt{while\ (\ }\underline{\mathtt{E}}\mathtt{\ )\ s}}$$

Semantics:

The *do* and *while* constructs allow statements to be repeated while a particular expression evaluates to **true**. *while* provides loops with the test on entry, the *do* version on exit.

The *for* loop is really a form of the *while* loop, which is so common that many languages give it a special construct. A loop variable is initialised to a certain value, and the statements in the body of the loop are executed while the expression (calculated at the top of the loop) evaluates to **true**. After the statements in the loop body are executed, the <*next*> statement is executed, which ought to alter the loop variable in some way. Control then passes back to 'top' of the loop (the evaluation of the invariant expression).

Abstraction Mechanisms

Syntax:

```
<function_header>   ::=    <type_expr> function <id> ( <paramlist> )
<procedure_header>  ::=    procedure <id> ( <paramlist> )
<transaction_header>::=    transaction <id> ( <paramlist> ) uses <id>
<program_header>    ::=    program ( <paramlist> )

<abstraction>       ::=    <header> <statementblock>

<return_statement>  ::=    return [<expression>];
<trans_termination> ::=    (rollback | commit);
```

Type Rules:

$$\forall g : \text{Functions} \cdot \text{id}(g) = \underline{f} \wedge \text{arity}(g) = \underline{n} \Rightarrow$$
$$((\forall i{:}1\ldots\underline{n} \cdot \text{type}(\text{param}(g,i)) \preccurlyeq \underline{P_i}) \Rightarrow \text{type}(g) \preccurlyeq \underline{T}) \wedge$$
$$((\forall i{:}1\ldots\underline{n} \cdot \underline{P_i} \preccurlyeq \text{type}(\text{param}(g,i))) \Rightarrow \underline{T} \preccurlyeq \text{type}(g))$$

$$\boxed{\underline{\mathtt{T}}\mathtt{\ function\ }\underline{\mathtt{f}}\mathtt{(}\underline{\mathtt{P}_1}\mathtt{\ i}_1\mathtt{,\ \ldots\ }\underline{\mathtt{P}_n}\mathtt{\ i}_n\mathtt{)}}$$

Semantics:

D^b allows abstractions over statements and expressions. Statements can be abstracted over primarily by the *procedure* construct, which can contain *value* or *variable* parameters. Expressions can be abstracted over through the *function* abstraction, which can only take *value* parameters but has an associated type. *value* parameters are treated from within abstraction bodies as though they were locally-defined constants.

The *program* abstraction abstracts over statements and is implicitly executed at start-up. It may take only value parameters and termination is through the *return* statement. A *program* may start any transaction on any database, however the *onfail* clause must be included in any *begin* statements.

The *transaction* abstraction abstracts over statements which access or alter a database in some way. They may take *variable* or *value* parameters and must specify which database they will alter in the header. It is illegal to invoke a transaction from within a *procedure* or *function* body. It is also illegal to invoke a transaction which operates on database *d* from within a transaction which does not operate on database *d*. Transactions differ from other transactions in their method of termination: they either *commit* or *rollback*. If it commits, changes to the database are made final and control returns to the callee as normal. If it rolls back, changes to the database made by it and all transactions which it started are cancelled, and control passes back to the callee's *onfail* statement. In the transaction body, the database with which the transaction interacts with is available through the *database* keyword.

Both *function* and *procedure* abstractions terminate through the *return* statement, where procedures must not have a return value and functions must. *function* and *procedure* adhere to the requirements for the provision of "read-only" and "update" operators respectively. *transaction* adheres to the requirements for provision of explicit transaction boundaries and transaction termination. *program* is primarily supplied such that stand-alone applications can be written in D^{b} – however their use is somewhat limited at present (see "Further Work").

Abstraction Invocations

Syntax:

```
<procedure_invoc>   ::=   call <id>( <arglist> )
<function_invoc>    ::=   <id>( <arglist> )
<trans_invoc>       ::=   begin <id>( <arglist> ) [<onfail>]
<onfail>            ::=   onfailure <statement>
<arglist>           ::=   <argument> (, <argument>)*
<argument>          ::=   <expression> | (asvar <assignable>)
```

Type Rules:

$$\begin{array}{c}
\exists f : \text{Functions} \cdot \text{name}(f) = \underline{x} \wedge \text{arity}(f) = \underline{n} \wedge (\forall i{:}1\ldots n \cdot A_i \preccurlyeq \text{type}(\text{param}(f,i))) \wedge \\
\neg \exists g : \text{Functions} \cdot \text{name}(g) = x \wedge \text{arity}(g) = n \wedge \\
(\forall i{:}1\ldots n \cdot A_i \preccurlyeq \text{type}(\text{param}(g,i)) \wedge \text{type}(\text{param}(g,i)) \preccurlyeq \text{type}(\text{param}(f,i))) \\
\hline
\boxed{\underline{x}\ (\ \underline{A}_1,\ \underline{A}_2,\ \ldots\ \underline{A}_n\ )} \\
\hline
\text{type}(f)
\end{array}$$

$$\begin{array}{c}
\exists p : \text{Procedures} \cdot \text{name}(p) = x \wedge \text{arity}(f) = n \wedge \\
\forall i{:}1\ldots n \cdot\ (\text{isvar}(A_i) \Rightarrow \text{isvar}(\text{param}(p,i)) \wedge \text{type}(\text{param}(p,i)) = A_i) \vee \\
(\neg\text{isvar}(A_i) \Rightarrow \neg\text{isvar}(\text{param}(p,i)) \wedge A_i \preccurlyeq \text{type}(\text{param}(p,i))) \\
\hline
\boxed{\textbf{call}\ \underline{x}\ (\ \underline{A}_1,\ \underline{A}_2,\ \ldots\ \underline{A}_n\ )} \\
\hline
\end{array}$$

It is an error to specify a software component with functions or procedures defined in such a way that the run-time version of a particular invocation is possibly ambiguous. Given the following definition:

$$\begin{array}{l}
\text{same\_tree}(f,g) \overset{\text{def}}{=} (\text{name}(f) = \text{name}(g) \wedge \text{arity}(f) = \text{arity}(g) \wedge \\
\qquad (\forall i{:}1\ldots\text{arity}(f)\ \exists t \cdot \text{type}(\text{param}(f,i)) \preccurlyeq t \wedge \text{type}(\text{param}(g,i)) \preccurlyeq t\ ))
\end{array}$$

The following constraint must be true of the state of any given software component prior to execution:

$$\forall f, g : \text{Functions} \cdot \text{same\_tree}(f,g) \Rightarrow$$
$$\quad \exists h : \text{Functions} \cdot \text{same\_tree}(f,h) \wedge \forall i{:}1\ldots\text{arity}(f)$$
$$\qquad \exists t : \text{Types} \cdot t \preccurlyeq \text{type}(\text{param}(f,i)) \wedge t \preccurlyeq \text{type}(\text{param}(g,i)) \wedge$$
$$\qquad\quad (\nexists g\ : \text{Types} \cdot t \prec g \wedge g \preccurlyeq \text{type}(\text{param}(f,i)) \wedge g \preccurlyeq \text{type}(\text{param}(g,i)))$$
$$\qquad\quad \wedge\ \text{type}(\text{param}(h,i)) = t$$

Similar rules affect the read-only parameters of procedures.

Semantics:

The argument to a *variable* parameter must in itself be a variable and marked as such by the caller – otherwise it will be treated as a value when looking for potential targets. The value of the variable passed as an argument may alter during the execution of the statement. The argument to a *value* parameter is guaranteed to remain constant during execution.

Both procedures and transactions abstract over statements, and are invoked as statements themselves. Transaction invocations have an optional *onfail* clause, which allows the user a statement to be executed if the transaction terminates with the *rollback* statement. If such a statement is omitted, then the caller ('parent') transaction fails also. It is an error for a *begin* statement in a *program* body to omit the *onfail* clause.

Procedures and functions can have 'more-specific' versions. The actual version executed at run-time is the version whose value parameter types match the 'most-specific type' of the arguments most accurately.

Relational Projection

Syntax:

```
<project_op> ::= project { <attribute_list> } ( <expression> )
```

Type Rules:

$\underline{A} \in \text{TupleTypes} \wedge \forall i{:}(1\ldots n)\ \exists a{:}\text{attributes}(A) \cdot \text{id}(a) = \underline{x}_i$

$\boxed{\texttt{project}\{\underline{x}_1,\ \underline{x}_2,\ \ldots\ \underline{x}_n\}\ (\ \underline{A}\ )}$

$\text{tuple}[\{\underline{x}_i,\ \underline{x}_2,\ \ldots\ \underline{x}_n\}]$

$\underline{A} \in \text{RelationTypes} \wedge \forall i{:}(1\ldots n)\ \exists a{:}\text{attributes}(A) \cdot \text{id}(a) = \underline{x}_i$

$\boxed{\texttt{project}\{\underline{x}_1,\ \underline{x}_2,\ \ldots\ \underline{x}_n\}\ (\ \underline{A}\ )}$

$\text{relation}[\{\underline{x}_i,\ \underline{x}_2,\ \ldots\ \underline{x}_n\}]$

Semantics:

The *project* operator is a read-only operator which returns a tuple or relation-valued result, such that the attributes of the result are equal to the set of attributes specified in the operator.

In the tuple-variation of the operator, all attributes in the result are value-equal to the corresponding attributes in the argument. In the relation-variation, the set of tuples of the result is equal to the set of tuples achieved by performing the tuple-project operator on each tuple in the relation.

Relational Rename

Syntax:

```
<rename_op>   ::= rename { <rename_list> } ( <expression> )
```

Type Rules:

$\underline{A} \in \text{TupleTypes} \wedge \forall i:(1\ldots n) \cdot (\exists a:\text{attributes}(\underline{A}) \cdot \underline{x_i} = \text{id}(a)) \wedge$
$\nexists a:\text{attributes}(\underline{A}) \cdot \underline{y_i} = \text{id}(a)) \wedge \forall j:(1\ldots n) \cdot (i \neq j \Rightarrow \underline{y_i} \neq \underline{y_j})$

rename{x_1 to y_1, x_2 to y_2, … x_n to y_n} (A)

tuple[(attributes(A)-$\{x_1, x_2, \ldots x_n\}$) $\cup$ $\{y_1, y_2, \ldots y_n\}$]

$\underline{A} \in \text{RelationTypes} \wedge \forall i:(1\ldots n) \cdot (\exists a:\text{attributes}(\underline{A}) \cdot \underline{x_i} = \text{id}(a)) \wedge$
$\nexists a:\text{attributes}(\underline{A}) \cdot \underline{y_i} = \text{id}(a)) \wedge \forall j:(1\ldots n) \cdot (i \neq j \Rightarrow \underline{y_i} \neq \underline{y_j})$

rename{x_1 to y_1, x_2 to y_2, … x_n to y_n} (A)

relation[(attributes(A)-$\{x_1, x_2, \ldots x_n\}$) $\cup$ $\{y_1, y_2, \ldots y_n\}$]

Semantics:

The *rename* operator is a read-only operator which returns a tuple or relation-valued result, such that the attributes of the result correspond to the attributes of the argument, with specific attributes having their identifiers changed.

Relational Extend

Syntax:

```
<extend_op>   ::= extend[: <id>] { <extend_expr> } ( <expression> )
<extend_expr> ::= <type_expr> <id> = <expression>
```

Type Rules:

$\underline{A} \in \text{TupleTypes} \wedge (\nexists a:\text{attributes}(\underline{A}) \cdot \text{id}(a) = \underline{x}) \wedge \underline{E} \preccurlyeq \underline{T}$

extend{$\underline{T}$ $\underline{x}$ = $\underline{E}$} ($\underline{A}$)

tuple[attributes($\underline{A}$) $\cup$ $\{<\underline{T}, \underline{x}>\}$]

$\underline{A} \in \text{RelationTypes} \wedge (\nexists a:\text{attributes}(\underline{A}) \cdot \text{id}(a) = \underline{x}) \wedge \underline{E} \preccurlyeq \underline{T}$

extend{$\underline{T}$ $\underline{x}$ = $\underline{E}$} ($\underline{A}$)

relation[attributes($\underline{A}$) $\cup$ $\{<\underline{T}, \underline{x}>\}$]

Semantics:

The *extend* operator is a read-only operator which returns a tuple or relation-valued result (corresponding to the type of its argument), such that the result contains an extra attribute as specified.

The user may specify an identifier with the relation-*extend* operation, which the expression can reference. The expression is evaluated for each tuple in the argument, with the tuple value being substituted for the specific identifier each time.

Tuple Wrap

Syntax:

```
<wrap_op>    ::= wrap { <attribute_list> as <id> } ( <expression> )
```

Type Rules:

$\underline{A} \in \text{TupleTypes} \wedge \forall i{:}(1\ldots n)\ (\exists a{:}\text{attributes}(\underline{A}) \cdot \text{id}(a) = \underline{x_i}) \wedge (\nexists a{:}\text{attributes}(\underline{A}) \cdot \text{id}(a) = \underline{y})$

wrap{$\underline{x_1}$, $\underline{x_2}$, … $\underline{x_n}$ as $\underline{y}$} ($\underline{A}$)

$\text{tuple}[\ (\text{attributes}(A)\text{-}\{\underline{x_1}, \underline{x_2}, \ldots \underline{x_n}\}) \cup \{\langle\text{tuple}[\{\underline{x_1}, \underline{x_2}, \ldots \underline{x_n}\}], \underline{y}\rangle\}\ ]$

$\underline{A} \in \text{RelationTypes} \wedge \forall i{:}(1\ldots n)\ (\exists a{:}\text{attributes}(\underline{A}) \cdot \text{id}(a) = \underline{x_i}) \wedge (\nexists a{:}\text{attributes}(\underline{A}) \cdot \text{id}(a) = \underline{y})$

wrap{$\underline{x_1}$, $\underline{x_2}$, … $\underline{x_n}$ as $\underline{y}$} ($\underline{A}$)

$\text{relation}[\ (\text{attributes}(A)\text{-}\{\underline{x_1}, \underline{x_2}, \ldots \underline{x_n}\}) \cup \{\langle\text{tuple}[\{\underline{x_1}, \underline{x_2}, \ldots \underline{x_n}\}], \underline{y}\rangle\}\ ]$

Semantics:

The *wrap* operator nests the attributes of a particular tuple or relation into another tuple-valued attribute.

Tuple Unwrap

Syntax:

```
<unwrap_op>  ::= unwrap { <id> } ( <expression> )
```

Type Rules:

$\underline{A} \in \text{TupleTypes} \wedge \exists a{:}\text{attributes}(\underline{A}) \cdot \text{id}(a) = \underline{x} \wedge \text{type}(a) \in \text{TupleTypes} \wedge$
$\forall b{:}(\text{attributes}(\underline{A}))\ \forall c{:}\text{attributes}(\text{type}(a)) \cdot b \neq a \Rightarrow \text{id}(b) \neq \text{id}(c)$

unwrap{$\underline{x}$} ($\underline{A}$)

$\text{tuple}[\ (\text{attributes}(\underline{A})\text{-}\{\underline{x}\}) \cup \text{attributes}(\text{type}(a))\ ]$

$\underline{A} \in \text{RelationTypes} \wedge \exists a{:}\text{attributes}(\underline{A}) \cdot \text{id}(a) = \underline{x} \wedge \text{type}(a) \in \text{TupleTypes} \wedge$
$\forall b{:}(\text{attributes}(\underline{A}))\ \forall c{:}\text{attributes}(\text{type}(a)) \cdot b \neq a \Rightarrow \text{id}(b) \neq \text{id}(c)$

unwrap{$\underline{x}$} ($\underline{A}$)

$\text{relation}[\ (\text{attributes}(\underline{A})\text{-}\{\underline{x}\}) \cup \text{attributes}(\text{type}(a))\ ]$

Semantics:

The *unwrap* operator performs tuple-level un-nesting, in that it takes a relational value with a tuple-typed attribute, and 'un-nests' the attributes of that tuple-type into the original value.

Relation Group

Syntax:

```
        <group_op>   ::= group{ <attribute_list> as <id> }( <expression> )
```

Type Rule:

$$\underline{A} \in \text{RelationTypes} \wedge (\forall i{:}(1\ldots n)\ \exists a{:}\text{attributes}(\underline{A}) \cdot \text{id}(a) = \underline{x_i}) \wedge (\nexists a{:}\text{attributes}(\underline{A}) \cdot \text{id}(a) = \underline{y})$$

$$\boxed{\texttt{group\{}\underline{x}_1\texttt{, }\underline{x}_2\texttt{, \ldots\ }\underline{x}_n\texttt{ as }\underline{y}\texttt{\} (}\underline{A}\texttt{)}}$$

$$\text{relation}[\ (\text{attributes}(A)\text{-}\{\underline{x_1}, \underline{x_2}, \ldots\ \underline{x_n}\}) \cup \{\langle\text{relation}[\{\underline{x_1}, \underline{x_2}, \ldots\ \underline{x_n}\}], \underline{y}\rangle\}\ ]$$

Semantics:

The *group* operator performs relation-level nesting, with the *y* attribute in the result consisting a relation containing attributes *x1, x2, … xn* corresponding to tuples in the argument where attributes in the result agree on value.

Relation Ungroup

Syntax:

```
        <ungroup_op> ::= ungroup { <id> } ( <expression> )
```

Type Rules:

$$\underline{A} \in \text{RelationTypes} \wedge \exists a{:}\text{attributes}(\underline{A}) \cdot \text{id}(a) = \underline{x} \wedge \text{type}(a) \in \text{RelationTypes} \wedge$$
$$\forall b{:}(\text{attributes}(\underline{A}))\forall c{:}\text{attributes}(\text{type}(a)) \cdot b \neq a \Rightarrow \text{id}(b) \neq \text{id}(c)$$

$$\boxed{\texttt{ungroup\{ }\underline{x}\texttt{ \} (}\underline{A}\texttt{)}}$$

$$\text{relation}[\ (\text{attributes}(\underline{A})\text{-}\{\underline{x}\}) \cup \text{attributes}(\text{type}(a))\ ]$$

Semantics:

The *ungroup* operator unnests relations from within attributes of other relations.

Relation Restrict

Syntax:

```
        <restr_op>   ::= restrict[<ren>] { <expression> }
                                         ( <expression> )
```

Type Rule:

$$\underline{A} \in \text{RelationTypes} \wedge \underline{E} = \text{Boolean}$$

$$\boxed{\texttt{restrict\{ }\underline{E}\texttt{ \} (}\underline{A}\texttt{)}}$$

$$\underline{A}$$

Semantics:

The *restrict* operator returns the same type as its argument, with each tuple in the result corresponding to a tuple in the argument for which the given expression evaluates to **true.**

Relation Update

Syntax:

```
        <update_op>   ::= update[<ren>] { <update_expr> } ( <expression> )
        <update_expr>::= <id> := <expression>
```

Type Rule:

$$\underline{A} \in \text{RelationTypes} \wedge \exists \text{a:attributes}(\underline{A}) \cdot \text{id(a)} = \underline{i} \wedge \underline{E} \preccurlyeq \text{type(a)}$$

$$\boxed{\texttt{update\{}\underline{\texttt{i}}\texttt{ := }\underline{\texttt{E}}\texttt{\}(}\underline{\texttt{A}}\texttt{)}}$$

$$\underline{A}$$

Semantics:

The *update* operator returns the same relation as its argument, but with the value of attribute *id* in each resultant tuple being set to the value of the the expression over each tuple of the argument.

Relational Natural Join

Syntax:

```
        <join_op>    ::= join( <expression>, <expression> )
```

Type Rules:

$$\begin{aligned}
&\underline{A} \in \text{RelationTypes} \wedge \underline{B} \in \text{RelationTypes} \wedge \exists \text{R:RelationTypes} \cdot \\
&\qquad (\forall \text{a:attributes}(\underline{A}) \cdot \\
&\qquad\qquad ((\exists\ \text{b:attributes}(\underline{B}) \cdot \text{id(a)} = \text{id(b)}\ ) \Rightarrow \\
&\qquad\qquad\qquad (\exists \text{t:Types} \cdot \text{type(a)} \preccurlyeq \text{t} \wedge \text{type(b)} \preccurlyeq \text{t} \wedge \\
&\qquad\qquad\qquad\qquad \nexists \text{g:Types} \cdot \text{type(a)} \preccurlyeq \text{g} \wedge \text{type(b)} \preccurlyeq \text{g} \wedge \text{g} \prec \text{t}) \\
&\qquad\qquad\qquad \wedge \langle \text{t, id(a)} \rangle \in \text{attributes(R)}) \\
&\qquad\qquad \wedge ((\nexists\ \text{b:attributes}(\underline{B}) \cdot \text{id(a)} = \text{id(b)}\ ) \Rightarrow \text{a} \in \text{attributes(R)})\ ) \wedge \\
&\qquad (\forall \text{b:attributes}(\underline{B}) \cdot (\nexists\ \text{a:attributes}(\underline{A}) \cdot \text{id(a)} = \text{id(b)}) \Rightarrow \text{b} \in \text{attributes(R)})\ ) \wedge \\
&\qquad \nexists\ \text{r : attributes(R)} \cdot (\nexists\ \text{a:attributes}(\underline{A}) \cdot \text{id(a)} = \text{id(r)}) \wedge (\nexists\ \text{b:attributes}(\underline{B}) \cdot \text{id(b)} = \text{id(r)})
\end{aligned}$$

$$\boxed{\texttt{join(}\underline{\texttt{A}}\texttt{, }\underline{\texttt{B}}\texttt{)}}$$

$$\text{R}$$

Semantics:

The *join* operator implements the relational *natural-join* operator, and allows two relations to be concatenated. Each tuple in the resultant relation is formed by the combination of each tuple from relation *A* with each tuple in relation *B* where the two tuples agree on the common attributes of *A* and *B*. Where *A* and *B* have no common attributes, the operator implements ‘cross-product’ semantics.

Relational Union / Intersection / Difference

Syntax:

```
<union_op>   ::= union( <expression >, <expression> )
<inter_op>   ::= intersection( <expression>, <expression> )
<diff_op>    ::= difference( <expression>, <expression> )
```

Type Rules:

$\underline{A} \in \text{RelationTypes} \wedge \underline{B} \in \text{RelationTypes} \wedge \exists R : \text{RelationTypes} \cdot$

$(\forall a{:}\text{attributes}(\underline{A})\ \exists b{:}\text{attributes}(\underline{B}) \cdot \text{id}(a) = \text{id}(b) \wedge$

$\exists t{:}\text{Types} \cdot \text{type}(a) \preccurlyeq t \wedge \text{type}(b) \preccurlyeq t \wedge$

$\nexists s{:}\text{Types} \cdot \text{type}(a) \preccurlyeq s \wedge \text{type}(b) \preccurlyeq s \wedge s \prec t$

$\wedge <t, \text{id}(a)> \in \text{attributes}(R)\ ) \wedge$

$(\ \forall b{:}\text{attributes}(\underline{B})\ \exists a{:}\text{attributes}(\underline{A}) \cdot \text{id}(a) = \text{id}(b)\ ) \wedge$

$(\ \nexists r{:}\text{attributes}(R) \cdot (\nexists a{:}\text{attributes}(\underline{A}) \cdot \text{id}(a) = \text{id}(r)) \wedge (\nexists b{:}\text{attributes}(\underline{B}) \cdot \text{id}(b) = \text{id}(r))\ )$

union($\underline{A}$, $\underline{B}$)
intersection($\underline{A}$, $\underline{B}$)
difference($\underline{A}$, $\underline{B}$)

R

Semantics:

The *union* operator takes two arguments and returns a relation value which contains all the tuples in the first argument and all the tuples in the second argument, and no others.

The *intersection* operator takes two arguments and returns a relation value which contains all the tuples which are present in both arguments, and no others.

The *difference* operator takes two arguments and returns a relation value which contains all the tuples which are in the first argument but not in the second argument, and no others.

Additional Operators

Syntax:

```
<tuple_extr> ::= extract ( <expression> )
<rel_count>  ::= count ( <expression> )
<member>     ::= ismember( <expression>, <expression> )
```

Type Rules:

$\underline{A} \in \text{RelationTypes}$

extract ($\underline{A}$)

tuple[attributes($\underline{A}$)]

$\underline{A} \in \text{RelationTypes}$

count ($\underline{A}$)

integer

$\underline{A} \in \text{TupleTypes} \wedge \underline{B} \in \text{RelationTypes} \cdot$

$(\ \text{A}a{:}\text{attributes}(\underline{A})\ \text{E}b{:}\text{attributes}(\underline{B}) \cdot \text{id}(a) = \text{id}(b) \wedge \text{type}(A) \preccurlyeq \text{type}(b)\ ) \wedge$

$\text{A}b{:}\text{attributes}(\underline{B})\ \text{E}a{:}\text{attributes}(\underline{A}) \cdot \text{id}(a) = \text{id}(b)$

ismember($\underline{A}$, $\underline{B}$)

boolean

Semantics:

The *count* operator returns the cardinality of a relation. The *extract* operator returns the tuple member of a relation of cardinality one. Invoking this operator over a relation which does not have a cardinality of one will result in a run-time error. The *ismember* operator returns whether or not the first argument is a member of the second.

3. IMPLEMENTING D^b

A sample implementation of the language is also required, which will feature the main components of D^b. Requirements for the implementation, presented first, before the main structure of the compiler is determined. The implementation stage is then discussed, which mainly concerns .NET aspects of the project.

Both the *Ellipse/Circle* and *Parallelogram/Rectangle/Rhombus/Square* examples used are taken from [Date & Darwen, 2000], as they are the examples which the authors use, and fit in well with the 'specialisation-by-constraint' type model.

3.1 Compiler Requirements

The implementation of the language shall be in the form of a compiler. The compiler shall take as input a string in the D^b language, and produce as output a .NET portable executable file which implements the input string.

The implementation must support a subset of the language, such that any features missing are orthogonal to those features which are supported.

The compiler must identify all syntactic and semantic errors in the input string. All errors must have a well-defined associated error code and an explanation of the error for the user.

All errors must be reported to the user in the standard format of:

```
        <FILENAME>(<LINE>,<COLUMN>) : error <ERRORCODE>: <EXPLANATION>
```

Where each error code will begin with "`DFLAT`" and be followed by a four-digit integer. Where possible, the compiler should attempt to recover from an error and continue compilation. Where recovery is not reliably possible, the compiler should terminate. If any error is found in the source description, the output file will not be generated.

If the user specifies the output to be a library it is an error for the *program* construct to be present in the source description. If the user specifies the output to be an executable it is an error for the *program* construct not to be present. The PE file will require no explicit setting of 'trust' to be executed, i.e. it must be capable of being proved verifiably type safe by the .NET tool `peverify.exe`. The output will primarily consist of up to six separate classes, each corresponding to one of:

`Functions`	contains methods which implement each function described in the source description.
`Procedures`	contains methods which implement each procedure described in the source description.
`Types`	contains classes which implement each type defined in the source description.
`Transactions`	contains methods which implement each transaction described in the source description.
`Databases`	contains classes which correspond to each database described in the source description.
`Program`	contains a single static method which defines the starting point of the application.

Arguments to the compiler may be specified by the user through the command line. Arguments supported must include:

`/IN:<FILENAME>`	Specifies the file to be used for input
`/OUT:<FILENAME>`	Specifies the name of the PE file to be created
`/TARGET:<TARGETTYPE>`	Specifies the type of output (`EXE` or `DLL`)

The /IN switch is mandatory.
The /OUT switch is optional : where not specified, the output file is the name of the input file without its extension, plus "`.EXE`" in the case of an executable target, and "`.DLL`" in the case of a library target.
The /TARGET switch is optional : where not specified, the target is assumed to be an executable.

3.2 Compiler Design

The design of the compiler is expressed in natural English, with structural overviews given in simple UML (Unified Modelling Language) to show the primary classes involved and the relationships between them.

Compiler Structure

Compilers can generally be split into two groups, multi-pass and single-pass. Multi-pass compilers are split into well-defined components, each of which creates a representation of the original source description. The lexical analyser splits the source description into lexemes, which are used by the parser to create a complete parse tree. The semantic analyser assigns meaning to this parse tree and creates an equivalent representation in an intermediate language, from which the final output is created by the code generator. Each stage passes through the entire source description (in whatever form).

Parser-driven compilers have the parser as the main component, which calls upon the lexical analyser, semantic analyser and code generation components as required. This leads to less overhead in the components communicating and is often faster, as only one pass is required. Languages which force force "declaration before use" lend themselves well to this technique. As D^b is one such language, the compiler will be parser-driven.

Lexical Analysis

The scanner is the interface between the parser and the user's textual representation of the component to be compiled. Its primary purpose is to group characters from the source description into lexemes (or tokens), which represent the source description. Each lexeme consists of a lexeme type and a value, which is used to denote which particular instance of a type this lexeme refers to. A '*Scanner*' class will perform the operations necessary to generate lexemes from a particular character stream. The actual lexeme splitting action can be implemented in terms of deterministic finite state automata which embody the definition of the regular expressions defining the lexemes. The state transition diagram which identifies identifiers as well as integer and floating-point constants is show in figure [1].

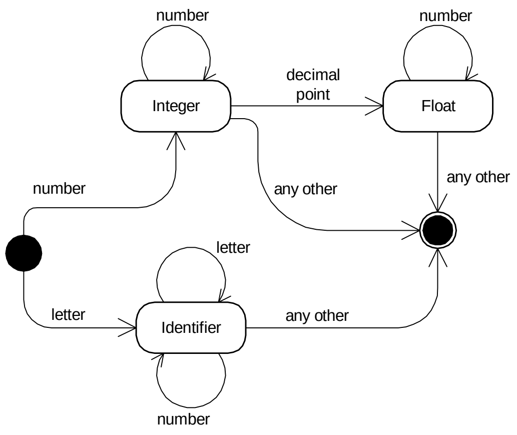


Figure 1: Basic Lexical Analyser Finite State Machine

As there are areas where the type of an expression will not be known immediately, a '*LexemeStore*' class will act as a supplier of lexemes which have already been read. This allows the expressions to be parsed when additional information has been added. To support the use of placeholder identifiers in expressions, the '*LexemeProxy*' class will act as a converter, converting specific lexemes to others as they are found. This allows normal expression-parsing techniques to be used over expressions using placeholders such as *value*.

The final conceptual design for the scanner module appears as figure [2].

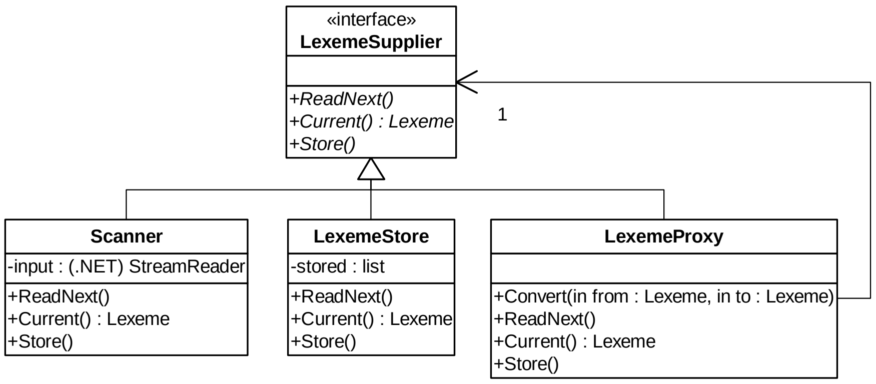


Figure 2: Lexical Analyser Classes

Parser classes

The parser's main task is to determine the structure of the source description by analysing the sequence of lexemes from the scanner module. It does this by applying the grammar rules of the language. If a string does not fit into available grammar rules then the string is not valid in the language. Otherwise the source description will be assigned a meaning depending on its structure.

As D^b has the beneficial properties of being non-left-recursive and LL1 (indicating that only the next symbol need be examined to determine the next grammar production), a simple, efficient and effective method of parsing is top-down recursive descent. The design rule for the parser classes will be to use internal methods for parsing items from a specific grammar rule, and to create a new parser class when that grammar rule can appear elsewhere. This allows parser classes of common syntactic categories to be re-used among other parser classes.

All parser classes will be reading tokens of some sort, and the grammar rules insist on a particular sequence. A base class 'Parser' will provide helper methods to subclasses. Implementations for other main syntactic categories will be subclasses of 'Parser'. 'StatementParser' acts as a superclass for the different types of statement bodies that can be parsed. Each of the main abstraction mechanisms (functions, procedures etc) have a corresponding implementation to ensure statements legal in their respective bodies are handled correctly. The structure of the parser classes is shown in figure [3].

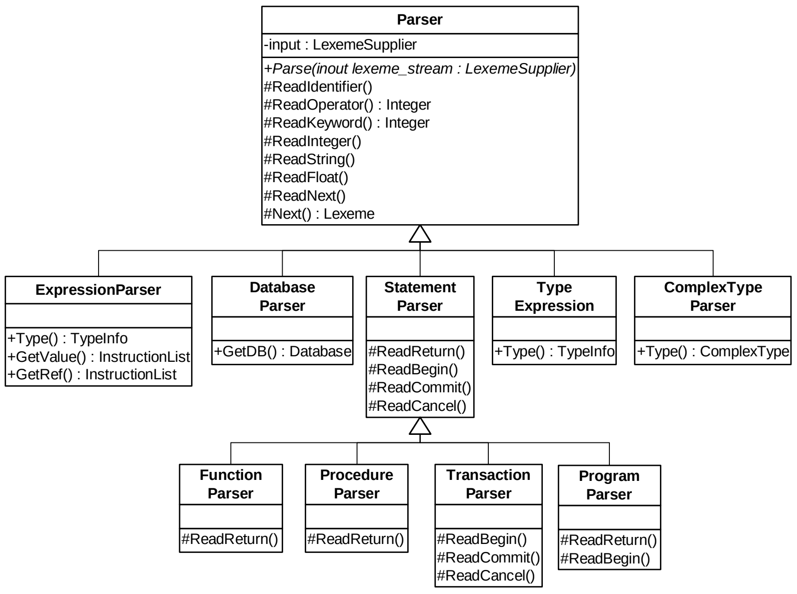


Figure 3: Parser Classes

Semantic Classes

One of the principle functions the compiler performs is enforcing type safety. This entails keeping track of the type hierarchy. A number of classes are required to model the type system of D^b, as shown in figure [4].

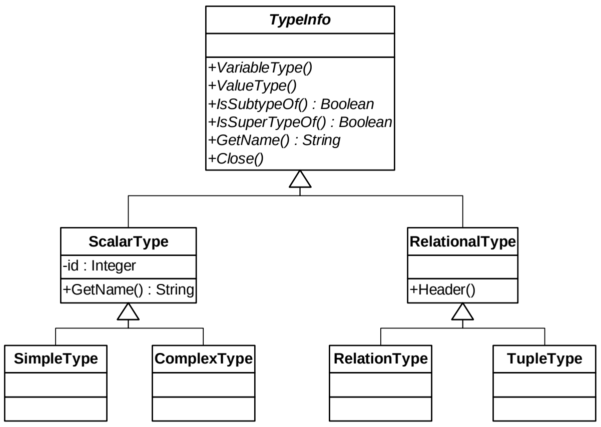


Figure 4: D Type Classes

All types are subtypes of the *TypeInfo* class. All types have a name which can be accessed via the *GetName* method. The result is returned as a string, and is primarily used for displaying to the user during error handling. Every type can also be tested to see if it is a subtype or supertype of another, which is essential for many type-compatibility tests. The final method *Close* allows for final checks to be performed before the respective types in the output are completed. For instance, a complex type ensures that every pair of non-distinct subtypes have a common subtype. This final check must be performed after the user has been given the chance to define such subtypes.

Each *ScalarType* is identified by an identifier, and have standard routines for handling subtypes and supertypes. User-defined simple types have not yet been implemented, and as such each *SimpleType* is merely an alias for a type which already exists and is supplied by the .NET Framework. Meanwhile each *ComplexType* relates to a user-defined complex type and has four respective classes, each corresponding to those required in the final output.

Each instance of *RelationalType* is either a tuple or relation type, but either has a *RelationalHeader* which can be accessed to determine whether particular relational actions are valid. The implementations of *TupleType* and *RelationType* handle the implementation in the final output of their specific instances.

In order to enforce strong type checking, the compiler must ensure that all operations are well-defined over the type of their arguments. To achieve this, expressions use a stack which mimics the .NET runtime stack: but instead of using values, it uses types. By checking the types of objects at the top of the stack, as each operation is found and 'executed', the types of the arguments are checked against a list of types which the operation accepts. If a match is found, then the arguments are popped off the stack and the result type of the particular operation is added. If no match is found, then a type error has occurred. Each class can generate instructions for .NET operations as required. The hierarchy of classes used is shown in figure [5].

The evaluation stack contains zero or more objects of type *StackEntry*. This abstract base class contains operations to determine the local variable the entry corresponds to, whether or not it can be updated, and the underlying type. *ValueEntry* is a subclass which stands for entries which cannot be updated. *LocalVariable* and *VariableParameter* stand for local variables and update parameters respectively. *ComponentEntry* stands for instances of type "x.y" where x can be a tuple or complex object. A component of an object shares attributes with its parent object, such as mutability and the base local, and so has a reference to its parent object.

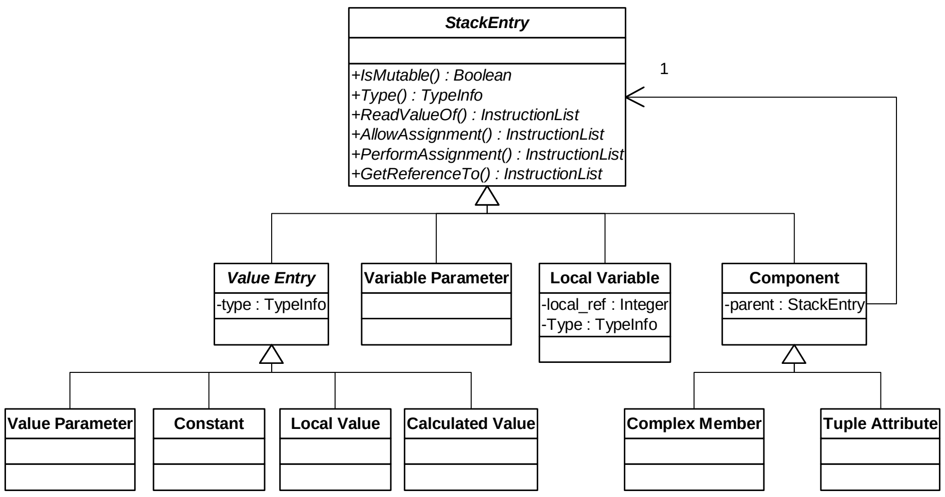


Figure 5: Expression Stack Classes

StackEntry also contains three methods which return instructions to be executed in order to perform a particular type of operations. *ReadValueOf* will ensure that the object at the top of the execution stack during execution is the value of the corresponding object. The *GetReferenceTo* method will ensure that a reference to the current object is at the top of the stack. *AllowAssignment* sets up the stack such that a value can be assigned to the current object later, and *AssignTo* perform the actual assignment. The reason for these two methods being separate is due to the .NET Common Intermediate Language method of assignment. Instead of pushing the target and then the value to the stack and performing an assignment operation, the value is pushed onto the stack and the assignment operation specifies the target. For example, the sequence of instructions in assigning the value '3' to 'x.y' is "`push x`, `push 3`, `stfld y`". Different stack entries will require different operations to be executed before the expression is evaluated: these operations are fetched through the *AllowAssignment* method.

Code Generation

In a typical compiler, code generation takes the form of two stages: intermediate code generation and output code generation. Intermediate code is usually simple, and machine-independent. Output code is generated from the conversion of the intermediate instruction sequence to the instruction set of the target machine. This structure separates the front-end of the compiler (language-specific) from the back-end (machine-specific). Optimisations may be made to the instructions at the intermediate or output stages.

The implementation abstracts over the intermediate instruction list by creating an *InstructionList* class. This has a number of methods in its interface, which can be used to add simple instructions to the list. Instances of *InstructionList* can be compiled to given a .NET CIL generator (provided by the 'Reflection Emit' section of the .NET Framework). Each type of instruction in turn can compile itself to the CIL generator. A further abstraction is made in the form of a similar class, *StatementList*, which represents procedure, function, transaction and program definitions. Each statement in D^b has a corresponding class, which can compile itself to an *InstructionList* instance.

As efficiency is not a priority of the sample compiler, no optimisation phase is currently implemented. Such a phase could be added later at both statement and instruction levels in both of these classes, inside the *CompileTo* method.

3.3 Compiler Implementation

The .NET Type System

The .NET type system is very different from the type system prescribed by the 'D' model. In .NET, there is a clear division in the type system between value types and reference types. Value types are designed to be small and lightweight, and lack many of the features available to reference types. They are copied by value cannot have subtypes or supertypes.

Reference types can be classes or interfaces. Interfaces provide no implementation code and purely define the publicly accessible sections of those classes which implement them. Classes always inherit from one other class (with the exception of *object*, the superclass of all others), but may implement multiple interfaces. In .NET, subclasses inherit all of the structure and operations of their parent. There is no separation of subtyping and inheritance, so all inherited classes must be treated as subtypes (various restrictions are enforced on the subclass to ensure this). The method to be invoked by an object is determined by the message and the exact type of the object the message is sent to – the decision is not affected by the types of the arguments of the message. Reference types are always passed by reference.

Obvious problems are immediately apparent: the 'D' inheritance model requires multiple inheritance. .NET also has no (built-in) concept of specialisation by constraint, nor does it support multi-methods. All these problems have to be solved in order to build a D^b implementation which targets the .NET platform.

Run-time Errors

There are six errors which can occur during the execution of the output:

1. Invalid Tuple Join – a join of two tuples is attempted and the values of common attributes do not match.
2. Invalid Tuple Extraction – an attempt is made to extract a tuple from a relation when the relation has more than one tuple.
3. Invalid Type Cast – an attempt was made to downcast a value of a type to a subtype when the value is not a member of the subtype.
4. Invalid Database Data – an error occurred while attempting to retrieve the values of database relations from persistent storage.
5. Transaction Rollback – a transaction called from an external component failed.
6. Out Of Range – An attempt was made to access an element of an array outwith the bounds of the array.

If these occur in a user program, the result is the termination of the program together with a suitable error message. If however they occur as a result of an external .NET component accessing the output file, then the error must be signalled back to that component somehow. The .NET exception mechanism can be used for this task. When an error is detected, a new object is created and 'thrown'. It is the responsibility of the caller to 'catch' such errors. The 'OutOfRange' exception is thrown by the .NET String class and so the compiled code can just let this exception 'fall through'.

A class for each of the remaining five errors could be created in each output PE file. This is counter-intuitive however as the errors are not specific to the source description. It might also lead to confusion over different exception objects when a software component accesses more than one D^b assembly. Instead these errors are collected together and implemented as a .NET library, `dlib.dll`. This library contains all exception classes for run-time errors, so that all output files throw the same exceptions. The downside to this is that all .NET components which access a D^b assembly must also reference `dlib.dll`.

Complex Type Implementation

Types in D^b may be inherited from multiple types. As .NET does not allow a class to inherit from multiple parent classes, D^b types must be implemented in terms of interfaces. Each type may have more than one possible representation, which can then be implemented in terms of the type's interface.

This approach allows representations of types to be defined in other languages, with the compiler merely suppling a representation which matches the complex definition. A simplified version of the *Circle/Ellipse* example will be used for the next few examples, with the types being declared as:

```
complex type Ellipse
{
        float a;
        float b;
}

complex type Circle : Ellipse when value.a = value.b
{
        float r;
}
generalisation Ellipse(value.r, value.r)
specialisation Circle(Ellipse.a);
```

A subtype may have components defined which share the same identifiers as components in the supertype. To protect against possible ambiguity, each method to retrieve components from representations is prefixed with the type name. This allows representations to correctly answer requests for component values.

The .NET type system does not directly support specialisation by constraint, so additional classes must be added in order to achieve the same effect. The compiler generates an addition class (the 'value holder') for each type. These are effectively wrapper classes, adding a level of indirection between a program wishing to use values of the type and the actual representation of each individual value.

Each holder class supports methods to access all components of their corresponding D^b type – the class delegates responsibility of accessing components to a representation class satisfying the corresponding interface. On creation of a new holder object, the most specific type of the value is calculated by the holder class. The holder class also contains methods for testing if a value of a type is of a subtype and for 'down-casting' – creating a value of a subtype from a value of a supertype, as prescribed by the 'D' model. Finally, each holder class contains a member which returns the underlying value. This is required solely for the implementation of specialisation by constraint, and is inaccessible to external components. An overview of the implementation of the *ellipse*/*circle* type hierarchy is shown in figure [6].

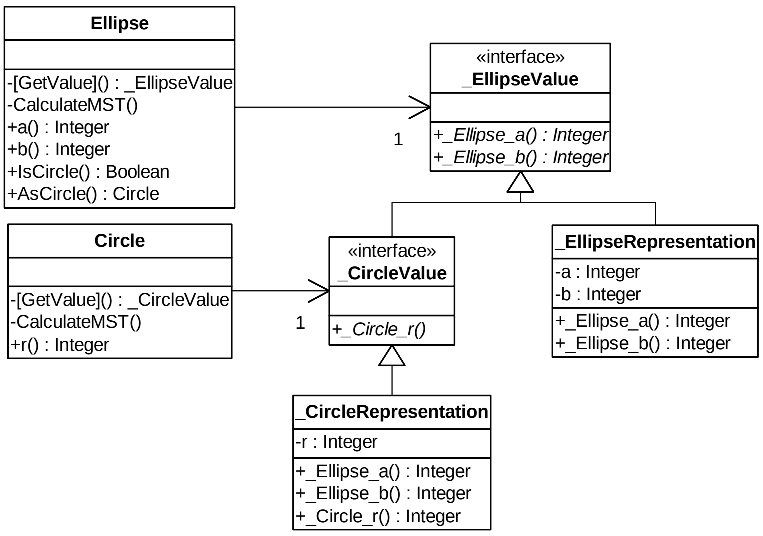


Figure 6: Implementation of Ellipse/Circle Values

The temptation to have holder classes inherit from each other must be avoided, as this doesn't support multiple inheritance. Having a similar interface method as the representation classes almost works, but means that values of subtypes would appear to inherit the same structure as their supertypes. For example, consider the following C# source snippet:

```
Circle c = new Circle(3.0f);
float area = c.a * c.b * PI; // ILLEGAL: Circles have no ‘a’ or ‘b’ components
```

This is a consequence of subtyping and inheritance being inextricably linked in .NET. To counteract this, the compiler will define implicit conversions for values to values of their supertypes. The holder type also provides the ability to have the representation of its value supplied to it, rather than each component being expressed separately. This allows external .NET components to define representations for D^b types, albeit in a rather limited fashion (no externally-defined representations will be used to represent the result of calculations, for instance).

Once a holder type has been created, its value is constant. Additional classes must be provided to store variables of each type. These are fairly simple classes and consist of accessors (where each accessor merely delegates responsibility to its corresponding holder object) and mutators (which create a new holder type based on a supplied argument and the previous value of the variable).

Classes corresponding to variables must have all their components as variables also. It is very important that after a component has been changed, the variable re-evaluates its most specific type. The compiler can take care of this trivially in the case of compiling D programs, and the variable classes take care of this when used in other .NET programs. The variable classes have operators overloaded such that they implement by-value semantics when used in other .NET languages.

Implementing “Specialisation by Constraint” on Complex Types

Implementing specialisation by constraint requires that the representation of values be altered depending on the value of their components. The main challenge is allowing the representation to be altered after the holder type has been created, i.e. the *CalculateMST* method.

The implementation of *CalculateMST* is done in two ways, depending on the position of the type in the type hierarchy. If the type *T* has no siblings (either base types or which are sole subtypes of their parent(s)) then the *IsC* test is performed for each child type *C*. If the *IsC* test evaluates to true, then a holder type of *C* is created according to the user-defined specialisation expression, then the value of the new value of *C* is copied into the current object. This method deteriorates to a simple ‘no operation’ for types with no subtypes. The C# equivalent for the *CalculateMST* method of *Ellipse* is shown below as an example:

```
private void Ellipse::CalculateMST()
{
      if ( MSTCheck_IsCircle() )
      {
            value = Circle.Specialisation(this);
      }
}
```

Note that *MSTCheck_* functions are used during specialisation by constraint, and it is these functions which implement the corresponding subtyping constraints. The *Is_* functions perform a .NET type test on the type of the value the holder is pointing to, which is significantly more efficient. The *Circle.Specialisation* method is implemented as (again, C# equivalent):

```
internal static CircleValue Circle::Specialisation(Ellipse e)
{
      return new Circle(e.a, e.pos).value;
}
```

Thus if *Circle* itself has a subtype, then the most specific type of the specialised *Circle* will be evaluated within the constructor call, and so the ‘value’ field in the original object of type *Ellipse* will always be of the most specific representation.

This method needs slightly amended when extended to multiple-inheritance. Consider the type hierarchy of some shapes depicted in figure [7].

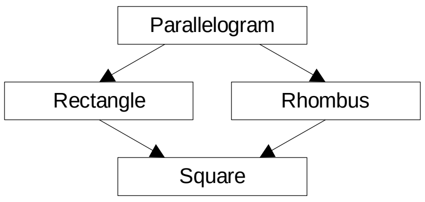


Figure 7: Parallelogram Type Hierarchy

The *CalculateMST* method in type *Rectangle* will have a test to see if the object is of type *Square*. The test will involve checking if the object is also of type *Rhombus*. To do this, it is necessary to convert the type to a *Parallelogram* and then check if that value is also of type *Rhombus*. Implementing this as a simple constructor call leads to an infinitely recursive loop (the *Parallelogram* constructor specialises to a *Rectangle* which converts to *Parallelogram*, etc). Another constructor (accessible only to compiler-generated methods) is created in *Parallelogram* which takes an object of type *ParallelogramValue* and which does not perform specialisation of its value – a "lazy" constructor. Its sole purpose is the provision of this "checking if a value is also a sibling" operation. The *MSTCheck_IsRhombus* method can then be invoked, and if the result is true, the *Rectangle* should specialise its value to a *Square*.

Here lies another potential problem. The specialisation function for *Square* takes two parameters, of types *Rectangle* and *Rhombus*. To create the *Rhombus* value requires a specialisation from *Parallelogram* to *Rhombus*. However the specialisation function also re-calculates the most-specific type of the value (e.g. *Circle* example above). For *Rhombus* this involves testing if the value is of type *Square*, and another infinitely recursive loop appears. To solve this, another specialisation function is required which does not perform further specialisation of its value – a "lazy" specialisation. Again, its sole purpose is to allow the implementation of "specialisation by constraint" over types with siblings. With sibling types having an added specialisation function and the parent of multiple subtypes having a lazy constructor, specialisation by constraint can be implemented in .NET such that it is invisible to the users of the holder classes involved.

Implementing Relational Types

Each relational type with a different relational header expressed in the source description has a corresponding class in the output file. Where two relational types are specified such that their relational headers differ only by the order of attributes, they are implemented as the same type, with the compiler performing the re-ordering as required.

Tuple type implementations are all derived from a single base class `D.Tuple`, which is located in the `dlib` library. This class has no publicly accessible members. All tuple classes consist of accessors and modifiers to their members, as well as methods which implicitly convert up to supertypes, and which cast down to subtypes. As the most specific type of tuple values depends entirely on the most specific type of each of its attributes, specialisation by constraint does not require any extra classes.

Relation type implementations are all derived from a single base class `D.Relation`, which is also located in the `dlib` library. This class offers inherited classes (those relation types referenced in the source description) functionality for implementing relations, i.e. adding and retrieving tuples etc. Derived classes must implement type tests, to allow multi-method semantics on methods with relation parameter types. The advantage of forcing all such relations to use a common base type is that the implementation of the tuple bodies of relations can be altered while causing no disruption to existing relation types.

At present relations are implemented as a hash table, with the hash function taking into account the value of all attributes in the tuple. In future the candidate keys of the relation could be implemented as indexes to the tuple bodies. Efficient ways to implement candidate keys which involve attributes of complex, tuple or even relation types would have to be investigated.

Relational types are severely limited when used in other .NET languages. Relation operators such as *join* cannot be used, as to do so would require the type hierarchy to be built into the compiled file somehow. Implementing a *TypeSystem* class with each output leads to potential problems when a .NET component attempts to perform relational operations on types from each assembly. Allowing some form of generic tuple or relation-type to be created by external .NET assemblies would alleviate the problem, and should be possible in a more interpreted solution, such as when D^b is implemented in a DBMS.

Tuple-valued relational operators are implemented inline, that is to say that the compiler inserts the instructions required directly into the instruction stream. Relation-valued operators which require loops (such as *join*) are implemented as calls functions which implement the required semantics. As the instruction sequences for these operations are so much larger, this cuts down on the output file size. Such internally-generated methods have *private* accessibility and cannot be accessed from other .NET components.

Implementing Multi-Methods

Languages which implement multi-methods allow the exact method to be executed based on the message and the closest match of the exact types of the arguments to the types of parameters. To emulate this in .NET, logical tests must be added to each method implementation so that control is passed to other methods if the arguments are of suitable types. (The remainder of this explanation will deal with functions – procedure implementation is similar but with the added complication of the types of variable parameters having to be equal in order for two procedures to be considered non-distinct).

To implement this, functions of the same name which have compatible parameter types are grouped together. They are then arranged into a lattice such that the function with the most-specific types for parameters as the root node, and the functions with the least-specific parameter types as leaf nodes. For instance, consider the following function declarations:

```
float function f(Ellipse value a, Circle value b);
float function f(Circle value a, Ellipse value b);
float function f(Circle value a, Circle value b);    # required!
```

These declarations (where *Ellipse* and *Rectangle* are subtypes of *Shape*, and *Circle* a subtype of *Ellipse*) would lead to the tree show in figure [8]. The tests each function must consider are determined by the type of parameters of its parent.

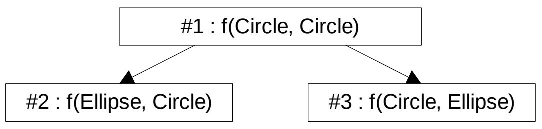


Figure 8: Function Hierarchy

Here function #2 takes arguments of type *Ellipse, Circle*. Its parent (function #1) takes arguments of type *Circle, Circle*. The argument list differs by the type of argument *a,* therefore the definition of function #2 after multi-method calculation would look something similar to:

```
float function f(Ellipse value a, Circle value b)
{
        if ( a.IsCircle() )
                return f(a.AsCircle(), b);
        # rest of function body here, as defined by user
}
```

As the 'value holder' classes for complex types have been implemented in terms of distinct .NET classes, the overloading mechanism of .NET is sufficient for the above technique to implement multi-method semantics.

Implementing Transactions

Transactions must either succeed completely (successfully terminate) or have no action. This is implemented by having each transaction correspond to a .NET method. The *commit* statement is implemented as transforming the stacks for all relations in the database from [… A B] to [… B], where

B is the copy of the relation for the transaction. The *rollback* statement is implemented as popping the top value off all database stacks and throwing an exception.

The extended form of the *begin* statement (with the addition *onfailure* statement block) is implemented as a try…catch block. If the transaction that is started fails, then the exception will be caught here and the *onfailure* statement list will be executed. If the short version of the *begin* statement is used, then any child transactions failing leads to the failure of the current transaction too. This transaction will implicitly throw the exception as well, to be handled by the parent transaction or program. In the case of the transaction being called from an external .NET program, it is the responsibility of the program to catch the exception.

Implementing Databases

All databases defined in the source description have a corresponding class in the final output. This class has a series of static fields which are implemented as stacks. Each of these corresponds to a particular relation in the database. When a transaction over a particular database is started, copies of all the relations are made. That transaction will then be manipulating only the values at the top of the stack. If the transaction fails, then the value at the top of all the stacks will be discarded. If the transactions commits, then the value at the top of the stack will replace the value underneath it (ie in the transaction which called it). If the stack has one entry and the transaction commits, the values of the relations are persistently stored.

Persistence is implemented by allowing all types to be able to write their values to an output stream, and recreate values from an input stream. They all follow a common format, which is XML (eXtendable Markup Language, [URL 4]). All types are capable of this and it allows persistence of databases to be implemented in a straightforward manner.

Each database has a corresponding database file, which is located in the same directory as the output file (more accurately, the database file will be considered to be in the same directory as the output file is executed from). When the first relation is requested by a transaction, the values of all relations in the database are loaded from the corresponding database file. When the last transaction commits, all relations are serialised to the database file. A sample database file is shown in figure [9].

```
<?xml version="1.0"?>
<database db="myDatabase">
    <relation id="shapes" typename="relation{Ellipse e}">
        <tuple>
            <complex typename="Ellipse" id="e">
                <float id="a">1.5</float>
                <float id="b">2.7</float>
                <complex typename="Position" id="centre">
                    <float id="x">0.0</float>
                    <float id="y">9.0</float>
                </complex>
            </complex>
        </tuple>
    </relation>

    <relation id="students" typename="relation{Student s}">
        <tuple>
            <complex typename="Student" id="s">
                <integer>20</integer>
                <string id="name">"Peter Nicol"</string>
            </complex>
        </tuple>
    </relation>
</database>
```

Figure 9: Sample Database XML File (indentation added for presentation)

4. EVALUATION

Implementation Testing

The implementation consists largely of those sections which are well-defined (namely the lexical analysis and parser sections) and those sections which are not (areas which deal with semantics). Well-defined sections can be tested to ensure the implementation meets its specification. Other areas are harder to test, due to the lack of a complete specification. The testing procedures will therefore be split along the syntax/semantic divide.

The lexical analyser class can be tested by supplying a series of strings and checking to ensure that the lexemes supplied correspond to the input string. As the series of strings the scanner must process is potentially infinite, the strings must be chosen by hand to mimic certain scenarios. The strings are chosen to check that each type of lexeme is being detected, and that combinations of different types of lexeme are also detected correctly (when in a string together). This is not an exhaustive test, and it is entirely possible that the lexical analyser fails to recognise an extreme case, such as an identifier a million characters long. However the tests, coupled with the systematic implementation, should prove adequate.

The parsing classes could be tested by ensuring that they detect all grammar rule infringements. While this is in theory possible, the number of tests involved makes it prohibitive. The sequences of lexemes were instead chosen by a human, designed to be indicative of the main legal uses and the main errors. If the grammar rule meets the lexeme stream, the parser is expected to accept it or signal an "end of file" error (in the case of the lexeme stream being shorter than the grammar rule). If it doesn't meet the grammar rule, then the parser should give some other error.

Where type-rules are associated with grammar rules, they define which syntactically valid strings have a definite meaning in the language. That the compiler enforces these can be tested by supplying expressions to the corresponding parser, such that the rules are either broken (in which case the parser should return an error) or obeyed (accept the string). The set of type tests for the *join* operator is given in figure [10].

join(A, B) => Result X = Any type.

A	B	Rationale	Result
[any simple]	X	Not defined over non-relational types	Error
[any complex]	X	Not defined over non-relational types	Error
tuple{float a}	tuple{float a}	Correct over equal attribute sets	tuple{float a}
relation{float a}	relation{float a}	Correct over equal attribute sets	relation{float a}
tuple{float a}	tuple{integer a}	Common attributes have distinct types	Error
relation{float a}	relation{integer a}	Common attributes have distinct types	Error
tuple{float a, float b}	tuple{float a, float c}	Different header with common attributes	tuple{float a, float b, float c}
relation{float a, float b}	relation{float a, float c}	Different header with common attributes	relation{float a, float b, float c}
tuple{float a}	tuple{float b}	Different header, no common attributes	tuple{float a, float b}
relation{float a}	relation{float b}	Different header, no common attributes	relation{float a, float b}
tuple{float a}	relation{float a}	Cannot join relation+tuple values	Error
tuple{}	tuple{}	Correct over header with zero attributes	tuple{}
relation{}	relation{}	Correct over header with zero attributes	relation{}

Figure 10 : Type tests for the *join* operator.

The other main components of the language deal are harder to test. For example the code generation stage produces an object of *InstructionList,* the result of which may be to calculate a value. There are no hard constraints on how the *InstructionList* actually calculates that result, the order of term evaluation etc, as long as the terms are evaluated correctly. The given list is not required to be 'optimal' in any sense, and so cannot be tested through equality. Instead a series of tests are defined for each construct in the language, such that the execution of these tests can determine whether the semantics are implemented as required. To aid this, the implementation adds an output statement, *print*, which takes a single expression as argument and displays its type and value in the console. This allows control flow to be traced and values to be tracked. This allows the observer to determine which most-specific function has been chosen, for example.

Examples of the data sets used are presented in Appendix B.

Implementation Evaluation

The implementation is in the form of a compiler which accepts a subset of D^b as input and generates .NET-compatible output. The sections of the language which are not accepted were chosen in order that the missing features are orthogonal to the other aspects of the language which are available.

Candidate keys over relation types cannot currently be processed. If the implementation accepted these, it would also have to force all tuple values in relations to be uniquely identifiable by the combination of those keys. As relational operators were implemented in a rather naïve and simple manner, this would have required that each tuple be compared with other tuples already in the relation and tested on the combination of attributes in each key. More advanced techniques for implementing relations have features such as indexes which are used to find where each tuple will be located. Finding the location of the tuple and maintaining key integrity could be linked together in such a system at a much lower cost.

Virtual relations in databases were not implemented either – in order to be worthwhile, they must be accessed as though they were real relations, i.e. be updateable. The implementation of this appears to rely heavily on the concept of functional dependencies and the relationship between relations (including 'foreign keys'). As such keys are not fully supported, it did not seem worthwhile implementing virtual relations.

User-defined *simple types* have not been implemented as there was simply not enough time. Allowing such type definitions would have complicated the compiler and raised questions about how 'standard' the standard types of D^b are. Specialisation by constraint currently requires that the implementation of the base type 'know' about its subtypes and the constraints between them. If the simple types of D^b were allowed to be modified in some way by a particular component then the types would require different implementations between different .NET assemblies – a peculiar notion. Instead user-defined simple types aren't implemented, and the built-in simple types correspond directly to .NET types, which makes the output more accessible to external components.

Persistent data is supported through each database having a corresponding XML file. The component of each value is stored with it such that values can be re-constructed as they were before. An improvement would be to allow the type system to persist also, such that as the values are restored, their respective types are loaded by .NET from their respective assemblies. This would ensure that the store was type-safe. A similar persistent store is implemented in [Harrison & Naeem, 2000].

Currently the implementation does not distinguish between distinct types which have a common supertype – i.e. it assumes all subtypes are non-distinct. This forces all pairs of subtypes to have a common subtype, causing an exponential rate of growth in the number of extra subtypes which must be defined, as shown in figure [11].

Immediate Subtypes	Extra Subtypes Required
1	0
2	1
3	4
4	11
5	26
6	57
7	120
…	…
n	$(\sum_{r=1}^{r=n} {}^{n}C_{r}) - n$

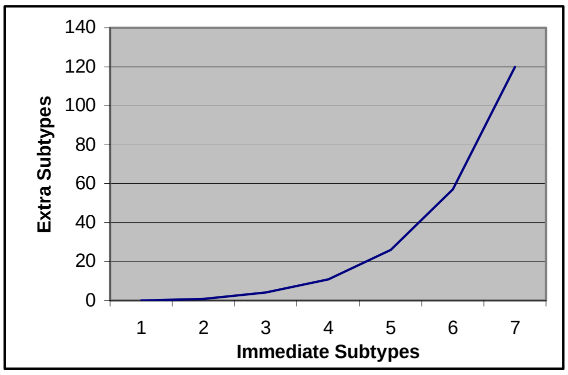


Figure 11: Extra subtypes required for non-distinct subtypes

An improvement would be to statically analyse the constraints to determine which pairs are non-distinct. This approach would not identify all distinct types (for example the constraint may be a user-defined boolean function for which no inverse is available), and it might well be more effective to allow users to simply specify which subtypes are distinct from others.

Language Evaluation

The language can be evaluated against the requirements of the 'D' model and the language design principles described previously. D^b seems to satisfy all prescriptions of the relational aspects of the model, although there is a lack of support for the 'dropping' of types, functions etc, and no support transactions dynamically adding and removing relational variables from the database. Such mechanisms could possibly be added in if D^b is implemented as a DBMS in the future.

D^b does not follow the *Principle of Locality*. It may be possible to allow functions, procedures etc to be defined locally, however the implementation of locally defined types which are part of some larger type hierarchy may prove difficult. A solution would greatly improve the language. The *Principle of Type Completeness* is largely followed, with the (enforced) exception of only relations being stored in the database, contrary to the common notion of "persistence orthogonal to type" [ref]. There is a slight issue in only built-in types can be *generated*, i.e. relation and tuple types are parameterised by types. A future addition to the language should allow user-defined types to be similarly parameterised and thus offer *parametric polymorphism.* Abstraction mechanisms are provided over statements and expressions, through procedures and functions respectively. Abstractions over declarations are also useful and provide for the division of a software system into smaller components. These could perhaps be added in such a way to leverage the .NET 'namespace' system.

The principle of locality is supported only in a very limited way. To allow function declarations (for just one example) to be local would require other functions to be duplicated and called during the local scope. For example, f may be of type $T \rightarrow S$ and be defined globally. If f is defined again locally to be of type $T' \rightarrow S'$, then all calls to f with arguments of type T would have to be tested to see if the local function was required : if not, then the original f would be invoked again. The issue is compounded with more complex function hierarchies. For this reason it is not currently supported.

The language manages to keep its syntax rules fairly low in number through re-using grammar components such <*expression*>, which helps keep the language as general as possible. Statement blocks are suitable everywhere a statement is expected, allowing statements to be used together or alone, as required. Using statements or relational operators together does not affect their individual semantics, and so the language can be said to be *orthogonal* in this regard.

Project Evaluation

The first stage of implementation was the creation of a prototype interpreter for a subset of the "Tutorial D" language, as described in [Date & Darwen, 2000]. This was written in C++ and was very basic, but did allow a restricted form of specialisation by constraint (no multiple supertypes) and multi-method matching. This allowed the requirements to be further investigated, and worked well as several issues were uncovered through this very basic implementation.

A preliminary language was then created. This attempted to contain the main aspects of the final language according to the requirements. Unfortunately as the language was constantly re-evaluated against various language design principles over time, changes had to be made. This led to the speed of implementation being dramatically reduced.

The implementation proper consisted of the construction of a compiler for the language. This was created in C#, chosen as it is the standard language of .NET and supports library classes which directly support compilers and interpreters targeting the .NET Framework. The main problem was again time, as C# and .NET are relatively new technologies (.NET 1.0 was released just two months prior to project completion) which had to be learnt, slowing progress. As a result several planned features were not implemented.

In the planning of future projects, more time would be allocated to perform tasks which required the development of a new skill or new technology. More accurate methods for specification (however non-formal) would be introduced at a much earlier stage in the project.

5. CONCLUSIONS

During the course of this project, the “D” language model was examined and a language was created which sought to adhere to its requirements. A preliminary language was created which was then iteratively refined by re-evaluation against language design principles. A compiler was created which implements a subset of the language, featuring the main components of the “D” model and creating .NET compatible output.

The ‘D’ model prescribes that it be possible to write full standalone software components in the language. The decision was therefore made for D^b to be imperative, rather than the declarative style of other relational database languages. A chief problem resulting from this decision was the integration of the relational operators into an imperative style. D^b allows relational operations to be expressed as functions, with attributes and expressions expressed between the operator name and the arguments. Such syntax is not completely satisfactory, and it was noted that allowing functions to be treated as values would help alleviate the problem, as well as making the language more powerful. It is, in hindsight, feared that such a style may limit the usefulness of the language, as many developers have existing expertise in SQL and its more declarative style. An interesting avenue of investigation would be to determine how ‘usable’ existing SQL users find D^b, perhaps by using tests such as those described in [Reisner, 1981].

D^b features a separate language construct for transactions, so that the header of each abstraction mechanism can act as a contract between the caller and the callee – a property which the author believes to be highly beneficial. It was noted that a future addition to the language of a construct which accesses a database but does not update it may also prove useful.

D^b also features the type inheritance model described in [Date & Darwen, 2000]. While the benefits of constructing a language which learns from SQL’s mistakes seem clear, various question marks remain over the type system proposed. A subtyping relationship states that instances of one type may be substituted in any context where an instance of its parent type is expected. In the ‘D’ type system, this holds true only for values, and so it seems that the type system involved features *value inheritance* rather than *type inheritance*.

In object-oriented terminology, a class consists of attributes, where attributes may include the equivalent of functions or procedures, as well as members. The term *inheritance* generally refers to the ability of a subclass to be defined in terms of another class, while having the capability to modify and extend its attributes. In the ‘D’ type system, while a subtype can certainly modify the operations defined over the supertype, it is not possible to extend the structure of the supertype. All attributes in the subtype must be fully derivable from the supertype, so no additional information can be added. It is concluded that the form of *inheritance* used in the ‘D’ model is rather different from *inheritance* as it is defined in the wider object-oriented world. While it is fully possible that the author has simply not made the necessary ‘paradigm shift’, it seems unlikely that a type system where the structure of the supertype restricts the subtype can be truly useful in the development of large-scale software.

6. FURTHER WORK

Model Extensions

It would be interesting to be able to have a more general type system, where functions are also treated as values (and indeed types as values, as promoted in [Donahue & Demers, 1985]). This introduces a question of function subtypes. At present, functions which are *less specific* than another are substitutable for their *more specific* counterparts if and only if they have the same return type, with *more-specific* functions never being substitutable for *less-specific* versions.

If values of function types are to be treated equally within the 'D' model, they must have 'most-specific' types. The most-specific type of any value is the type which defines the smallest set of values of which that value is a member. This could be considered to be the type of the function which the user associates with it during declaration. However, consider:

```
Ellipse function X(integer value v)
{
        return Circle(v);
}
```

Here the function (value) X is of *declared* type *integer → Ellipse*, but the *most specific* type is clearly *integer → Circle*. By considering the declared types of values returned from the function, the most specific type is found.

Such problems as the storage of functions in the database have not yet been considered in great detail. If a way is found which facilitates the treatment of functions as values, then it would allow less awkward syntax for relational operators, e.g. `restrict:t(rel, t.x < 3)`. Braces after relational operators would then be reserved for listing attributes, rather than attributes and expressions. It would also make the language much more powerful, as shown below.

Language Developments

Currently, the only way to change the value of a variable is through the assignment statement. This may in some cases lead to somewhat convoluted code. For example, the following expression updates a specific tuple in a relation:

```
database.JobAllocations := union(
    update:u{u.JobTitle := "Lecturer"}
      ( restrict:r{r.Name="David Frame"}(database.JobAllocations) ),
    restrict:r{not r.Name="David Frame"}(database.JobAllocations)
);
```

More constructs could be added to cover common cases (such as an equivalent *update* statement) – but this comes at the cost of adding complexity to the language. Another solution would be to allow functions and procedures to be passed as arguments, and allow operators to be specified such that they accept some form of generic type. Such improvements could lead to a procedure defined as:

```
procedure update_rel( relation{generic X} var target,
                      boolean function(tuple{X}) test),
                      procedure(tuple{X} var) change )
{
        target := union( update( restrict(target, test), change ),
                       restrict(target, not test) );
}
```

With the example expression changing to:

```
call update_rel(database.JobAllocations,
                tuple.Name = "David Frame",
                tuple.JobTitle := "Lecturer");
```

The language would be powerful enough that it would be possible for such constructs to be entirely user-defined. Such a language would be much more difficult to implement, but would be very powerful indeed.

D^b should probably be made more practical, such as the introduction of a *break* statement (to exit the current loop). String support needs drastically improved, in order to support some form of *substring* operator, as well as the introduction of control-codes to allow quotes to appear within string literals. Additional relational operators may also be useful, an example of which might be a *maximum* function which return the maximum value from a relation with a single attribute.

The language does not currently support input or output to the console or files. Ideally such additions to the language should be based around the .NET Framework's I/O class hierarchy, and it was originally planned to implement this so these classes could be accessed by D^b programs. The problem with the merging of the .NET and 'D' type systems however was too great and instead separate built-in types were planned, each corresponding to a particular I/O class. This is a somewhat unsuitable solution, as it leads to either such types being unduly restrictive, or a whole host of types required in order to support I/O: `binary_input_tcp`, `ascii_output_tcp`, `ascii_output_file`, etc. This matter was not satisfactorily resolved and so I/O was skipped from this version of D^b altogether, which leads to programs in D^b being rather limited in that their input is limited to databases and command-line arguments. This will certainly have to be improved upon in future versions.

It is disappointing that there was not enough time to fully investigate and formally specify the 'D' type system. The type-checking rules specified do not specify when exactly an object of a type can be replaced by another, or which function body is invoked under certain conditions. The specification of the semantics of the D^b (possibly using similar techniques to those described in [Schmidt, 1996]) would be greatly beneficial.

True .NET Support

More work is required in order to ascertain the level at which the 'D' type system and the .NET type system could be combined, in order that D^b programs could access externally defined .NET components. It may be possible to analyse .NET classes to determine which modify the object and which do not (similar to 'const' methods in C++), and those methods to be invoked on variables and values respectively.

The language should really also include .NET accessibility information, such as *public*/*private*. This would allow the user to explicitly state which constructs to allow to be called from external components and which are to be used solely for the component being compiled.

Implementation Improvements

At present, the sample compiler supports only a subset of the full 'D' model. Some aspects, such as candidate keys, seem so important to the relational model of data that they must be implemented in any future development. The relationships between relation types must also be specified, especially concerning database relations. 'virtual' database relations should also be supported in such a form that they are treated identically to 'real' relations.

The next logical step in the development of D^b is the support of the language (and the 'D' model) by a fully-fledged Database Management System (DBMS). Such a system would be a stand-alone application with which other applications communicate in order to store data. Implementing D^b as an interpreter may well be more efficient for executing ad-hoc queries. Sections of the existing compiler could possibly be used to support compiled transactions.

Efficiency was not a prime requirement of the current implementation, and this would certainly have to be improved in future, with the use of more advanced techniques to implement the relational operators. The implementation of multiple-dispatch could also be vastly improved over the rather naïve current implementation, perhaps by implementing compressed dispatch tables, as defined in [Dujardin et al, 1998].

7. LIST OF REFERENCES

- [Atkinson & Buneman, 1988] "Types and Persistence in Database Programming Languages", Malcolm P. Atkinson & O. Peter Buneman, ACM Computing Surveys, Vol.19, No.2, June 1987
- [Codd, 1970] "A Relational Model of Data for Large Shared Data Banks", E.F. Codd, Communications of the ACM, Vol. 13, No.6, June 1970. pp377-387
- [Date & Darwen, 2000] "Foundation for Future Database Systems – The Third Manifesto (2nd Edition)", C.J. Date & Hugh Darwen, Addison-Wesley, ISBN 0-201-70928-7
- [Donahue & Demers, 1985] "Data Types Are Values", James Donahue & Alan Demers, ACM Transactions on Programming Languages and Systems, Vol.7, No.3, July 1985, pp426-445
- [Dujardin et al, 1998] "Fast Algorithms for Compressed Multimethod Dispatch Table Generation", Eric Dujardin & Eric Amiel & Eric Simon, ACM Transactions on Programming Languages and Systems, Vol. 20, No.1, January 1998, pp 116-165.
- [Harrison & Naeem, 2000] "POOL: A Persistent Object-Oriented Language", C.J. Harrison & Majid Naeem, Proceeding of ACM Symposium of Applied Computing, 2000, Vol. 2, pp764-766
- [Reisner, 1981] "Human Factors Studies of Database Languages: A Survey and Assessment", Phyllis Reisner, ACM Computing Surveys, Vol. 13, No. 1, March 1981 – pp13-31
- [Schmidt 1996] "Programming Language Semantics", David A. Schmidt, ACM Computing Surveys, Vol. 28, No.1, March 1996.
- [Stroustrup, 2000] "The C++ Programming Language: Special Edition", Bjarne Stroustrup, Addison-Wesley, ISBN 0-201-70073-5
- [URL 1] American National Standards Institute, http://www.ansi.org/
- [URL 2] International Standards Organisation, http://www.iso.org
- [URL 3] Java Documentation & Training, http://developer.java.sun.com/developer/infodocs/
- [URL 4] XML Standard, http://www.w3.org/XML/

Appendix A : Full D^b Grammar

General

```
<Letter>       ::=    a | b | c | d | e | f | g | h | I | j | k | l | m | n | o | p |
                      q | r | s | t | u | v | w | x | y | z | A | B | C | D | E | G |
                      H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W |
                      X | Y | Z
<Digit>        ::=    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<Id>           ::=    <Letter> (<Letter> | <Digit>)*
<Number>       ::=    (<Digit>)+ [. <Digit> (<Digit>)*]
<AnyChar>      ::=    Any unicode character
<NonCR>        ::=    <AnyChar> not carriage return
<NonQuote>     ::=    <AnyChar> not double-quote character
<NonApost>     ::=    <AnyChar> not apostrophe
<String>       ::=    " (<NonQuote>)* "
<Character>    ::=    ' <AnyChar> '

<Source>       ::=    ( <TopLevel> )*
<TopLevel>     ::=    <Function> | <Procedure> | <Transaction> | <Program> |
                      <Database> | <Type>
```

Functions

```
<Function>     ::=    <FuncHeader> (; | <StmntBlock>)
<FuncHeader>   ::=    <TypeExpr> function <id>( <FuncPlist> )
<FuncPlist>    ::=    <FuncParam> (, <FuncParam>) *
<FuncParam>    ::=    <TypeExpr> value <id>
```

Procedures

```
<Procedure>    ::=    <ProcHeader> (; | <StmntBlock>)
<ProcHeader>   ::=    procedure <id>( <ProcPlist> )
<ProcPlist>    ::=    <ProcParam> (, <ProcParam>) *
<ProcParam>    ::=    <TypeExpr> (value | variable) <id>
```

Transactions

```
<Transaction>  ::=    <TransHeader> (; | <StmntBlock>)
<TransHeader>  ::=    transaction <id>( <ProcPlist> ) uses <id>
```

Programs

```
<Program>      ::=    program ( <ProcPlist> ) <StmntBlock>
```

Type Expressions

```
<TypeExpr>     ::=    <id> | <GenType>
<GenType>      ::=    <TupleType> | <RelationType>
<TupleType>    ::=    tuple <RelHeader>
<RelHeader>    ::=    { [ <TypeExpr> <id> ( , <TypeExpr> <id> ) ] }
<RelationType> ::=    relation <RelHeader> <KeyList>
<KeyList>      ::=    [ <KeyDef> (, <KeyDef>)* ]
<KeyDef>       ::=    key( <AttributeList> )
```

Type Definitions

```
<Type>         ::=    <Simple> | <Complex> | <Abstract>
<simple>       ::=    simple type <id> (<SubtypeRel> | <NewType>) ;
<SubtypeRel>   ::=    : <STypeList>
<STypeList>    ::=    <id> [<Constraint> | (, <id>)* ]
<Constraint>   ::=    when <Expression>
<NewType>      ::=    underlying type <TypeExpr>
<Complex>      ::=    complex type <id> [<SubtypeRel>] <MemList> <GenSpec> ;
<MemList>      ::=    { (<TypeExpr> <id>;)* }
<GenSpec>      ::=    ( <GenExpr> )* [<SpecExpr>]
<GenExpr>      ::=    generalisation <id> <FunctionArgs>
<SpecExpr>     ::=    specialisation <id> <FunctionArgs>
<Abstract>     ::=    abstract type <id> <SubtypeRel> ;
```

Databases

```
<Database>      ::=    database <id> { (<DB_Entry>)* }
<DB_Entry>      ::=    <ForeignKey> | <DB_RelVar>
<DB_RelVar>     ::=    <RelationType> (<id> | <Virt_RelVar>) ;
<Virt_RelVar>   ::=    virtual <id> := <Expression>
<ForeignKey>    ::=    foreign key <id>.<id> (.<id>)* to <id> (.<id>)*
```

Statements

```
<Stmnt>         ::=    <StmntBlock> | <If> | <Switch> | <While> | <Do> | <For> |
                       <Begin> | <Call> | <Return> | <Commit> | <Cancel> | <Let> |
                       <LocalDecl>
<StmntBlock>    ::=    { ( <Stmnt> )* }
<If>            ::=    if ( <Expression> ) <Stmnt>
<Switch>        ::=    switch ( <Expression> ) { (<Case>)* [<Other>] }
<Case>          ::=    case <Expression>: <Stmnt>
<Other>         ::=    other: <Stmnt>
<While>         ::=    while ( <Expression> ) <Stmnt>
<Do>            ::=    do <Stmnt> while ( <Expression> );
<For>           ::=    for (<Init>; <Invariant>; <Next>) <Stmnt>
<Init>          ::=    <TypeExpr> <Id> := <Expression>
<Invariant>     ::=    <Expression>
<Next>          ::=    <Let>
<Begin>         ::=    begin <id>( <ProcArgList> ) [onfailure <Stmnt>] ;
<Call>          ::=    call <id>( <Proc_ArgList> );
<ProcArgList>   ::=    (<ValueArg> | <VarArg>)
<ValueArg>      ::=    <Expression>
<VarArg>        ::=    asvar <id> (. <id>)*
<Return>        ::=    return [<Expression>];
<Commit>        ::=    commit;
<Cancel>        ::=    cancel;
<Let>           ::=    <id> (. <id>)* := <Expression> ;
<LocalDecl>     ::=    <TypeExpr> (value | variable) <id> = <Expression> ;
```

Expressions

```
<Expression>    ::=    <LogOR>
<LogOR>         ::=    <LogAND> ( or <LogAND> )*
<LogAND>        ::=    <LogNeg> ( and <LogNeg> ) *
<LogNeg>        ::=    [not] <LogComp>
<LogComp>       ::=    <TypeTest> ( ( < | > | <= | >= | = ) <TypeTest> ) *
<TypeTest>      ::=    <AddExpr> ( (is | as) <TypeExpr> ) *
<AddExpr>       ::=    <MulExpr> ( ( + | - ) <MulExpr> ) *
<MulExpr>       ::=    <Term> ( ( * | / ) <Term> ) *
<Term>          ::=    <UnaryOp> | <Atomic> | <SubExpr> | <RelOp> | <GenSelInvoc>
<UnaryOp>       ::=    ( + | - ) <Term>
<Atomic>        ::=    (<id> <IdPostFix>) | <Literal>
<IdPostFix>     ::=    ( (. <id>)* ) | <FunctionArgs> | [ <Expression> ]
<FunctionArgs>  ::=    ( <Expression> ( , <Expression> ) * )
<SubExpr>       ::=    ( <Expression> )
<Literal>       ::=    <integer> | <char> | (<string> [ [ <Expression> ] ]) |
                       <float> | <bool>
<GenSelInvoc>   ::=    <GenType> <FunctionArgs>

<RelOp>         ::=    <RelProject> | <RelRename> | <RelExtend> | <RelWrap> |
                       <RelUnWrap> | <RelGroup> | <RelUnGroup> | <RelRestrict> |
                       <RelUpdate> | <RelNoSpecial> | <RelExtract> | <RelCount> |
                       <RelMember>
<RelProject>    ::=    project { <AttributeList> } <SubExpr>
<AttributeList>::=     [ <id> ( , <id> )* ]
<RelRename>     ::=    rename { <RenameList> } <SubExpr>
<RenameList>    ::=    [ <id> to <id> ( , <id> to <id> ) ]
<RelExtend>     ::=    extend : <id> { <TypeExpr> <id> := <Expression> } <SubExpr>
<RelWrap>       ::=    wrap { <AttributeList> as <id> } <SubExpr>
<RelUnWrap>     ::=    unwrap { <id> } <SubExpr>
<RelGroup>      ::=    group { <AttributeList> as <id> } <SubExpr>
<RelUnGroup>    ::=    ungroup { <id> } <SubExpr>
<RelRestrict>   ::=    restrict : <id> { <Expression> } <SubExpr>
<RelUpdate>     ::=    update : <id> { <id> := <Expression> } <SubExpr>
<RelNoSpecial>  ::=    ( join | union | intersection | minus )( <Expression> ,
                                                                 <Expression> )
<RelExtract>    ::=    extract <SubExpr>
<RelCount>      ::=    count <SubExpr>
<RelMember>     ::=    ismember ( <Expression>, <Expression> )
```

Appendix B: Example Test Data

This appendix contains some of the data used to test the implementation. Not all the tests can be presented here but it should give the impression of the different levels of testing required.

Testing Lexical Analysis

The first stage involves checking that all lexeme types are recognised correctly. The lexical analyser splits the string into the following 'types':

integer, identifier, stringliteral, float, character, operator, keyword, eof.

Strings with each of the items (bar 'eof', which marks the end of the file and is implicitly at the end of all strings) are supplied and then the lexemes read back, to ensure that they were correctly identified. The text strings are read from a file, with the string on odd lines and the expected result on even (where a result line of "#" indicates that an error should occur).

An example of the tests for integers are: (result lines in italics):

```
3
integer 3, eof.
3494
integer 3494, eof.
-3494
operator -, integer 3494, eof.
+304
operator +, integer 304, eof.
0  4
integer 0, integer 4, eof.
  345
integer 345, eof.
```

Unary operators such as + and – are detected at the parsing stage. The tests for floats include:

```
3.494
float 3.494, eof.
0.0
float 0.0, eof.
-4.5
operator -, float 4.5, eof.
3.
#
0.4.5
#
```

Again the unary operators are picked up at the parsing stage. The last two errors show that numbers must be expressed after the decimal point (although as shown, zero is acceptable), and that the decimal point may only appear once. While this could be picked up at the parsing stage also, it is so obvious a mistake that the lexical analyser has been written to catch it. Identifiers are tested similarly:

```
a
identifier a, eof.
abab
identifier abab, eof.
a495
identifier a495, eof.
a3.4
identifier a3, operator ., integer 4, eof.
a2.b
identifier a2, operator ., identifier b, eof.
```

These test that identifiers must start with a letter but can then be followed by numbers. The inclusion of the syntactically incorrect "a3.4" is to ensure that the "3.4" isn't picked up as a float.

Operators are tested first individually, and then ensuring that it is possible to embed them in other expressions. For example:

```
a+3
identifier a, operator +, integer 3, eof.
1>a
integer 1, operator >, identifier a, eof.
3.4>>=>
float 3.4, operator >, operator >=, operator >, eof.
5> =2
integer 5, operator >, operator =, integer 2, eof.
```

The third test ensures that operators of double length are detected correctly, and the fourth that putting whitespace between the two characters of a double length operator allows them to be detected as separate operators.

The identification of keywords and textual operators is also performed at the lexical analyser level:

```
and or not
operator and, operator or, operator not, eof.
restrict procedure tuple
keyword procedure, keyword procedure, keyword tuple, eof.
not42
identifier not42, eof.
notb
identifier notb, eof.
```

Tests similar to the last two are performed to ensure that identifiers which start with keywords are not cut short. Finally, the tests for string and character literals are performed. The tests ensure that the internals of the literal are not handled in the same way.

```
""
string literal "", eof.
"3.4.5.6 494"
string literal "3.4.5.6 494", eof.
"'d'"
string literal "", eof.
'd'
character 'd', eof.
'"'
character '"', eof.
'''
character ''', eof.
"
#
'
#
''
#
'34'
#
```

These ensure that the general lexical analysis rules are turned off while reading the internals of the literals. The final four tests check for invalid literals: strings and literals with no end, characters with no character specified and characters with more than one character specified. Finally, the lexical analyser must also remove comments from input:

```
#nothing to see here
eof.
34#.2
integer 34, eof.
```

Testing Parser Classes

The parser classes cannot be exhaustively tested, as large numbers of the production rules can potentially be infinitely long. Instead a selection of the main errors were chosen along with legal strings for each main syntactic category. Some examples of this for the *function* parser are shown below.

```
#Error: Missing type
function x

#Error: No missing name
integer function ()

#Error: Missing parameters
integer function y {}

#Error: Parameter missing value/variable qualifier
integer function x(integer p) {}

#Error: Functions not allowed variable parameters
integer function y(integer variable x)

#Error: Parameter missing type
integer function y(p) {}

#Error: Parameter missing type
integer function y(value p) {}

#Error: Parameter missing name
integer function y(integer value) {}

#Error: Missing comma between parameters
integer function y(integer value v integer value q) {}

#Error: Begin not valid from within function body
integer function y(integer value x)
{
	begin t(x);
}

#Error: Commit not valid from within function body
integer function y(integer value x)
{
	commit;
}

#Error: Cancel not valid from within function body
integer function y(integer value x)
{
	cancel;
}

#Error: Return used without expression in function body
integer function y(integer value x)
{
	return;
}
```

Testing Type Rules

The type rules were tested by creating files with the variations of each construct in, and compiling them to see if the correct results were obtained. Where required, complex types were defined such that *T* was a base type, *S* and *R* were both subtypes of *T*, and *U* was a subtype of both *S* and *R*. *T* was defined as:

```
complex type T { integer x; }
```

Assignment

```
integer variable x := 9;                        # Correct
integer variable x := "Hello World";            # Error
string variable  x := "Hello World";            # Correct

tuple{} variable x := 9;                        # Error
tuple{} variable x := tuple{}();                # Correct
tuple{T x} variable x := tuple{T y}(v);         # Error
tuple{T x} variable x := tuple{S x}(v);         # Correct
tuple{S x} variable x := tuple{R x}(v);         # Error

tuple{T x} value y = tuple{T x}( T(3) );
y := x                                          # Error
```

General Expressions

```
T variable x        = T(9);
R variable y        = R(3);
integer variable z  = 6;
tuple{integer x} p  = tuple{integer x}(z);

x + x;                            # Error (no + for T x T)
x + y;                            # Error (no + for T x R)
z + x;                            # Error (no + for integer x T)
z + z;                            # Success
z + (z);                          # Success
p + p;                            # Error (no + for tuple x tuple)
p.x + z;                          # Success
x.x + p.x;                        # Success

              [Similar tests performed for *, /, unary ops etc]

x is R;                           # Success
x is integer;                     # Error (integer not subtype of T)
y is T;                           # Error (T is not subtype of R)
x as R;                           # Success
x is integer;                     # Error (integer not subtype of T)
y is T;                           # Error (T is not subtype of R)

length( "Hello" );                # Success
length( concat("C","S") );        # Success
length( true );                   # Error (no length() for boolean)
concat( x, x );                   # Error (no concat() for T x T)
concat( 3, 7 );                   # Error (no concat() for integer,
                                                            integer)

"Hello"[9]                        # Success
"hello"[1] = 'e'                  # Success
```

If Statements

```
if ( 3 )                          # Error (non-boolean)
if ( 3.14 )                       # Error (non-boolean)
if ( true )                       # Success
if ( false )                      # Success
if ( tuple{}() )                  # Error (non-boolean)
```

```
        if ( 3.1 = 3.1 )                # Success
```

Repetition Statements

```
        for(integer x := 3.4;           # Error (float not subtype of int)
        for(integer y := 3; y;          # Error (test not boolean)
        for(integer y := 3; true;
              y := y + 9) …             # Success

        do x := x+1; while ( x < 3 );   # Success
        do x := x-1; while ( x );       # Failure (test not boolean)

        while ( true ) x := x+1;        # Success
        while ( x ) x := x-2;           # Failure (test not boolean)
```

Testing Function Constraints

```
        R function x(T value y);   # Success
        R function x(T value y);   # Success (re-declaration)
        R function x(S value x);   # Success (more-specific)
        T function x(R value x);   # Error: cannot generalise return type of
                                          more-specific function

        R function x(T value y, T value y);     # Success
        R function x(S value x, R value y);     # Success
        S function x(R value x, S value y);     # Success
              # Error: No function x(U, U) defined

        R function x(T value y, T value y);     # Success
        R function x(S value x, R value y);     # Success
        S function x(R value x, S value y);     # Success
        U function x(U value x, U value y);     # Success
```

Testing Semantics

The semantics were tested by adding a *print* construct which displays a specific value. Examples of this in action are:

```
        # List of 0 to 9
        for (integer j := 0; j < 10; j := j+1)
              print j;

        # Displays the result of the join of two tuple values
        print join(   tuple{integer a, integer b}(9,2),
                      tuple{integer b, integer c}(2,10)       );

        # Tests specialisation by constraint [Expected is f(Circle)]
        integer function f(Ellipse e)
        {
              print "In F(Ellipse)"; return 1;
        }
        integer function f(Circle c)
        {
              print "In F(Circle)"; return 2;
        }
        print f( Ellipse(2,2) );

        # Tests if-semantics
        if ( 3 = 4 ) print "Uh-oh"; else print "Okay";
```

The above examples show how the result of operations (such as *join*) were checked as well as the trace of execution checking that the structured programming constructs operate as planned.