

DATABASE IMPLEMENTATION ON AN OBJECT-ORIENTED PROCESSOR ARCHITECTURE


A thesis submitted in partial fulfilment
of the requirements of the
University of Abertay Dundee
for the degree of Doctor of Philosophy

Louis David Natanson

University of Abertay Dundee

July 4, 1995

I certify that this thesis is the true and accurate version
of the thesis approved by the examiners.

Signed 
Director of Studies

Date *21/2/95*

Abstract

The advent of an object-oriented processor, the REKURSIV, allowed the possibility of investigating the application of object-oriented techniques to all the levels of a software system's architecture. This work is concerned with the implementation of a database system on the REKURSIV. A database system was implemented with an architecture structured as

- An external level provided by DEAL, a database query language with functions.
- A conceptual level consisting of an implementation of the relational algebra.
- an internal level provided by the REKURSIV system.

The mapping of the external to the conceptual levels is achieved through a recursive descent interpreter which was machine generated from a syntax specification.

The software providing the conceptual level was systematically derived from a formal algebraic specification of the relational algebra.

The internal level was experimentally investigated to quantify the nature of the contribution made to computational power by the REKURSIV's *architectural* innovations.

The contributions made by this work are:

- the methodology exposed for program derivation (in class based languages) from algebraic specifications;
- the treatment of the notion of *domain* within formal specification;
- the development of a top-down parser generator;
- the establishment of a quantitative performance profile for the REKURSIV.

Acknowledgements

At a personal level I would like to thank Ian Colligan and Colin Fraser of the University of Abertay Dundee for their encouragement throughout.

I thank Allan Milne for his intellectual robustness and curiosity and for realising that my only mode of communication is argument.

Thanks also to Andy Wakelin for his insights and challenges.

Most of all, at all levels intellectual and personal, for showing me this whole enjoyable side of life and patiently putting up with me, I thank Bill Samson my supervisor.

Contents

1	Introduction	7
1.1	Background	7
1.2	Scope of the study	9
1.3	Objectives	12
1.4	Summary	12
2	Literature review	16
2.1	Hardware	17
2.1.1	Programming Language support	17
2.1.2	Database machines	23
2.2	Database Query Languages	24
2.2.1	Deductive Database Systems	26
2.2.2	Functional, Deductive and Object-oriented Databases .	29
2.3	Program derivation	32

2.4	Summary	33
3	The REKURSIV	36
4	The language DEAL	44
4.1	Introduction	44
4.2	Syntax	47
4.3	Using the DEAL interpreter	52
4.4	Conclusion	55
5	Implementing the language	56
5.1	Introduction	56
5.2	The topmost levels	59
5.3	Function definitions	66
5.4	Expressions	72
5.4.1	Expressions involving binary operators	75
5.4.2	Selection and Projection	77
5.4.3	Link elements	83
5.4.4	Function Application	90
5.5	The execution phase	91
5.5.1	Binary Operators	92
5.5.2	The Unary operators	94

5.5.3	Variables and Constants	94
5.5.4	Function Application	95
5.5.5	Statements	96
5.6	A complete example	97
5.7	Summary	100
6	Specification of the Relational Algebra	101
6.1	Introduction	101
6.2	Formal techniques	102
6.3	Deriving programs from algebraic specifications	112
6.4	Specifying the Relational Algebra	120
6.5	Specifying a relational operator	123
6.6	Efficiency and refinement	127
6.7	2-3 trees	131
6.8	Query optimisation	142
6.9	Conclusion	146
7	Performance evaluation	148
7.1	Harland's claims	148
7.2	A preliminary experiment	150
7.3	Medium scale benchmarks	154

7.4	Large scale benchmarks	170
7.4.1	Hashing	170
7.4.2	AVL Trees	174
8	Conclusions	177
8.1	Qualitative results	177
8.2	The use of the REKURSIV for database work	179
8.3	The verdict on the REKURSIV	182
8.4	The failure	184
8.5	The future	185
A	Code for the preliminary experiment	205
B	An SML specification	213
C	Implementing Interpreters	224
C.1	Introduction	224
C.2	An example	225
C.3	Translation	231
C.3.1	Grammars	231
C.3.2	Lexical Analysis	232
C.3.3	The syntax analyser	236

CONTENTS 6

C.3.4 Adding semantic and interpretive actions	242
C.4 An interpreter generator	244
C.5 Summary	253
D An SML specification based on 2–3 trees	255

Chapter 1

Introduction

1.1 Background

In November 1988, Dundee Institute of Technology won a grant under the Department of Trade and Industry's Awareness Initiative in Object-oriented programming. The grant included the award of a REKURSIV processor [52] board (manufactured by Linn Smart Computing, Glasgow) to be hosted on a Sun workstation. The aim of the initiative was to develop applications software to run on REKURSIV systems.

Workers at the Institute had for several years been actively cooperating with workers at other institutions in the development of a relational language DEAL (DEductive ALgebra) [26]. This work had included both query

language implementation and application : HQL (an Historical Query Language) [90] and Graphical Databases [112]. Given this experience the natural goal of the group was to work on DEAL in the context of the REKURSIV.

The language DEAL makes use of an extended relational algebra. The extensions aim to provide facilities useful to knowledge processing - user defined functions, recursion and to some extent deductions.

Turning to the target machine, the REKURSIV is a microcodeable processor that utilises a persistent object store. The processor is object-oriented in these ways

- Objects in memory are addressed by unique **object identifiers**.
- Object images on disk are in direct correspondence with their main memory image.
- Objects' types (classes) and sizes are stored in parallel with their contents allowing hardware type checking or hardware assisted dynamic binding and method lookup.
- The virtual memory manager deals with (arbitrarily sized) objects rather than fixed size pages and so strategies to keep the most useful *objects* in primary storage can be employed (rather than strategies that keep an area or page of memory that contains a useful object as

well as parts of other objects)

In addition, the REKURSIV's microcode level has stacks available to it which facilitate recursive processing.

The Smalltalk-like language Lingo ([53]) provides a convenient programmer interface to the underlying REKURSIV hardware, obviating the need to microcode. Harland, the REKURSIV's designer, has claimed (informally within conversations during the progress of the initiative) that the purpose of the REKURSIV is to execute Lingo programs and so it can be assumed that the microcode support for Lingo is near optimal in its use of the hardware capability.

1.2 Scope of the study

This study concerns the implementation of a complete database system based on DEAL on the REKURSIV. The general aim is to attempt to quantify the advantages that the REKURSIV architecture and environment (or aspects of it) offer for a particular approach to query language development.

The approach is based on a formal attitude towards the relational algebra. The relational algebra, rather than an object-oriented model, is chosen since it is a well studied and mathematically stable model. Chapter 6 discusses

the algebraic specification of the relational algebra and the derivation of programs from such a specification. A side benefit of such a specification is that database terms such as domain and attribute, which are used differently by different authors, can be given precise and unambiguous meanings.

The concern of the study is limited to memory-resident data sets. This decision is based on these observations :

- A key feature of the REKURSIV is its object-oriented store. The issue of its performance is more clearly aired in the absence of unassociated disk operations which, anyway, are largely beyond an implementor's control. Disk operations cannot be completely avoided in a system that has a virtual memory. An intention behind the restriction to data sets that could reside in physically existing memory is to avoid the contamination of performance results by factors attributable to the system that hosts the REKURSIV, that is a SUN workstation and its operating system.

Moreover, much database machine work [4] [6] [15] [59] [28] has centred on increasing bandwidth of data flow from the data store (disk) to the processing elements by using parallel processors and placing processors as close to the disk surface as possible. Such a fundamental level of hardware configuration was not available for this study and so the work

reported here does not attempt to contribute to the database machine domain.

- The computational effort that we are primarily interested in observing emanates from the deductive nature of the queries. Such queries typically involve recursion and the building of closures rather than scans through large relations.

The choice of a relational algebra based query language rather than a logic programming (Prolog) approach is made given the following considerations :

- The security of the system is easier to control, and its integrity and consistency rules are more succinctly expressed, within a framework of types, schemas and keys, all of which are absent from Prolog.
- The resolution proof procedure of Prolog is not transparent to the programmer: some queries are only successfully expressed by judicious use of the cut whose correct positioning is determined by considering the route taken through rules by the Prolog proof mechanism. In addition, there can be no consolidation of proof computation (in the form of lemmas or the storage of a derived clause) without recourse to metalevel predicates `assert()` and `retract` (pages 94 to 96 of [42]). Prolog concerns itself with the process of the proof, giving the deduction itself as

a side effect.

1.3 Objectives

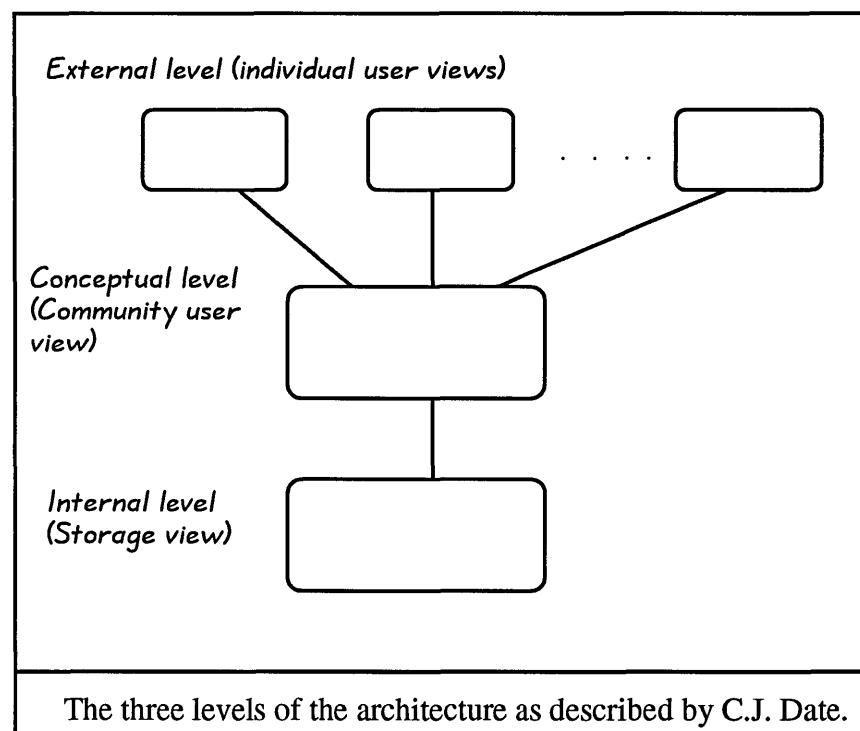
The objectives of the work carried out in this study are to

- Implement a database system on the REKURSIV processor.
- Use the implementation to investigate the performance of the REKURSIV.
- Evaluate the results of performance experiments in terms of positive contributions to computation made by different aspects of the REKURSIV architecture.

1.4 Summary

This chapter has introduced the main ingredients of the work – object-orientation, the REKURSIV, formal specification, deductive query languages and the relational algebra. This disparate collection of domains is brought to bear on the central task being undertaken (the development of a database system) so as to exercise an object-oriented processor and evaluate its impact on performance and the software engineering life cycle.

To clarify the interrelation of these domains to the work, consider the architectural diagram which is based on Date's ([24]) generalisation of the ANSI/SPARC Study Group on Database Management Systems architecture ([107])



- The *external* level concerns itself with the way data is viewed by users. This is provided through the language DEAL for which an interpreter (written in the language Lingo) was constructed. There is also an "embedded" DEAL in the sense that the interpreter object can be interro-

gated from any Lingo code. A description of DEAL and the synthesis strategy used by the interpreter are discussed in chapter 4. The analysis (lexical and syntactical) phases of the interpreter are discussed in chapter 5, along with the construction of a generalised translator generator.

The *conceptual* level consists of abstract representations of the database. This is provided by a set of Lingo objects modelling the relational algebra – relational tables and relational operators. This level separates the external level from the storage details of the database. The implementation of these Lingo objects was derived from a formal specification (written in Standard ML). This aspect of the work is reported in chapter 6.

The *internal level* concerns the way data is physically stored. In this case data is stored in the REKURSIV's persistent object store. At this level it is the performance of the REKURSIV that is of interest. The performance evaluation of the REKURSIV is reported in chapter 7 where two storage strategies (hash tables and balanced trees) are compared (both on the REKURSIV and a Smalltalk/V system on an IBM PC) and a general performance profile for the REKURSIV is established.

The REKURSIV itself is described in chapter 3.

The next chapter contains a literature review supporting this work.

Chapter 2

Literature review

This chapter depicts, with reference to a body of literature, the climate which has influenced work on the project. The structure of the chapter follows the last chapter's description of the different levels in a database system's architecture:

- The first section looks at specialised hardware support for advanced language systems and then, specifically, database systems.
- The second section covers briefly the historical context of the development of database query languages based on the relational model.
- The third section reviews the sources that have informed the formal algebraic approach to program derivation that was used to implement

the relational model.

2.1 Hardware

2.1.1 Programming Language support

During the late 1970s and throughout the 1980s, workers in many computing domains proposed architectures for machines (virtual or physical) that would better support their computational paradigm. Looking first at language based paradigms, it is instructive to consider the following:

- **Functional Programming –**

An evaluation strategy for applicative languages, known as SECD (Stack, Environment, Control list, Dump) inspired abstract machines ([1, 56]) of which arguably the best known is Cardelli's Functional Abstract Machine (FAM, [14]) used in the University of Edinburgh's implementation of Standard ML. The hardware support for Lisp described in [105] has its roots in the SECD approach.

A more modern evaluation strategy is based on super-combinators and lambda lifting ([60]) which in turn derives from the combinator approach of Turner ([108]). Again, the history of this line of development

includes the definition of abstract machines and then their embodiment in hardware. For the simple combinator approach, the combinators themselves can be regarded as the instruction set for a machine ([11]) which is usually implemented virtually but has been constructed out of hardware ([17]). ALICE (the Applicative Language Idealised Computing Engine) developed at Imperial College ([22, 23]) is an example of a hardware embodiment of a combinator machine. The hardware was not customised however. Instead, ALICE utilised the INMOS transputer (a parallel processing element) and took advantage of opportunities for parallel evaluation afforded by applicative programs.

Abstract machines that support the specialised combinators (super-combinators) found by lambda lifting to suit a particular applicative program include the G machine ([64]) and the Three Instruction Machine (TIM) of Fairbairn and Wray ([29]).

As well as these machine designs, thought has also been given to the operating system layer, often as a *tour de force* in functional programming ([67, 1, 104]).

A complete 'functional programming workstation' is the Symbolics LISP machine ([114, 44]). This machine is often associated with its object-oriented component known as *flavors* ([79]) and so is consid-

ered under the next heading.

- **Object-oriented programming** – Perhaps the largest language development in this area is that of Smalltalk-80 described by Goldberg in [39]. Strictly speaking the term ‘Smalltalk-80’ refers to a *system* and not just a programming language, since Goldberg’s book (acknowledged as definitive) covers a language, an operating system and a programming environment. The original goal of the team at Xerox Palo Alto Research Center was to provide the complete software for a personal information management system, the Dynabook ([68]), an advanced idea for the mid 1970s before the advent of the personal computer. From the outset, the language development effort was affected by the search for efficient implementations since the system aimed to present an accessible graphical interface to users which involved resource hungry components such as a windowing system, icons and menus. Indeed the early success of Smalltalk-72 ([98]) concentrated on graphics: the graphical Pygmalion system ([99]) inspired the Star office system ([100]), a precursor to today’s graphical user interfaces such as Microsoft Windows. With the experience of deficiencies in Smalltalk-72, Smalltalk-76 ([61]) established the essential message passing syntax of today’s Smalltalk systems and introduced an intermediate language to which Smalltalk

expressions were translated. This intermediate language increased execution speeds dramatically ([62]) and was the forerunner of the Smalltalk-80 virtual machine established by Goldberg and others along with the language in [39]. Two other works published at the time have also become the definitive texts on Smalltalk systems: Goldberg [40] described the programming environment and in [70] (edited by Krasner) the experiences and conclusions of teams who had implemented the Smalltalk-80 virtual machine on a spectrum of hardware platforms from Motorola 68000 systems to Digital Equipment's VAX minicomputers. Of the ten implementations compared there, only three were on microprogrammable customised hardware: two on the Xerox Dolphin and a third on the Xerox Dorado ([86, 87]). The Dorado performed best of the set of implementations and became the machine used for Smalltalk by Xerox. This is perhaps hardly surprising since the Dorado utilised Emitter-Couple Logic (ECL) technology to achieve a short instruction time (as opposed to the MOS technologies utilised in its competitors).

Ungar and Patterson, writing in 1987 ([111]), describe a Reduced Instruction Set Computer (RISC [85]) approach to implementing Smalltalk called SOAR (Smalltalk On A RISC). Based on simulation experiments,

they claim a marginally superior performance to the Dorado, despite the SOAR having an instruction time over 5 times that of the Dorado. Interestingly, they show that the performance advantage of the Dorado over Motorola 68010 systems is in line with what one would expect from the ratio of their instruction times. Taken together, these two assertions indicate that SOAR's performance advantage is attributable to an *architectural* difference rather than a different underlying implementation technology.

The Symbolics LISP machine mentioned previously ([114, 44]) may appear an odd candidate to support Smalltalk-like object-orientation. Many of the difficulties in the execution of object-oriented programs, emanate from their dynamic nature:

- **run time type checking** – since new types can be created actually at run time, this appears unavoidable.
- **dynamic binding of messages** – since routines are associated with data structures, and the types of data structures are not known until run time, the addresses of the routines to be invoked cannot be computed at compile time.

- **dynamic storage management** – the message passing paradigm, together with information hiding and encapsulation, favours heap (dynamic) memory use. Indeed, [111] reports that Smalltalk programs tend to generate garbage ten times faster than most Lisp programs.

Given this, run time support for object-orientation involves many tables and levels of indirection (method look-up tables, object identifiers and so on). Lisp machines, optimised for the classical list structure, the ‘cons’ of a ‘head’ atom to a ‘tail’ list, provide this support readily. In addition, Lisp in general does not differentiate between data and code – everything is either an atom or a list, including functions which via λ -expressions can be treated as data and generated at run time. This property directly supports the dynamic message binding of object-orientation.

The Symbolics, the Dorado and the SOAR are examples of ‘tagged’ architectures where some bits of every machine word are used to denote the kind of object the word represents. In the case of the Dorado and the SOAR a single bit is used to differentiate between integers and pointers. The Symbolics uses its tags to differentiate between atoms and lists. Lists are further differentiated to support a more

efficient storage regime known as 'CDR coding' ([114]) which reduces the number of pointers that need to be stored.

2.1.2 Database machines

A number of dedicated database processors have been designed and built. In general, the approach has been to increase the bandwidth of database systems by introducing parallelism and placing processing power as close to the disk storage system as possible.

The following are two representatives of the multiprocessor approach:

- GAMMA ([28]) consisted of 17 VAX 11/750 processors connected via a high speed token ring network. Although a distributed system, query processing was centrally controlled.
- RDBM ([95]) contained specialised processors for sorting and supporting binary relational operations. These special function processors shared a memory. It also contained a content addressable memory. All the hardware was centrally controlled by a minicomputer.

The VERSO machine ([33]) used a device akin to a finite state automaton to filter data more or less as it comes off the disk (actually out of the buffers to which the disc controller had direct memory access). In this way it is

capable of selection and projection as well as binary operations.

The emphasis on parallelism led to much work on devising parallel algorithms for relational operations and configuring multiprocessor systems ([8, 9, 10]).

2.2 Database Query Languages

Following Codd's seminal work ([18]) on the relational model languages for information retrieval moved from being procedural and record-oriented (such as COBOL, where programmers had to involve themselves with the intricacies of strategies to perform tasks) to non-procedural languages based on the relational calculus. Notable among these is the language QUEL of the INGRES database management system ([103, 55]). QUEL realised the notion of 'tuple variables' introduced by Codd in a proposed language ALPHA ([19]) which are existentially quantified variables. The general form of QUEL queries is:

```
RANGE OF <tuple variable> IS <relation>
```

```
RETRIEVE
```

```
  (<relation>.<attribute>)
```

```
WHERE <predicate>
```

where both the RANGE and RETRIEVE clauses can take more than one operand.

An example where the relation Human has scheme (age, gender, name) is

```
RANGE OF X IS Human
    RETRIEVE (X.gender, X.name)
WHERE X.age >= 18
```

which retrieves the names and genders of all humans aged 18 or more.

At about the same time, another relational calculus based language, SQL ([16]) was developed by a team at IBM. The above query could be given in SQL as:

```
SELECT gender, name
FROM Human
WHERE age >= 18
```

which is superficially very similar to QUEL. SQL, however, differs significantly from QUEL in that it allows the formation of intermediate relations and operations of set union and set difference on these and also a form of nesting sub-queries. These differences give it the complete power of the re-

lational calculus by overcoming the lack of universal quantification and also some of the character of the relational algebra since sets can be worked with.

An interactive form of the relational calculus known as Query By Example, QBE ([116]) is interesting as it allowed users to formulate queries by filling in example values in on-screen forms. Gray ([42]) draws out the interesting correspondence between QBE's example elements and Prolog's variables.

2.2.1 Deductive Database Systems

With the success of the relational model and the widespread adoption of SQL, much interest has emerged in attempting to create database systems which enhance database querying techniques by allowing logical inference. Such systems may be termed *Knowledge based systems*.

The following example on ancestry, often used in discussion of deductive capability, will be referred to throughout this subsection to elucidate the concerns in this area.

We may have certain facts stored in a database concerning parenthood. That is we may have a relation, **parent** say, with scheme (name, child). The membership of a tuple such as ('louis', 'ruth') in the relation **parent** expresses the fact that it is true that 'louis' is a parent of 'ruth'. We may also have

knowledge based on some rules rather than the simple facts contained in the parent relation. For example, we know that in order for X to be a grandparent of Y , there must be a Z such that the tuples (X,Z) and (Z,Y) are present in the relation parent.

Another relationship we may be interested in is that of ancestor: here P is an ancestor of Q if there is a set of tuples $(P,I_1),(I_1,I_2),\dots,(I_n,Q)$ for some n (perhaps zero).

A difficulty with SQL is its ‘flatness’ – since it has no means of embodying *indefinite* nesting of queries. Although a query can be formulated for the grandparent relationship above, the same cannot be done for the more general ancestor (unless a limit is artificially placed on the number of generations to look back). More formally, it is not possible to compute the *transitive closure* of a relation ([3]).

By contrast, the language Prolog allows a succinct modelling of the above.

```
parent(louis, ruth).
parent(odette, louis).
.
.
parent(elias, odette).

grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
ancestor(X,Y)    :- parent(X,Y).
ancestor(P,Q)    :- parent(P,I), ancestor(I,Q).
```

At this point, the Prolog system contains both facts and rules for deriving

new facts (such as 'elias' is an ancestor of 'ruth'). These new facts, though, are not derived until the system is appropriately queried with, for example, `?- ancestor(A,B).` which would retrieve all ancestors.

The manner in which the marriage of facts and rules within a deductive system is achieved has been the characterising feature of deductive query languages. The tension exists since, on the one hand, Prolog has excellent deductive capability and on the other, relational database systems support the storage and retrieval of facts.

A number of language systems have been designed for data models other than the relational model and in particular the Functional Data Model:

- DAPLEX ([97]) models the rules of a knowledge base through intentionally defined functions;
- FQL ([12]) operates on streams akin to the lazily evaluated lists of functional programming languages such as Miranda ([109]);
- FDL ([88]) addresses some deficiencies in DAPLEX – computational completeness, uniform storage regime for all functions and support for arbitrary construction of types.

For the relational model, attempts have been made to build on the success of SQL:

- SQUIRREL ([113]) extends the syntax of SQL to allow the inclusion of rules and their manipulation by allowing relations to contain logic statements;
- LQL ([96]) has logic-based extensions to SQL, where rules can be expressed with left hand sides as in Prolog and right hand sides SQL expressions.

DEAL ([26]) by contrast extends SQL by allowing recursion and the definition of relation returning functions. DEAL has, however, no real notion of *rules* and so cannot be classified as a deductive database system any more than a general purpose programming language which happens to have *relations* amongst its built-in types. More information on this can be found in chapter 4 since DEAL is the chosen language for this work.

2.2.2 Functional, Deductive and Object-oriented Databases

There is much interplay between these three approaches. Each approach has a characteristic essence:

- Functional Database systems, such as Buneman's FQL ([12]) and Shipman's DAPLEX ([97]), make use of functional data models based on *'the fundamental concept of function to model relationships among real world objects'* (Gray et al, [43]). Two kinds of item are present within the functional data model: entities (that model real world objects) and scalars (reals, integers, strings and so on). Functions map items to items. Multi-valued functions are allowed for flexibility. Built-in type constructors allow definition of sequences and tuples. Functions can be combined in various ways: function composition and restriction are common. In the functional data model view, the distinction between stored and derived data is removed: queries (requests for answers) are *'essentially requests for a value of a function, given argument values'* (Folius et al,[30], as quoted in Gray, ([43])).
- Deductive database systems *'... contain inference rules which can be used to deduce new facts from those stored explicitly'* (Frost, [31]).

Non-deductive systems may also contain rules which serve as *integrity constraints* restricting the permissible database states. In contrast, deductive systems, although they may also allow the expression of integrity constraints, contain inference rules with which to deduce *new* facts. Deductive processes, such as resolution, are directly supported

by the system and so the set of inference *rules* is specified by the user rather than the *process* of inference and deduction.

- Object-oriented database systems generally have origins in object-oriented programming languages: entities from the real world are represented as objects which encapsulate structure and behaviour. All objects are members of a class or type and can only be accessed and manipulated through operations defined on their class (Date, [24]). In object-oriented database systems, both the data and programs associated with an object are stored. The approach can be summarised as ‘*embedding semantics into database objects*’ (Date, [24]). The relationship between object-oriented data models and semantic data models is close (Gray [43]).

All three approaches above concern themselves with the relationship between what can be termed, coarsely, *code* and *data*. Both the functional data model and object-oriented model remove the distinction largely by only providing access to *operations* (code). Deductive systems are based on the uniform treatment of data whether stored as facts (data) or deduced by the application of rules (code).

2.3 Program derivation

Much space is given in chapter 6 to the discipline under which the implementation of the conceptual level of the database architecture has been achieved. In this section, the historical background to the discipline is covered.

The interpretation of abstract data types as many-sorted algebras (a collection of named sets and operators between them) is due to Morris ([80]), extended by Guttag ([45, 46]) and largely formalised by Goguen ([35, 36]). The key insight of this work was to abstract data types away from their representations and to show that the *relationships* between their operators characterised them. A significant contribution in [36] was the application of ideas from category theory, a branch of mathematics that is used to reveal ‘natural’ characteristics of algebraic structures that may be hidden by representation detail.

Defining the semantics of operations by axioms was introduced by Hoare ([57]).

Specification languages incorporating a formal notation for abstract data types and abstract operations were introduced by both Guttag (LARCH, [47]) and Goguen (OBJ, [37]).

An early equational program (to insert values into 2–3 trees) was pro-

vided by Hoffman and O'Donnel ([58]). This was extended by the inclusion of removal of values by Reade ([89]). (Reade's SML specifications were used by the author to derive the balanced tree implementations used in the performance experiments reported in chapter 7).

The design of programs by refinement of abstract data types towards 'implementations' based on abstract models of concrete representations is discussed in [20, 65].

Goguen and others ([32, 38]) incorporated a facility to support state information in abstract data types in the algebraic specification language OBJ2.

A survey by Samson and Wakelin ([93]) on algebraic specification of databases reveals that little work has been done on the specification of database operations (rather than queries). In particular, they detect a lack in the treatment of the idea of attribute domains and recommend further work.

2.4 Summary

This chapter has collected the main contexts in which this work has been undertaken.

Specifically, these are:

- Specialised hardware platforms – the work of the project was carried out on a REKURSIV processor. The background of database machines, machines to support the functional programming paradigm and object-oriented platforms have been described. A detailed description of the REKURSIV is to be found in the next chapter.
- Database Query Languages – the language implemented within the project, DEAL, has been placed within a spectrum of other language approaches. DEAL, which can be characterised as a ‘database query language with functions’ is based on the relational model and falls short of providing a deductive database system. Chapter 4 describes the language in more detail. Chapter 5 describes the implementation of the language.
- Program derivation – most of the underlying computational machinery of the project that supports relational algebra operations was obtained by deriving programs (in the language Lingo) from formal algebraic specifications written in SML. Chapter 6 describes this derivation process in greater detail and goes on to show how the implementation can be further refined.

The next chapter returns to the internal level of the database architecture

with a presentation of the hardware used for the project – the REKURSIV.

Chapter 3

The REKURSIV

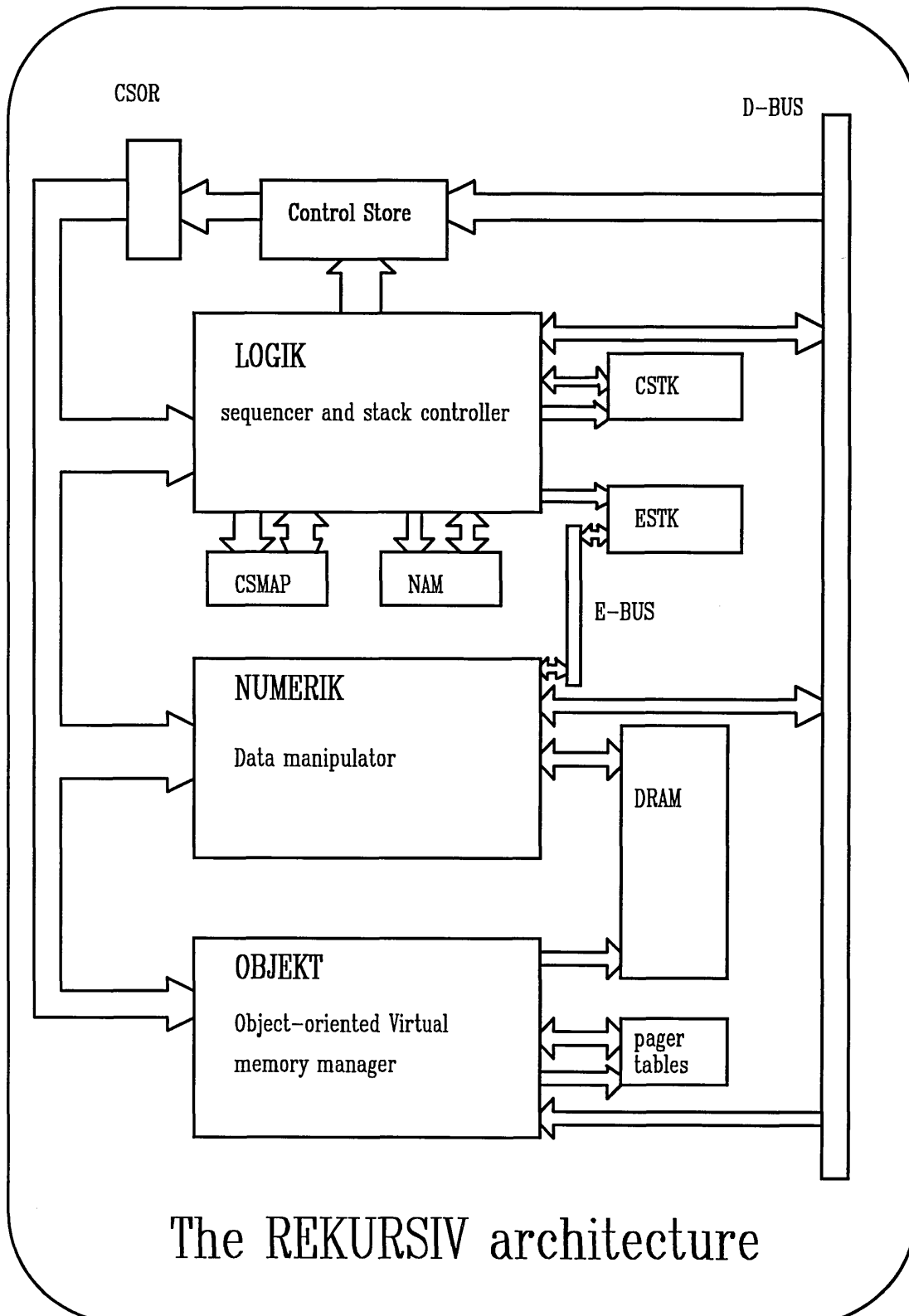
The REKURSIV processor differs from a conventional architecture in two principal ways.

- **Data Types.** At the machine level, a conventional architecture provides the programmer with a memory consisting of an array of equally sized cells each holding a bit pattern. The REKURSIV provides a structured space of objects, each having a *type* and *size* associated with it. The type and size of an object are retrieved from memory in parallel with the actual data parts of the object and can be inspected and used to determine execution sequence at the microcode level.

Harland's intention behind the design of the REKURSIV is to narrow the *semantic gap* that exists conventionally, where, at the programmer level, complex data types are used to maximise expressivity whereas at the machine level these data types are implemented by complex mechanisms involving several memory accesses and much processing. On the REKURSIV the burden of type checking is placed on the machine hardware. The chore of bundling and unbundling data between its high level structured organisation and a collection of machine words is removed.

- An Object Store. The REKURSIV addresses memory by unique *object identifiers* which are the only method of memory access available to the programmer. The provision of a virtual memory is facilitated by using the same representation for an object's disk image as it has in physical memory. This allows memory management strategies which seek to maintain frequently used *objects* in real memory as opposed to frequently accessed *pages*. In addition this mechanism allows the object store to persist.

The REKURSIV is constructed from a set of proprietary chips called LOGIK, NUMERIK, OBJEKT and KLOK.



- LOGIK is the sequencer that controls the microprogram execution. It is connected (by separate data paths) to various memories containing
 - the microcode (in the Control Store).
 - the map between machine level instructions and microcode sequences (in the Control Store Map)
 - abstract instructions (the NAM)
 - a stack for use by the microcode level (the CSTK)

LOGIK also has addressing logic for another stack memory (the ESTK), which is used by NUMERIK as an evaluation stack.

- NUMERIK takes the place of a conventional ALU containing sixteen thirty two bit registers. NUMERIK is connected to its own stack (the evaluation stack or ESTK) whose addressing is controlled by LOGIK. In addition it is connected to the main memory (the DRAM) of the object store (which is managed by OBJEKT)
- OBJEKT manages the object store. It contains circuitry to create new object identifiers, create space in the DRAM for objects, generate (and range check) addresses into the DRAM. It is connected to two memories

- the DRAM — this is the main memory where the contents of objects are actually stored. OBJEKT handles all aspects of addressing this - indexing, range checking, allocating and deallocating space.
- the pager tables — these take the place of the page tables in a conventional virtual memory. The object identifier, size, type and first word (of the contents or *representation*) of each object physically present in the DRAM is stored here. When servicing a request for an object, OBJEKT addresses the pager tables using the bottom 16 bits of the object identifier. The object identifier found in the pager tables is then compared with the required one — a match indicates the object is in the DRAM. If there is no match the object must be swapped in from DISK and OBJEKT handles the communication with the disk processor (DP) to effect this.

The Rekursiv's pedigree

Clearly the design of the Rekursiv did not occur in a vacuum. At the time that Harland's [52] book was published (1988), RISC architectures were the apparent way forward for processor design and indeed Harland devotes a

section of his book to a discussion between proponents of the RISC and EISC schools of thought. A major line of argument that Harland supports is that RISC architectures do not tolerate changes of control flow. Many of the advantages brought about by RISC features such as instruction caches and pipelining, are antagonised by such changes.

It is perhaps unfortunate that Harland does not capitalise on a previous work [51] to connect this line of argument and his concept of the semantic gap more closely with the question of types and their promotion to first class citizens – entities on which computation can be performed and which are a primary means of programmer expression. Instead, the microcodability of the Rekursiv is stressed strongly throughout the book.

It is instructive to view the progress and learning curve of project groups under the Object-oriented initiative which in effect became the sole theatre in which the Rekursiv showed itself to the world. Originally many groups anticipated microcoding instruction sets tailored towards their problem areas. When the Rekursivs were delivered, the only language compiler supplied with them was a C compiler that compiled code which executed in one of the Rekursiv's stacks and allowed a very primitive interface to the object store. At the time no mention was made of any other software for the Rekursiv, including Lingo, and many groups concentrated on adding to the microcoded

instruction set that supported C ([102]).

About six months into the project, it became apparent that some groups had received copies of the language Lingo (as well as a Prolog and a Forth implemented in Lingo) on an ad hoc basis, normally because they had been finding the C compiler inadequate and had been in communication with Linn-Smart. At the second Rekursiv workshop, reports by these groups on the efficacy of Lingo circulated and in the ensuing discussion it became quite clear that the Rekursiv was intended primarily as a Lingo engine and that in fact the language predated the processor. By the end of the initiative, with the demise of Linn-Smart the focus was completely on Lingo. Indeed many groups supported the intention to attempt to carry on **Lingo** development on other platforms.

This 'shifting goal-posts' period was unfortunate since in retrospect a clearer and cleaner justification for the Rekursiv could have been made by focussing on the design of the language Lingo. Its ancestry is the programming through types school of thought. Proponents of this line of attack include Harland himself ([51]), Burstall and Lampson ([13]) with their language Pebble and Milner ([77],[78]) and the polymorphism of SML.

The flat world of a conventional view of memory (and in this respect von Neumann machines and RISC are equivalent) sits uneasily with program-

ming through types. Since these machines do not support the storage of semantics along with data and do not support the variety of sizes and shapes of the abstract structures on a programmer's palette, they are forced to resort to tortuous control flows to manage expressive programs. Yet changes in control flow are precisely what defeats the features that could increase their bandwidth.

So this is the semantic gap – advances in technology (applied to an essentially unchanged architecture) will give the same improvement in performance to software produced from inexpressive C as they do to software produced from expressive functional or object-oriented languages and so C will always be used by preference. (Or put in other terms, RISCs support C and tolerate Smalltalk, say, only by more or less translating to C and playing by the rules of the game in a C world!).

This vicious circle can perhaps be broken by an architectural change that allows technology to support the expressivity that language design has given the programmer. If types are part of the palette (just as arithmetic and decision making are conventionally) then the architecture should allow the technology to work directly on types (just as the hardware works directly on arithmetic and decision making).

Chapter 4

The language DEAL

4.1 Introduction

Traditionally, database management systems were designed to meet needs from business data processing applications. Areas such as Computer Assisted or Automated Design are better supported by languages of Turing equivalent power and with expressivity at least as high as that of modern programming languages ([92]).

For several years, a group at Dundee Institute of Technology had been involved with the development and utilisation of a relational query language, DEAL ([26],[90],[112]). Some of this work was directed at using DEAL to show that its enhanced expressivity made problems in certain application

areas more tractible. These areas included –

- History – relational databases with an inbuilt model of time [90] allow selection predicates to involve temporal relations. This is of use in a wide variety of areas including, within engineering, design version and configuration control.
- Graphics – CAD systems necessitate the integration of the Database Management System with the ability to view and operate on database objects graphically within the same language ([112]).

The usefulness of the language DEAL in real applications at the institute was limited by the efficiency of the implementation and the language was very much used as a research model against which to test ideas for further language development [92] and to carry out experiments in algebraic specification of the Relational Algebra [91].

DEAL (DEductive ALgebra) is a relational language. DEAL's proposer and designer, Deen, ([26]) was attempting to provide 'a unified framework for both conventional and deductive database processing.'

Rather than supporting knowledge based systems by providing Prolog, say, with an interface to an underlying relational database, in DEAL the relational language is extended. "Deductions" are regarded as the generation

of new facts from existing facts (extensional database) using deduction rules (intensional database).

Despite Deen's nomenclature and terminology, it is hard to see that modern interpretations of the words 'deduction' and 'deductive' are appropriate to DEAL. Date ([24]), writing on the use of such terms, describes deductive DBMS as follows:

Deductive DBMS: A DBMS that supports the proof-theoretic view of databases, and in particular is capable of deducing additional information from the extensional database by applying inferential (or deductive) rules that are stored in the intensional database. A deductive DBMS will almost certainly support recursive rules and so perform recursive query processing.

As will be seen in this chapter's description of DEAL, the language has no real notion of 'inferential rule' in any deep sense. It does have functions, which can be called recursively. These however are imperative, result-returning subroutines that can modify variables. Some syntactic features (link variables) allow the programmer to write functions that have a surface similarity to the rules of Prolog. However, deduction is not directly supported any more than it is in a general purpose programming language such as C or Pascal.

4.2 Syntax

The concrete syntax of the DEAL interpreter that was implemented is described by the following extended BNF where the metasymbols { and } are being used to denote zero or more occurrences of the enclosed and the metasymbols [and] are used to indicate the optional (zero or once) occurrence of the enclosed. In addition, all non-terminals of the grammar are enclosed in angle brackets, < and >; terminal strings are enclosed in quotation marks, "; entities neither enclosed in angle brackets or quotation marks denote terminal *classes* whose syntactic description is not further expanded. An example of this is Identifier which denotes all terminal character strings which start with an alphabetic character and are followed by (zero or more) alphabetic or numeric characters.

```

<input>      ::= <input1> ";" { <input1> ";" }
<input1>    ::= <defn> | <filecommand> | <expr>
<expr>      ::= <term> { <binOp> <term> }
<term>      ::= <factor> [<block1>] [<block2>]
              {<arithOp2><factor> [<block1>] [<block2>]}
<factor>    ::= Relation | Integer | String | <function>
              | Var | LVar | LAMVar | Identifier
              | "(" <expr> ")" | <linkblock>
<linkblock> ::= <block1> "where" <expr> "{"<predicatelists>"}"
<predicatelists> ::= <predicate> {","<predicate>}
<block1>    ::= "[" <selectionList> "]"
<block2>    ::= "where" <condition>
<setOp>     ::= "*?" | "++" | "--" | "**"
<binOp>     ::= <setOp> | <arithOp1>
<arithOp1> ::= "+" | "-"
<arithOp2> ::= "*" | "/"
<stmt>      ::= <asgn> | <whileStatement> | <ifStatement>
              | "{" <stmtList> "}"
<asgn>      ::= <asName> ":@" <expr>
<whileStatement> ::= "while" <cond> <stmt>
<ifStatement>  ::= "if" <cond> <stmt> ["else" <stmt>]
<stmtList>    ::= { <stmt> ";" }
<asName>     ::= Var | Relation | Identifier | LAMVar | Function
<cond>       ::= "(" <predicate> ")"
<selectionList> ::= <expr> { "," <expr> }
<condition>  ::= <predicate> { "and" <predicate> }
<defn>       ::= "func" <funcName> "(" <paramList> ")" <stmtList>
<paramList>  ::= [<param> ":" <declaration>]
              { "," <param> ":" <declaration> }
<param>      ::= Identifier
<declaration> ::= "int" | "rel" | "at" | "char"
<header>     ::= [ "(" <argList> ")" ]
<argList>    ::= <expr> {","<expr>}
<funcName>   ::= Identifier
<predicate>  ::= <expr> <relOp> <expr>
<relOp>      ::= ">" | "<" | ">=" | "<=" | "=" | "!=" | "||" | "&&"
<constant>  ::= Integer | String

```

```

<function>      ::= "card" "(" <expr> ")" | "#" "(" <expr> ")"
                  | Function "(" <argList> ")"
<filecommand>  ::= "run" String | "load" (Identifier | Relation)
                  | "save" Relation

```

DEAL allows SQL-like queries. For example, given a relation **EMP** (for employees) with scheme **ENAME, DNO, SAL** (employee name, department number and salary) we can have

- **EMP [ENAME]** – gives the relation containing just employee names.
- **EMP [ENAME, SAL]** – gives the relation containing employee names and their salaries.
- **EMP [ENAME,DNO,NEWSAL := 1.1 * SAL]** – gives a new relation where each employee's salary is increased by 10 per cent.
- **EMP where SAL > 15000** – gives the relation with scheme **ENAME, DNO, SAL** where each employee earns over 15000. This is an example of a *tuple predicate*.

Functions can be defined in DEAL. For example

```

func fac (n : int )
{
  if n = 0
    fac := 1
  else
    fac := n * fac(n-1);

```

```
};
```

The parameters to a function can also include relations and attribute names.

A more relevant example of a function that returns relations, is the following:

```
func ancestor( x : char)
{
  temp := (parent where childname = x ) [ parname];
  if (card(temp) = 0)
    ancestor := temp
  else
    ancestor := temp++ancestor(temp);
};
```

Here, the existence of a relation `parent` with scheme `(parname, childname)` is assumed within which each tuple represents a parent relationship. Given the above function definition, evaluating an expression such as

```
ancestor("Rachel_Natanson");
```

would result in a relation with scheme `parname` where each tuple contains either a parent of "Rachel_Natanson" or the parent of another member of the relation. In other words the result is the set of ancestors of "Rachel_Natanson" that are known to the system via the relation `parent`.

Some explanation of the form of the `ancestor` function is needed: on the face of it, the function takes a single parameter of type "char" and yet it is

being *called*, within the recursive step, with the actual parameter `temp` which must have type relation as a consequence of the preceding assignment. The semantics of function application employed by the interpreter are such that, should an actual argument to a function be a relation where a simpler type was expected, the actual argument is considered to be a collection of individual values to which the function is applied in turn and then the individual results are combined together using the relational union operator. This is similar to an implicit *map* operator, as used within functional programming to apply the same function to all the elements of a list. Where the conventional *map* is related to the *list* constructor commonly known as `cons`, the implied operator here is related to the *relational* operator union.

Clearly, to begin to approach the problem of calculating such things as transitive closures some ‘higher order’ construct is necessary. Given the basic nature of the DEAL approach, the mechanism as above was chosen as representing a trade-off between semantic and syntactic opacity (which was already perceived to be high).

The “deductive” nature of DEAL is apparent in two facets – functions and the possibility of recursion allow the computation of transitive closures. Additionally, a syntactic feature proposed by Deen ([26]), called ‘link elements’, allows queries to be expressed in a Prolog like form (this feature was

not implemented by Sadeghi ([90])). As an example of this, consider the classic query 'Paul likes everyone who likes wine' against a relation *likes* with schema (name,object). An expression that returns the answer relation is –

```
[ name := "Paul", object := x ] where likes {x=name, object="wine"};
```

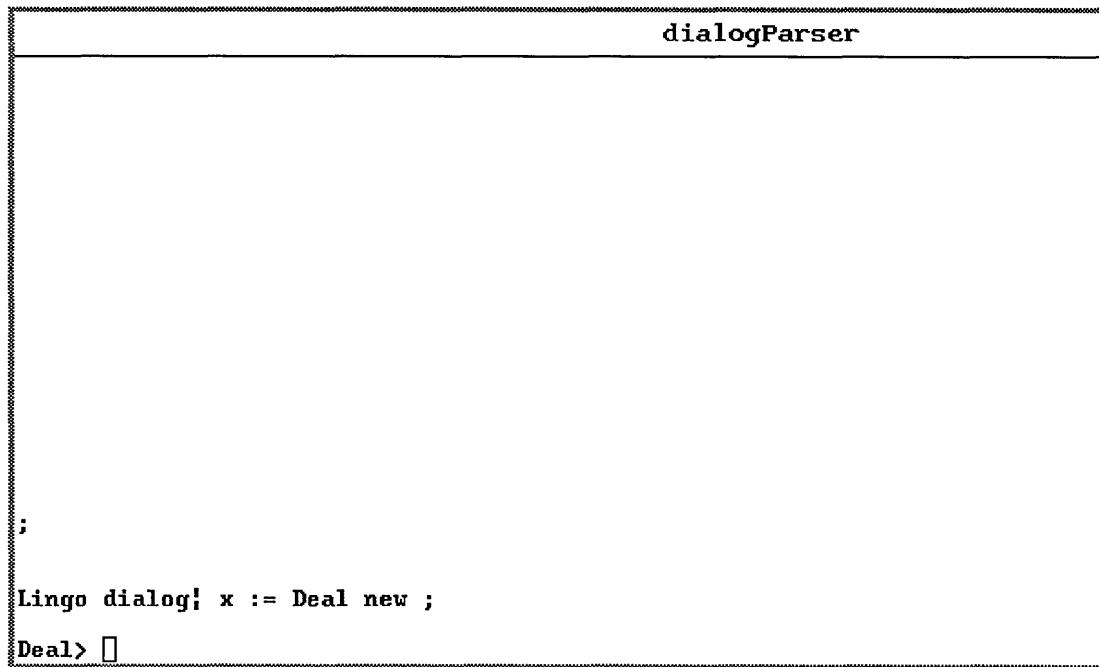
The result can be unioned with the original relation *likes* and the final result used to update *likes*. (Note that this concrete syntax is not exactly that proposed by Deen and was adopted in order to facilitate parsing. The tension in the marriage of the syntax of SQL with that of Prolog reaches breaking point here. The underlying model, though, (in terms of the abstract syntax) does allow unification to be established via relational algebra operations).

4.3 Using the DEAL interpreter

An interactive session with the DEAL interpreter is started by creating an instance of a Deal class object. For example, assuming *x* is a Lingo process variable:

```
x := Deal new;
```

will initiate an interactive session that will end when a <ctrl> D is entered for end of file (as shown in the following diagram).



The object x still exists, with whatever environment the Deal object had, and can be queried as in:

```
x ask:"parts where pweight > 14" ;
```

This query returns an object of Relation class, the relation being the subset of the parts relation whose weight attribute is greater than 14.

The diagram shows this in operation.

```
dialogParser
;
Lingo dialog! x ask: "parts where pweight > 14" ;
string pnum string pname string pcolour int pweight string pcity

p6 cog red 19 london
p3 screw blue 17 rome
p2 bolt green 17 paris

Lingo dialog! □
```

Within an interactive session, the DEAL interpreter recognises two categories of statement.

- **Queries** – the purpose of the language is to answer relational algebra queries such as

```
parts where pweight > 14
```

but the interpreter will accept any expression – relational or arithmetic – and print its result.

- **Environmental statements** – these change the context in which queries are evaluated. Functions that take parameters and return results, for example can be defined; Relations can be loaded from text files and so on.

4.4 Conclusion

This chapter has introduced the major syntactic and semantic features of the language DEAL. The next chapter, chapter 5, describes the detail of the analysis and synthesis phases of the interpreter. The execution of DEAL programs proceeds by traversing data structures, synthesised during interpretation, and invoking more primitive operations (such as arithmetic and relational algebra operators) as indicated by the synthesised data structures.

Following chapter 5, chapter 6 describes the development of the main set of these operations, those involved in the relational algebra, from a formal specification.

Chapter 5

Implementing the language

5.1 Introduction

The last chapter gave a description of the syntax of DEAL. This chapter deals with its implementation – the method by which the language interpreter was effected.

The term interpreter embraces the collective action of a number of objects. Its overall action can be demarcated into three phases:

- **Analysis** - the *recognition* of the basic lexical elements of the language (keywords, literal constants and so on) and the syntactic structures. The former is effected in a subphase known as *Lexical Analysis* and the latter by *Syntax Analysis*.

- **Synthesis** - the *construction* of data structures containing the essential information (discerned in the analysis phase) needed to carry out the intended computation.
- **Execution** - effecting the computation.

Separate objects are used for lexical analysis and syntax analysis and these are termed the ‘scanner’ and the ‘parser’ respectively.

The parser is viewed as the ‘root’ object of the interpreter since the main thread of execution through the interpreter is contained within it as follows:

- The code to carry out the actions of the synthesis and execution phases is interspersed through the code of the parser.
- The parser calls methods of the scanner as required.

It is important to distinguish the strategy employed here from the conventional model of an interpreter or compiler. Conventionally, phases of a translator such as lexical analysis or syntax analysis, produce entire data structures which are then operated upon by a following phase. The syntax analyser, for example, normally concerns itself with building a *parse tree* which the semantic analysis phase and code generation phase (or evaluation phase, for an interpreter) utilise. The strategy upon which this work is based does not build an *explicit* parse tree; this tree does exist, though, as

the *thread of execution* through the syntax analyser's recogniser procedures. A consequence of this is that the processing of this *implicit* parse tree is intimately bound with its construction; that is that semantic analysis and synthesis (code generation or evaluation) occurs concurrently with syntax analysis. The interpreter, in a sense, *parser driven*.

The general strategy for the interpreter's analysis phases (lexical analysis and syntax analysis) is covered in appendix C. The syntax analyser uses a predictive top-down method (recursive descent) to recognise the language's syntactic classes. Code to effect semantic actions is interspersed within the parser's code. In general, these semantic actions are synthetic and construct data structures which the interpreter can then traverse at an appropriate point and thus 'execute' the original DEAL source code.

The objects within these synthesised data structures approximate to the 'object code' that the compiler translates source code into. These are coded in Lingo. Underlying these objects is a collection of objects providing primitive functionality to support relational algebra operations. This last layer is dealt with in the next chapter.

Given the above, the operation of the interpreter will be described in the following way –

The BNF of a syntactic entity will be given and the points at which

semantic actions will be inserted will be noted. The semantic actions will then be described along with any objects contained within the data structures that the semantic actions synthesise.

5.2 The topmost levels

The *distinguished symbol* of DEAL's grammar is `<input>`. The production that defines it is

```
<input> ::= <input1> ";" { <input1> ";" }
```

The recogniser procedure (in Lingo) for `<input>` is an instance method of the interpreter object. In the following diagram, a heavier type has been used to emphasize the similarity of the recogniser procedure's structure and the extended BNF production on which it was styled:

```
self input {}
{
  self input1 ;
  self mustBe: ";" ;
  while ((Vector ["|" "Relation" "Integer" "String" "Var"
                 "Identifier" "(" "func" "run"
                 "Function" "card" "#" "load" "save" ])
         includes: (scanner token)) do
  {
    self input1 ;
    self mustBe: ";" ;
  }
}
```

The condition controlling the while in the above represents the lookahead and predictive nature of the parsing strategy. The condition amounts to ‘does the current token belong to the *director* set for <input1> (i.e. the set of tokens that can appear on the extreme left of an instance of <input1>)’.

The above code is, however, only reproduced here so that, in what follows, the code necessary for *parsing* can be distinguished from the code inserted to effect *semantic* actions. The procedure above may appear a little dense. This is because, in practice, the code for all recogniser procedures was generated

automatically by a *parser generator* (discussed in appendix C) which was constructed specifically for this work. The parser generator also assisted in the insertion of code for semantic actions.

The semantic actions, $A_1 \dots A_3$, associated with this production are indicated by annotating the BNF thus:

$$\langle \text{input} \rangle ::= A_1 \langle \text{input1} \rangle A_2 \text{ ";" } \{ \langle \text{input1} \rangle A_3 \text{ ";" } \}$$

The recogniser method has code for these actions interspersed amongst the code given above for parsing at the points indicated by the annotated BNF. The following diagram is intended to illustrate this by reproducing the original parsing code in a lighter print:

```
self input {}
{
  ... code for action A1 ...
  self input1;
  ... code for action A2 ...

  self mustBe: ";" ;
  while ((Vector [{" "Relation" "Integer" "String" "Var"
                  "Identifier" "(" "func" "run"
                  "Function" "card" "#" "load" "save" }])
         includes: (scanner token)) do
  {
    self input1;
    ... code for action A3 ...
    self mustBe: ";" ;
  }
}
```

In this case the actions (associated with *this production*) are

- A_1 – open the initialisation file ‘init.deal’, execute it, then close it and then initialise the exception handling mechanism.

Once the file (init.deal) is opened, the scanner is informed to take its input from it and the same parsing loop as the above is executed until the end of file is reached. The file is then closed, the scanner informed to take its input from standard input and the exception handling mechanism directed to return control to this point in the process (so that

syntax and run-time errors during the session will result in control coming back to the recognition of the almost top level syntactic entity `<input1>`). After this initialisation phase, a prompt 'Deal>' is printed on the user window to indicate that the interpreter is in interactive mode.

- A_2 – At this point, the actions associated with `<input1>` have been executed and so the prompt, 'Deal>' is printed on the user's window.
- A_3 – Again the actions associated with `<input1>` have been executed and so the prompt is printed.

The distinguished symbol `<input>` is not particularly interesting since it is merely describing that a session with the interpreter consists of an indefinite sequence of `<input1>`s separated by semi-colons. The above example does serve the purpose of elucidating the manner in which the implementation is to be described.

Turning to the syntactic entity `<input1>`, its BNF is

```
<input1> ::= <defn> | <filecommand> | <expr>
```

This is expressing the differentiation of top-level DEAL statements into the categories

- `<defn>` – a function definition.
- `<filecommand>` – The two commands ‘load’ and ‘save’ allow retrieval and storage of relations in external files in which their schemes are also described. The ‘run’ command executes DEAL statements contained in a file. This facility is intended to be used primarily for storage of function definitions.
- `<expr>` – this last category represents expressions which the user wishes to be evaluated and the result shown.

It is only this last alternative that has a semantic action associated with it (*at this level* – the others have semantic actions within the recognition procedures that are called as a consequence of their own recognition). The actions required are to evaluate the data structure synthesised by the recogniser procedure for `<expr>` and then print the result. More concretely, actions are associated as so:

$$\langle \text{input1} \rangle ::= \langle \text{defn} \rangle \mid \langle \text{filecommand} \rangle \mid A_1 \langle \text{expr} \rangle A_2$$

The actions are as follows

- A_1 – ‘remember’ the value returned by the call to the recognition procedure `<expr>` (this will be a data structure whose traversal leads to the evaluation of the recognised expression). This ‘remembering’ is

effected by assigning the return value of the procedure for `<expr>` to a local (to the method for `<input1>`) variable. This local variable is called `result`.

- A_2 – evaluate the expression and print the result. The data structures representing expressions are so arranged that their traversal is effected via a method with message selector `evaluateWith: and:` which takes two arguments. The first argument is the global environment, a dictionary containing the current bindings of all global DEAL variables; the second is the current local environment, a dictionary for accessing the current local DEAL data area (function parameters, locals and so on). The interpreter maintains two Lingo variables for these: `globals` and `locals`. The action A_2 is thus effected by the inclusion of the following Lingo code.

```
self println: (result evaluateWith: globals and: locals);
```

These two actions are inserted into the recogniser procedure for `<input1>` as indicated in the following diagram where a lighter print is being used again to indicate parsing code:

```
self input1 [ result ]
{
  if Vector ["func"] includes: (scanner token) then
    { self funcDefn }
  else
    { if Vector ["load" "save" "run"] includes: (scanner token) then
      { self filecommand}
      else
        { /* A1 - `remember' the expr */
          result := self expr;
          /* A2 - evaluate the expr and print */
          self println: (result evaluateWith: globals and: locals);
        }
      }
}
```

Having now established the principle of operation of the interpreter the next two sections will deal with the detail of handling the major language aspects of function definitions and expressions.

5.3 Function definitions

Function definitions have their top level syntax described in the production for <defn> as follows:

```
<defn> ::= "func" <funcName> "(" <paramList> ")" <stmtlist>
```

As a concrete example, to facilitate explanation, consider the following

DEAL function definition for the factorial function

```
func factorial( n : int)
{
  if n=0 then
    factorial := 1;
  else
    factorial := n * factorial(n-1);
}
```

Even though DEAL is more concerned with computation involving relations, the above example is useful to elucidate the interpretation of *function definitions* – the next section that deals with the interpretation of expressions will provide the detailed operation of the interpreter in handling relational computation as well as function invocations.

Considering first the concrete example of factorial, the overall action required of the interpreter when the definition has been completely recognised is to enter the data structure representing the function's body into the global dictionary (and use the string "factorial" as the key). In addition, the parameter list - in this case containing only the string "n", is stored in a symbol table within the scanner (in a dictionary associated with the string "factorial" as the key) so that later when the function is invoked a local environment can be built – this will consist of a dictionary whose keys are the function's name (factorial) and all the formal parameters' names.

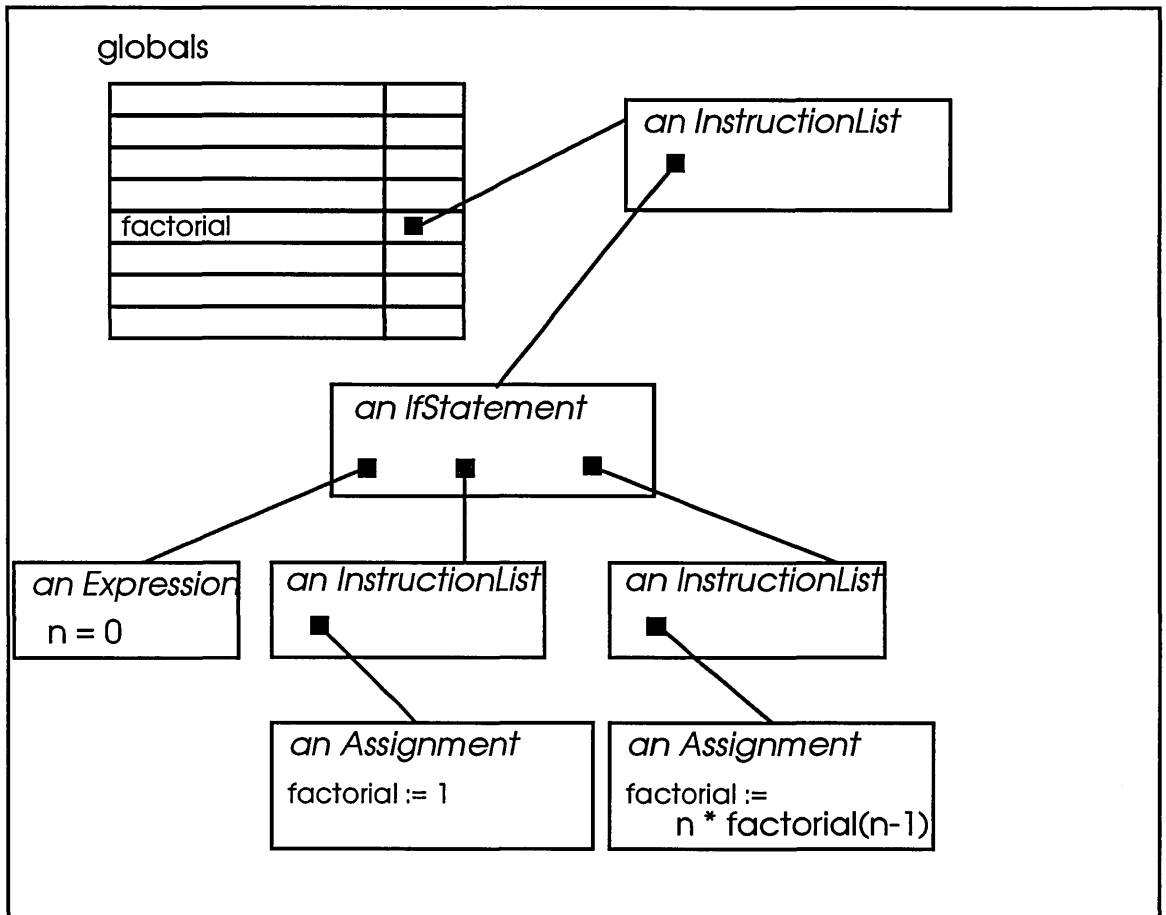
The function body is represented by an instance of *InstructionList*, which

is a collection class. The members of the collection are the ‘compiled’ forms of the individual statements from the function body and fall into one of the following classes (whose behaviour will be described later):

- **WhileStatement**
- **IfStatement**
- **Assignment**

The only statement in the factorial function’s body is an if statement.

The following diagram depicts the situation:



Turning now to the general syntax for a function definition (rather than the concrete example of the factorial function), actions are associated as follows:

`<defn> ::= "func" A1 <funcName> "(" A2 <paramList> ")" A3 <stmtlist> A4`

- A₁ – the formal parameters to a function can be of type integer, rela-

tion, character string or attribute (used to pass the name of a table's column). These are denoted by 'int', 'rel', 'char' and 'at'. The scanner maintains two symbol tables so that it can check that an identifier is in scope. The first of these holds information about parameters of the first three types (called LVars for 'Local Variables') and the second holds information about attribute parameters (called LAVars for 'Local Attribute Variables').

The first action to be performed is to inform the scanner to create new symbol tables. The name of the function is also 'remembered' at this point (by assigning to a local Lingo variable) so that at the end of this recognition procedure the binding of the function name with the data structure describing how to perform it can be included in the interpreter's global symbol table. In the following, name is a Lingo local variable used for 'remembering':

```
.
.
/* Have entered a new local scope :
   start new symbol tables in scanner */
scanner freshLAVars; scanner freshLVars;
/* recognise and 'remember' the function's name */
name := self funcName;
.
.
```


- A_2 – when function calls are made, local environments are created associating formal parameters with actual parameters. The purpose of this action is to ‘remember’ the list of formal parameter names (which will be returned by calling the recogniser procedure for `<paramList>`) so that it can later (in action A_3) be associated with the function. In the following, `params` is a Lingo local variable used for ‘remembering’:

```
.  
.br/>params := self paramList;  
.  
.
```

- A_3 – The scanner maintains a dictionary that associates function names with a list of their formal parameter names which is later used at function application to create the local environment. Within this action, the previously remembered parameter name list is entered into the appropriate dictionary within the scanner under the function’s name.

In addition, the data structure representing the statement body of the function definition (returned by the recogniser procedure for `<stmt>`) is remembered in the local Lingo variable `body`.

```
.  
.br/>scanner bindFunction: name to: params;
```

```
body := self stmt;  
.  
.
```

- A_4 – the global environment (a dictionary) can now be updated with an entry associating the functions name with the data structure representing its body (remembered during A_3 in the local body).

```
.  
.  
globals at: name put: body;  
.  
.
```

The analysis and synthesis associated with function definitions is now followed by the analysis and synthesis of expressions.

5.4 Expressions

The form of expressions is defined by the following Extended BNF productions (where the metasymbols [and] are used to denote the optional single occurrence of the enclosed and { and } are used to denote the optional multiple occurrence of the enclosed):

```
<expr> ::= <term> { <binop> <term> }
```

```
<term> ::= <factor> [<block1>] [<block2>] { <arithop2> <factor> [<block1>]  
  [<block2>]}
```

```
<factor> ::= <Relation> | <Integer> | <String> | <Function> | <Var>  
  | <LVar> | <LAVar> | <Identifier> | "(" <expr> ")"  
  | <linkblock>
```

The syntactic entity `<binop>` represents the binary operators on relations and the lowest precedence binary operators (addition and subtraction) of integer arithmetic. The entity `<arithop2>` represents the binary arithmetic operators multiplication and division. The further productions defining `<binop>` and `<arithop2>` are not reproduced here for the sake of clarity. The parsing procedures for these recognise the terminal character sequences that represent them and return appropriate Lingo classes which the higher level recogniser procedures for `<term>` and `<expr>` can instantiate with appropriate instance data. The following table describes the operators:

Terminal characters	Description	Class	Syntactic class
*?	intersection	Intersection	<binop>
++	union	Union	<binop>
--	difference	Difference	<binop>
**	cartesian product	CartProd	<binop>
+	integer addition	Plus	<binop>
-	integer subtraction	Minus	<binop>
*	integer multiplication	Times	<arithop2>
/	integer division	Divide	<arithop2>

The entities in the production for <factor> mainly represent items that have a value (either as literal constants or as bindings to a variable). For example, the entity <Integer> represents integers and its recogniser procedure returns objects of type Constant which contain the actual value of the integer constant. The recognition of variables produces objects which when ‘evaluated’ in the execution phase return the values they are currently bound to by looking up the appropriate global or local environment.

The entities <block1> and <block2> relate to *projection* and *selection* in relational expressions which will be dealt with later.

The next four subsections give a detailed treatment of `<expression>`. It is convenient to cover (in the first subsection) expressions involving the binary operators. The second subsection deals with relational expressions involving `<block1>` and `<block2>`.

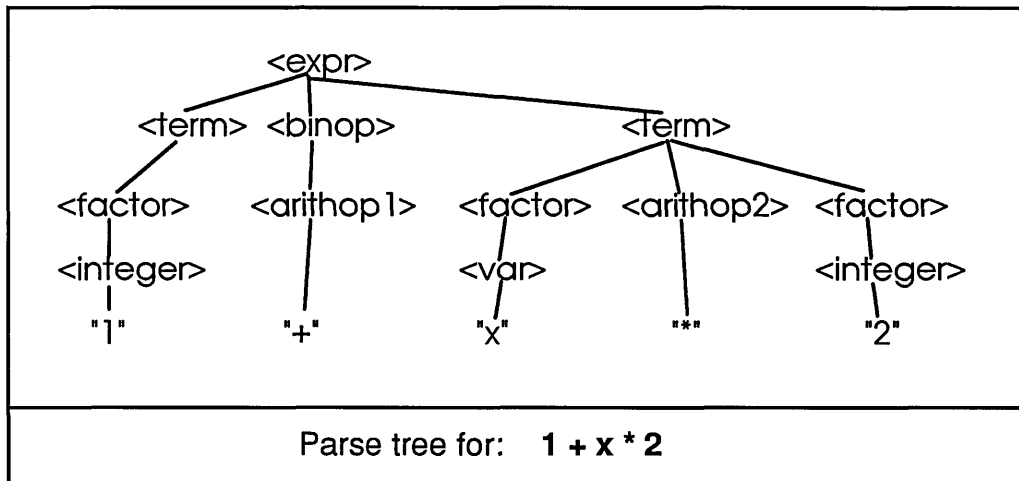
In the third subsection, the treatment of DEAL's *link elements* is explained by considering the entity `<linkblock>`. and the fourth subsection deals with function application.

5.4.1 Expressions involving binary operators

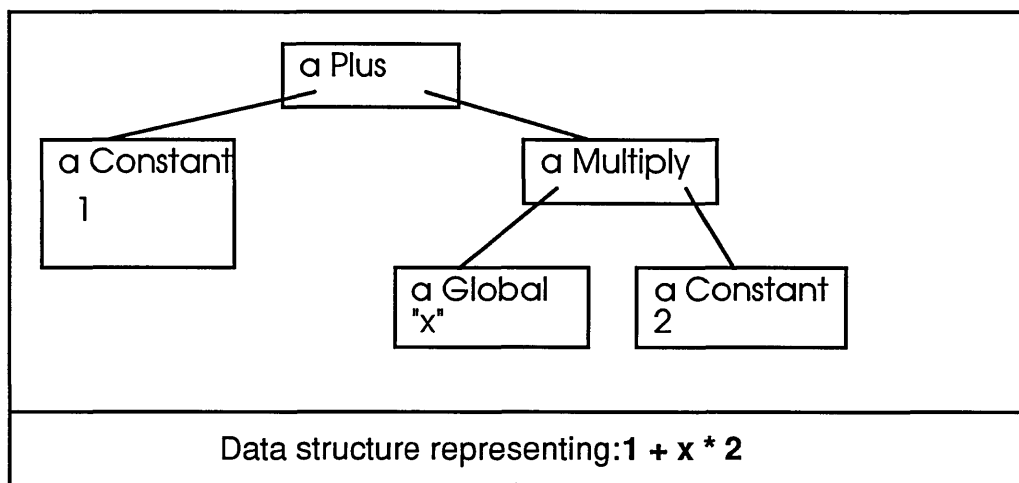
For now, to elucidate the general strategy, consider the integer expression:

$$1 + x * 2$$

where `x` is a global variable. The parse tree for this expression is given in the following figure where the leaves have been annotated with the terminal sequences from the source code:



The data structure created by the synthesis phase can be depicted as follows:



5.4.2 Selection and Projection

Before returning to the complete treatment of `<expr>`, two of its optional component clauses `<block1>` and `<block2>` will be described.

Both these entities essentially qualify a basic relational expression. The former, `<block1>`, is defined by the productions:

```
<block1>      ::= "[" <projectlist> "]"
<projectlist> ::= Identifier [ "!=" <expr> ]
               { ",", Identifier [ "!=" <expr> ] }
```

and the latter, `<block2>`, by:

```
<block2>      ::= "where" <condition>
<condition>   ::= <predicate> { "and" <predicate> }
<predicate>   ::= <expr> <relop> <expr>
<relop>       ::= ">" | "<" | ">=" | "<=" | "=" | "!="
```

The purpose of `<block1>` is to allow certain fields to be projected from the base relational expression which the clause qualifies. In the following, for example, the base relational expression consists of the variable parts, and the qualifier specifies that the `pnum` and `pweight` fields should be projected:

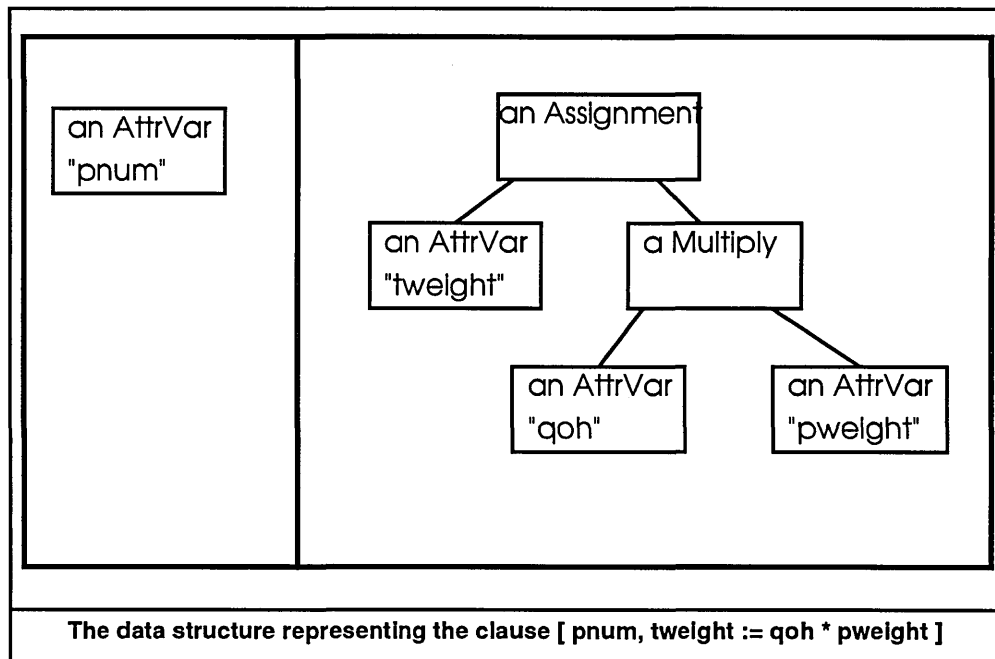
```
parts [ pnum, pweight]
```

Furthermore, the `<block1>` qualifier can be used to rename fields or indeed to calculate new fields. In the following, the relation `parts` has a scheme which includes `pnum` for part number, `qoh` for quantity on hand and `pweight`

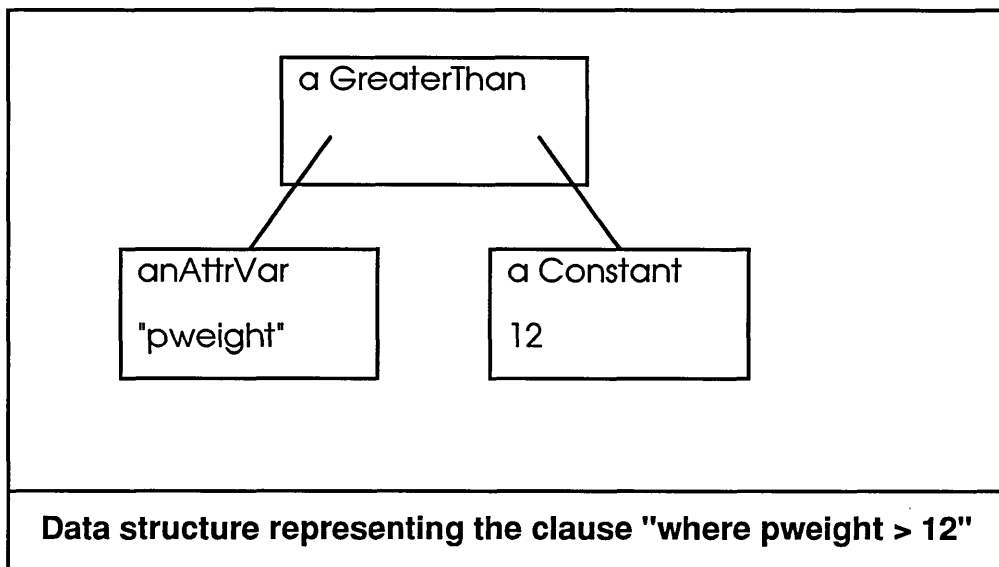
for the weight of a part. The expression evaluates to a relation whose scheme consists of `partno` (the part's number) and `tweight` which will be the total weight of all the parts in stock with that part number.

```
parts [ partno := pnum, tweight := qoh * pweight ]
```

To deal with this, the recogniser procedure for `<projectList>` returns a list each of whose entries is either an object representing the field name to be projected or an `Assignment` object and this list is returned unmodified by the recogniser procedure for `<block1>`. The clause `[pnum, tweight := qoh * pweight]` for example would result in the return of the data structure depicted in the following diagram:



The `<block2>` clause allows the application of a selection on the basic relational expression (the order of application of projection and selection should both a `<block1>` and `<block2>` be present is dealt with later). The recogniser procedure for `<block2>` returns a data structure containing the essential information needed to carry out the selection. For example, the clause `where pweight > 12` would result in the structure illustrated in the following diagram:



A more complete description of `<expr>` can now be given. The extended BNF definition is annotated as follows:

$$\langle \text{expr} \rangle ::= A_1 \langle \text{term} \rangle \{ A_2 \langle \text{binop} \rangle A_3 \langle \text{term} \rangle \} A_4$$

In describing the actions use of two local (to the recogniser procedure for

<expr>) variables, `theOp` and `result`, is made:

- A_1 – remember the result returned from the call to the recogniser procedure for <term>.

```
result := self term;
```

- A_2 – remember the class of the operator (which is returned from the recogniser procedure for <binop>).

```
theOp := self binOp;
```

- A_3 – by instantiating the operator recognised in A_2 , combine the subexpression built so far (in `result`) with the subexpression returned by the next call to the procedure for <term>. Recall that the local variable `theOp` contains the *class* of the recognised operator. All such operators have a class method (with selector `of:` and `and:`) for instantiation. The `of:` and `and:` parameters supply the left and right subexpressions to the instantiated operator:

```
result := theOp of: result and: (self term);
```

- A_4 – the complete expression has been recognised and so the variable `result` contains the data structure representing the expression; this is returned (to the procedure that called the procedure for <expr>).

```
  ^ result;
```

The treatment for `<term>` follows a similar line, but is slightly complicated by the need to ensure that should both a `<block1>` (for projection) *and* a `<block2>` be present, the resultant data structure when ‘executed’ will result in the selection being performed *before* the projection (since the selection may involve fields which do not appear in the projection).

```
<term> ::= A1 <factor> [A2<block1>] [A3<block2>]A4
          {A5<arithop2>A6<factor> [A7<block1>] [A8<block2>]A9}A10
```

The actions:

- A_1 – remember the result of the call to the procedure for `<factor>`.

```
  result := self factor;
```

- A_2 – if this action is taken there *is* a `<block1>` clause. Set a flag to mark that a projection must be constructed and remember the result returned by the call to the procedure for `<block1>`.

```
  projectFlag := Boolean true;
  projectParams := self block1;
```

- A_3 – similarly, if this action is taken there *is* a `<block2>` clause. Set a flag to mark that a selection must be constructed and remember the result returned by the call to the procedure for `<block2>`.

```
selectFlag := Boolean true;
selectParams := self block2;
```

- A_4 – if the selection flag is set, construct the selection on the result constructed so far and clear the flag. Then perform a similar action if the projection flag is set:

```
if selectFlag do
{
  result := Select of: result where: selectParams;
  selectFlag := Boolean false;
}
if projectFlag do
{
  result := Project of: result over: selectParams;
  projectFlag := Boolean false;
}
```

- A_5 – remember the class of the operator returned by the call to the procedure for <arithop2>.

```
theOp := self arithop2;
```

- A_6 – remember the result of the call to the procedure for <factor>

```
tempResult := self factor;
```

- A_7 – this is exactly the same as action A_2 .
- A_8 – this is exactly the same as action A_3 .

- A_9 – this is similar to action A_4 in that any necessary selection and projection is constructed. In addition, the operator remembered in action A_6 is used to combine the expression constructed so far with the subexpression stored in `tempResult`.

```
if selectFlag do
{
  tempResult := Select of: tempResult where: selectParams;
  selectFlag := Boolean false;
}
if projectFlag do
{
  tempResult := Project of: tempResult over: selectParams;
  projectFlag := Boolean false;
}

result := theOp of: result and: tempResult;
```

- A_{10} – the complete term has been recognised and so the variable `result` contains the data structure representing the term; this is returned (to the procedure that called the procedure for `<term>`).

5.4.3 Link elements

An alternative in the defining production for `<factor>` is `<linkblock>`. This is where the *link elements* of DEAL are used. Recall the example DEAL statement:

```
[name := "Paul", object := X ] where likes {name=X, object="wine"};
```

In the above, X is known as a *link* element. Link elements are a syntactic feature proposed by Deen to allow queries to be expressed in a Prolog like form. The above query represents a relation with scheme (name,object). The relation will have, for every tuple existing in the relation likes with object field value "wine", a tuple with object value derived from the name field of likes and its name field with value "Paul".

To make this a little more concrete, suppose the relation likes is as follows:

name	object
Paul	beer
Bill	beer
Bill	wine
Louis	wine
Bill	Paul

The result of the query will be the following relation:

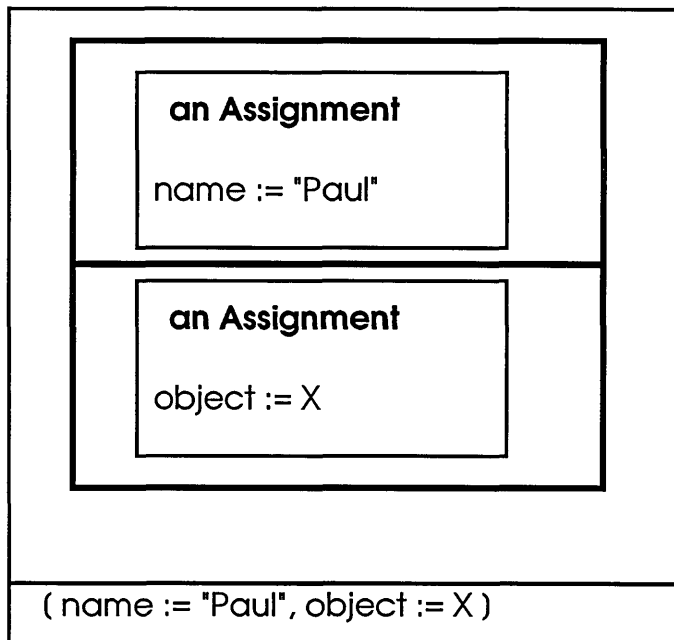
name	object
Paul	Bill
Paul	Louis

The extended BNF for `<linkblock>` (and related components) is:

```
<linkblock> ::= <block1> "where" <expr> "{" <predicatelists> "}"
```

```
<predicatelists> ::= <predicate> { "," <predicate> }
```

With reference to the example query, the call to the procedure for `<block1>` will return a list, called `projList1`, of two objects (both of which are actually instances of `Assignment`). The diagram shows this:

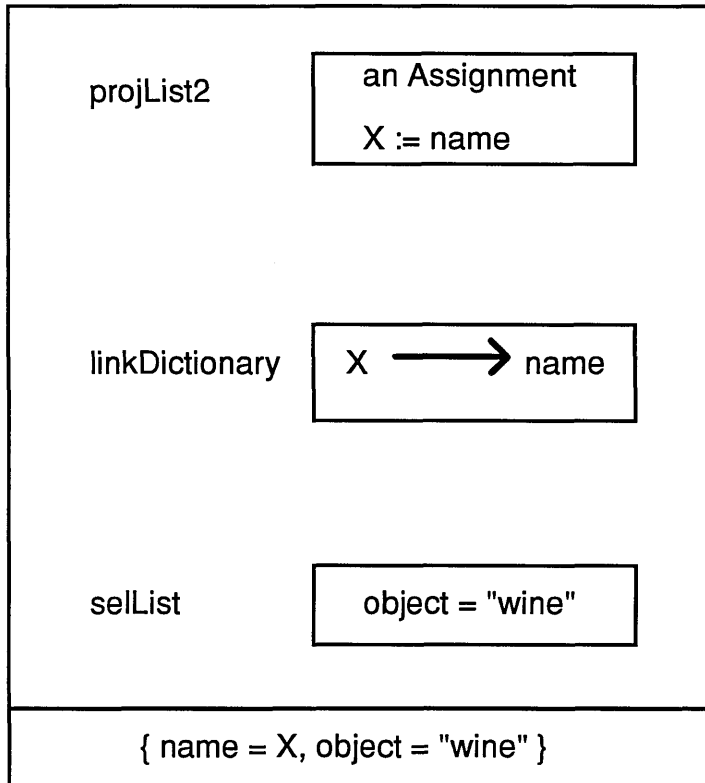


The call to the procedure for `<predicatelists>` will return a list containing the two predicates `'name = X'` and `'object = "wine"'`.

The first of these lists is scanned and must consist of Assignment objects. Whenever an Assignment has as its right hand side a single variable that is not in scope, it is assumed to be a link variable and added to a list (called `links`). For the example, `links`, will be a list containing a single element, the string `"X"`.

The second list (of predicates) is scanned; whenever a predicate is found involving a test for equality whose right hand side consists solely of a link element and whose left hand side solely of an attribute name, an Assignment object is created assigning the attribute name to the link variable. If an Assignment was created this is appended to a list called `projList2` and the association of the link element to the attribute name is recorded in a dictionary called `linkDictionary`. Otherwise the predicate is appended to a list of predicates called `selList`.

For the example, at this stage the relevant data structures (`projList2`, `linkDictionary` and `selList`) can be depicted as:



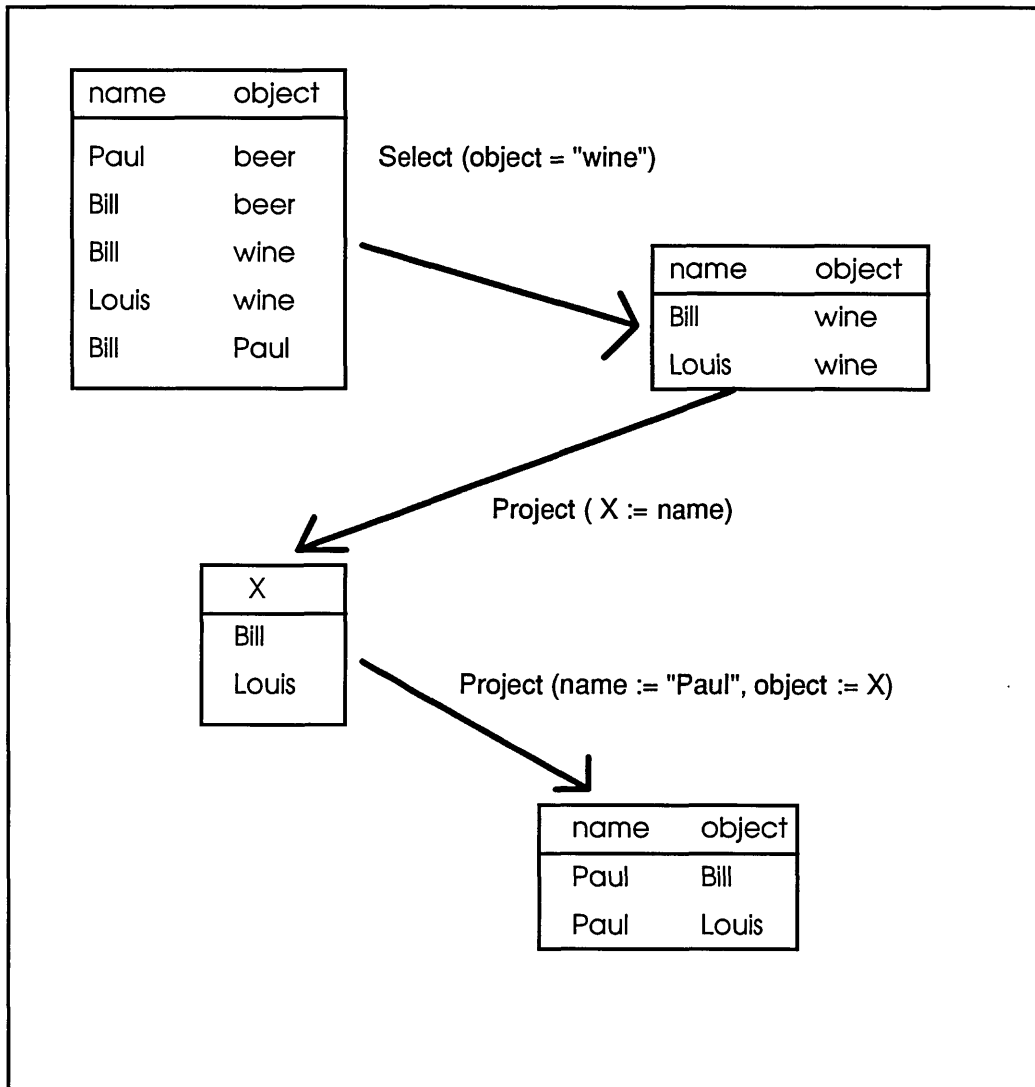
The logical conjunction of the individual predicates in the list `selList` is constructed and called `pred`.

The complete computation for the query can now be constructed. The run time order of the operations is to first of all evaluate the base expression in the `<linkblock>`. Then a selection (according to the predicate `pred`) is applied. To this is applied a projection (according to the information contained within `projList2`). Recall that this list will contain *Assignments*

(in this particular case ‘ $X := \text{name}$ ’). The projection operation in this case will reduce to a renaming of the field (from name to X).

Finally a second projection is applied, using the information contained in `projList1`. Again, in this example these are both assignments. The first, ‘ $\text{name} := \text{"Paul"}$ ’, is used to compute a new field, named name , for each tuple containing the value "Paul" . The second, ‘ $\text{object} := X$ ’, reduces to renaming the X field (of the result so far) to object .

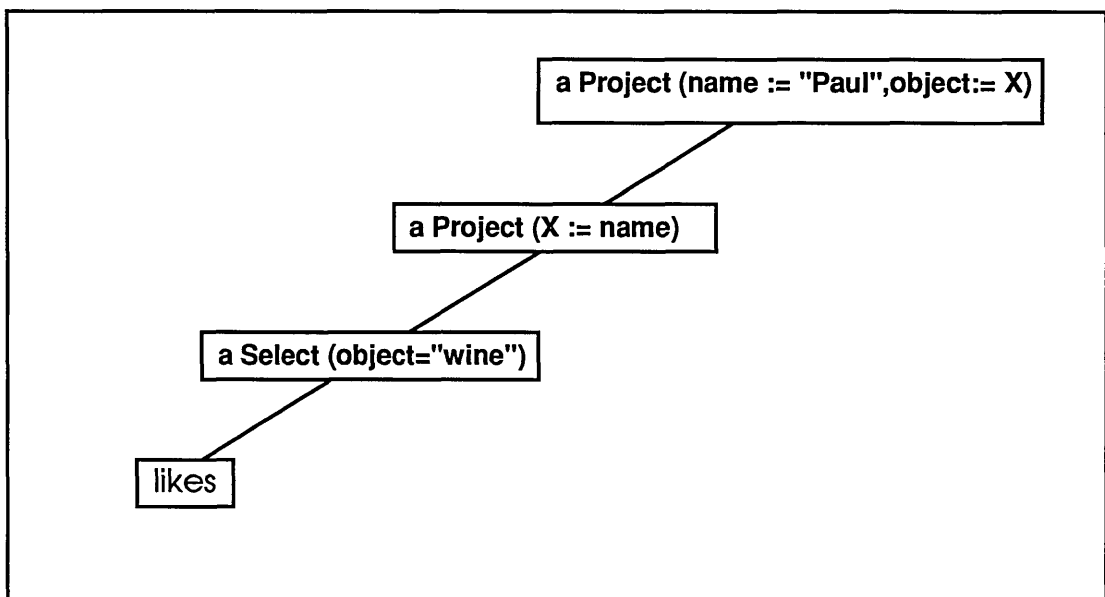
The following diagram traces the computation (within the context of the example).



The notion of link elements and the above exposition demonstrate a similarity with the principle of *unification* which underlies Prolog systems. This similarity is only of *intent*. Where unification is a symbolic computation, the

above is a computation on *relations*.

It must also be borne in mind that the computation (associated with link elements) described above is carried out *later* in the execution phase. The result of the *synthetic* phase is the data structure represented in the following diagram:



5.4.4 Function Application

Turning now to function applications, these have the form

```
<function> ::= Function "(" <argList> ")"
```

The entity `Function` above is unquoted: it can be considered to be a terminal *class* rather than a terminal token; it is recognised by the scanner and consists of an identifier that is the name of a previously declared (within the current interpreter session) function.

The recogniser procedure for `<arglist>` returns a list of objects representing expressions (one for each argument). An instance of the class `FunApp` is created. These objects hold three pieces of instance data:

- the *name* of the function.
- the list of formal parameter names of the function. These were stored (within a dictionary residing in the scanner) when the function declaration was analysed.
- the list of argument expressions.

This completes the description of the synthetic phase of the interpreter. The next section describes the execution phase.

5.5 The execution phase

The result of the analytic and synthetic phases is a data structure, all of whose nodes are objects which possess a method with selector `evaluateWith:and:.`

The two parameters to this method supply a global and a local environment respectively.

This set of objects together constitute the ‘virtual machine’ operators that supports the interpreter’s execution. It is convenient to group the discussion of these operators. The following subsections cover in turn the binary operators, the unary relational operators such as `Select` and `Project`, the operators representing variables and constants, the function application operator and finally the operators representing DEAL statements (`Assignment`, `IfStatement`, `WhileStatement`).

5.5.1 Binary Operators

All the binary operators of DEAL are represented by objects whose classes are all subclasses of the class `BinOp`. The inherited behaviour of these objects allows them to be instantiated (via a class method with selector `of:and:`). The two parameters to this method are used to point to the left and right subexpressions which are the operands of the operator.

The responsive behaviour of these objects to the `evaluateWith:and:` message is to first pass on the message to their left subexpression and then to their right subexpression. At that point the two results are used as the operands to the operation that the object represents and the overall result

constitutes the object's response.

The following table gives the Lingo class names used in the implementation of the interpreter as well as a description of their response when evaluated. All these classes have BinOp as their superclass:

Class	operation
CartProd	cartesian product of relations
Difference	difference of relations
Divide	integer division
Equal	equality test on either strings or integers
GreaterThan	the $>$ operation on either strings or integers
GreaterThanOrEqual	the \geq operation on either strings or integers
Intersection	intersection of relations
LessThan	the $<$ operation on either strings or integers
LessThanOrEqual	the \leq operation on either strings or integers
Minus	integer subtraction
Multiply	integer multiplication
NotEqual	inequality test on either strings or integers
Plus	integer addition

5.5.2 The Unary operators

These operators all operate on a single relation. Select and Project require in addition another operand; for Select this represents the boolean expression that is the criterion for selecting tuples from the relation; for Project this other operand is the list of fields to project from the relation. Recall that this list may also contain assignments indicating that a new field (computed from fields of the original relation) is to be derived.

The Hash operator is intended to be applied to a relation consisting of a single tuple with only one field and then to return the value of that sole attribute.

Class	operation
Select	selection
Project	projection
Card	cardinality
Hash	coercion

5.5.3 Variables and Constants

Constants are represented by objects of the class Constant. These have an instance variable holding the actual value being represented.

The various kinds of DEAL variable are:

- Global variables represented by instances of `Global`.
- Attribute references represented by instances of `AttrVar`.
- Local variables represented by instances of `LVar`.
- Local attribute variables represented by instances of `LAVar`.

Each of these contains an instance variable holding their string representation. Their responsive behaviour to being evaluated is to access the appropriate environment (global or local) using the string as the key.

5.5.4 Function Application

Function application nodes are represented by `FunApp` objects. These have three instance variables. One holds the function name, the second the list of formal parameters and the third the list of arguments (that is, the data structures for the expressions which evaluate to the arguments).

When evaluated, `FunApp` objects respond by building a new local environment, a dictionary associating with each formal parameter name the result of the evaluation of the corresponding argument. This new environment also contains an entry for the function name. The statement body of the function

(retrieved from an interpreter symbol table) is then evaluated (within the new local environment).

After execution of the body, the function result is then retrieved from the environment (where it has been stored under the function name) and is returned as the overall result.

The above operation is deviated from slightly in the case where an argument evaluates to a relation whereas the parameter list indicates an expectation of some kind of atomic value (such as a string or an integer). In this case, the function application is iterated through each tuple of the argument, treating the tuple as an atomic value. The union of all the individual results is returned as the overall result.

5.5.5 Statements

Statements are represented by instances of `Assignment`, `IfStatement` and `WhileStatement`. Groups of statements are represented by instances of `InstructionList`.

The evaluation of an `InstructionList` is straightforward: each individual statement is evaluated within the same global and local environment passed to the `InstructionList`.

Assignments are represented by `Assignment` objects which contain two in-

stance variables. One holds the string representation of the variable involved in the assignment; the other holds the expression to be evaluated.

The control statements are represented by instances of `IfStatement` and `WhileStatement`. `IfStatement` objects contain a boolean expression and two instances of `InstructionList`: one holding the statements for the true branch, the other the statements for the false branch.

Similarly, `WhileStatement` objects hold a boolean expression and a single `InstructionList` representing the statements within their loop bodies.

5.6 A complete example

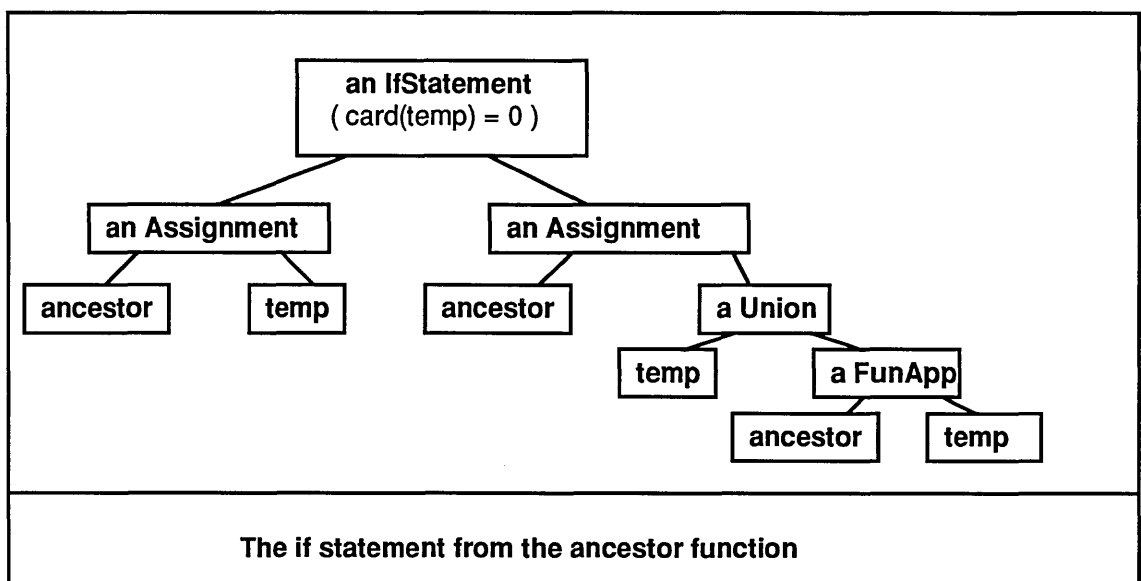
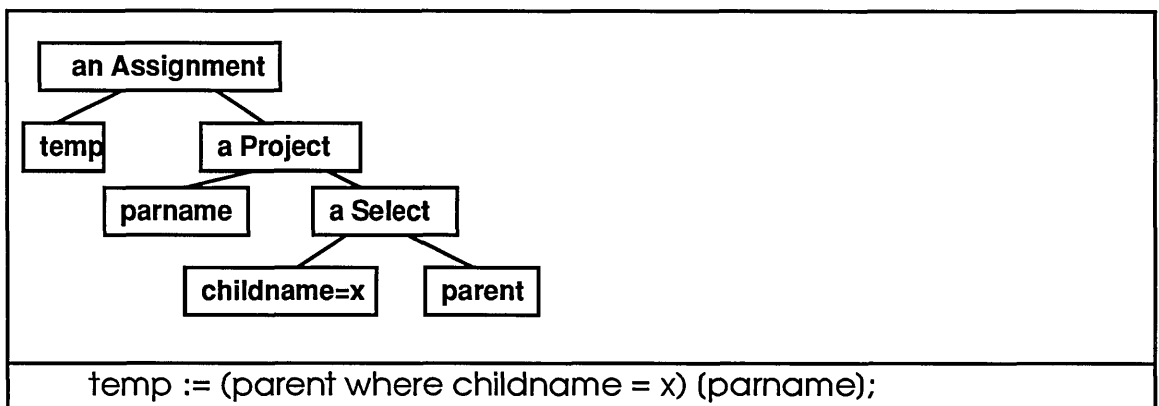
As a complete example, consider the function `ancestor` (and its application) described in the previous chapter:

```
func ancestor( x : char )
{
  temp := (parent where childname = x) [parname];
  if (card( temp ) = 0)
    ancestor := temp
  else
    ancestor := temp ++ ancestor(temp);
};

ancestor("Rachel_Natanson");
```

The above 'session' with the DEAL interpreter consists of two parts: the first a function definition, the second the application of the function.

The interpreter will synthesise a data structure for the function definition consisting of two statement objects as depicted in the following diagrams:



This data structure (representing the statement body of the function) is stored in the global environment dictionary bound to the function name, ancestor.

In addition, tables within the interpreter will be updated to record the formal parameter information (names and types).

The application of the ancestor function proceeds as follows:

- The interpreter's tables are used to retrieve the parameter information for the function ancestor.
- each argument is evaluated. In this case the only argument is the string literal "Rachel_Natanson".
- The type of each argument is checked against the declared type of the functions formal parameters. In this case, the argument is of type char a was the formal parameter so execution proceeds unimpaired.
- A new local environment is created. This is a dictionary object containing associations of each formal parameter name and the value of the corresponding argument. In addition, there is a binding under the name of the function, ancestor in this case, which is used for temporary storage of the function result.

- the statement body is retrieved (from the global environment) and each statement is passed the global and local environments and executed in turn. In general this may involve recursion.
- When the above execution terminates, the value contained in the local environment under the binding of the function name, ancestor, is returned as the value of the expression `ancestor("Rachel_Natanson")`.

The recursive step above proceeds similarly except that the check of the argument type reveals a clash since the argument evaluates to a value of type "rel" instead of the expected "char". In such a case, the argument is treated as a list of values (of the expected type). The function is then applied to each value in the list and all these individual results are unioned to form the single result for the original function application.

5.7 Summary

This chapter has detailed the operation of the DEAL interpreter. Underlying the execution phase is a virtual machine consisting of objects which effect the computation.

The next chapter covers the development of the relational operators that underpin the objects within the virtual machine.

Chapter 6

Specification of the Relational Algebra

6.1 Introduction

Specification is the cornerstone of the process of software construction – without a specification phase, the *acceptability* of a software product can only be based on consumers' reaction to the software's operation. It is hard to conceive how an *anticipation* of this reaction can usefully inform the construction process. Even though specification is primarily concerned with communication between clients and developers, the *form* that a specification takes can significantly affect the software development process.

This chapter presents a justification for the use of *formal* specification techniques and gives an overview of the two major classes of techniques: model-based and algebraic. Approaches to the specification of database systems are examined and the appropriateness of algebraic specification to the particular work being reported here is demonstrated. Conclusions are drawn as to the efficacy of the methodology for the development of certain kinds of software for the REKURSIV/Lingo system.

6.2 Formal techniques

The inadequacy of natural language to express precisely the intended behaviour of computer systems has been cited throughout the half-century history of digital computation. On the other hand, it appears unrealistic to base software construction on a theory so mathematical that the majority of programmers would not be able to avail themselves of the problem solving leverage which the theory enables.

Schach ([94]) finds an interesting case study in informal specifications whose history spans some sixteen years. A demonstration of a technique for constructing and proving a product correct (an ALGOL procedure for a text processing problem specified in English) was given by Naur in 1969 ([83]).

CHAPTER 6. SPECIFICATION OF THE RELATIONAL ALGEBRA103

Four faults were found in the 26 line procedure, 1 by Leavenworth ([71]) and 3 by London ([73]) who corrected these and gave a formal correctness proof. In 1975 Goodenough and Gerhart ([41]) found three further faults in London's work and produced a new set of specifications (two of the seven discovered faults were considered to be specification faults.).

Meyer in 1985 ([75]), writing to promote the use of formal specification techniques (to ease the detection of contradictions, ambiguities and omissions contained in English specifications), detected 12 faults in the work of Goodenough and Gerhart. He presented mathematical specifications to correct all the faults and then produced English specifications by paraphrasing the mathematical specifications. Interestingly, an ambiguity in Goodenough and Gerhart's work which is pointed out by Meyer is again present in Meyer's own English paraphrases, according to Schach ([94]).

A major argument against the use of formal techniques, which has an intuitive appeal is that the software production process lengthens in time and cost since correctness proofs and the necessary mathematical skills are not within the usual armoury of system development teams and the software still has to be written !

It is extremely difficult to gather evidence to test the hypothesis that formal specification techniques shortens overall product development time,

since clients are unlikely to be able to afford product development under two different regimes in order to provide the control sample for a statistical significance test. Some non-quantitative data does however appear. Correctness proving is not necessary for all aspects of software and is not even the main fruit of formal techniques. If programs can be *derived* from a formal specification through a systematic method, the likelihood of introducing errors is diminished. It has also been found that inspecting formal specifications easily reveals faults ([84], [48]) and that the writing of formal specifications can be taught to software professionals (with only school mathematics) in a relatively short time ([48]). The use of formal specification may not adversely affect overall software development costs: Hall and Pfleeger ([49]) report on the application of formal methods in a large industrial project (about 50 person years effort). They conclude that the use of formal methods appeared to yield high quality software at no greater cost than conventional methods.

Given the above, it is clear that natural language is far from ideal for program specification. Semi-formal techniques such as those (from systems analysis) advocated by DeMarco ([27]), Yourdon ([115]) and Gane and Sarsen ([34]) have been used in a wide range of application areas. They (and their hybridisations) help clarify the medium-scale structure of large systems by allowing their description in terms of annotated diagrams. Each technique

has at its core a syntax for these diagrams and practitioners have developed aesthetics and rules of thumb with which to inspect diagrams for signs of ambiguity, contradiction and omission. Computer Aided Software Engineering (CASE) tools are now available to assist system analysis according to these regimes.

More formal still are techniques such as Finite State Machines (FSM) and Petri nets. Again, these techniques have associated diagrammatic representations which allow the development of an aesthetic that detects likely problem areas in specifications under development. Unlike the techniques of the previous paragraph, FSMs and Petri nets have a mathematical basis which allows properties of systems to be deduced without recourse to the diagrammatic representations.

FSMs are ideally suited to handle the complexity of event driven systems but give no insight into the management of data flow. Specifying large systems by using FSMs is cumbersome because of the proliferation of states since there is no concept of modularisation and encapsulation and so FSMs are not useful for clarifying the complexity of large systems.

Petri nets bear some similarity to FSMs but also go some way towards expressing data flow (or at least the inherent synchronisation requirements). Their main strength has been the ability to cope with (and express) tim-

ing and synchronisation requirements. For this reason their use has been strongest in real time systems development.

The mainstream fully formal techniques can be broadly classified as either **model-theoretic** or **algebraic**. In the model-theoretic camp, the specification language Z ([101]) is arguably the most widely used (the other contender being the Vienna Development Method, (VDM [66])). Z specifications consist of *schemata* interspersed with explanatory English text. Each schema consists of two sections – a *declarations* section that contains variable declarations (typing information) and a *predicates* section which constrains the values the variables can take. Schemata can be combined under the *schema calculus*.

Essentially, Z allows the expression (using set theory and first order logic) of the invariant aspects of the global state space of a system and then the consequent changes to that state when operated on by procedures and functions.

Algebraic techniques, by contrast, define objects by the relationship of the operators on the object through equational rules. This approach has its roots in abstract data type methods. The essence of the methodology is to give an abstract denotation of the values that variables of a type can take and to relate the operations on the type through equations.

CHAPTER 6. SPECIFICATION OF THE RELATIONAL ALGEBRA107

As an example, we can specify a type **Natural** with an operation **add** as follows

the type **Natural** has denotations *One* and *Succ(x)* (where *x* denotes a **Natural**). This means that the following are legitimate **Natural** denotations

One

Succ(One)

Succ(Succ(One))

The operator *add* can be defined by a set of equations

$$add(One, y) = Succ(y)$$

$$add(Succ(x), y) = Succ(add(x), y)$$

This form of specification has come to be called a *constructor system of equations*, after ([106]), which is a restricted form of definition common to algebraists (who use equations to define algebraic structures such as groups, rings, vector spaces and categories themselves).

In such a system, a distinction is made between passive operators (functions) which are used to construct or denote data values of a type (the *Succ* function of the above) and active functions whose definition is the purpose

of the equations. The passive functions are normally termed **constructor functions** and these may be constant (i.e. they have no domain) as *One* in the above.

The form of the equations in such a system is restricted in that

- the left hand side always has the form $f(e_1, e_2, \dots, e_n)$ where f is an active function and the e_i are *patterns* involving variables and constructor functions (perhaps constant).
- variables on the right of an equation are always introduced on the left

These restrictions are exactly those enforced for pattern-matching in functional programming languages such as Standard ML ([78],[54]), which allows the possibility of executing specifications of this kind. In addition, the act of compilation (especially the type checking phase) gives the specifier some confidence that the notation has at least been used sensibly and correctly.

It is important to realise that the resulting computation (of an executable specification) is purely symbolic and could be represented as a succession of substitutions justified by the equations in the specification.

Another view on the above process is that a program has been developed in a declarative style by ‘programming through types’ and that the meanings

CHAPTER 6. SPECIFICATION OF THE RELATIONAL ALGEBRA 109

of types have been specified purely symbolically.

For clarity, the terms *type*, *class* and *abstract data type* will here be used with the following meanings –

- *types* – these essentially provide a partition of the value space of a programming language. Compilers may also use type information for representation purposes. Most compilers also use ‘type-checking’ (to a lesser or greater extent) to assist programmers avoid logical errors. The arguments to operators must obey certain type rules which may be slightly relaxed for built-in operators of a language but are strictly enforced for arguments to procedures and functions.
- *classes* – these derive from simula ([21, 7]). They provide a *behavioural* partition of the value space of a programming language. The permissible operations on a data item which has a class are defined within the class along with the data objects necessary to perform these operations. The data item can only be manipulated via the operations defined within its class and access to its internal data is denied (instance data).

In general, languages allow a class to be defined as a subclass of some other class, in which case the permissible operations (and instance

data) are *inherited*. In addition, some class-based languages, such as Smalltalk and Lingo, treat classes themselves as elements in the value space, thus allowing computation to be performed on them. This is often paraphrased as ‘classes are first class citizens of the language’.

- *abstract data types* – these package the operators on the defined data (including data construction operators) and hide the implementation details (such as internal supporting data structures). Abstract data types and classes can be seen as *operationally* equivalent although, in general, languages that support abstract data types do not support subclassing as above and restrict the level of computation that can be performed on abstract data types themselves (it is for example, unusual to be able to check the type of an object at run time). An exception to this is the functional programming language ‘Pebble’ proposed by Burstall and Lampson ([13]) which includes types themselves as first class citizens on which computation can proceed.

In the database field, there is also the notion of *domain* ([24]), which intuitively appears like a type above since domains are used to delineate the set of values that an attribute may have. In reality though, the situation is more complex when one considers the need to check the validity of set operations such as union – where type compatibility (rather than type checking)

is required on the domains of the operands. A lack of coherent approach to this aspect of specification is reported by Samson ([93]).

Given the above, the use of algebraic specification for the work being reported here, is a considered choice. A summary of the reasons is

- There is a coherence in the use of an algebraic technique to specify the Relational *algebra*.
- A methodology exists for deriving implementations from algebraic specifications. Indeed (as reported below) this methodology can be significantly streamlined where the target language for implementation is class-based.
- The notion of abstract data types (which form the backbone abstraction for algebraic specification) corresponds in a natural way to the concept of classes.
- For database work specifically, problems, such as the specification of the domain concept, can be resolved in a clear and regular manner by the use of abstract data types (rather than the limited typing of model-theoretic specification strategies).

6.3 Deriving programs from algebraic specifications

This section describes in general terms the derivation process. Succeeding sections describe the actual application of the process within the work of the project and its subsequent development into more efficient implementations.

We start with an approach using the language SML which is applicable to any imperative implementation language, and not especially Lingo. The approach is based on the following stages —

- Consider all the data types that the program will encounter and specify all these as abstract data types using constructor functions.
- define each operation as an operator on these abstract data types, using pattern matching in the usual way to provide case analysis
- define **destructor** functions — these are used to extract the arguments of any nonconstant constructor functions
- eliminate pattern matching in all function definitions (except destructors)
- choose an implementation language representation for each abstract data type

CHAPTER 6. SPECIFICATION OF THE RELATIONAL ALGEBRA113

- code the constructors and destructors as result returning functions in the implementation language
- the definition of all the operators is now expressed entirely in terms of function application. Coding these is simply a matter of transliteration.

An example — sequences

As an example, consider the representation of sequences of integers and the implementation of methods to reverse the sequence and to append an integer to the right hand end of a sequence.

- an abstract datatype —

```
abstype seq = empty | cons of int * seq
```

This says that a `seq` (a sequence) is either the constant sequence, `empty`, or can be constructed from an application of the function `cons` to an `(integer,sequence)` pair.

- the operators —

– right append —

```
fun rap(empty,i) = cons(i,empty)
  | rap(cons(h,t),i) = cons(h,rap(t,i))
```

Notice the use of pattern matching here. The first line states that right appending an integer to the empty sequence results in a sequence containing just that integer, which is obtained by applying `cons` to the integer and `empty` (which is a valid sequence).

The second line matches the case where an integer is being right appended to a nonempty sequence; this sequence, being nonempty, must be constructible by an application of `cons` to some integer, `h` say, and some sequence, `t` say. Clearly this recursion will come to an end.

– the reverse function, similarly —

```
fun rev(empty) = empty
  | rev(cons(h,t)) = rap(rev(t),h)
```

- Define destructor functions. In both the function definitions above pattern matching has been used to break a constructed item into its component parts. Here we would define destructor functions —

```
– fun head(empty) = raise seqFault
  | head(cons(h,t)) = h
– fun tail(empty) = raise seqFault
  | tail(cons(h,t)) = t
```

The `raise` expressions here are part of SML's exception mechanism. They represent the (abnormal) termination of a computation.

CHAPTER 6. SPECIFICATION OF THE RELATIONAL ALGEBRA115

- Eliminate pattern matching by using destructor functions —

```
fun rap(s,i) = if s = empty then
  cons(i,empty)
else
  let val h = head(s) in
    let val t = tail(s) in
      cons(h,rap(t,i));
    end
  end;
end;
```

and similarly for rev.

- Choose a representation in the implementation language. In general, we declare a class of objects, each with a single instance variable as in —

```
Seq is Object
  [ sequence ]
```

and we code instance methods to access this data —

```
sequence [] sequence.
sequence:s [] { sequence := s ; }.
```

- Code the constructor and destructor functions. The constructors, which create new items of the class, are naturally coded as class methods —

```
empty [] { ^ ( super new ) ; }.
cons: anInteger with: aSequence []
  { ^ (super new sequence: (Pair of:anInteger and:aSequence));}.
```

CHAPTER 6. SPECIFICATION OF THE RELATIONAL ALGEBRA116

Notice the use of a class `Pair`. This is appropriate, pairs (indeed n-tuples) are a built-in type in SML which has no direct correspondent in Lingo so we code it. `Pair` has two instance methods: **first** and **second**; these are destructor functions for extracting the components of a pair. With these, the destructor functions for `Seq` are coded as instance methods —

```
- head [] { if (sequence = nil ) then
            raise seqException
          else
            ^ (sequence first);
        }.

- tail [] { if (sequence = nil ) then
            raise seqException
          else
            ^ (sequence second);
        }.
```

- Code the operators. Again, we use instance methods since one of their arguments is always an instance of this class.

```
self rap: anInteger [ h t ]
{
  if sequence = nil then
    ^ Seq consOf: anInteger with: self
  else
  {
    h := self head; t := self tail ;
    ^ Seq consOf:h with:( t rap: anInteger);
  }
}.
```

Which is clearly in one to one correspondence with the original SML.

A refinement of the technique

One source of inefficiency in an implementation derived as above comes from pattern matching. When translating into a language such as Pascal or C, pattern matching would make use of tag fields within a variant record or a union type. In Lingo, we can do better by using inheritance. Consider again the SML for the seq abstype —

```
abstype seq = empty | cons of int * seq
```

We can represent sequences by a class Seq, as before, and have two specialisations Empty and Cons. The constructor functions **empty** and **cons** become instance creating class methods of Seq's subclasses Empty and Cons respectively —

```
Seq is Object  
[ ]
```

Which needs no class methods since only its subclasses should ever be instantiated.

```
Empty is Seq  
[ ]
```

CHAPTER 6. SPECIFICATION OF THE RELATIONAL ALGEBRA118

Which needs no class methods — simply creating an object of this type with it's inherited new method is enough.

Cons, whose instances are nonempty is coded along with destructors head and tail as instance methods —

```
Cons is Seq
[ sequence ]
{
  of:anInteger and:aSeq []
  {
    ^ ((super new) sequence: (Pair of:anInteger with:aSeq));
  }.
  head [] { ^ sequence first; }.
  tail [] { ^ sequence second; }.
  sequence [] { ^ sequence; }.
  sequence:s [] { sequence := s; }.
```

Now if again we consider the SML function rev

```
fun rev(empty) = empty
  | rev(cons(h,t)) = rap(rev(t),h)
```

we can view rev as having two specialisations, one a function which has as domain only those instances of the abstype that are empty, the other has the complementary domain of all nonempty seqs. We add an instance method within the Empty class as

```
rev [] { ^ Empty new }.
```

and an instance method to the Cons class as


```
self rev [] { ^ ((( self tail) rev) rap:(self head)); }.
```

Which is concise. Notice this facility cannot be mirrored in SML since SML has no facilities to specialise a type.

The refinement outlined above has a stronger result than merely providing a succinct implementation. The following observations can be made on the effect of removing (from the technique) the need to remove pattern matching

–

- The implementation no longer makes use of an underlying *data structure* and is thus more abstract and closer to the purely denotational specification.
- Selection (changes in control flow in implementations derived by the conventional technique) has been removed and replaced by use of the classing (or typing) mechanism of the implementation language. As processor architectures move to support object-orientation (and thus remove the problems associated with changes in control flow) this approach is favoured. In particular, advantage can be taken of this when implementing in the language Lingo on the REKURSIV which has hardware support for typing.

6.4 Specifying the Relational Algebra

The previous section described the general approach to program derivation. Within this section, the fundamental abtypes used in the actual application of the approach to the implementation of the relational algebra are described. The next section will detail the derivation of a particular operator (a relational join). The complete specification does however appear in appendix B.

In order to model the relational algebra, 7 abstract data types were defined in Standard ML. The declarations of these, describing also the constructor functions for the abstract data types are:

- Relations are declared as

```
relation = rel of (scheme * tupset);
```

This expresses the intention that a relation is a pair drawn from the product of the set of *schemes* and the set of *tupsets* (these types are defined below).

- Schemes are defined to contain information about the attributes involved in a relation's tuples. Each attribute has a name as well as a type, which are modelled by the abtypes *name* and *domain* respec-

CHAPTER 6. SPECIFICATION OF THE RELATIONAL ALGEBRA 121

tively. Schemes also hold a list of *names* which are used to denote the keys of the relation:

```
abstype scheme = sch of(( name list)*((domain * name) list))
abstype domain = dom of string
abstype name    = nam of string
```

- The data within a relation is modelled as a set of tuples by

```
abstype tupset = set of (tuple list);
```

- Tuples contain data values which are abstractly represented by the abstract data type *attribute* which has a constructor for each actual type of data that the relations can hold (in this case just strings and integers).

```
abstype tuple = tup of (attribute list)
abstype attribute = ival of int | cval of string
```

The above are merely the first parts of complete abstype declarations. Each introduces the definition of operations on the abstract type. For the type *Relation*, the key functions (together with their signatures are):

- `select : relation × (tuple → bool) → relation`
- `project : relation × (name list) → relation`

CHAPTER 6. SPECIFICATION OF THE RELATIONAL ALGEBRA122

- `cartprod` : `relation × relation → relation`
- `union` : `relation × relation → relation`
- `difference` : `relation × relation → relation`
- `intersection` : `relation × relation → relation`
- `equijoin` : `relation × relation × name → relation`

As a specimen, the detailed specification of the `equijoin` operator is elaborated in the next section.

A detail to notice within the above is the signature of `select`: this allows the application of any function whose type (or signature) is `tuple → bool` to the tuples within a relation, giving a great degree of freedom in selection predicates.

In addition to the functions listed above there are numerous operators whose use is to check the validity of an operation (union compatibility, for example) as well as service functions to create elements of the abstract types (since the constructor functions of an abstype can only be used within the abstype).

The treatment of domains is not restrictive. Any other candidate type for attributes within relations (`relation`, `even`) could simply be modelled by

adding another constructor to the *abstype*. The function *validdom* ensures the type safety of the set operations.

6.5 Specifying a relational operator

The previous section only sketched the broad terms of the formal specification of the relational algebra. This section gives the detail of the specification of a particular operator, an equijoin. This operator has been chosen as the ‘specimen’ since its efficiency within this *abstract* specification is poor. The following sections within this chapter will then demonstrate the process of replacing the *abstract* specification with a more *concrete* specification within which an equijoin operation can be specified with a better efficiency.

The signature of the operator is given by

$$\text{equijoin} : \text{relation} \times \text{relation} \times \text{name} \rightarrow \text{relation}$$

This indicates that the function takes three arguments: the two operand relations for the join and the *name* of an attribute from the scheme of the first operand. It is assumed, for clarity, that the second operand relation’s scheme has a single *key* attribute and that the join is to be performed on the named attribute from the first operand and the *key* attribute from the second.

At the level of the abstype relation, this operator is coded in SML as:

```
fun equijoin(rel(s1,ts1),rel(s2,ts2),n) =  
    rel(schappend(s1,s2),tjoin(ts1,s1,ts2,s2,n))
```

The purpose served by this function is twofold. Firstly, pattern matching is being used to extract the scheme information from the two operand relations and pass them along with the bodies (tupsets) of the relations to the function `tjoin` which actually performs the join on the bodies. (The scheme information is needed to match attributes). Secondly, the resulting relation has a scheme derived by appending the schemes of the two operand relations.

The operator `tjoin` (at the level of the abstype tupset) contains the join algorithm. The bodies of relations (modelled by the abstype tupset) are essentially lists of tuples (which are themselves essentially lists of attributes). Within the operator `tjoin`, it is necessary to traverse the first operand (a list) and, for each element, extract the join attribute and search the second tupset for a match (this will be require a linear search). The success of this search determines whether or not to append tuples from the two tupsets and insert the new tuple into the resultant relation.

In more detail, the definition of `tjoin` requires the consideration of two cases: when the first operand is empty and when it is not.

CHAPTER 6. SPECIFICATION OF THE RELATIONAL ALGEBRA 125

- the operand is empty. Clearly the join result is also empty:

```
tjoin(set(nil),s1,ts2,s2,n) = set(nil)
```

- the operand is non-empty. In this case, pattern matching is used to extract the 'head' tuple from the first operand

```
tjoin(set(h::t),s1,ts2,s2,n) =
```

To extract the join attribute from h the operator `tupleproj` is used. This takes three arguments: a tuple, a scheme and a list of names. The operator 'projects' the attributes from the tuple whose names appear in the list of names. The SML expression:

```
let val firstPart = tupleproj(h,s1,[n]) in
```

extracts the join attribute (named n) and stores the result in a local SML value named `firstPart` (actually a list containing only one element).

To search for a matching tuple in the second tupset (named `ts2`) the function `atKey` is used which given a key and a set of tuples returns the matching tuple from the set:

```
let val partner = atKey( firstPart,ts2) in
```

Since sets of tuples are fundamentally stored as lists, the efficiency of the function `atKey` is $O(n)$.

If the search is successful the two tuples can be joined with the operator `tupappend` and inserted into the overall result with `fastinsert` since the join cannot introduce duplicate values if the two original relationals were duplicate free.

Put together, the SML for the complete operator `tjoin` is:

```
fun tjoin(set(nil),s1,ts2,s2,n) = set(nil)
|   tjoin(set(h::t),s1,ts2,s2,n) =
    let val firstPart = tupleproj(h,s1,[n]) in
      let val partner = atKey( firstPart,ts2) in
        if tupnull(partner) then
          tjoin(set(t),s1,ts2,s2,n)
        else
          fastinsert(tupappend(h,partner),tjoin(set(t),s1,ts2,s2,n))
        end
      end
    end
```

As can be seen the efficiency of this algorithm (and hence its Lingo counterpart) is determined by the required number of calls to the function `tjoin` and the efficiency of the searching function `atKey` which, as indicated previously is $O(n)$. This gives an overall efficiency of $O(n^2)$.

Clearly, the component to work on in order to improve the efficiency is the searching function, since the number of calls to the `tjoin` function

(which will always be equal to the cardinality of the first operand) cannot be reduced.

The next section describes the development of the specification to incorporate more efficient algorithms.

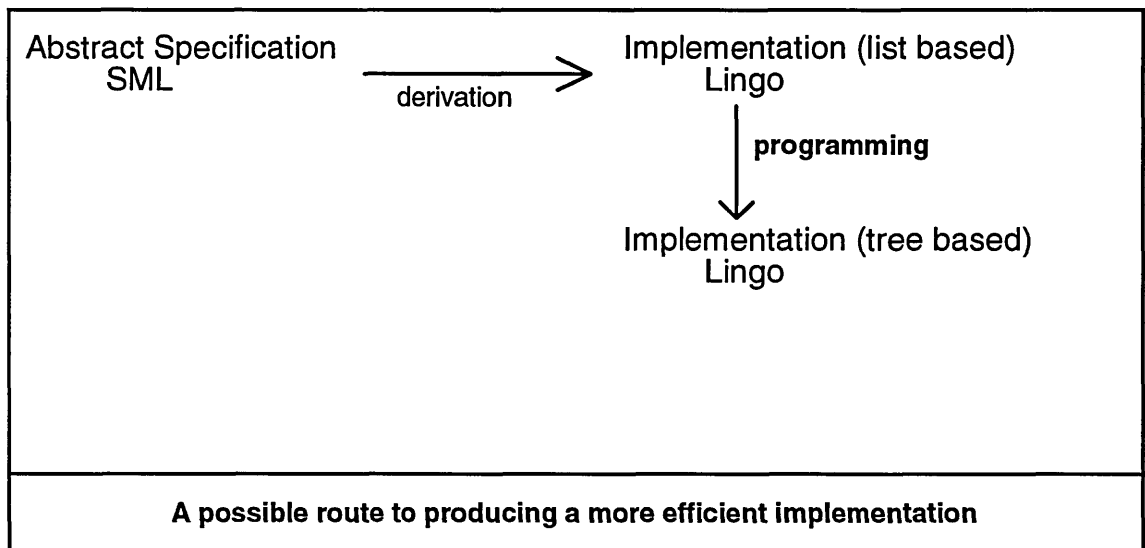
6.6 Efficiency and refinement

A difficulty associated with the program derivation outlined in the last sections is that the efficiency of the algorithms is low. Searching on a key, since it is linear, has $O(n)$. Storage techniques such as tree structures and hash tables [24] offer advantages for such searching ($O(\log(n))$ for trees, and constant order for hashing). The inefficiency of the join operator specified in the previous section (within a list based specification) is particularly marked.

It is important to notice that this inefficiency emanates from the basic *data structure* (in this case lists) embodied within the specification rather than the fact that the implementation has been *derived*.

A possible way to proceed to a more efficient implementation is to work on the derived Lingo program and incorporate a data structure which allows more efficient searching, since this is the fundamental issue affecting the efficiency of the relational operators. Candidate structures are hash tables

and trees. The following diagram illustrates the possible route:



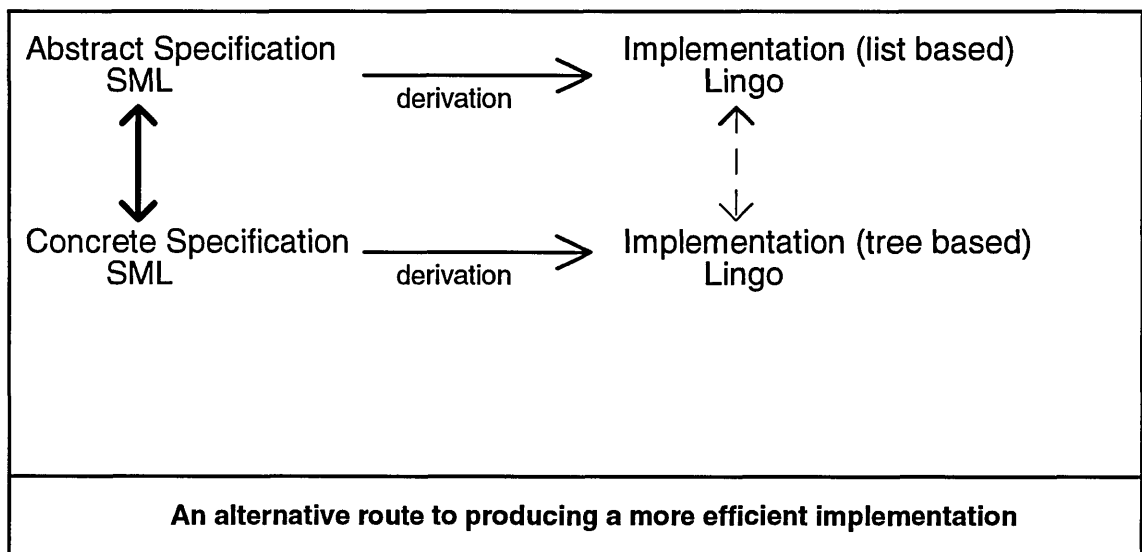
This is not wholly satisfactory since the relationship between the list based implementation and the tree based implementation (in terms of program properties) is intractable from the formal point of view depending as it does on referentially opaque programming. Some of the benefits of an initial formal specification are lost by the introduction of this programming step.

An alternative approach is to formally specify the data structure (trees, say) that is intended to be used and use the derivation technique to derive a more efficient Lingo implementation. The correspondence of the two formal specifications can be established through reasoning since they are equational and the correspondence between the two implementations can be reasonably inferred since they have been produced by a derivation method that preserves

program properties.

Both formal specifications specify the same thing, but the list based specification is in a sense more abstract since it deals with the relationship of operators whereas the tree based specification is more concrete since it deals with aspects of an underlying data structure.

This approach, which is what was chosen for this work (but not followed in its entirety), is depicted in the following diagram:



The correspondence between the abstract and concrete specifications is established in the following way. An *abstraction* function is defined relating objects within the concrete specification and objects within the abstract specification (in this case the objects will be the representations of relations). For example, if the abstype crelation models relations in a more

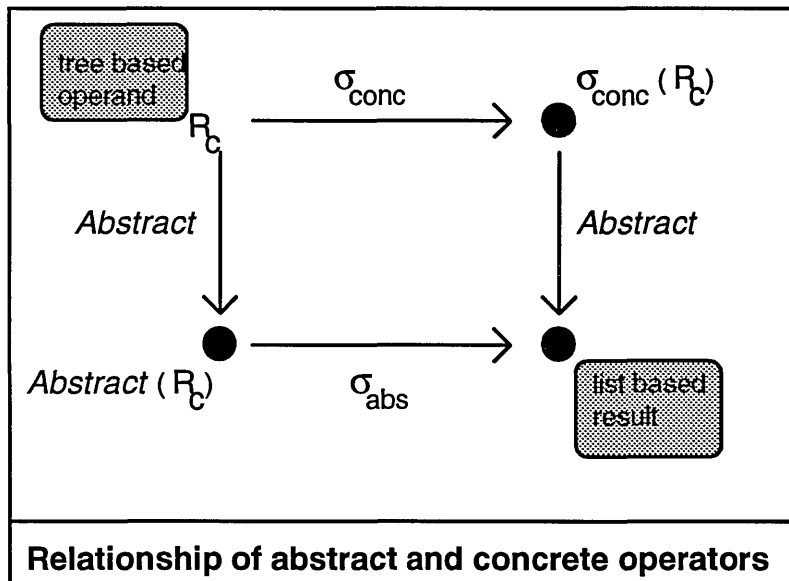
'concrete' specification and the abstype relation models relations in the abstract specification, an abstraction function (called *Abstract*, say) would be defined with signature

`Abstract : crelation → relation`

In principle, there could be many concrete specifications, each having an associated abstraction function mapping to the abstract specification. To show that the concrete specification of an operator (select, for example) has at least the properties of its counterpart in the abstract specification, it is required to prove that the result of applying the abstraction function to the result of the concrete operator on the concrete relation is equivalent to the result of applying the abstract operator to the result of applying the abstraction function to the concrete relation. For example, if *Abstract* denotes the abstraction function, σ_{conc} and σ_{abs} the concrete and abstract select operators (respectively) and R_c the concrete relation, this can be expressed as

$$Abstract(\sigma_{conc}(R_c)) = \sigma_{abs}(Abstract(R_c))$$

The above equality can be represented by the statement that the following diagram 'commutes' (a notion from category theory on which the formal foundation of this approach is based) in that the resultant (abstract) relations obtained via either computational route are equivalent:



In the case that such a proof is intractable, the process still has the advantage that with the equational nature of the specification and the use of pattern matching, the differentiation and characterisation of test cases is automatic, allowing conventional testing to be carried out in a very disciplined way.

6.7 2–3 trees

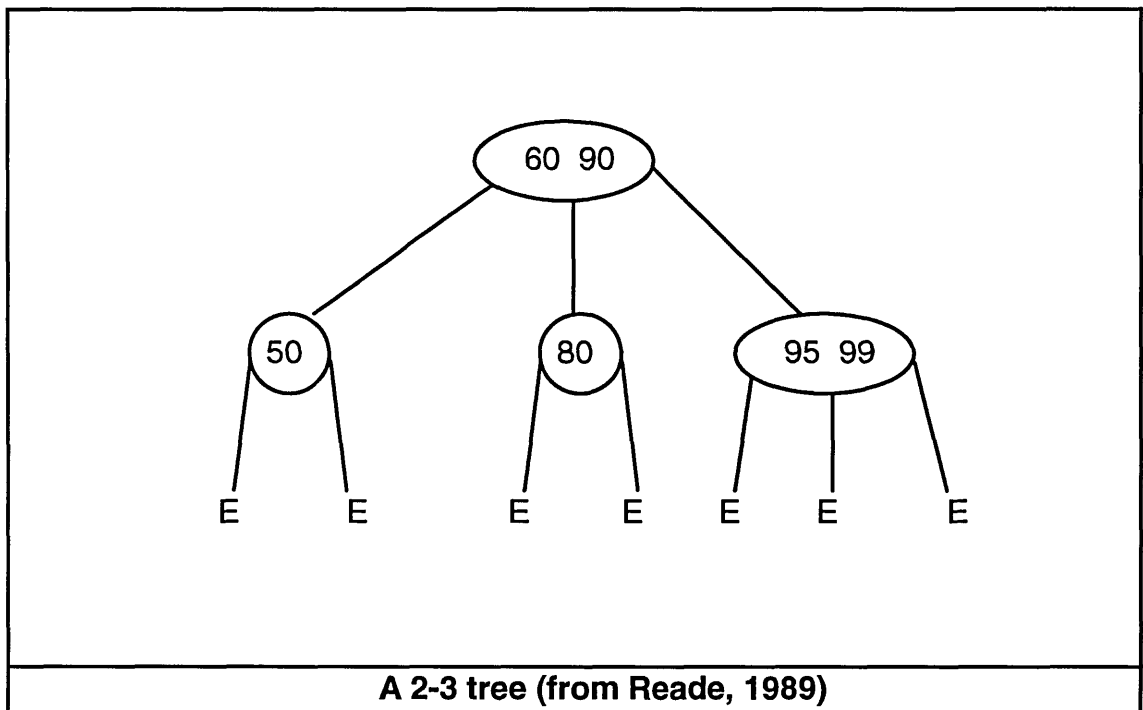
Moving now to the specific case of the relational algebra, a candidate data structure that would improve the efficiency of operators is the balanced tree

CHAPTER 6. SPECIFICATION OF THE RELATIONAL ALGEBRA132

structure known as a 2-3 tree.

Balanced trees are often used for storing large sets of indexed data items and algorithms of $O(\log(n))$ exist for storage and retrieval. 2-3 trees are a special case where each subtree is either empty, a node containing a value, a left subtree and a right subtree (a 2-node) or a node containing two values and a left, middle and right subtrees (a 3-node).

The diagram gives an example 2-3 tree, where E denotes an empty subtree:



In addition, 2-3 trees are:

- **ordered** – the value contained in a 2-node is greater than any value found in the left subtree and less than any value to be found in the right subtree. The 'left' value in a 3-node is greater than any to be found in the left subtree and less than any value in the middle subtree and the 'right' value is greater than any in the middle subtree and less

than any in the right subtree.

- **balanced** – all subtrees of any node have the same depth.

Reade ([89]) presents an equational program in SML for insertion into 2–3 trees which is proved to maintain the ordering and balance properties.

The program makes use of the following datatype declaration

```
datatype 'a tree23
= E
| Tr2 of 'a tree23 * 'a * 'a tree23
| Tr3 of 'a tree23 * 'a * 'a tree23 * 'a * 'a tree23
| Put of 'a tree23 * 'a * 'a tree23
```

The constructor functions E, Tr2 and Tr3 are used to model empty subtrees, 2–nodes and 3–nodes respectively. The constructor Put is used for nodes that are created during insertion and then removed during rebalancing. The details are in Reade ([89]) along with the proofs and algorithms for removal.

The 2–3 tree given in the last diagram would be represented in SML as $\text{Tr3}(\text{Tr2}(\text{E}, 50, \text{E}), 60, \text{Tr2}(\text{E}, 80, \text{E}), 90, \text{Tr3}(\text{E}, 95, \text{E}, 99))$

For the purposes of the relational algebra, 2–3 trees were incorporated into the formal specification described in section 6.4 to provide a more concrete specification. The counterpart (within the concrete specification) of the

abstract specification's abstype tupset (which is used to hold the data set 'body' for relations) is the following:

```
abstype tree23
  = E
  | Tr2 of tree23 * tag * tuple * tree23
  | Tr3 of tree23 * tag * tuple * tree23 * tag * tuple * tree23
  | Put of tree23 * tag * tuple * tree23
```

Reade's values are replace by pairs of tags and tuples. The tag component is some unique key value used to order and identify information for retrieval. Since an ordering relationship is defined for the abstype tuple, the type tag was implemented as a type synonym for tuple. The tag component will contain attributes drawn from the tuple component according to the scheme defined for the relation in which the tuple resides.

As an example, (taken from Date [24]), consider the body of the relation 'supplier', with fields 'snum' (supplier number), 'sname' (supplier name), 'status' (status value) and 'city' (location):

snum	sname	status	city
s1	smith	20	london
s2	jones	10	paris
s3	blake	30	paris
s4	clark	20	london
s5	adams	30	athens

The field 'snum' can be used for the index tag since it uniquely identifies tuples.

The concrete counterpart of the abstype relation is:

```
abstype crelation = crel of scheme * tree23
```

In order to establish the correspondence between the concrete and abstract specifications, abstraction functions were defined on both the abstype tree23 (mapping to the abstype tupset) and the abstype crelation (mapping to the abstype relation). Proofs, utilising these abstraction functions, of the correspondence of the results of concrete and abstract operations are not given since they are laborious and not practically feasible without an automated 'proof assistant'. No proof assistant is generally available for SML at present.

Counterpart operators (of a sample of those in the abstract specification) were defined on these abstypes. The detailed SML code is to be found in appendix D. For the purposes of this section, an equijoin operator is detailed. The operator is named cjoin (for concrete join). It takes three arguments: the two operand relations of the join and the name of an attribute field from the scheme of the first operand. It is assumed (for clarity) that the sec-

ond operand uses a single attribute for its index tags, and that the join is performed on the named attribute from the first operand and the tagging attribute from the second.

At the level of the abstype `crelation` this operator is coded in SML as:

```
fun cjoin(crel(s1,ts1),crel(s2,ts2),n) =  
    crel(schappend(s1,s2),ctreejoin(ts1,s1,ts2,s2,n))
```

This function is fairly cosmetic: pattern matching is being used to extract the schemes of the two operand relations and pass them along with the tree bodies of the relations to the function (at the tree level) which actually performs on the join on the bodies. The scheme information is needed to match attributes from named fields. In addition, the resulting relation (whose body is computed by the function `ctreejoin`) has a scheme derived by appending the schemes of the two operands.

The operator `ctreejoin` (at the level of the abstype `tree23`) is altogether more complicated given the separate cases that must be considered. Briefly, the algorithm involves traversing the first (tree) operand, and at each node that contains values, extracting the join attributes from the values and using these as the look up tags in the second tree operand. If the look up fails, the attributes do not contribute to the join result; if the look up succeeds the retrieved value from the second operand is appended to the value at the

node under inspection and the resulting tuple is inserted into the join result. The traversal is controlled by recursion through subtrees and subtree results are recomposed by performing unions.

In more detail, there are four primary cases to consider. These are dealt with by pattern matching on the first (traversed) tree operand and are:

- the operand is an 'Empty' node – that is it is the value E. Clearly the join result consists of the empty tree E. This case terminates the recursion. In fact, in the case where the second operand is E, it is pointless to continue as well. In SML, where the underscore symbol, `_`, is used for anonymous unification:

```
ctreejoin(E,_,_,_,_) = E |
ctreejoin(_,_,E,_,_) = E
```

- the operand is a Put node. This represents an error, since Put nodes should only exist transiently as trees are rebalanced on the addition of a value. The computation is aborted by raising an exception.

```
ctreejoin(Put(____),____) = raise putException
```

- the operand is a '2-node': that is, it has been constructed by an application of `Tr2`, as in:

```
ctreejoin(Tr2(left, tag, value, right),s1,ts2,s2,n)
```

The lookup tag (for retrieval from the second operand) is computed by projecting out from value the attribute named by the argument n. The tupleproj operator computes this from three arguments: the tuple from which to project, the scheme relating the attribute field names to position, and a list of attribute field names to project out. In this case, it is most convenient to have a locally scoped value in SML, to avoid unnecessarily repeated computation:

```
let val firstPart = tupleproj(value,s1,[n]) in
```

The look up (in the second operand) can now be performed, using firstPart as the tag. As was noted in the development of the join operator based on the abstract, list-based specification, the efficiency of the search function is crucial to the efficiency of the join algorithm. The tree-based counterpart to the atKey function (from the list based specification) is the operator at, which, given a tag, returns a tuple from a tree if the tag is found and a null tuple if the tag is not present. Again, a local value is used:

```
let val partner = at(firstPart,ts2) in
```

The efficiency of the at function is clearly crucial. It returns a tuple given a tag and a relation body (a tree). The algorithm for this is to

compare the presented tag with those at the root of the tree (there may be one or two at the root depending on whether the root is a 2-node or 3-node). If the tag is present in the root, the search is successful; if not, then the search is directed to the appropriate subtree as a result of the comparison, bearing in mind that the tree is *ordered*. The number of comparisons made is bounded by the depth of the tree. Since the tree is *balanced*, the depth of a tree containing n items is proportional to $\log(n)$ and so the efficiency of this search by the function `at` is $O(\log(n))$ as opposed to $O(n)$ for the function `atKey` within the list-based implementation.

Now, if `partner` has a null value, the current node contributes nothing to the eventual result which will be the union of the join operator applied to the left and right subtrees.

On the other hand, if `partner` has a non-null value, the tuple formed by ‘joining’ `value` with `partner` should be in the result. The joined tuple is computed by using the `tupappend` operator as in:

```
tupappend(value,partner)
```

In order to insert it into the result, a tag value must be computed. Since the scheme of the resulting relation is formed by merging the

schemes of the two operands, the tag value for the new tuple is formed by concatenating the tag values of the two tuples from which the new tuple was formed:

```
tupappend(tag,firstPart)
```

This tuple is inserted, by its tag value, into the union of the results for the left and right subtrees. Put together, the SML for the 2–node case is:

```
ctreejoin(Tr2(left,tag,value,right),s1,ts2,s2,n) =
  let val firstPart = tupleproj(value,s1,[n]) in
    let val partner = at(firstPart,ts2) in
      if tupnull(partner) then (* doesn't contribute *)
        treeunion(ctreejoin(left1,s1,ts2,s2,n),
                  ctreejoin(right1,s1,ts2,s2,n))
      else
        insert23(tupappend(tag,firstPart), (* the tag *)
                 tupappend(value,partner), (* the value *)
                 treeunion(ctreejoin(left1,s1,ts2,s2,n),
                             ctreejoin(right1,s1,ts2,s2,n)))
```

- The fourth case is that of the 3–node. This is a straightforward extension of the 2–node case, although, since 3–nodes contain two values as opposed to the single value in 2–nodes, two look ups are performed. This leads to four possible cases: both look ups succeed, both fail, the ‘left’ succeeds where the ‘right’ fails and vice versa. The SML is consequently long (roughly 4 times as long as the 2–node case) but is not

detailed here since nothing new is added (the code is to be found in appendix D).

The purpose in elaborating the detail of the development above was to establish that efficiency gains can be accomplished by manipulations at the formal specification level. As can be seen, the efficiency of the tree-based join on two relations depends linearly on the cardinality of the first relation and depends on the search efficiency algorithm for the second (which is $O(\log(n))$). Hence the overall efficiency is $O(n \log(n))$. Therefore, an implementation in Lingo *derived* (by the methodology explored in the previous section) from this more concrete specification, would itself be $O(n \log(n))$.

6.8 Query optimisation

The previous section elaborated one aspect of improving the efficiency of implementations of relational operators. Another aspect is that of improving the efficiency of *compositions* of these operators where a reordering of the operations can produce an equivalent overall result but at less cost – query optimisation. This section deals with demonstrating how query optimisation can be incorporated into the algebraic approach to program derivation.

Date ([24]) identifies four broad stages in the query optimisation process:

1. Cast the query into some internal representation
2. Convert to a canonical form
3. Choose candidate low-level procedures
4. Generate query plans and choose the cheapest

Although no query optimisation process was incorporated into this work, the approach reported within this chapter can provide a basis for limited query optimisation in two ways.

Firstly, the assurance of state preservation properties of the ‘low-level procedures’ allows query modification to take place safely. The correctness of a mathematically justified reordering of operators within a relational algebra expression assures the correctness of the same reordering of operators in the physical implementation.

Secondly, since the reordering rules can be expressed as *equational* equivalences, the interrelationship of operators can be defined through an algebraic specification in SML.

In order to demonstrate the approach and scratch the surface of query optimisation, the rest of this section will concentrate on two query optimisation rules reported by Date ([24]).

The first rule is that where a projection is followed by another projection, only the second projection needs to be carried out. The second rule is that where a projection is followed by a selection, the equivalent result can be obtained by performing the selection first and then the projection.

In order to incorporate these specimen query optimisation rules, another layer is introduced to the existing specification. This layer contains an SML abstype to represent relational algebra *expressions* themselves with constructor functions defined for each relational algebra operator. Essentially the abstype provides the the internal representation alluded to by Date.

In the case of the specimen, only two operators are concerned: Select and Project. In addition, there is a constructor in the abstype to allow 'named' relations to appear in relational algebra expressions (otherwise it is impossible to express the notion of operand).

```
abstype relExpr = Project of relExpr * nameList
                | Select of relExpr * whereClause
                | Literal of relName
```

The equational equivalences can now be expressed using pattern matching within the definition of a function *optimise* which maps *relExpr* to *relExpr*. Both rules (the removal of redundant projections and the reordering of projections followed by selections) are individually straightforward but

interfere together in the sense that the reordering performed by the second rule may generate redundant projections to be removed by the first rule.

Consider the following SML:

```
fun optimise(Project(Project(anExpr,nameList1),nameList2) =  
    Project(optimise(anExpr,nameList2))
```

Although this would successfully remove redundant projections which were originally adjacent within the relational expression, it would fail to cater for adjacencies produced by reorderings produced by the second optimisation rule. In a sense, the optimisation of redundant projections is being performed ‘from left to right’ whereas the optimisation of projections followed by selections moves projections ‘from right to left’.

In order to counter this interference, recursion is used to control the order of optimisation from right to left:

```
fun optimise(Literal(x)) = Literal(x)  
| optimise (Project(x,p)) = let val y = optimise(x) in  
    if isProject(y) then  
        y  
    else  
        Project(y,p)  
    end  
| optimise (Select(x,w)) = let val y = optimise(x) in  
    if isProject(y) then  
        let val Project(z,p) = y in  
            Project(Select(z,w),p)  
        end  
    else
```

```
        Select(y,w)
    end
```

The function `isProject`, given a relational expression, returns a boolean: true if the expression's ultimate operator is a project and false otherwise.

```
fun isProject(Project(x,n)) = true
| isProject(Select(r,w))   = false
| isProject(Literal(r))   = false
```

A Lingo derivation of this optimisation layer would then form an optimisation component within the DEAL architecture (chapter 5) and come into operation to process the synthesised data structures before their 'execution'.

The precise point at which to perform the optimisation, however, would depend on the nature of the optimisation strategies employed.

6.9 Conclusion

The areas of algebraic specification, abstract data type theory, functional programming and class-based object-oriented programming languages appear naturally together as weapons in an armoury serving the conquest of different phases of the software life cycle. These approaches allow an adaptive life cycle to be adopted and are particularly well suited to the development of a

CHAPTER 6. SPECIFICATION OF THE RELATIONAL ALGEBRA 147

platform system (such as a computational engine providing support for the relational algebra).

Other aspects of a system may favour different formal techniques. For example, the specification phase of a programming language development may be better suited by BNF and denotational semantics.

In addition, the implementation technique corresponds well with the hardware support for class-based languages afforded by the REKURSIV processor.

The equational, referentially transparent nature of the technique provides a sound basis for further work. For example, some approaches to query optimisation can be formulated as equations stating the equivalence of results obtained by performing operations in different orders. These equivalences, as well as the conditions under which they hold, can be specified equationally allowing access to a query optimisation layer within the architecture of the system being reported here.

The following chapter describes experiments carried out on the REKURSIV system to investigate its performance for various 'component' activities among which are handling of data sets using tree based storage strategies and hashing techniques.

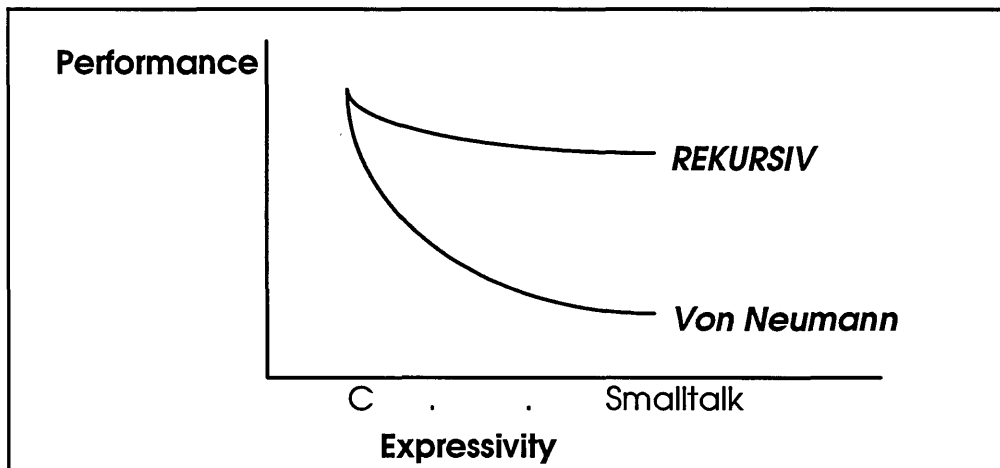
Chapter 7

Performance evaluation

To evaluate the performance of the Lingo/REKURSIV system's *architecture*, a means to factor out the effect of its implementation technology (in terms of semiconductor integration scale and its effect on processor clock rates) has to be found. The approach taken for this work is to distill a hypothesis (from the work of the system's designer and implementor) and to design experiments that test this hypothesis.

7.1 Harland's claims

The general claim by Harland for the REKURSIV system is that it narrows the semantic gap ([52]). The figure expresses this notion informally. The



horizontal axis imagines a spectrum of programming languages ranged according to their 'expressivity', where *C* is taken to be at the low end and languages such as Smalltalk are taken to be highly expressive (intermediate to these may lie languages such as Pascal, Ada, Lisp, Prolog and the functional languages.).

The vertical axis indicates performance, perhaps for a given task or perhaps for a suite.

The notion of expressivity is too diffuse to be quantified, and the curves in the graphs are merely a suggestion that performance approaches a lower bound as expressivity increases.

Harland's argument is that the REKURSIV can be implemented in any technology and so can perform as well as a C machine built on a regular von Neumann architecture (or a RISC variant). The *gradients* of the performance graph, however, are unaffected by implementation technology but are affected by the architectural arrangement of the underlying technology.

7.2 A preliminary experiment

To illustrate the approach, consider the following, preliminary, experiment.

Two benchmarks were coded in each of Smalltalk, Lingo and C. The Smalltalk programs were executed on both an IBM RS/6000 (using Gnu Smalltalk-80) and an IBM Personal Computer (using Digitalk's Smalltalk/V for Windows, an 80386SX processor running at 16MHz with 12 MBytes of memory and Microsoft Windows version 3.1). The Lingo programs were executed on the REKURSIV. The C programs were also executed on all three systems.

The first benchmark evaluates an arithmetic expression

$$(11+(10+(9+(8+(7+(6+(5+(4+(3+(2+1)))))))))))$$

represented as a tree (using instances of classes to represent nodes in the Smalltalk/Lingo code, and using union structures (variant records) in the C code). The code (in Smalltalk and C) for this test is given in Appendix A.

In the second benchmark, the contents of a 100 element (integer) array are computed by multiplying elements of two other arrays (the i th element is computed by multiplying the i th element of the first array by the $(100-i)$ th element of the second array. The first array contains the integers 1 to 100, the second has the integers 100 down to 1. Again, the code is given in Appendix A.

The figures give the number of evaluations per second.

- Benchmark 1

	Evaluations per second	
	Smalltalk/Lingo	C
RS6000	588	29400
REKURSIV	3333	22700
IBM PC	1111	3570

- Benchmark 2

	Evaluations per second	
	Smalltalk/Lingo	C
RS6000	50	12500
REKURSIV	116	10000
IBM PC	53	1640

The first benchmark can be seen as making use of expressive features of Smalltalk and Lingo which are absent in C. (The computation's control flow is explicitly programmer controlled in the C versions whereas it is embedded in the behaviour of objects in the other versions.).

The second benchmark represents a task which is naturally expressed in a similar way in both the Smalltalk/Lingo and C versions.

The effect of implementation technology can be factored out by considering the ratios of values for the first benchmark to those of the second.

In particular, the ratios for the C implementations are –

RS6000	2.35
REKURSIV	2.27
IBM PC	2.17

These figures represent the ratio of computational effort expended in executing the benchmarks. Across platforms, the second benchmark appears to

require a factor of about 2.25 as much time as the first benchmark to execute. The similarity of these figures is expected under Harland's hypothesis – the regular von Neumann features of the platforms are being exercised in both cases.

Turning to the ratios for the Smalltalk/Lingo implementations we have –

RS6000	12
REKURSIV	29
IBM PC	21

These figures indicate that making use of object-oriented features has changed the relative computational costs of the two benchmarks across all platforms. Loosely, the first benchmark has become 'easier' to perform (than the second) when use is made of inheritance and polymorphism. (The other possibility, that the second task has become 'harder' is unlikely since the computation involved in the second benchmark does not make any special use of object-orientation and the hardware's reaction to this kind of computation has already been demonstrated through the C programs). The more likely explanation is that all three systems have narrowed the semantic gap but to differing extents by being allowed to exercise their facilities for object-orientation.

7.3 Medium scale benchmarks

As implementations of Smalltalk-80 on various processors began to proliferate in the early 1980s, interest was shown in measuring their relative performance. [70] (the so-called ‘green book’) accounts the experiences of implementing teams and in particular, a chapter ([74]) reports facilities for objectively comparing the efficiency of implementations. These facilities are generally known as the ‘Smalltalk-80 benchmarks’.

Many of the micro benchmarks are only relevant to Smalltalk-80 implementations (based on the ‘blue book’ [39]) and exercise particular bytecodes and primitives of the Smalltalk-80 virtual machine. Such benchmarks are not directly applicable to measuring the efficiency of Lingo implemented on the REKURSIV. Examples in this category are –

- **testTextScanning** – this tests the speed of the (primitive) method that displays characters on the screen. Within the Lingo implementation, this speed is largely determined by the performance of the X-windows system running on the host Sun workstation).
- **testBitBlt** – this tests the block transfer of pixel values, an important feature in early Smalltalk-80 systems with their emphasis on the construction of Graphical User Interfaces without direct windowing and

event support from an underlying operating system.

- **testLoadThisContext** – this measures how quickly the current context (the execution environment containing the local variables and so on) can be pushed onto the stack. Within the Lingo implementation it is hard to see how this operation could be isolated and in any case the significance of the time for its execution is probably less for Lingo than for Smalltalk systems. In Smalltalk for example, control structures such as an if statement are not an inherent part of the language. Instead, they are synthesised by methods (within appropriate classes) which take **Blocks** (code objects) as arguments. An if-then-else statement, for example, is forged by including a method **ifTrue: ifFalse:** in the class **Boolean**, so that we may for instance write

```
x < y
  ifTrue: [ y := y -x. ]
  ifFalse: [ ^ y ].
```

(Pairs of square brackets delineate **Blocks**).

The execution of either branch will require the pushing of a context on to the stack. Lingo however, whilst still allowing the synthesis of

control structures, has the more common varieties available in the core language which are recognised by the compiler and efficiently compiled.

None of the macro benchmarks from [74] appear to be of direct relevance for measuring the Lingo system since they mainly deal with the methods involved in providing the programmer environment.

For this study, the spirit of the Smalltalk-80 micro benchmarks was used to model a suite of tests which measure slightly coarser grain activity (at roughly the statement, rather than the primitive/bytecode level). The following tables describe the tests. Each table groups tests that exercise similar implementation activities.

The first group exercise access to the execution environment – instance variables, class variables, locals and method arguments –

Environment		
	test name	description
A	testLoadInstVar	return the value of an instance variable
B	testLoadTempNRef	return a local
C	testLoadTempNRef2	assign to and then return a local
D	testLiteralNRef	return an integer literal
E	testLoadLiteralIndirect	return the value of a class variable
F	testPopStoreInstVar1	assign an integer literal to an instance variable
G	testPopStoreTemp1	assign an integer literal to a local

The arithmetic tests are performed for both ‘small’ integers and integers since with small integers (16 bit) the Smalltalk/V implementation can use the processor’s ALU directly –

Arithmetic		
	test name	description
H	test3Plus4	$3 + 4$
I	test3LessThan4	$3 < 4$
J	test3Times4	$3 * 4$
K	test3Div4	$3 / 4$
L	test35000Plus45000	$35000 + 45000$
M	test35000LessThan45000	$35000 < 45000$
N	test35000Times45000	$35000 * 45000$
O	test35000Div45000	$35000 / 45000$

The control flow tests exercise selection and iteration –

Control Flow		
	test name	description
P	testShortBranch	if false then .. else ..
Q	testWhile (mean time per iteration over 2000 iterations)	an empty while loop

Array manipulation is a fundamental activity (objects of whatever complexity can be modelled as arrays containing object identifiers). Character strings are stored (conceptually) as arrays of characters, yet require packing and unpacking of 8 bit quantities for storage efficiency –

Array and String manipulation		
	test name	description
R	<code>testArrayAt</code>	accessing an array element
S	<code>testArrayAtPut</code>	assigning to an array element
T	<code>testStringAt</code>	accessing a character within a string
U	<code>testStringAtPut</code>	assigning to a character within a string
V	<code>testSize</code>	finding the size of a string

The last group of tests concentrate on strongly object-oriented aspects –

Object operations		
	test name	description
W	testEq	testing object equality
X	testClass	determining the class of an object
Y	testValue	performing a Block
Z	testCreate	creating an object

The raw results obtained (in microseconds) for the two systems (Lingo and Smalltalk) are presented in the following tables along with the performance ratio of Lingo to Smalltalk (the Smalltalk time divided by the Lingo time). The timings obtained for the Lingo system were highly consistent, typically varying by less than 2 per cent across several (5 or more) repetitions of a test. Occasional 'rogue' results were obtained (and excluded from the averaging process). These were associated with the triggering of garbage collections as a consequence of the REKURSIV's pager tables being full. The Smalltalk system showed a far greater variability in timings (15 per cent). This was attributed to the incremental garbage collection strategy employed and the coarse resolution of the timer (1 millisecond).

Environment				
	test name	Lingo time	Smalltalk/V time	Performance ratio
A	testLoadInstVar	1.19	2.20	1.85
B	testLoadTempNRef	1.41	8.30	5.89
C	testLoadTempNRef2	2.71	13.60	5.01
D	testLiteralNRef	0.58	6.60	11.38
E	testLoadLiteralIndirect	2.40	9.80	4.08
F	testPopStoreInstVar1	1.45	8.20	5.66
G	testPopStoreTemp1	1.84	5.40	2.93

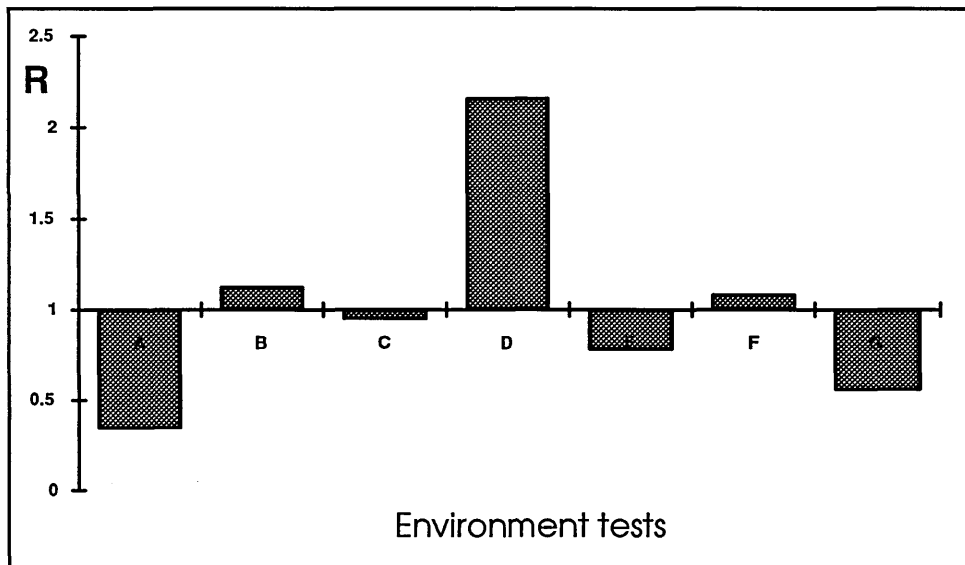
It is clear that the performance *profiles* of the two systems differ, even within this narrow spectrum of activities. This difference can be emphasised by computing a **normalised performance ratio** – the mean of the absolute ratios in the group is taken, and a new score is calculated as a proportion of that mean. A score of 1 indicates that the Lingo system has performed to an expectation in line with the general expectation based on the results for the test group as a whole. A score greater than 1 indicates that the Lingo system has operated better than expected on a particular test. A spread of scores

indicates that the Lingo system reacts differentially to these test activities (and that the system is not merely faster or slower overall than the Smalltalk system).

The mean for this group is 5.26 and the normalised performance ratios,

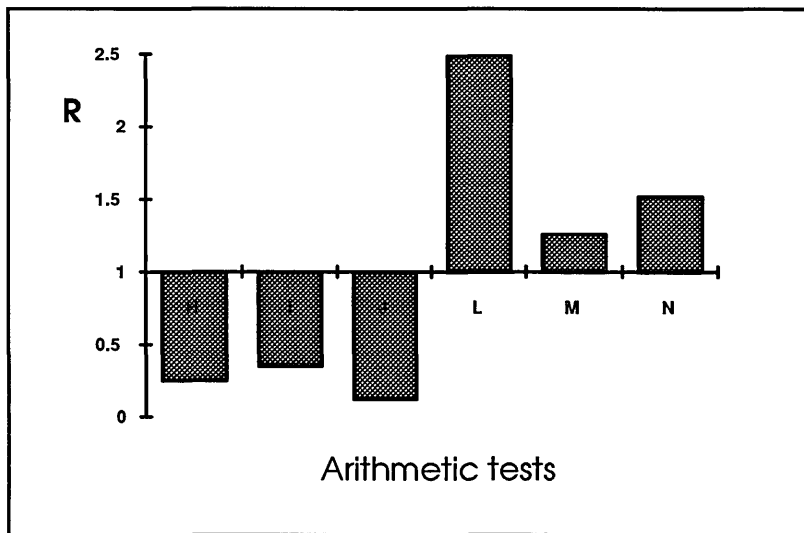
R , are

Environment		
	test name	R
A	testLoadInstVar	0.35
B	testLoadTempNRef	1.12
C	testLoadTempNRef2	0.95
D	testLiteralNRef	2.16
E	testLoadLiteralIndirect	0.78
F	testPopStoreInstVar1	1.08
G	testPopStoreTemp1	0.56



In the Arithmetic group, the normalised performance ratios have been calculated excluding the timings for the division tests since these timings were so large for the Smalltalk system. The mean (unnormalised) performance ratio was 13.1. It is unsurprising that the Lingo system operates similarly for arithmetic on both 16 and 32 bit integers since it is essentially a 32 bit machine.

Arithmetic				
	test name	Lingo time	Smalltalk time	<i>R</i> ratio
H	test3Plus4	3.97	13.10	0.25
I	test3LessThan4	3.48	15.80	0.35
J	test3Times4	11.04	16.90	0.12
K	test3Div4	20.20	900.70	<0.01
L	test35000Plus45000	5.24	170.70	2.49
M	test35000LessThan45000	4.74	78.00	1.26
N	test35000Times45000	12.43	247.20	1.52
O	test35000Div45000	21.69	3931.00	<0.01

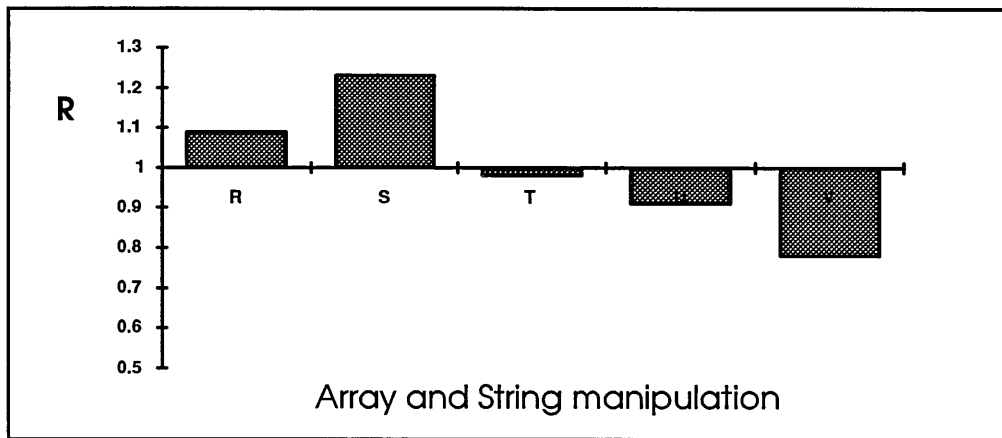


The Control Flow group has only two tests; the same analysis is computed though for the sake of completeness (The mean unnormalised performance ratio for the group was 3.85).

Control Flow				
	test name	Lingo time	Smalltalk time	<i>R</i> ratio
P	testShortBranch	2.60	7.70	0.77
Q	testWhile	5.21	10.46	1.23

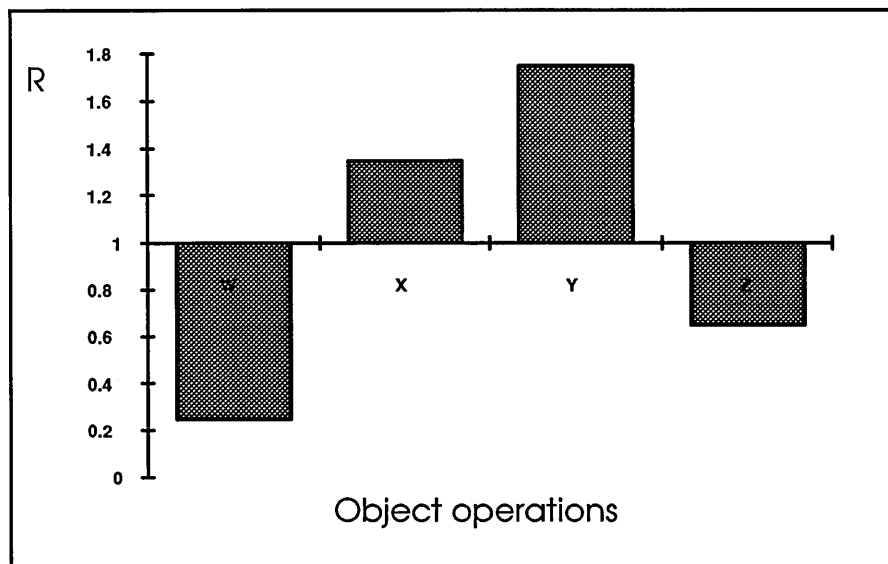
The Array and String manipulation group tests displayed the smallest variation in performance ratios and also the smallest mean performance ratio (2.51).

Array and String manipulation				
	test name	Lingo time	Smalltalk time	R ratio
R	testArrayAt	12.28	33.50	1.09
S	testArrayAtPut	12.29	38.00	1.23
T	testStringAt	12.69	31.30	0.98
U	testStringAtPut	16.50	37.80	0.91
V	testSize	11.97	23.60	0.78

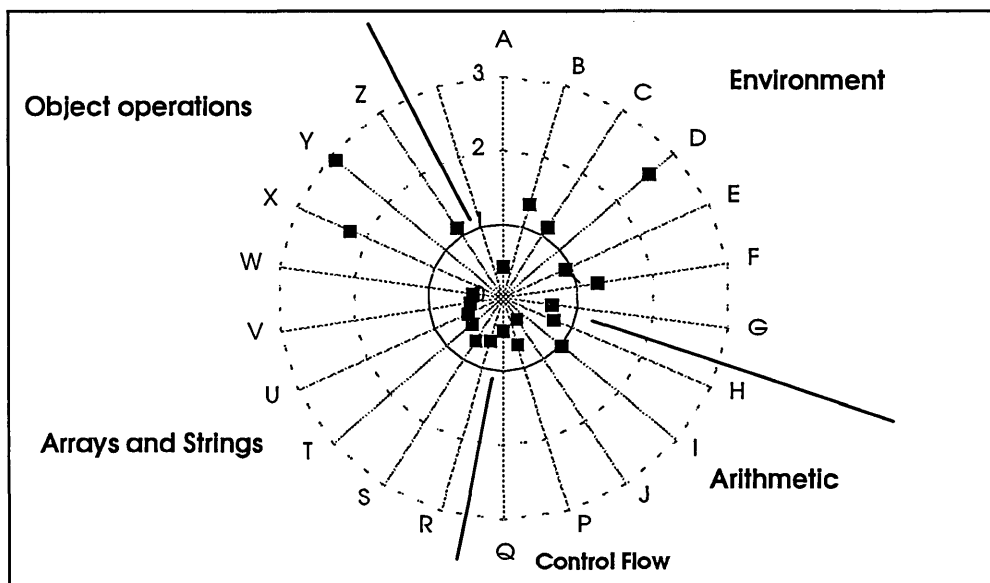


The last group, Object operations, exhibited a wide variation of ratios and the largest mean performance ratio.

Object operations				
	test name	Lingo time	Smalltalk time	<i>R</i> ratio
W	testEq	12.51	22.50	0.25
X	testClass	2.38	23.60	1.35
Y	testValue	16.65	215.20	1.75
Z	testCreate	13.18	65.90	0.65



The following radar diagram summarises the medium scale testing of the REKURSIV/Lingo system by giving its relative profile (against the Smalltalk/V system used in the tests). The circle in the middle (the grid line for an R value of 1) indicates parity – roughly, if the two systems reacted similarly to different tasks, points on the diagram would be distributed on the circle. Points outside the circle indicate that the REKURSIV system performs the task more efficiently than would be expected from a comparison of averages.



Overall performance profile (REKURSIV:Smalltalk)

This profile confirms that the REKURSIV is a success to the extent that it supports the squeezing of the semantic gap for Smalltalk-like object-oriented programming languages. Attribute *Y*, `testValue`, is involved in every method call (message send) and so is fundamental to the execution of programs. Class determination (attribute *X*, `testClass`) is an important feature in object-oriented programming and the REKURSIV's hardware sup-

port for this has had a positive effect.

Further discussion and evaluation of these results is to be found in chapter 8.

7.4 Large scale benchmarks

In order to assess the large scale behaviour (for database systems) of the REKURSIV/Lingo system, two general experiments were conceived. Both relate to the storage of relations and the retrieval of tuples within them.

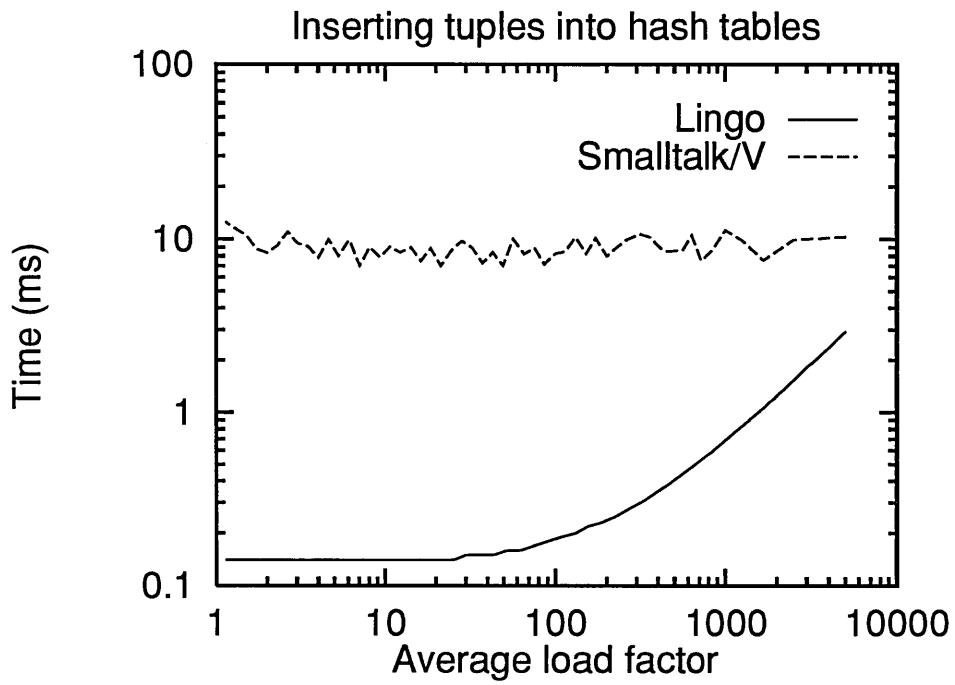
7.4.1 Hashing

In the first such experiment, a hashing scheme was devised for the storage of tuples of a relation. Under the scheme, a hash table consists of a set of buckets and the size of this set is fixed. The hash function determines the bucket address in which a tuple should be stored. Each bucket is a **Dictionary** object – these are associative structures which hold (key,value) associations. **Dictionary** is a built in collection class in both Smalltalk/V and Lingo.

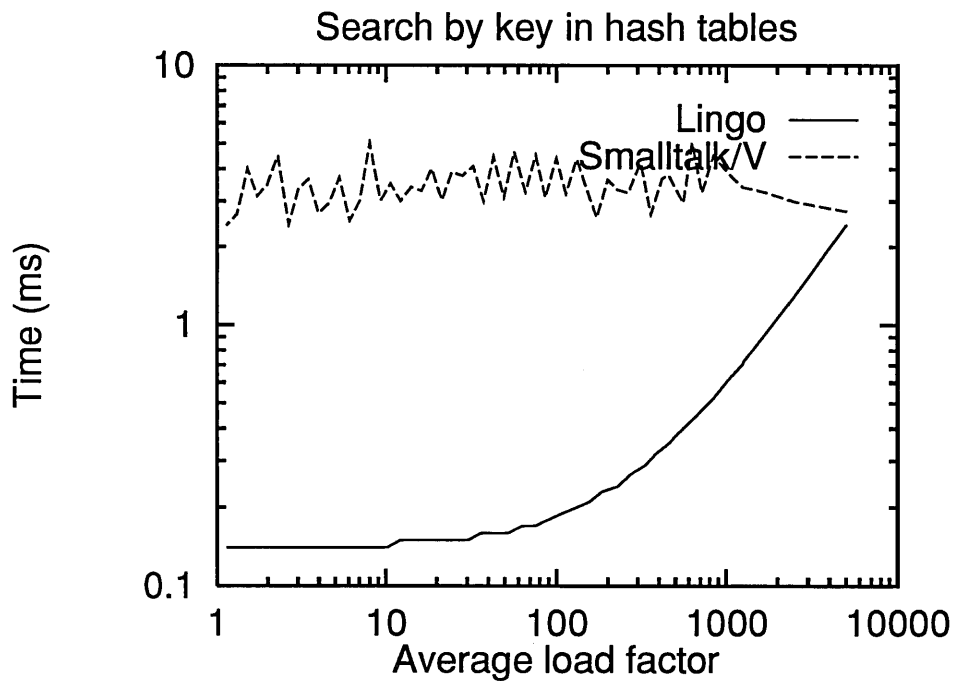
Collisions are not handled in any way. Since a **Dictionary** can hold an indefinite number of associations there is no need to do so. Once dictionaries

start to contain a great number of entries though, accessing the hash table can become very time consuming. The scheme can be used as the basis for an extensible hashing method, where the hash table itself is increased in size when the **average load factor** (the number of tuples in the relation divided by the number of buckets in the table) rises above a threshold of acceptability. Indeed, a purpose of this first experiment is to determine this threshold of acceptability.

The above (inextensible) hashing scheme was programmed in both Lingo and Smalltalk/V, and used to store a relation of cardinality 5000. The size of the hash table was varied from 1 to 5000 buckets, giving a range of ultimate average load factors of from 1 to 5000. The first graph shows the average time taken (in milliseconds) to insert a tuple into a hash table in both the Lingo and Smalltalk/V systems.



The second graph shows the average time taken to search a (full) hashtable for a particular record.



Points to the extreme left of these graphs represent the situation when the hash table storage has reduced to storage in a dictionary (since the hash table has now only one bucket – which is a dictionary!).

The two systems, Smalltalk and Lingo implement dictionaries differently. In Smalltalk, dictionaries are implemented via extensible hash tables and the results confirm this – the performance is independent of load factor and so there is no advantage in creating an extensible form of the experiments hashing scheme since there is no clear load factor at which to trigger the extension of the hash table.

In the Lingo system, however, dictionaries are implemented as array-like

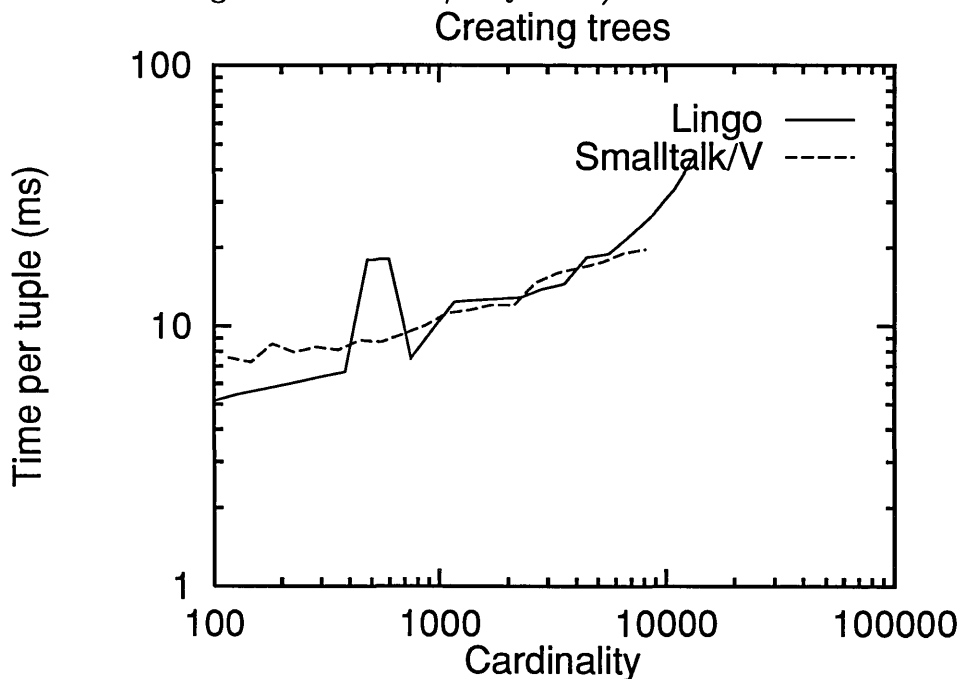
objects. The addition of a key,value association (the `at: put:` method) is managed by linearly searching the array for an existing association with the new key and the overwriting of the old association if this exists or the addition of a new array element if the key is not already present. This method is microcoded. The results indicate that basing an extensible hashing scheme on this prototype scheme would be advantageous if the scheme triggered an extension when the average load factor rose above 10. At these levels, the Lingo system would be performing better than the Smalltalk system by a factor of 70 (for creation of relations) and 8 (for searching through relations). This second factor is in line with an expectation based on the simple experiments reported in section 7.2 earlier within this chapter. The first factor demonstrates the advantage the Lingo system can gain through microcoding features such as memory allocation (where Smalltalk must make calls to an operating system.)

7.4.2 AVL Trees

In the second experiment, the tuples of a relation were stored in a semi-balanced tree structure. Each node of the tree can contain (the object identifiers of) up to two tuples and (the object identifiers of) up to three subtrees. The insertion of a tuple is managed in such a way that the lengths of the

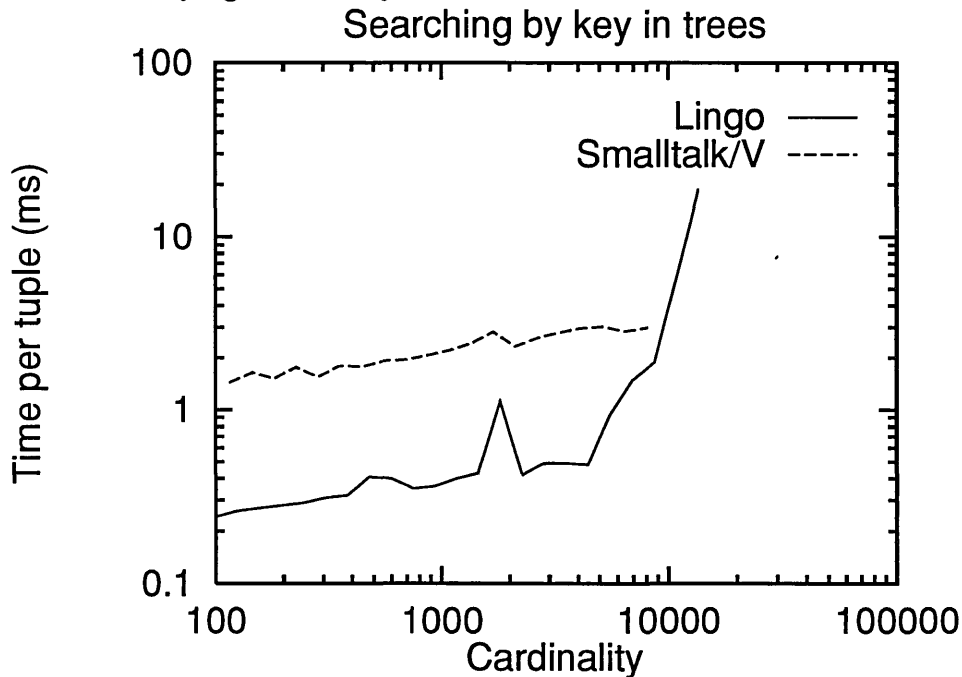
routes from the root node to a leaf never differ by more than one (complete balancing is not ensured). The tree is also ordered in the sense that the correct route to the insertion location for an incoming tuple is determined completely by key comparisons as is the search for the presence of a tuple. The balancing ensures that the number of comparisons that are made is within one of the optimum.

The following graph shows the average time (per tuple, in milliseconds) taken to create a tree against the ultimate cardinality of the tree (again for both the Lingo and Smalltalk/V systems).



The next graph shows the average time taken to search for a tuple by key

in trees of varying cardinality.



This second graph confirms a general performance ratio of 8 to 1 in favour of the Lingo system when manipulating data structures (the sharp deterioration in the Lingo performance for cardinalities in excess of 8000 or so is attributed to the a great increase in 'pager table full' faults and their consequent garbage collections and disk accesses). Taken together with the results for the simple hashing scheme, the graphs indicate that for both the Lingo and Smalltalk systems hash storage and tree storage perform similarly for searching through relations of cardinality less than a few thousand. Further discussion and evaluation of these results is to be found in chapter 8.

Chapter 8

Conclusions

8.1 Qualitative results

The contributions of this work fall into two camps. On the qualitative side, the experience of software construction for the Lingo/REKURSIV has yielded results that are transferable to object-oriented development platforms such as Smalltalk.

These contributions (other than the implementation of a database system) are

- In the field of Formal Specification methods, the methodology exposed in chapter 6 for program derivation is an adaptation new to this work.

The methodology can only be applied to derive programs in class based

languages. A relationship between an equational style of functional programming and object-oriented programming has been established.

The formal specification itself is novel in that it tackles the nature of domains.

- The development of a top-down parser generator (chapter 5) is novel in that the usual automated generation strategy is table driven. The advantage gained by using a top down strategy lies in the ease with which semantic actions can be attached. The dynamic binding of Smalltalk (or Lingo) is the essential property that allows the inclusion of semantic actions to be relatively transparent.
- The main contribution of the work, though, is to arrive at quantitative results on the performance of the REKURSIV detailed in the following section.

Future work, based on these contributions, that the author would like to engage in concern:

- A movement away from this project's tight coupling with the relational model, towards the functional model and a suitable query language such as FDL [88]. This would allow functional language implementation techniques (such as the supercombinator approach investigated by

Khan [69] to be employed, as well as providing a harmonious and more uniform treatment of the different levels of the architecture.

- The further development of compiler writing tools. A difficulty with the parser-generator developed within this project is the difficulty of disentangling syntactic and semantic definitions within the source file. Visual programming techniques within a well considered user interface may yield a truly useable compiler work bench. In addition, work should be done to incorporate a more formal approach to semantics specification.
- The refinement of program derivation from algebraic specifications. The question of state in such specifications and the inclusion of higher-order functions remain as challenges.

8.2 The use of the REKURSIV for database work

The results, particularly from chapter 7, indicate that the REKURSIV is not an ideal target for database implementations in its present form.

A crucial factor in database work is the efficient storage and retrieval

of large sets of data. The experiments comparing hash storage and tree storage, indicate the REKURSIV's performance is fine until, put crudely, it is 'full', at which point there is a catastrophic deterioration in performance. The tragedy is that this 'fullness' is fallacious – the real physical memory is perfectly capable of containing the small relations of cardinality 8000 or so and is not completely allocated; it is the *pager tables* that are full since the system, in pursuance of the relational operation, has generated in excess of 64K objects. The ensuing garbage collection (and its associated disk accesses) then completely swamps performance as the system tries to make space in the pager tables for another object identifier whilst keeping as many *associated* object identifiers in place as possible. The majority of resident objects are associated by virtue of representing tuples from the operands of whatever relational operation is currently being performed.

Larger pager tables would allow the degradation point not to be reached until higher cardinalities were encountered but would not remove the problem altogether. Alternatively, an investigation on optimal garbage collection strategies may ameliorate the catastrophic degradation for *particular* data storage regimes.

The REKURSIV recognises some classes of objects as *compact*. Compact objects have their types and values coded into their 40-bit object identifiers

and do not require use of the pager tables. The use of compact objects seems unrealistic, however, since the Lingo system itself must be modified to recognise them and it is not completely clear how this can be done.

A radical approach to database implementation using the REKURSIV would be to remove it from the HADES configuration altogether and provide it with its own disk processor rather than the artificial arrangement of using the host computer's Unix file store. In addition, for dedicated database work, it is unnecessary to have the REKURSIV configured for the general purpose programming language Lingo with the overheads that provision of general purpose power entail. In principle, storage regimes, such as balanced trees and extensible hash tables, crucial aspects of database work, could be microcoded and support any REKURSIV configuration. At present, the only built-in associative storage structure available on the REKURSIV is the **Dictionary** class, whose methods are microcoded. However, these are implemented as linear array-like structures for which search and lookup has $O(n)$ efficiency. Associative structures based on AVL trees (where lookup has $O(\log(n))$ efficiency) would perform better for sufficiently large cardinality n (the number of elements stored in the structure). Microcoding AVL trees would decrease the threshold cardinality at which advantage is gained. This benefit, however, would only hold as long as the cardinality was small enough

to avoid disk processing (through either the physical object store or the pager tables being full) at which point the input/output cost becomes the important factor.

8.3 The verdict on the REKURSIV

The experimentation for this is presented in chapter 7. How is this to be judged? Clearly, it is impossible to give a positive verdict on the REKURSIV – the world has already denied this with the demise of Linn-Smart Computing.

On technical grounds the REKURSIV is a success. The results in chapter 7 show that its computational profile *is* different from a conventional processor and that its profile favours the fundamental operations that support object-orientation (class determination, message sending, method look-up and so on).

The REKURSIV's handling of large data sets is rather harder to discern – a rather good performance suddenly worsens as the mechanics of the virtual memory management system are brought into play. Partly this is unfair to the REKURSIV, since it was designed to operate with its own disk processor managing the swap space rather than communicate with a unix file system

through some registers on the VME bus of the host Sun workstation. On the other hand, the problem really emanates from the pager tables being too small (or fixed in size at all) rather than main object memory being full. This detail does not seriously detract from the technical success of the REKURSIV chip set.

The real problem lies in the REKURSIV project itself. It is extremely doubtful that *architectural* advances in processor design can be successful at least when presented on their own without taking significant advantage of an advance in the underlying implementation technology. Harland has argued that his architecture squeezes the semantic gap and that advances in underlying technology if applied to the REKURSIV would maintain its advantage over conventional processors. This misses the point that these technological advances are so great that they completely swamp architectural advantage. In addition, technological advance occurs in reaction to a need. The advent of RISC processors in the late nineteen eighties generated a need for fast cache technology. Today, the fruits of cache technology have been applied to non-RISC processors such as the Intel 80486 to allow processor clock rates an order of magnitude greater than were possible six years ago.

It is also fallacious to believe that there is such a thing as a conventional processor. The distribution of 'intelligence' throughout a computer (interrupt

mechanisms, direct memory access, floating-point and graphics processors, intelligent disc controllers and so on) have completely distanced the computer from its ancestors. These advances represent continued effort at streamlining the computation process rather than attempts to revolutionise the computational basis. This last is a vain goal – in the end all processing machines are Turing equivalent.

8.4 The failure

The last section paints a bleak picture. What are the lessons to be learned?

It is instructive to remember that in the field of processor design, as in so much of Computing, ideas achieve success if the cost of implementing them will obviously be recouped quickly. The success of the personal computer was not due to its technical merit. It did not immediately revolutionise people's work practice. Instead it insinuated itself into indispensability through gradual stages of usefulness, most of which were advances in software applications.

The REKURSIV was too large a bite to swallow. On all fronts it was a novelty.

- Users required a Sun workstation to use their REKURSIV. No other

processor in this target market was hosted in this way. The Sun workstation itself became cheaper and more powerful than its embedded REKURSIV.

- The language Lingo was a proprietary product. Smalltalk or C++ would have given an aroma of familiarity.
- The system could not communicate with any existing software or data systems.

8.5 The future

Putting to one side the political and economic factors surrounding the REKURSIV project Harland's work has shown an object-oriented processor is feasible. To move into the future the following recommendation can be made.

A standard for processor architectures should be established. The SPARC standard is an example of such a standard. The specification (SPARC) is separated from the implementation. This allows a gradualist introduction of coprocessor support since the processor-coprocessor interface is defined. From there, an investigation of the processing needs of object-oriented applications could delineate specific activities to build coprocessor support for (this approach has been successfully used for the introduction of graphics

terminals and dedicated X-terminals).

What such an investigation would reveal is beyond the author. High performance message sending and hardware typing support, the technical successes of the REKURSIV, will no doubt be important but it is too early to devise a clearly interfaced mechanism that will allow a conventional processor to successfully share its burden with an object-oriented coprocessor.

The irony is, in the end, that despite the seductive naturalness of object-orientation, the real question is "What do we mean by an object?".

Bibliography

- [1] Abramsky, S. and Sykes, R. 1985. SECD-M: A Virtual Machine for Applicative Programming. In: *Functional Programming Languages and Computer Architectures, LNCS 201*, London: Springer-Verlag.
- [2] Aho, A.V., Sethi, R. and Ullman, J.D. 1986. *Compilers : Principles, Techniques and Tools*, Reading, Mass: Addison-Wesley.
- [3] Atkinson, M.P. and Buneman, O.P. 1989. *Types and Persistence in Database Programming Languages*, Persistent Programming Research Report 17-85, Universities of Glasgow and St. Andrews.
- [4] Babb, E. 1979. Implementing a relational database by means of specialised hardware. *ACM Transactions on Database Systems*, 4(1): 1-29.

- [5] Backus, J. 1959. The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM conference. In: *Proceedings International Conference on Information Processing*, Paris: UNESCO, 125-132.
- [6] Bell, D. 1990. Database Machines. In: *A practical guide to Database systems* edited by S.M. Deen. London: Pitman, chapter 11.
- [7] Birtwhistle, G., Dahl, O.J., Myhrhaug, B. and Nygaard, K. 1973. *SIM-ULA begin*, New York: Petrocelli Charter.
- [8] Bitton, D., Boral, H., De Witt, D.J. and Wilkinson, W.W. 1983. Parallel Algorithms for the Execution of Relational Database Operations. *ACM Transactions on Database Systems*, **8 (3)**, 324-353.
- [9] Bitton, D., De Witt, D.J., Hsiao, D.K. and Menon, J. 1984. A Taxonomy of Parallel sorting. *ACM Computing Surveys*, **16 (3)**, 287-318.
- [10] Boral, H. and De Witt, D.J. 1981. Processor Allocation Strategies for Multiprocessor Database Machines. *ACM Transactions on Database Systems*, **6 (2)**, 227-254.

-
- [11] Boutel, B. 1987. Combinators as Machine Code for Implementing Functional Languages. In: *Functional Programming: Languages, Tools and Architectures*, edited by S. Eisenbach. Chichester: Ellis Horwood.
- [12] Buneman, P., and Frankel, R.E. 1979. FQL – A Functional Query Language. In: *Proceedings of SIGMOD79 Conference, Boston*, edited by P.A. Bernstein, 52–59.
- [13] Burstall, R.M. and Lampson, B. 1984. A Kernel Language for Abstract Data Types and Modules. In: *Proceedings of Semantics of Data Types Conference, Sophia–Antipolis, France, LNCS 1173*, London: Springer–Verlag.
- [14] Cardelli, L. 1983. ML under UNIX. In: *Polymorphism: The ML/LCF/Hope Newsletter*, 1 (3).
- [15] Cesarini, F. and Salza, S. (Editors). 1987. *Database Machine performance: modelling methods and evaluation strategies, LNCS 257*, London: Springer–Verlag.
- [16] Chamberlin, D.D. et al. 1976. SEQUEL 2: a unified approach to data definition, manipulation and control. *IBM journal of Research and Development*, 20 (6), 560–575.

-
- [17] Clarke, T.J.W., Gladstone, P.J.S., MacLean, C.D. and Norman, A.C. 1980. SKIM – the S, K, I Reduction Machine. In: *Lisp Conference, Stanford, 1980.*
- [18] Codd, E.F. 1970. A relational model of data for large shared data banks. *Communications of the ACM*, **13 (6)**, 377–387.
- [19] Codd, E.F. 1971. A database sublanguage founded on the relational calculus. In: *Proceedings of the 1971 ACM–SIGFIDET Workshop on Data Description, Access and Control, San Diego, California*, 35–68.
- [20] Correll, C.H. 1978. Proving Programs Correct through Refinement. *Acta Informatica*, 121–132.
- [21] Dahl, O.J. and Nygaard, K. 1966. Simula – an ALGOL–based simulation language. *Communications of the ACM*, **9(9)**, 671–678.
- [22] Darlington, J. and Reeve, M.J. 1981. ALICE, a Multiprocessor Reduction Machine for the Parallel Evaluation of Applicative Languages. In: *Proceedings ACM Conference of Functional Programming Languages and Computer Architectures, Portsmouth, New Hampshire, 1981*, ACM, 166–170.

- [23] Darlington, J. and Reeve, M.J. 1983. ALICE and the Parallel Evaluation of Logic Programs. In: *Proceedings of the 10th Annual ACM/IEEE Symposium on Computer Architecture, Stockholm, 1983*, ACM.
- [24] Date, C.J. 1991. *An Introduction to Database Management systems, Volume I*, Reading, Mass: Addison-Wesley.
- [25] Davie, A.J.T. and R. Morrison, R. 1981. *Recursive Descent Compiling*, Chichester: Ellis Horwood.
- [26] Deen, S.M. 1985. DEAL - a relational language with deduction, functions and recursion. *Data and Knowledge Engineering*, 1.
- [27] DeMarco, T. 1978. *Structured Analysis and System Specification*, New York: Yourdon Press.
- [28] De Witt, D.J. et al. 1986. Gamma - a high performance dataflow database machine. In: *Proceedings of the International Conference on Very Large Databases, 1986*, 228-237.
- [29] Fairbairn, J. and Wray, S. 1987. Tim - A simple, lazy abstract machine to execute supercombinators. In: *Proceedings of the Third Conference on Functional Programming Languages and Computer Architecture, LNCS 274*. Berlin: Springer-Verlag.

- [30] Folinus, J.J., Madnick, S.E. and Shutzmann, H.B. 1974 Virtual information in database systems. *ACM SIGFIDET*, 6.
- [31] Frost, R.A. 1986. *Introduction to Knowledge Based Systems*. Collins, London.
- [32] Futatsugi, K., Goguen, J., Meseguer, J. and Okada, K. 1987. Parametrized Programming in OBJ2. In: *Proceedings of the 9th International Conference on Software Engineering, Monterey*, 51–60.
- [33] Gamerman, S. and Scholl, M. 1985. Hardware versus Software Data Filtering – the VERSO experience. In: *Proceedings of the 4th International Workshop on Database Machines, Grand Bahama Islands*. Berlin: Springer-Verlag, 125–136.
- [34] Gane, C. and Sarsen, T. 1979. *Structured Systems Analysis: tools and techniques*, Englewood Cliffs, NJ: Prentice-Hall.
- [35] Goguen, J.A., Thatcher, J.W., Wagner, E.G. and Wright, J.B. 1977. Initial Algebra Semantics and Continuous Algebras. *Journal of the ACM*, 24, 68–95.
- [36] Goguen, J.A., Thatcher, J.W. and Wagner, E.G. 1978. An initial algebra approach to the specification, correctness and implementation of

- abstract data types. In: *Current trends in Programming Methodology, Volume IV*, edited by R. Yeh. Prentice-Hall.
- [37] Goguen, J.A. 1983. *Parameterized Programming*. Menlo Park, Ca.: SRI International Computer Science Lab.
- [38] Goguen, J.A. and Meseguer. 1987. Unifying Functional, Object-oriented and Relational Programming. In: *Research Directions in Object-oriented Programming*, edited by B.Shriver and P. Wegner, Cambridge, Mass.: MIT press, 417-477.
- [39] Goldberg, A. and Robson, D. 1983. *Smalltalk-80 : the language and its implementation*, Reading, Mass.: Addison-Wesley.
- [40] Goldberg, A. 1984. *Smalltalk-80 : the interactive programming environment*, Reading, Mass.: Addison-Wesley.
- [41] Goodenough, J.B. and Gerhart, S.L. 1975. Towards a theory of test data selection. *IEEE Trans. on Software Engineering*, SE-1 (June), 156-173.
- [42] Gray, P. 1984. *Logic, Algebra and Databases*, Chichester: Ellis Horwood.

- [43] Gray, P.M.D., Paton, N.W. and Kulkarni, K.G. 1992. *Object-Oriented Databases: a Semantic Data Model Approach*, Prentice-Hall International.
- [44] Greenblatt, R.D., Knight, T.F., Holloway, J. et al. 1984. The LISP Machine. In: *Interactive Programming Environments*, edited by D.R. Barstow, H.E. Shrobe and E. Sandwell. New York: McGraw-Hill, 326–352.
- [45] Guttag, J.V. 1977. Abstract Data Types and the Development of Data Structures, *Communications of the ACM*, 20, 396–404.
- [46] Guttag, J.V., Horowitz, E. and Musser, D.R. 1978. Abstract Data Types and Software Validation, *Communications of the ACM*, 21, 1048–1064.
- [47] Guttag, J.V. and Horning, J.J. 1983. *Preliminary Report on the Larch Shared Language*. Palo Alto, Ca.: Xerox Corporation.
- [48] Hall, A. 1990. Seven Myths of Formal Methods. *IEEE Software*, 4 (September), 11–19.

- [49] Hall, A. and Pfleeger, S.L. May 1995. Some metrics from a formal development. In: *IEE Colloquium on: Practical Applications of Formal Methods*.
- [50] Held, G.D., Stonebraker, M. and Wong, E. 1975. INGRES – A relational database management system. In: *Proceedings of AFIPS 1975 NCC Vol. 44*. Montvale N.J.: AFIPS Press, 409–416.
- [51] Harland, D.M. 1984. *Polymorphic Programming Languages*, Chichester: Ellis Horwood.
- [52] Harland, D.M. 1988. *REKURSIV: an object-oriented architecture*, Chichester: Ellis Horwood.
- [53] Harland, D.M. 1989. *A Guide To Lingo*, Glasgow: Linn Smart Computing Limited.
- [54] Harper, R., MacQueen, D. and Milner, R. 1986. *Standard ML, ECS-LFCS-86-2*, Edinburgh: Laboratory for Foundations of Computer Science, University of Edinburgh.
- [55] Held, G.D., Stonebraker, M. and Wong, E. 1975. INGRES – A relational database management system. In: *Proceedings of AFIPS 1975 NCC Vol. 44*. Montvale N.J.: AFIPS Press, 409–416.

- [56] Henderson, P. 1980. *Functional Programming: Application and Implementation*. Englewood Cliffs, NJ: Prentice Hall International.
- [57] Hoare, C.A.R. 1969. An Axiomatic Basis for Computer Programs. *Communications of the ACM*, 12, 576–580.
- [58] Hoffman, C.M. and O’Donnell, M.J. 1982. Programming with Equations. *ACM Transactions on Programming Languages and Systems*, 4 (6 January), 83–112.
- [59] Hsiao, D.K. 1983. *Advanced database machine architecture*, Englewood Cliffs, NJ: Prentice Hall International.
- [60] . Hughes, J. 1982. *Graph reduction with Super-Combinators, PRG-28*, Oxford: Oxford University Computing Laboratory.
- [61] Ingalls, D.H.H. 1978. The Smalltalk-76 Programming System Design and Implementation. In: *Proceedings of the 5th POPL, Tucson, Arizona*, 9–17.
- [62] Ingalls, D.H.H. 1984. The Evolution of the Smalltalk-80 Virtual Machine. In: *Smalltalk-80 : bits of history, words of advice*, edited by G Krasner. Reading, Mass.: Addison-Wesley, 153–174.

- [63] Johnson, S.C. 1975. *Yacc - yet another compiler compiler*, C.S. Technical report 32, Murray Hill, NJ: Bell Telephone Laboratories.
- [64] Johnsson, T. 1984. Efficient Computation of Lazy Evaluation. *ACM SIGPLAN*, 19 (6), 58-69.
- [65] Jones, C.B. 1980. *Software Development: A Rigorous Approach*, Englewood Cliffs, NJ.: Prentice-Hall International.
- [66] Jones, C.B. 1986. *Systematic Software Development using VDM*, Englewood Cliffs, NJ: Prentice Hall International.
- [67] Jones, S.B. and Sinclair, A.F. 1989. Functional Programming and Operating Systems. *The Computer Journal*, 32 (2), 162-174.
- [68] Kay, A. and Goldberg, A. 1977. Personal Dynamic Media. *Computer*, 10 (3), 31-42.
- [69] Khan, M.Y. 1993. *The Implementation of Functional Languages on an Object-oriented Architecture*, PhD Thesis, Dundee Institute of Technology.
- [70] Krasner, G. (Editor). 1983. *Smalltalk-80 : bits of history, words of advice*, Reading, Mass.: Addison-Wesley.

- [71] Leavenworth, B. 1970. Review #19420. *Computing Reviews*, 11 (July), 396-397.
- [72] Lesk, M.E. and Schmidt, E. 1975. *Lex - A Lexical Analyzer Generator*, C.S. Technical Report No. 39, Murray Hill, NJ: Bell Telephone Laboratories.
- [73] London, R.L. 1971. Software Reliability through Proving Programs Correct. In: *Proceedings of IEEE International Symposium on Fault Tolerant Computing*, Los Alamitos, CA: IEEE.
- [74] McCall, K. 1983. "The Smalltalk-80 Benchmarks". In: *Smalltalk-80 : bits of history, words of advice*, edited by G Krasner. Reading, Mass.: Addison-Wesley, 153-174.
- [75] Meyer, B. 1985. On Formalism in Specifications. *IEEE Software*, 2 (January), 6-26.
- [76] Milne, A.C. 1984. *A Syntax Analyser and Manipulation Tool*. M.Sc. thesis, University of St. Andrews.
- [77] Milner, R. 1979. A Theory of Type Polymorphism in Programming. *Journal of Comp. Sys. Sci.*, 17 (3), 1979.

- [78] Milner, R. 1984. A Proposal for Standard ML. In: *Proceedings of ACM Symposium on Lisp and Functional Programming, Austin, Texas*, ACM.
- [79] Moon, D. and Weinreb, D. 1986. Object-oriented Programming with Flavors. In: *Proceedings of the 1st OOPSLA, Portland, Oregon*. 1–8.
- [80] Morris, J.H. 1973. Types are not Sets. In: *Proceedings of the 1st ACM symposium on Principles of Programming Languages*, 120–124.
- [81] Natanson, L.D., Samson, W.B. and Wakelin, A.W. 1991. A Recursive Database Query Language on an Object-Oriented Processor. In: *Applications of Supercomputers in Engineering II*. edited by C.A. Brebbia, D. Howard and A. Peters. Southampton: Computational Mechanics Publications, 191–205.
- [82] Naur, P. 1960. Report on the Algorithmic Language ALGOL60. *Communications of the ACM*, **3** (5), 229–314.
- [83] Naur, P. 1969 Programming by Action Clusters. *BIT*, **9** (3), 250–258.
- [84] Nix, C.J. and Collins, B.P. 1988. The Use of Software Engineering, Including the Z Notation, in the Development of CICS. *Quality Assurance*, **14** (September), 103–110.

- [85] Patterson, D.A. 1985. Reduced Instruction Set Computers. *Communications of the ACM*, **28** (1), 8–21.
- [86] Peter Deutsch, L. 1983. *The Dorado Smalltalk-80 Implementation: Hardware Architecture's Impact on Software Architecture*. Reading Mass.: Addison-Wesley.
- [87] Pier, K.A. 1983. A Retrospective on the Dorado, a High Performance Personal Computer. In: *Proceedings of the 10th Annual Symposium on Computer Architecture, Stockholm*, 252–269.
- [88] Poulouvassilis, A. 1992. The Implementation of FDL, a Functional Database Language. *The Computer Journal*, **35** (2), 119–128.
- [89] Reade, C.M.P. 1989. *Balanced Trees with Removals: an Exercise in Rewriting and Proof*, Brunel University Computer Studies Technical Report CSTR-89-3.
- [90] Sadeghi, R. 1988. HQL - A Historical Query Language. In: *Proceedings of BNCOD6*, Cambridge: Cambridge University Press.
- [91] Samson, W.B, Deen, S.M. Wakelin, A.W. and Sadeghi R. 1987. Formalising the Relational Algebra — Some specifications, observations,

- problems and suggestions. Presented at: *Formal Methods Workshop, Teesside Polytechnic (UK)*.
- [92] Samson, W.B. and Wakelin, A.W. 1989. PEARL – a database query language for the integration of data and knowledge bases. In: *Proceedings of the International conference on AI in industry and government, Hyderabad, India*, edited by P. Balagurusamy, London: Macmillan.
- [93] Samson, W.B. and Wakelin, A.W. 1992. Algebraic Specification of Databases – a survey from a Database Perspective. In: *Specification of Database Systems, Glasgow 91*, edited by D.J. Harper and M.C. Norrie. Berlin: Springer-Verlag.
- [94] Schach, S.R. 1993. *Software Engineering, 2nd Edition*, Homewood, IL: Aksen Associates, 160–161
- [95] Schweppe, H. Zeidler, H. Hell, W. et al. 1983. RDBM – a Dedicated Multiprocessor System for Database Management. In: *Advanced Database Machine Architecture*, edited by D.K. Hsiao. Prentice-Hall, 36–86.
- [96] Shao, J., Bell, D.A. and Hull, M.E.C. 1988. LQL: A Unified Language for Deductive Database Systems. In: *Proceedings of IFIP International*

Conference on the Role of AI in Databases and Information Systems.

- [97] Shipman, D.W. 1981. The functional data model and the data language DAPLEX. *ACM transactions on Database Systems*, 6 (1).
- [98] Shoch, J.F. 1979. An Overview of the Programming Language Smalltalk-72. *ACM SIGPLAN Notices*, 14 (9), 64-73.
- [99] Smith, D.E. 1975. *Pygmalion, a Creative Programming Environment*. AI memo 26, Cambridge Mass.: Massachusetts Institute of Technology AI lab.
- [100] Smith, D.C, Irby, C., Kimball, R. et al. 1983. Designing the Star User Interface. In: *Integrated Interactive Computing Systems*, edited by P. Degano and E. Sandwell, Amsterdam: North-Holland, 297-313.
- [101] Spivey, J.M. 1992. *The Z notation: a refernce manual, second edition*, New York: Prentice Hall.
- [102] Stabler, H., Drummond, B., Rose, S. and Harland, D. 1989. *The REKURSIV C instructions*, Glasgow: Linn Smart Computing Limited.

- [103] Stonebraker, M., Wong, E. and Kreps, P. 1976. The Design and Implementation of INGRES. *ACM Transactions on Database Systems*, **1** (3), 189–222.
- [104] Stoye, W.: 1985. *The implementation of functional languages using custom hardware*. Ph.D. thesis, Technical report 81. University of Cambridge.
- [105] Sussman, G.J. 1981. Scheme–79–Lisp on a Chip. *IEEE Computer*, **14** (7), 10–21.
- [106] S Thatte, S. 1987. A Refinement of strong sequentiality for term rewriting with constructors. *Information and Computation*, **72** (1), 46–55
- [107] Tsichritzis, D.C. and King, A. (editors). 1978. The ansi/X3/SPARC DBMS Framework: report of the study group on Data Base Management Systems. *Information systems* (3).
- [108] Turner, D. 1979. A New Implementation Technique for Applicative Languages. *Software Practice and Experience*, **9**
- [109] Turner, D. 1985. Miranda – a non–strict functional language with polymorphic types. In: *Proceedings of the conference on Functional Pro-*

- programming Languages and Computer Architecture, LNCS 201*, Berlin: Springer-Verlag.
- [110] Ullman, J.D. 1982. *Principles of Database Systems*. Rockville, Md.: Computer Science Press.
- [111] Ungar, D. and Patterson, D. 1987. What Price Smalltalk? *IEEE Computer*, 18 (1), 67-74.
- [112] Wakelin, A.W. 1988. A database query language for operations on graphical objects. Ph.D. thesis, Dundee Institute of Technology.
- [113] Waugh, K.G, Williams, M.H., Kong, Q. Salvani, S. and Chen, G. 1990. Designing SQUIRREL: an extended SQL for a deductive database system. *The Computer Journal*, 33 (6), 535-546.
- [114] Weinreb, D. and Moon, D. 1981. *LISP Machine manual*, Cambridge Mass.: Symbolics Inc.
- [115] Yourdon, E. and Constantine, L.L. 1979. *Structured Design: Fundamentals of a Discipline of Computer Program and System Design* Englewood cliffs, NJ: Prentice Hall.
- [116] Zloof, M.M. 1977. Query By Example: a Database Language. *IBM Systems Journal*, 16, 324-343.

Appendix A

Code for the preliminary experiment

In Smalltalk, nodes involving addition are modelled by the class **Plus** -

```
Object subclass: #Plus
  instanceVariableNames: 'left right'
  classVariableNames: ''
  poolDictionaries: '' !
!Plus class methods!
of: a and: b
"create a new node representing a + b "
  ^ ((super new) left: a right: b)
!!
!Plus methods!
"private - set instance variables of newly created instance"
left: a right: b
  left := a.
  right := b.
```

```
!  
compute  
"return the result of the addition for this node"  
  ^ ((left compute) +(right compute))  
!!
```

For this scheme to work, **Number** objects (which may reside in the left and right branches of nodes) must be able to respond to **compute** messages

-

```
!Number methods!  
compute  
  ^ self  
!!
```

The actual test is contained as a class method, **test**, of the class **Bench**, which also has a class method, **init**, to create the original tree -

```
Object subclass: #Bench  
  instanceVariableNames: ''  
  classVariableNames: 'theTree'  
  poolDictionaries: '' !  
!Bench class methods!  
init  
  | a b c |  
  a := Plus of: 2 and: 1.  
  b := 3.  
  [ b <= 11.] whileTrue:  
  [  
    a := Plus of: b and: a.  
    b := b + 1.  
  ].  
theTree := a.
```



```
theTree printNl .
!  
  
test | a b c |  
a := 0.  
[ a <= 1000. ] whileTrue:  
[  
  b := 1.  
  [ b <= 1000 .] whileTrue:  
  [  
    c := theTree compute.  
    b := b + 1.  
  ].  
  a printNl.  
  a := a + 1.  
].  
!!
```

The C version is -

```
#include <stdio.h>  
#define NUMBER 0  
#define PLUS 1  
#define MINUS 2  
#define TIMES 3  
#define DIVIDE 4  
typedef struct node  
{  
  int kind;  
  union  
{  
    int number;  
    struct nodes { struct node * left, * right; } nodes;  
  } body;  
} node ;
```

```
node * new()
{
    node * temp;
    temp = ( node * ) malloc(sizeof(node));
    return temp;
}

void show( x)
node * x;
{
    switch (x->kind)
    {
        case NUMBER : printf(" %d ", x->body.number); break;
        default : printf("("); show(x->body.nodes.left);
            switch (x-> kind)
            {
                case PLUS : printf("+"); break;
                case MINUS : printf("-"); break;
                case TIMES : printf("*"); break;
                case DIVIDE : printf("/"); break;
            }
            show(x->body.nodes.right);
            printf(")");
    }
}

int eval( x)
node * x;
{ int le, ri;
    switch (x->kind)
    {
        case NUMBER : return x->body.number;
        default : le = eval(x->body.nodes.left);
            ri = eval(x->body.nodes.right);
            switch (x-> kind)
            {
```

```
case PLUS : return (le + ri);
case MINUS : return (le - ri);
case TIMES : return (le * ri);
case DIVIDE : return (le / ri);
    }

}
}
main()
{
    node * top,*i,*j,*k;
    int how;
        int the,qu,quo=0;
    i = new();
    j = new();
    top = new();
    j->body.number=2;
    i->kind = NUMBER;
    i->body.number=1;
    j->kind = NUMBER;
    top->kind = PLUS;
    top->body.nodes.left = j;
    top->body.nodes.right = i;
    for (how = 3; how <= 11; how ++ )
    {
        j=new();
        j->body.number = how;
        j->kind = NUMBER;
        i = new();
        i->kind = PLUS;
        i->body.nodes.left = j;
        i->body.nodes.right = top;
        top = i;
    }
    show(top);
    while (1)
    {
```

```
for (the=0; the<1000; the++)
  qu = eval(top);
printf("%d\n",quo++);
}
}
```

In the second benchmark, the contents of a 100 element (integer) array are computed by multiplying elements of two other arrays (the *i*th element is computed by multiplying the *i*th element of the first array by the (100-*i*)th element of the second array. The first array contains the integers 1 to 100, the second has the integers 100 down to 1.

The Smalltalk for this is -

```
Object subclass: #Bench2
  instanceVariableNames: ''
  classVariableNames:
    'Ac Ab Aa '
  poolDictionaries: '' !
!Bench2 class methods !

init | i j k |
  Aa := Array new: 100.
  Ab := Array new: 100.
  Ac := Array new: 100.
  i := 1.
  [ i <= 100 ] whileTrue:
  [
    Aa at: i put: i.
    Ab at: (101-i) put: i.
    i := i+1.
  ].!

```

```
test | i j k q |
Transcript nextPutAll: 'going'; cr.
i := 1. j := 1.
[ j <= 1000. ] whileTrue:
[
  i := 1.
  [ i <= 1000. ] whileTrue:
  [
    k := 1.
    [ k <= 100. ] whileTrue:
    [
      Ac at: k put: ((Aa at: k ) * ( Ab at: (101-k))).
      k := k + 1.
    ].
    i := i + 1.
  ].
  j printOn: Transcript. Transcript cr.
  j := j + 1.
].! !
```

and the corresponding C code is -

```
main()
{
  int a[100],b[100],c[100];
  int i,j,k=0;

  for (i=0;i<=99;i++) { a[i]=i; b[99-i]=i;}

  while (1)
  {
    for(i=0;i<=1000;i++)
    {
      for (j=0;j<=99;j++)
```

```
        {  
c[j] = a[j] * b[99-j];  
        }  
    }  
    printf("%d\n",k++);  
}  
}
```

Appendix B

An SML specification

```
abstype domain = dom of string
with
  exception domexception
  fun validdom(dom s) = s = "int" orelse s = "string"
  fun nameofdom(dom s) = s
  fun displaydom(dom s) = print s
  fun makedom s = let val d = dom s in
                    if validdom d then d
                    else raise domexception
                  end
end

abstype name = nam of string
with
  fun nameofnam(nam s) = s
  fun makenam s = nam s
  fun nameq(nam s1, nam s2) = s1=s2
  fun displaynam(nam s) = print s
end
```

```
abstype attribute = ival of int | cval of string
with
```

```
  exception attexception
  fun makeival(i) = ival(i)
  fun makecval(s) = cval(s)
  fun getival(ival(i)) = i
    |getival(cval(c)) = raise attexception
  fun getcval(cval(c)) = c
    |getcval(ival(i)) = raise attexception
  fun type2string(ival(i)) = "int"
    |type2string(cval(c)) = "string"
  fun att2string(ival(i)) = makestring i
    |att2string(cval(c)) = c
  fun atteq(ival(_),cval(_))=false
    |atteq(cval(_),ival(_))=false
    |atteq(ival(x),ival(y))=x=y
    |atteq(cval(x),cval(y))=x=y
  fun attgt(ival(_),cval(_))=false
    |attgt(cval(_),ival(_))=false
    |attgt(ival(x),ival(y))=x>y
    |attgt(cval(x),cval(y))=x>y
  fun attlt(ival(_),cval(_))=false
    |attlt(cval(_),ival(_))=false
    |attlt(ival(x),ival(y))=x<y
    |attlt(cval(x),cval(y))=x<y
  fun attne(ival(_),cval(_))=false
    |attne(cval(_),ival(_))=false
    |attne(ival(x),ival(y))=x<>y
    |attne(cval(x),cval(y))=x<>y
  fun attge(a,b) = atteq(a,b) orelse attgt(a,b)
  fun attle(a,b) = atteq(a,b) orelse attlt(a,b)
end
```

```
abstype scheme = sch of ( (name list) * (( domain * name ) list ))
(* The first name list is the names of the key attributes *)
with
```

```
  exception schemeexception
```



```

fun second(x,y) = y
fun makesch (l1,l2) = sch (l1,l2)
fun schlength(sch (l1,l2)) = length(l2)
fun validscheme(sch(h::t,[])) = false
  | validscheme(sch([],_)) = true
  | validscheme(sch(h1::t1,h2::t2)) = isin(h1,h2::t2) andalso
    validscheme(sch(t1,h2::t2))
and isin(x,[]) = false
  | isin(x,(h1,h2)::t) = nameq(x,h2) orelse isin(x,t)
fun keyofscheme(sch(l1,l2)) = l1
fun namesinscheme(sch(q,h::t)) = second(h)::namesinscheme(sch(q,t))
  | namesinscheme(sch(_,[])) = []
fun bodyofscheme(sch(l1,l2)) = l2
fun schhd(sch (l1,l2)) = hd l2
fun schtl(sch (l1,l2)) = tl l2
fun schappend(sch(l11,l12),sch(l21,l22))=sch((l11@l21),(l12@l22))
fun schnull(sch (l1,l2)) = l2 = nil
fun equiv(s1:scheme, s2:scheme):bool = if schnull(s1) andalso schnull(s2)
  then true else
    if schlength(s1) <> schlength(s2) then false else
      nameofdom(first(schhd s1)) = nameofdom(first(schhd s2)) andalso
      equiv(sch([],schtl s1), sch([],schtl s2))
and first(x,y) = x
exception renamefault
fun rename(s,l3,l4) = let val s1 = sch(l3,l4) in
  if equiv(s,s1) andalso validscheme(s1) then s1 else
    raise renamefault
end
fun posinscheme(s:scheme,n:name) = if schnull s then raise schemeexception
  else if nameofnam(n) = nameofnam(second(schhd s)) then 1
    else 1 + posinscheme(sch([],schtl s),n)
(* ldn 13.2.91 *)
fun isinscheme(s:scheme,n:name) = if schnull s then false else
  if nameofnam(n) = nameofnam(second(schhd s)) then true
    else isinscheme(sch([],schtl s),n)
(* ldn 13.2.91 *)
fun domofnaminscheme(sch(_,[]),n) = raise schemeexception

```

```

    | domofnaminscheme(sch(_, (d1, n1)::st), n) =
      if nameq(n1, n) then d1 else domofnaminscheme(sch([], st), n)
(* ldn 13.2.91 *)
fun schproj(sch(_, []), h::t) = raise schemeexception
  | schproj(s, []) = sch([], [])
  | schproj(s, h::t) = if isinscheme(s, h) then
    schappend(sch([h], [(domofnaminscheme(s, h), h)]),
      schproj(s, t)) else raise schemeexception
(* cut out by ldn 13.2.91
fun schproj(sch(_, []), h::t) = raise schemeexception
  | schproj(s, []) = sch([], [])
  | schproj(sch(_, (d2, n2)::st2), h::t) = if nameq(n2, h) then
    schappend(sch([n2], [(d2, n2)]), schproj(sch([], st2), t)) else
    schproj(sch([], st2), t)
*)
fun scheme2string(sch(_, [])) = ""
  | scheme2string(sch (l1, (d, n)::t)) = (nameofdom(d)^" "^nameofnam(n)^
    " "^scheme2string(sch([], t))) (* ^"\n" *)
fun displayscheme s = print scheme2string(s)
end

abstype tuple = tup of (attribute list)
with
  exception tupexception1
  exception tupexception2
  fun maketup(al) = tup al
  fun tuphd (tup al) = hd al
  fun tuptl (tup al) = tl al
  fun tupeq(tup([], tup(h::t)))=false
    | tupeq(tup(h::t), tup([]))=false
    | tupeq(tup([], tup([])))=true
    | tupeq(tup(h1::t1), tup(h2::t2))=atteq(h1, h2) andalso tupeq(tup(t1), tup(
  fun tuplt(tup([], tup(h::t)))=false
    | tuplt(tup(h::t), tup([]))=false
    | tuplt(tup([], tup([])))=false
    | tuplt(tup(h1::t1), tup(h2::t2))
      = if attlt(h1, h2) then true

```

```

        else if atteq(h1,h2) then tuplt(tup(t1),tup(t2))
        else false
fun tupgt(tup [], tup _) = false
|  tupgt(tup(h::t), tup []) = true
|  tupgt(tup(h1::t1),tup(h2::t2)) = if attgt(h1,h2) then true
    else tupgt(tup(t1),tup(t2))
fun tupappend(tup l1, tup l2) = tup (l1@l2)
fun tuplength (tup al) = length al
fun tupnull(tup al) = al = nil
fun match(t:tuple ,s:scheme) = (tupnull(t) andalso schnull(s)) orelse
    if not(tupnull(t) orelse schnull(s)) then
        (type2string(tuphd t)=nameofdom(first(schhd s)))
        andalso match(tup(tuptl t),makesch([],schtl s))
    else false
fun tupnth(t:tuple,n:int) = if n<=0 then raise tupexception1
    else if tupnull t then raise tupexception1
        else if n=1 then tuphd(t) else tupnth(tup(tuptl t),n-1)
fun dot(t:tuple, s:scheme, n:name) = if not( match(t,s)) then
    raise tupexception2 else tupnth(t,posinscheme(s,n))
fun tupleproj(t,s,hnl::tnl)=tupappend(tup([dot(t,s,hnl)]),tupleproj(t,s,tn
| tupleproj(t,s,[])=tup([])
(* fun tupleproj(t,s,nl)=tupappend(tup([dot(t,s,hd(nl)])),tupleproj(t,s,tl(
| tupleproj(t,s,[])=tup([]) *)
fun tup2string(tup(h::t))=att2string(h)^" "^tup2string(tup t)(*"\n" *)
| tup2string(tup [])="\n"
end

datatype comparator = gt | ge | eq | le | lt | ne

abstype tupset = set of (tuple list)
with
    val emptyset = set([])
    fun maketupset(tl) = set(tl)
    fun thd(set(h::t)) = h
    fun ttl(set(h::t)) = t
    fun set2list(set(tl)) = tl
fun      taddattr(_,set([])) = emptyset

```

```

| taddattr(a,set(h::t)) = maketupset(tupappend(h,maketup([a]))::set2list
fun tsum(i,set([])) = 0
| tsum(i,set(h::t)) = getival(tupnth(h,i)) + tsum(i,set(t))

fun member(t:tuple,nil:tuple list) = false
| member(t,h::l)= if tupeq(t,h) then true else member(t,l)
(* fun tpartition(r1,set([]),_) = [makerel(schemeof(r1),emptyset)]
| tpartition(r1,set(h::t),n) =
    projsel(r1,n,h) ::(tpartition(r1,set(t),n)) *)
fun is_empty(set(s)) = length(s) = 0
(* insert to be used when there is no possibility of duplicates *)
fun fastinsert (t, set(l)) = set(t::l)
(* insert which guards against tuple duplication *)
fun safeinsert (t, set(l)) = if member(t,l) then set(l)
                             else set(t::l)

(* In an efficient implementation it is likely to be faster to
use the fastinsert for all insertions and eliminate duplicates by
sorting the list with a quicksort then looking for repeated adjacent
values in a final pass. This is Onlogn rather than On^2 *)
fun tunion(set([]), set(l)) = set l
| tunion(set(h::t),set(l)) = tunion(set(t),safeinsert(h,set(l)))
fun tintersect(set([]),set(l)) = set([])
| tintersect(set(h::t),set(l)) = if member(h,l) then fastinsert(h,
    tintersect(set(t),set(l))) else
    tintersect(set(t),set(l))
fun tdifference(set([]),set(l)) = set([])
| tdifference(set(h::t),set(l)) = if member(h,l) then tdifference(set(t)
    set(l)) else
    fastinsert(h,tdifference(set(t),set(l)))
fun tupleprod(t:tuple,set([])) = set([])
| tupleprod(t,set(h::l)) = fastinsert(tupappend(t,h), tupleprod(t,set(l))
fun tcartprod(set([]),s) = set([])
| tcartprod(set(h::t),s) = tunion(tupleprod(h,s), tcartprod(set(t),s))
fun tselect(set([]), cond:tuple -> bool) = set([])
| tselect(set(h::t),cond) = if cond(h) then fastinsert(h,tselect(set(t),
    cond))
    else tselect(set(t), cond)

```

```

(* PRECI-style select *)
fun tpre.sel(set ([]),s:scheme,n:name,c:comparator,a:attribute) = set ([])
|   tpre.sel(set (h::t),s,n,gt,a) = if attgt(dot(h,s,n),a) then
      fastinsert(h,tpre.sel(set (t),s,n,gt,a))
    else tpre.sel(set (t),s,n,gt,a)
|   tpre.sel(set (h::t),s,n,ge,a) = if attge(dot(h,s,n),a) then
      fastinsert(h,tpre.sel(set (t),s,n,ge,a))
    else tpre.sel(set (t),s,n,ge,a)
|   tpre.sel(set (h::t),s,n,eq,a) = if atteq(dot(h,s,n),a) then
      fastinsert(h,tpre.sel(set (t),s,n,eq,a))
    else tpre.sel(set (t),s,n,eq,a)
|   tpre.sel(set (h::t),s,n,le,a) = if attle(dot(h,s,n),a) then
      fastinsert(h,tpre.sel(set (t),s,n,le,a))
    else tpre.sel(set (t),s,n,le,a)
|   tpre.sel(set (h::t),s,n,lt,a) = if attlt(dot(h,s,n),a) then
      fastinsert(h,tpre.sel(set (t),s,n,lt,a))
    else tpre.sel(set (t),s,n,lt,a)
|   tpre.sel(set (h::t),s,n,ne,a) = if attne(dot(h,s,n),a) then
      fastinsert(h,tpre.sel(set (t),s,n,ne,a))
    else tpre.sel(set (t),s,n,ne,a)

fun tproject(set ([]),s:scheme, nl:name list) = set ([])
|   tproject(set (h::t),s, nl) = safeinsert(tupleproj(h,s,nl),
      tproject(set (t),s,nl))
fun tcard(set (ts))=length(ts)
fun set2string(set (h::t))=tup2string(h)^set2string(set t)(* ^"\n" *)
|   set2string(set [])=""
fun textend_by(set ([]),_)=set ([])
|   textend_by(set (h::t),m:tuple->attribute)=
      tunion(set ([tupappend(h,maketup([m(h)])])),textend_by(set (t),m))
fun tprojsel(set ([]),_,_,_) = set ([])
|   tprojsel(set (h::t),nl,tu,s) =
      if tupeq(tupleproj(h,s,nl),tu) then
        fastinsert(h,tprojsel(set (t),nl,tu,s))
      else
        tprojsel(set (t),nl,tu,s)
fun atKey(t,set (nil),s) = maketup(nil)

```

```

| atKey(t,set(h::t1),s) =
    let val tag = tupleproj(h,s,keyofscheme(s)) in
      if tupeq(h,t) then
        h
      else
        atKey(t,set(t1),s)
    end

fun tjoin(set(nil),s1,ts2,s2,n) = set(nil)
| tjoin(set(h::t),s1,ts2,s2,n) =
    let val firstPart = tupleproj(h,s1,[n]) in
      let val partner = atKey(firstPart,ts2,s2) in
        if tupnull(partner) then
          tjoin(set(t),s1,ts2,s2,n)
        else
          fastinsert(tupappend(h,partner),
                    tjoin(set(t),s1,ts2,s2,n))
        end
      end
    end

end

abstype relation = rel of (scheme*tupset)
with
  exception relexception1
  exception relexception2
  exception relexception3
  exception relexception4
  exception duplicate_keys_rel
(* fun partition(r1,rel(s,ts)) = tpartition(r1,ts,namesinscheme(s)) *)
fun setof(rel(s,ts)) = ts
fun schemeof(rel(s,ts)) = s
fun namelistof(rel(s,ts)) = namesinscheme(s)
fun makerel(s,t)=rel(s,t)
fun validrel(rel(s,t)) = validscheme(s) andalso
  if not(is_empty(t)) then match(hd(set2list(t)),s) else true
(* this validation is simplistic. Could be improved on *)
fun insert(t,rel(s,ts))=if not (match(t,s)) then raise relexception1

```

```

    else if member(tupleproj(t,s,keyofscheme(s)),
        set2list(tproject(ts,s,keyofscheme(s))))
        then raise duplicate_keys_rel else rel(s,fastinsert(t,ts))
fun union(rel(s1,ts1),rel(s2,ts2))= if not(equiv(s1,s2)) then
    raise relexception2
    else rel(s1,tunion(ts1,ts2))
fun intersect(rel(s1,ts1),rel(s2,ts2))= if not(equiv(s1,s2)) then
    raise relexception3
    else rel(s1,tintersect(ts1,ts2))
fun difference(rel(s1,ts1),rel(s2,ts2))= if not(equiv(s1,s2)) then
    raise relexception4
    else rel(s1,tdifference(ts1,ts2))
fun select(rel(s,ts),cond)=rel(s,tselect(ts,cond))
(* PRECI-style select *)
fun presel(rel(s,ts),n:name,c:comparator,a:attribute) =
    rel(s,tpresel(ts,s,n,c,a))
fun project(rel(s,ts),nl)=rel(schproj(s,nl),tproject(ts,s,nl))
fun cartprod(rel(s1,ts1),rel(s2,ts2))=rel(schappend(s1,s2),
    tcartprod(ts1,ts2))
fun cardinality(rel(s,ts))=tcard(ts)
fun degree(rel(s,ts))=schlength(s)
fun rel2string(rel(s,t))=scheme2string(s)^^"\n"^set2string(t)^^"\n"
fun projsel(rel(s,ts),nl,t) = rel(s,tprojsel(ts,nl,t,s))
fun sum(n,rel(s,ts)) = tsum(posinscheme(s,n),ts)
fun attrsum(n,r) = makeival(sum(n,r))
fun equijoin(rel(s1,ts1),rel(s2,ts2),n) =
    rel(schappend(s1,s2),tjoin(ts1,s1,ts2,s2,n))
end

val suppsch = makesch([makenam("snum")],[(makedom("string"),makenam("snum"))
(makedom("string"),makenam("sname")),
(makedom("int"),makenam("status")),
(makedom("string"),makenam("city"))])

val s1 = maketup([makecval("s1"),makecval("smith"),makeival(20),
    makecval("london")])
val s2 = maketup([makecval("s2"),makecval("jones"),makeival(10),makecval("pa

```

```
val s3 = maketup([makecval("s3"),makecval("blake"),makeival(30),makecval("pa
val s4 = maketup([makecval("s4"),makecval("clark"),makeival(20),makecval("lo
val s5 = maketup([makecval("s5"),makecval("adams"),makeival(30),makecval("at
val s6 = maketup([makecval("s6"),makecval("andy"),makeival(10),makecval("rom
val q1 = maketup([makeival(30)])

val sts= maketupset([s1,s2,s3,s4,s5])

val supprel = makerel(suppsch,sts)

val partssch = makesch([makenam("pnum")],[(makedom("string"),makenam("pnum")
      (makedom("string"),makenam("pname")),
      (makedom("string"),makenam("colour")),
      (makedom("int"),makenam("weight")),
      (makedom("string"),makenam("city"))])

val p1 = maketup([makecval("p1"),makecval("nut"),makecval("red"),
      makeival(12),makecval("london")])
val p2 = maketup([makecval("p2"),makecval("bolt"),makecval("green"),
      makeival(17),makecval("paris")])
val p3 = maketup([makecval("p3"),makecval("screw"),makecval("blue"),
      makeival(17),makecval("rome")])

val p4 = maketup([makecval("p4"),makecval("screw"),makecval("red"),
      makeival(14),makecval("london")])

val p5 = maketup([makecval("p5"),makecval("cam"),makecval("blue"),
      makeival(12),makecval("paris")])

val p6 = maketup([makecval("p6"),makecval("cog"),makecval("red"),
      makeival(19),makecval("london")])
val pts = maketupset([p1,p2,p3,p4,p5,p6])

val partsrel = makerel(partssch,pts)

val shipsch = makesch([makenam("snum"),makenam("pnum"),makenam("qty")],
```



```
[(makedom("string"), makenam("snum")),
 (makedom("string"), makenam("pnum")),
 (makedom("int"), makenam("qty")) ]

val sh1 = maketup([makecval("s1"), makecval("p1"), makeival(300)])
val sh2 = maketup([makecval("s1"), makecval("p2"), makeival(200)])
val sh3 = maketup([makecval("s1"), makecval("p3"), makeival(400)])
val sh4 = maketup([makecval("s1"), makecval("p4"), makeival(200)])
val sh5 = maketup([makecval("s1"), makecval("p5"), makeival(100)])
val sh6 = maketup([makecval("s1"), makecval("p6"), makeival(100)])
val sh7 = maketup([makecval("s2"), makecval("p1"), makeival(300)])
val sh8 = maketup([makecval("s2"), makecval("p2"), makeival(400)])
val sh9 = maketup([makecval("s3"), makecval("p2"), makeival(200)])
val sh10 = maketup([makecval("s4"), makecval("p2"), makeival(200)])
val sh11 = maketup([makecval("s4"), makecval("p4"), makeival(300)])
val sh12 = maketup([makecval("s4"), makecval("p5"), makeival(400)])

val shipts = maketupset([sh1, sh2, sh3, sh4, sh5, sh6, sh7, sh8, sh9, sh10, sh11, sh12])

val shiprel = makerel(shipsch, shipts)
```

Appendix C

Implementing Interpreters

C.1 Introduction

This appendix covers the principles involved in constructing the kind of interpreter outlined by the main thesis. Since there are no compiler tools (such as *Lex* [72] and *Yacc* [63] of the Unix system) available for the Lingo system, the interpreter was constructed from scratch. The following two sections illustrate the construction strategy by applying it to the implementation of a simple interpreter for an SQL-like language. Although the implementation language is Lingo, the techniques employed transfer readily to Smalltalk.

The last section of this appendix deals with the construction of a parser generator that was built to afford a similar functionality to *Yacc*. The Lingo

version of the parser generator was an essential tool in controlling the development of the DEAL interpreter since it allowed a separation of concerns (parsing as against semantic considerations) to minimise the complexity of the task.

C.2 An example

The following subsections will make use of a simple SQL-like language. The general form of an SQL query is

```
SELECT fields FROM tables WHERE predicate
```

in which the WHERE qualifying clause is optional.

It is assumed that a number of tables (relations) are known to the system and have names whose lexical formation is governed by the conventional rules for forming identifiers in a language such as Pascal. A *field* follows the same naming rules and refers to a column of a table. For example (taken from Date [24]), a relation named 'supplier', with fields 'snum' (supplier number), 'sname' (supplier name), 'status' (status value) and 'city' (location) is tabulated as

snum	sname	status	city
s1	smith	20	london
s2	jones	10	paris
s3	blake	30	paris
s4	clark	20	london
s5	adams	30	athens

Given this, a query that retrieves all supplier names and their locations is

```
SELECT sname, city FROM supplier
```

The resulting table is

sname	city
smith	london
jones	paris
blake	paris
clark	london
adams	athens

A query involving a WHERE clause can be used to retrieve the names of all suppliers whose status is less than or equal to 20 -

```
SELECT sname FROM supplier WHERE status <= 20
```

with resulting table

sname
smith
jones
clark

The FROM clause may name more than one table. If, in addition to the table supplier, we have the table parts given as

pnum	pname	colour	weight	city
p1	nut	red	12	london
p2	bolt	green	17	paris
p3	screw	blue	17	rome
p4	screw	red	14	london
p5	cam	blue	12	paris
p6	cog	red	19	london

and the table shipments (connecting suppliers and parts) given by

snum	pnum	qty
s1	p1	300
s2	p2	200
s1	p3	400
s1	p4	200
s1	p5	100
s1	p6	100
s1	p1	300
s2	p2	400
s3	p2	200
s4	p2	200
s4	p4	300
s4	p5	400

we may now retrieve the names of all suppliers who ship screws

```
SELECT sname FROM supplier, parts, shipments WHERE pname = "screw"
```

Although SQL is based on the relational *calculus*, the inclusion of set operations allows its use as a convenient syntactic interface to relational algebra. Consider the generalised SQL query –

```
SELECT  A1, ..., An
FROM    R1, ..., Rm
WHERE   Bθb
```

An equivalent in the relational algebra (Ullman, [110]) is the projection of attributes from the selection of tuples from a cartesian product (in practice, the cartesian product would be replaced by an appropriate join) –

$$\pi_{A_1, \dots, A_n}(\sigma_{B\theta b} R_1 \times \dots \times R_m)$$

This is excessively dense and opaque. An equivalent in Lingo is unthinkably large and unwieldy. As a taste, the first query above –

```
SELECT sname, city FROM suppliers
```

could be expressed in Lingo (given the appropriate class definitions) as

```
(tableDictionary at: "suppliers") project: ["sname" "city"]
```

The following section describes how to effect this transformation.

C.3 Translation

C.3.1 Grammars

The informal description of SQL syntax given in the preceding section is insufficiently detailed to form the basis for a parser. The BNF notation, due to Backus [5] and Naur [82], is usually used for this purpose –

```

<query>          ::= <selectF> | <selectFW>
<selectF>       ::= "SELECT" <fieldList> "FROM" <fromList>
<selectFW>      ::= "SELECT" <fieldList> "FROM" <fromList>
                  "WHERE" <predicateTerm>
<fieldList>     ::= <fieldName> | <fieldList> "," <fieldName>
<fieldName>    ::= Identifier
<fromList>     ::= TableName | <fromList> "," TableName
<predicateTerm> ::= <expression> <comparison> <expression>
<comparison>   ::= "=" | "<" | ">" | "<=" | ">=" | "<>"
<expression>   ::= <fieldName> | <constant>
<constant>    ::= String | Integer

```

Here, non-terminals such as <query> and <fieldList> (entities defined by appearing on the left hand side of some rule in the BNF description) are denoted by enclosure within angle brackets < and >. An entity that is quoted, such as "FROM", indicates that it is terminal and its component characters must appear exactly in the input stream.

Entities such as TableName example have no definition. We will consider these as terminal *classes* denoting entities whose syntactic structure is conventional and simple and analysed by a translator phase other than parsing.

In the case of `Integer`, for example, this denotes the class of integers whose members are easily recognised at the *lexical* rather than *syntactical* level. Similarly for the classes `Identifier` and `String`. `TableName` refers to the subset of the class `Identifier` that names relations known to the system (for example, `supplier`).

C.3.2 Lexical Analysis

The process of recognising a language's constructs from the arrangement of individual characters in an input stream is conventionally split into two phases

–

- **Lexical Analysis** concerns itself with the recognition of groups of characters (such as keywords of the language, identifiers, numbers and so on). Recognised groups are associated with *tokens* with which the lexical analyser (or scanner) communicates its analysis to other phases.
- **Syntax Analysis** is concerned with the recognition of structured patterns of tokens (such as language statements, expressions and so on). The syntax analyser (or parser) usually communicates its analysis to other phases by associating tree structures with a language construct. These structures may either be explicit data structures, or may be im-

PLICITLY constructed through program execution and the state of the procedure stack.

This section deals with the lexical analysis phase. The coding of this phase is tedious and error-prone as it deals with the input-output section of the interpreter. Lexical analyser generators are widely available, perhaps the best known being Lex [72].

Such tools have a power beyond simply providing lexical analysers within compilers. The author takes the view that the lexical analysis phase of an interpreter for a programming language warrants a rather simpler approach. The objective is to define a general class, Scanner say, which can be instantiated to provide a lexical analyser for any given language.

We observe that the microsyntax (with which lexical analysis concerns itself) of most languages reduces to just a few terminal classes. Most languages use the same rules for describing the syntax of integers and identifiers. Additionally most languages have the same rules for dealing with white space. Keywords usually form a subset of the terminal strings that could otherwise be considered to be identifiers. Keywords, identifiers and integers are separated by white space, punctuation characters or operator terminals (such as '<=').

Punctuation characters are those characters that cannot prefix other terminal strings. For example, '(' is usually a punctuation character, whereas '<' is not since the character may be the prefix of the terminal string '<='. Some characters (such as '<') have different lexical significance depending whether they appear on their own or grouped with other such characters. We call such characters **cryptic** characters, following the terminology used in the lexical analyser used for the object-oriented language Lingo [53].

Given a set of keywords, a set of punctuation characters and a set of cryptic characters, an algorithm to recognise terminal classes is straightforward to code. The preliminary decision on which terminal class is being recognised is based on the first non-white character in the source (that is the first character that is not a space, a tab or a newline).

- a decimal digit – an integer is being recognised; accumulate all following decimal digits and return the token 'Integer' (a string literal).
- a punctuation character – return a string containing just the punctuation character itself as the token and advance the input stream to the next character.
- a cryptic character – accumulate the character and any following cryptic characters into a string which is returned as the token.

- an alphabetic character – accumulate the character and any following numeric or alphabetic characters into a string. Then search the keywords vector, if a match is found return the string as the token; otherwise return the string ‘Identifier’

A class, `Scanner`, has been programmed in Lingo along the above lines. It has a class method for instantiation which takes as parameters collections of keywords, punctuation characters and cryptic characters and also a file descriptor for the source text (a stream or file). `Scanner`’s instance methods include `getToken` which returns the current token and advances through the source stream. There are also methods `theNumber` and `theIdentifier` which return the actual values found in the source for the tokens ‘Integer’ and ‘Identifier’ respectively.

Interestingly, some of the context sensitive aspects of a language are very easily handled by this approach using inheritance. Consider the Pascal expression –

`x + y ;`

With an instance of `Scanner` as above, `x` and `y` will be recognised as Identifiers although for semantic analysis purposes it may be more useful to recognise them as ‘Variables’ or ‘Functions’. As in either case they should have been previously defined and thus present in a symbol table it is possible

to determine their particular significance in the lexical analysis phase. To do this, a class, `PascalScanner` say, is defined as a specialisation of `Scanner`. It has an additional instance variable to hold a symbol table and access methods to place and look up entities in this table. The inherited `getToken` method is overridden in `PascalScanner`— the specialisation calls the superclass's `getToken` and then inspects the returned token. If it is 'Identifier' the actual terminal string is searched for in the symbol table. If a match is found, the appropriate token is returned ; if it is not matched, 'Identifier' is returned.

C.3.3 The syntax analyser

With the strategy adopted here, the syntax analyser is the driving spirit of the interpreter: no *explicit* parse tree is constructed to inform later analytic and synthetic phases. Instead, an *implicit* parse tree is contained within the thread of execution of the syntax analyser and so, in order to traverse the parse tree, the syntax analyser itself invokes further analytic and synthetic procedures.

The purpose, then, of the syntax analyser is to recognise the language's syntactic structures and then invoke appropriate semantic routines to effect the intent of the original source program.

The starting point for writing a syntax analyser is a description of the

grammar of the language in the form of a BNF specification.

The recursive descent method [25] allows a syntax analyser to be written almost directly from a BNF description. Each nonterminal in the grammar is represented by a method with the responsibility of recognising its own nonterminal's syntax. In addition there is a method, (**mustBe:**, say) that takes a token representing a terminal, and checks that the token it is passed is the same as that currently held by the lexical analyser. In Lingo, we can arrange all these methods (those representing nonterminals and **mustBe:**) as instance methods of a class, **Parser** say. An instance variable, **scanner**, holds the lexical analyser. Another instance variable, **token**, holds the last token returned by the lexical analyser. The method **mustBe:** is simply –

```
mustBe: aToken [ ]
{
  if (token = aToken) then
    { token := scanner getToken}
  else
    { "syntax error\n" printedOn: FileDescriptor output}
}
```

The methods for nonterminals are written by examining the right hand sides of their defining rules within the BNF. If there are no alternatives in the rule, that is the right hand side is merely a sequence, the method is coded as a sequence of calls: in the case of a nonterminal, a call to its associated method, in the case of a terminal, a call to the method **mustBe:** using the

token representing the terminal as an actual parameter.

For example, a BNF rule such as –

```
<selectF> ::= "SELECT" <fieldList> "FROM" <fromList>
```

would be coded as

```
self selectF []
{
  self mustBe: "SELECT";
  self fieldList;
  self mustBe: "FROM";
  self fromList;
}
```

Where a right hand side contains alternatives, each alternative is inspected to determine the set of terminals that can appear at its start. These sets are termed **director sets** since they are used to direct the parse. If these sets are not disjoint, the method will not be successful and the redefinition of the language should be attempted. If the director sets are disjoint, they can be used to decide which alternative rule should be followed by finding which of the sets the current token is a member of. Consider the rules

```
<expression> ::= <fieldName> | <constant>
<fieldName>  ::= Identifier
<constant>   ::= String | Integer
```


The rule for `expression` contains two alternatives. The director set for the first alternative contains only 'Identifier'. The second alternative's director set is { 'String', 'Integer' }. The method for `expression` is coded as

```
self expression []
{
  if (["Identifier"] includes: token) then
    { self fieldName; }
  else
    { self constant;}
}
```

Unfortunately, the cases where director sets are not disjoint are sufficiently common that consideration must be given to grammar manipulation. Frequently the problem arises since the natural way to express a sequence in BNF is to use recursion. Consider, for example the production

```
<fromList> ::= TableName | <fromList> "," TableName
```

The intention is to express that a `fromList` is a sequence of `TableNames` separated by commas. Variations of BNF (which we shall call Extended BNF or EBNF) allow iteration to be expressed. We shall use the metasymbol pairs '[' ']' and '{' '}' to indicate zero or one and zero or more (respectively) repetitions of the BNF fragments they enclose. This allows, for example, the above production to be rephrased as

```
<fromList> ::= TableName { "," TableName }
```

Rules containing the iteration metasympols { and } are coded by deriving the director set for the enclosed sequence. The iteration condition is then that the current token is in the director set. The method for fromList is

```
self fromList []
{
  self mustBe: 'TableName'.
  while ( [ "," ] includes: token)
  {
    self mustBe: ",";
    self mustBe: "TableName";
  }
}
```

The metasympols [and] are treated in a similar way, using ifTrue: rather than whileTrue:.

Another often occurring situation is that a BNF rule expresses that a sentence has two variants each of which starts with the same structure, but then finishes differently. For example we have –

```
<query>          ::= <selectF> | <selectFW>
<selectF>        ::= "SELECT" <fieldList> "FROM" <fromList>
<selectFW>       ::= "SELECT" <fieldList> "FROM" <fromList> "WHERE" <predicat
```

From this, a query always starts with a selectF, but may optionally have a WHERE clause. Just as with BNF, EBNF can be used to factor out the commonality and remove the disjunction in the first production (whose disjuncts have coincident director sets). i.e.

```
<query>          ::= "SELECT" <fieldList> "FROM" <fromList> [ <whereClause> ]  
<whereClause>   ::= "WHERE" <predicateTerm>
```

The reader is referred to Milne [76] for a fuller account of these issues.

An EBNF description of our sample language is

```
<query>          ::= "SELECT" <fieldList> "FROM" <fromList> [ <whereClause> ]  
<whereClause>   ::= "WHERE" <predicateTerm>  
<fieldList>     ::= <fieldName> { "," <fieldName> }  
<fieldName>    ::= Identifier  
<fromList>     ::= TableName { "," TableName }  
<predicateTerm> ::= <expression> <comparison> <expression>  
<comparison>   ::= "=" | "<" | ">" | "<=" | ">=" | "<>"  
<expression>   ::= <fieldName> | <constant>  
<constant>     ::= String | Integer
```

An issue that must be addressed is **error recovery**. The predictive nature of the recursive descent method means that when a syntax error does occur, the syntax analyser loses synchronisation with the source text being parsed and many consequential syntax errors are reported. It is not possible to recover by merely scanning till a statement terminator is found, since at the time of the error, the thread of execution will in general be at some deeply nested point due to the dependence on recursion. [25] gives an elegant algorithm for error recovery in such a situation, which merely adds a few lines to the `mustBe:` method. (Note that Lingo [53] provides an exception facility. Raising an exception strips back the procedure stack to its state at the moment of declaration of the exception. This mechanism can be used

to allow return to the top level of syntax analysis on encountering the first syntax error and was used for the DEAL implementation).

C.3.4 Adding semantic and interpretive actions

Now that a correct program can be recognised by the syntax analysis phase, we wish to invest meaning into its statements. This is the most imaginative part of the process of creating an interpreter. It is approached by associating actions with fragments of the BNF for the language. These actions are then effected by inserting lines of code within the parsing methods at the points indicated by their association with the BNF (bear in mind that the recursive descent method gives a one to one correspondence with the code of the parsing methods).

Consider the rule –

```
<query> ::= "SELECT" <fieldList> "FROM" <fromList> [ <whereClause> ]
```

We adopt the strategy that the ‘meaning’ of a query is to display its resulting relation. We can view the BNF as a framework on which to hang a prescription of how to determine a query’s meaning from its components, that is how to construct its meaning from the meanings of its components.

To do this, we associate **actions** $A_1 \dots A_3$ with points in the B.N.F. –

`<query> ::= "SELECT" <fieldList> A1"FROM" <fromList> A2 [<whereClause>]`

The informal description of these actions is

- A₁ – store the fieldList’s result (a list of fields to be projected from the relation resulting from the rest of the expression).
- A₂ – store the fromList’s result (a relation – the base that the rest of this expression is modifying in some way) as the result for query so far.
- A₃ – use the selection criteria returned by whereClause on the result of the query (which was stored in A₂). The resulting relation is stored as the result for query. We can arrange that the meaning of whereClause is a Lingo Module (Lingo’s counterpart to Smalltalk’s **BlockContext**’s – these are anonymous pieces of code, which can take parameters, and are similar to lambda expressions of the lambda calculus.). The returned module can be exactly that code which when passed to the **select:** method of **Relation** objects performs the selection.
- A₄ – perform the projection of the fields specified during A₁ and display the resulting display relation.

This can be coded in Lingo as

```
self query [ projectList result block ]
```

```

{
  self mustBe: "SELECT";
  projectList := self fieldList;      /* action A1 */
  self mustBe: "FROM";
  result := self fromList;           /* action A2 */
  if (["WHERE"] includes: token) then
  {
    block := self whereClause;       /* { action A3 */
    result := result select: block; /* {          */
  }
  (result project: projectList)      /* action A4 */
  printedOn: FileDescriptor output;
}

```

C.4 An interpreter generator

The strategy outlined in the preceding section can be turned on itself. Consider the extended BNF-

```

<grammar>      ::= <rule> { <rule> }
<rule>        ::= <nonterminal> " ::= " <ruleexp> ";"
<nonterminal> ::= "<" Identifier ">"
<ruleexp>     ::= <ruleterm> { "|" <ruleexp> }
<ruleterm>    ::= <rulefactor> { <rulefactor> }
<rulefactor>  ::= "[" <ruleexp> "]" | "{" <ruleexp> "}"
               | "(" <ruleexp> ")" | <nonterminal>
               | QuotedStringLiteral | Identifier

```

This describes the grammar of the extended BNF *itself* that has been used in this report (except that productions are terminated with a semi-colon), and yet is shorter than the BNF description of the example language

pursued in this paper. A parser for extended BNF can thus be written (using the strategies of the preceding section). In order for this EBNF-parser to be able to generate a syntax analyser for a presented language, it is only necessary to include within it interpretive actions that

- build and fill data structures capturing the essential information of the presented grammar.
- use these data structures to determine the director sets for all the non-terminals of the presented grammar
- create a class definition containing recogniser methods which make use of the determined director sets.

In detail, the above is effected in the following way: an abstract syntax tree is created (by the EBNF-parser) for the right hand side of every production encountered in the EBNF source. A symbol table associates, for each production, the non-terminal's name and the tree representing the production's right hand side. In addition, each symbol table entry has a field which can contain one of three values (`notStarted`, `inProgress` and `complete`). This field is used to mark the progress of director set computation (which is described more fully below) and is initially set to `notStarted`.

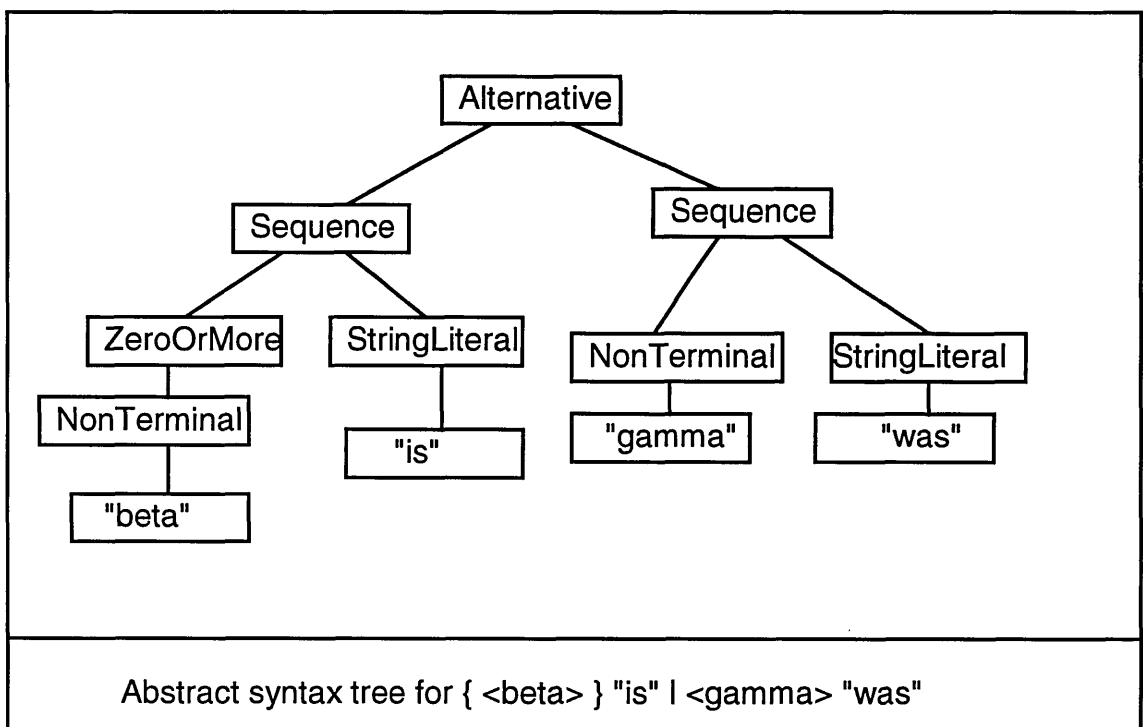
The abstract syntax trees make use of eight kinds of node, one for each type of unitary term within EBNF.

1. **NonTerminal**– these nodes contain the name of a non-terminal.
2. **StringLiteral**– these contain the character strings recognised by the EBNF-parser as `QuotedStringLiterals`.
3. **Identifier**– these correspond to the Identifier entities of the EBNF-parser and merely contain the character strings that were recognised.
4. **Alternative**– these nodes correspond to alternatives within the EBNF. They contain pointers to the two alternatives.
5. **Sequence**– these nodes correspond to a sequence of terms within the EBNF. They contain pointers to the lead term and the following terms.
6. **ZeroOrMore**– These nodes correspond to terms which are specified within the ‘zero or more’ iteration metasymbols { and }. The nodes contain a pointer to the iterated expression.
7. **ZeroOrOnce**– similarly, these nodes represent optional EBNF expressions (enclosed by the metasymbols [and]). The nodes contain a pointer to the optional expression.

8. **Once**— these nodes represent expressions that are enclosed within the metasymbols (and). Again, they contain a pointer to the parenthesised expression.

The following diagram represents the abstract syntax tree that would be created for the right hand side of the production:

`<alpha> ::= { <beta> } "is" | <gamma> "was"`



Once the symbol table has been built, it is traversed (linearly). As each entry is traversed, output is generated (the code of the computed parser). First, the procedure header for the non-terminal's recogniser procedure is output. For the example production above, for example, the following Lingo code would be generated:

```
self []  
{
```

The associated abstract syntax tree is then traversed (in post-order where the nodes are not singly-branched). The output generated depends on the type of node encountered. For **NonTerminal** nodes, a call to the corresponding recogniser procedure is generated.

In concrete terms, if *x* is a Lingo variable containing the **NonTerminal** node, *contents* is a **NonTerminal** method returning the string contained in a **NonTerminal** node and *print* is a Lingo output method, the following Lingo fragment is the action performed on encountering a **NonTerminal** node:

```
"self " print;  
(x contents) print;  
";\n" print;
```

For **StringLiteral** and **Identifier** nodes, appropriate calls to *mustBe* are output. Concretely (again assuming the variable *x* contains the node in question), the Lingo for the action is:

```
"self mustBe: " print;  
(x contents) print;  
";\n" print;
```

Sequence nodes are treated by generating code for their lead term and then their following terms.

```
(x lead) generate;  
(x following) generate;
```

For **Once** nodes, the algorithm is straightforward: output a left brace '{' (which is the Lingo token for starting a code block), generate the output for the expression the node points contents point to (by recursively calling the generate method) and finally output a right brace '}' (which is the Lingo token for ending a code block).

```
"{\n" print;  
(x contents) generate;  
"}\n" print;
```

The treatment of the remaining node types **Alternative**, **ZeroOrOnce** and **ZeroOrMore** makes use of director sets. These are computed via a method `getStarters:` which takes as a parameter a pointer to the expression whose director set is to be computed. Where this parameter is a non-terminal, it may be that this computation is merely a retrieval since the director set has already been computed. (The algorithm for `getStarters:`

is detailed more fully below since it is crucial to the parser generating strategy). Returning to the three node types in question, they are dealt with as follows:

```
Alternative  "if ((Vector [ " print;
              (self getStarters: (x left)) print;
              "]) includes: (scanner token)) then\n{" print;
              x left generate;
              "}\nelse\n{\n" print;
              x right generate;
              "}\n" print;

ZeroOrOnce  "if ((Vector [" print;
              (self getStarters: (x body)) print;
              "]) includes: (scanner token)) do\n{" print;
              x body generate;
              "}\n" print;

ZeroOrMore  "while ((Vector [" print;
              (self getStarters: (x body)) print;
              "]) includes: (scanner token)) do\n{" print;
              x body generate;
              "}\n" print;
```

Calculating director sets

The method `getStarters:` alluded to above is based on the following:

1. The director set for an expression that contains a sole **StringLiteral** or **Identifier** node is the singleton set containing the character string contents of the node.

2. In general, the director set for a **Sequence** node or a **Once** node is the director set of the first node in the sequence. However, since there is the possibility of null productions, the computation is more complex.

Consider the computation of the director set for the rule

{ alpha } beta gamma

Since an alpha term may not be present, the director set for the overall rule is computed as the *union* of the director sets for alpha and for beta. In addition, the parser generator checks that the two sets are disjoint and reports an error if they are not since this indicates that the grammar does not satisfy the LL(1) criterion.

3. For an **Alternative** node, the director set is computed from the union of the director sets of the node's component subexpressions (which are also checked for disjointness).
4. **ZeroOrOnce** and **ZeroOrMore** nodes have their director set computed from the director set of the expressions within their bodies. In addition, since both these node types indicate the presence of a null production within a rule, their director sets contain a special element `null` which indicates the presence of a null production to the algorithm (for use in step 2 above).

5. Finally, in the case of a `NonTerminal` node, the algorithm proceeds according to the setting of the state field (`notStarted`, `inProgress` or `complete`) within the non-terminal's symbol table entry.

If the state is `complete`, the director set has already been computed and is returned. If the state is `inProgress`, this indicates that the LL(1) criteria have not been met since left recursion (perhaps indirect) is present and an error report is generated.

If the state is `notStarted`, it is set to `inProgress`, and the tree representing the rule for the non-terminal is retrieved from the symbol table and presented to the algorithm. The resulting director set is stored in the symbol table (for later use) and also returned.

A slight refinement to the above allows the incorporation of semantic and interpretive actions, by introducing new metasympols `@` and `%`. These are used to indicate that the text they delimit (which should be Lingo fragments) is to be literally inserted into the generated parser at the indicated point. `@` delimits text to be inserted *prior* to the EBNF term that follows, `%` delimits text to be inserted *after* the recognition of the EBNF term it follows. In addition if `@` is used before the `::=` of a production the delimited text is inserted in the local variable declaration area of the recogniser method for

that nonterminal. As an example, the syntax and interpretive actions for query (as derived in the previous section) would be described by –

```
<query> @ projectList result block @
      ::= "SELECT" @ projectList := @ <fieldList>
         "FROM"   @ result      := @ <fromList>
         [ @ block := @ <whereClause>
           % result := result select: block; % ]
         % (result project: projectList)
           printedOn; FileDescriptor output; %
```

C.5 Summary

The recursive descent method of compiling has been shown to transfer naturally to implementation in Lingo and coding the lexical analysis phase is greatly simplified through inheritance.

The parsing method is transparent enough to allow programmers to easily include statements that carry out semantic actions - and the method is simple enough for programmers to carry out themselves.

The definition of an Extended B.N.F. in itself may take only a few lines. Paradoxically, the creation of a general **parser generator** which will generate a parser for a language from its presented EBNF is simpler than generating the parsers directly. A parser generator was constructed for Lingo.

Further work could be carried out to improve the interface to the parser

generator whose input files can quickly become unreadable since they carry so much information. In addition, the area of grammar manipulation (in order to achieve suitability for the recursive descent method) is important for a language of moderate syntactic complexity.

Appendix D

An SML specification based on

2–3 trees

```
abstype tree23 = E
  | Tr2 of tree23 * tuple * tuple * tree23
  | Tr3 of tree23 * tuple *tuple * tree23 * tuple * tuple * tree23
  | Put of tree23 * tuple * tuple * tree23
with
exception putException
exception atException

fun at (k :tuple , E:tree23) : tuple = raise atException
| at (_,Put(_,_,_,_)) = raise putException
| at (k, Tr2( t1, k1,v1, t2)) =
  if tupeq(k, k1) then
    v1
  else
    if tuplt(k , k1) then
      at(k,t1)
```

```
    else
      at(k,t2)
| at (k, Tr3(t1,k1,v1,t2,k2,v2,t3)) =
  if tupeq(k , k1) then v1
  else if tupeq(k,k2) then v2
  else if tuplt(k , k1) then at(k,t1)
  else if tuplt(k , k2) then at(k,t2)
  else at(k,t3)

fun at2 (k :tuple , E:tree23) : tuple = maketup(nil)
| at2 (_,Put(_,_,_,_)) = raise putException
| at2 (k, Tr2( t1, k1,v1, t2)) =
  if tupeq(k, k1) then
    v1
  else
    if tuplt(k , k1) then
      at2(k,t1)
    else
      at2(k,t2)
| at2 (k, Tr3(t1,k1,v1,t2,k2,v2,t3)) =
  if tupeq(k , k1) then v1
  else if tupeq(k,k2) then v2
  else if tuplt(k , k1) then at2(k,t1)
  else if tuplt(k , k2) then at2(k,t2)
  else at2(k,t3)

fun isMember (k:tuple , E:tree23) = false
| isMember (_,Put(_,_,_,_)) = raise putException
| isMember (k, Tr2( t1, k1,v1, t2)) =
  if tupeq(k, k1) then
    true
  else
    if tuplt(k , k1) then
      isMember(k,t1)
    else
      isMember(k,t2)
```

```

| isMember (k, Tr3(t1,k1,v1,t2,k2,v2,t3)) =
    if tupeq(k , k1) then true
    else if tupeq(k,k2) then true
    else if tuplt(k , k1) then isMember(k,t1)
    else if tuplt(k , k2) then isMember(k,t2)
    else isMember(k,t3)

fun put k v E = Put (E,k,v,E)
| put k v (Tr2(t1,k2,v2,t2))
    = if tupeq(k2 , k) then Tr2(t1,k,v,t2) else
    if tuplt(k, k2) then tr2(put k v t1, k2, v2, t2) else
    tr2(t1,k2, v2, put k v t2)
| put k v (Tr3(t1,k2,v2,t2,k3,v3,t3))
    = if tupeq(k,k2) then Tr3(t1,k2,v2,t2,k3,v3,t3) else
    if tupeq(k, k3) then Tr3(t1,k2,v2,t2,k3,v3,t3) else
    if tuplt(k, k2) then tr3(put k v t1,k2,v2,t2,k3,v3,t3) else
    if tuplt(k, k3) then tr3(t1,k2,v2,put k v t2,k3,v3,t3) else
    tr3(t1,k2,v2,t2,k3,v3,put k v t3)
| put k v y = raise putException
and
tr2(Put(t1,k1,v1,t2),k2,v2,t3) = Tr3(t1,k1,v1,t2,k2,v2,t3)
| tr2(t1,k1,v1,Put(t2,k2,v2,t3)) = Tr3(t1,k1,v1,t2,k2,v2,t3)
| tr2 other = Tr2 other
and
tr3(Put(t1,k1,v1,t2),k2,v2,t3,k3,v3,t4)
    = Put(Tr2(t1,k1,v1,t2),k2,v2,Tr2(t3,k3,v3,t4))
| tr3(t1,k1,v1,Put(t2,k2,v2,t3),k3,v3,t4)
    = Put(Tr2(t1,k1,v1,t2), k2,v2,Tr2(t3,k3,v3,t4))
| tr3(t1,k1,v1,t2,k2,v2,Put(t3,k3,v3,t4))
    = Put(Tr2(t1,k1,v1,t2), k2,v2,Tr2(t3,k3,v3,t4))
| tr3 other = Tr3 other;
fun checkTop(Put(t1,k,v,t2)) = Tr2(t1,k,v,t2)
| checkTop other = other
fun insert23( k,v, t) = checkTop (put k v t )
fun keyOf1(aTuple , s ) =
    tupleproj(aTuple,s,keyofscheme(s))

```

```

fun makeTree(nil,s : scheme) = E
  | makeTree(h::t,s) = insert23 (keyOf1(h,s),h,makeTree(t,s))

fun treeunion(Put(_,_,_,_),_) = raise putException |
  treeunion(E,x) = x |
  treeunion(x,E) = x |
  treeunion(Tr2(left1,key,value,right1),x) =
  let val belongs = isMember(key,x) in
if belongs then
  treeunion(left1,treeunion(right1,x))
else
  insert23(key,value,treeunion(left1,treeunion(right1,x)))
end
  |
  treeunion(Tr3(left,key1,value1,middle,key2,value2,right),x) =
  let val belongs1 = isMember(key1,x)
      and belongs2 = isMember(key2,x) in
if belongs1 andalso belongs2 then
  treeunion(left,treeunion(middle,treeunion(right,x)))
else
  if belongs2 andalso (not( belongs1)) then
    insert23(key1,value1,treeunion(left,
      treeunion(middle,treeunion(right,x))))
    else
  if belongs1 andalso (not( belongs2)) then
    insert23(key2,value2,treeunion(left,treeunion(middle,treeunion(right,x))
      else
    insert23(key1,value1,
      insert23(key2,value2,treeunion(left,
        treeunion(middle,treeunion(right,x))))))
end

fun ctreenjoin(Put(_,_,_,_),_,_,_,_,_)= raise putException |
  ctreenjoin(E,s1,ts2,s2,n2) = E |
  ctreenjoin(ts1,s1,E,s2,n2) = E |
  ctreenjoin(Tr2(left1,key,value,right1),s1,ts2,s2,n2) =

```

```

let val firstPart = tupleproj(value,s1,[n2]) in
  let val partner = at2(firstPart,ts2) in
    if tupnull(partner) then
      treeunion(ctreejoin(left1,s1,ts2,s2,n2),ctreejoin(right1,s1,ts2,s2,
    else
      insert23(
        tupappend(tupleproj(value,s1,keyofscheme(s1)),
          firstPart),
        tupappend(value,at(firstPart,ts2)),
        treeunion(ctreejoin(left1,s1,ts2,s2,n2),ctreejoin(right1,s1,ts2,s2
  end
end
|
ctreejoin(Tr3(left,key1,value1,middle,key2,value2,right),s1,ts2,s2,n2) =
let val firstPart1 = tupleproj(value1,s1,[n2])
  and firstPart2 = tupleproj(value2,s1,[n2]) in
  let val partner1 = at2(firstPart1,ts2)
    and partner2 = at2(firstPart2,ts2) in

    if tupnull(partner1) andalso tupnull(partner2) then
      treeunion(ctreejoin(left,s1,ts2,s2,n2),
        treeunion(ctreejoin(middle,s1,ts2,s2,n2),
          ctreejoin(right,s1,ts2,s2,n2)))
    else
      if tupnull(partner1) andalso not(tupnull(partner2)) then
        insert23(tupappend(tupleproj(value2,s1,keyofscheme(s1)),
          firstPart2),
          tupappend(value2,partner2),
          treeunion(ctreejoin(left,s1,ts2,s2,n2),
            treeunion(ctreejoin(middle,s1,ts2,s2,n2),
              ctreejoin(right,s1,ts2,s2,n2))))
      else
        if tupnull(partner2) andalso not(tupnull(partner1)) then
          insert23(tupappend(tupleproj(value1,s1,keyofscheme(s1)),
            firstPart1),
            tupappend(value1,partner1),
            treeunion(ctreejoin(left,s1,ts2,s2,n2),

```

```

treeunion(ctreejoin(middle,s1,ts2,s2,n2),
          ctreejoin(right,s1,ts2,s2,n2))))
else
insert23(tupappend(tupleproj(value2,s1,keyofscheme(s1)),
                  firstPart2),
         tupappend(value2,partner2),
insert23(tupappend(tupleproj(value2,s1,keyofscheme(s1)),
                  firstPart2),
         tupappend(value2,partner2),
treeunion(ctreejoin(left,s1,ts2,s2,n2),
treeunion(ctreejoin(middle,s1,ts2,s2,n2),
          ctreejoin(right,s1,ts2,s2,n2))))))

end
end
fun tree2string(Put(_,_,_)) = raise putException
| tree2string (E) = ""
| tree2string(Tr2(tree1,key,value,tree2)) =
    tree2string(tree1)^tup2string(value)^tree2string(tree2)
| tree2string(Tr3(tree1,key1,value1,tree2,key2,value2,tree3)) =
    tree2string(tree1)^tup2string(value1)^tree2string(tree2)
    ^tup2string(value2)^tree2string(tree3)
(*
fun absT (E) = nil
| absT(Put(_,_,_)) = raise putException
| absT(Tr2(tree1,value,tree2)) = absT(tree1)@[value]@absT(tree2)
| absT(Tr3(tree1,value1,tree2,value2,tree3))=
    absT(tree1)@[value1]@absT(tree2)@[value2]@absT(tree3)
*)
end

abstype crelation = crel of (scheme * tree23)
with
exception crelexception1
exception crelexception2
exception crelexception3
exception crelexception4

```

APPENDIX D. AN SML SPECIFICATION BASED ON 2-3 TREES 261

```
fun cmakerel (s,t) = crel(s,t)
fun keyOf(aTuple , crel(s,ts) ) =
    tupleproj(aTuple,s,keyofscheme(s))

fun crel2string(crel(s,ts)) =scheme2string(s)^\n"~tree2string(ts)^\n"

fun cinsert(t,crel(s,ts)) = if not (match(t,s)) then
    raise crelexception1
    else
    if isMember(keyOf(t,crel(s,ts)),ts) then
    raise crelexception2
    else
    crel(s,insert23(keyOf(t,crel(s,ts)),t,ts))
fun cunion(crel(s1,ts1),crel(s2,ts2)) =
    if not (equiv(s1,s2)) then
    raise crelexception3
    else
    crel(s1,treeunion(ts1,ts2))
fun cjoin(crel(s1,ts1),n1,crel(s2,ts2),n2) =
    crel(schappend(s1,s2),ctreejoin(ts1,s1,ts2,s2,n2))

end

val stree = makeTree([s1,s2,s3,s4,s5,s6],suppsch)
val stree1 = makeTree([s1,s2,s3],suppsch)
val stree2 = makeTree([s4,s5,s6],suppsch)
val suppcrel = cmakerel(suppsch,stree)
val supp1crel = cmakerel(suppsch,stree1)
val supp2crel = cmakerel(suppsch,stree2)
```

The two published papers cited below have been removed from the e-thesis due to copyright restrictions

Published Work

'Object-oriented Implementations from a Functional Specification'

Proceedings of the Software Quality Workshop,
June 1990, Dundee Institute of Technology

Published Work

'A Recursive Database Query Language on an Object-oriented Processor'.

In: Applications of Supercomputers in Engineering II.

Edited by C.A. Brebbia, D.Howard and A.Peters.

Southampton: Computational Mechanics Publications, 191-205.