J. Functional Programming **7** (2): 163–182, March 1997. Printed in the United Kingdom

NATURAL EXPERT: *a commercial functional programming environment*

NIGEL W. O. HUTCHISON, UTE NEUHAUS
Software AG, Uhlandstr 12, 64297 Darmstadt, Germany

MANFRED SCHMIDT-SCHAUSS
FB Informatik, J. W.Goethe-Universität, Postfach 11 19 32, 60054 Frankfurt am Main, Germany

CORDY V. HALL
Department of Computer Science, University of Glasgow, Glasgow, UK

Abstract

NATURAL EXPERT is a product that allows users to build knowledge-based systems. It uses a lazy functional language, NATURAL EXPERT LANGUAGE, to implement backward chaining and provide a reliable knowledge processing environment in which development can take place. Customers from all over the world buy the system and have used it to handle a variety of problems, including applications such as airplane servicing and bank loan assessment. Some of these are used 10,000 times or more per month.

Capsule Review

It is interesting to see a lazy functional language being applied in an industrial setting for expressing rules of expert systems. A functional language provides a solid basis for reasoning about rules, an aspect that is notoriously difficult to accomplish with, for example, production systems. Sometimes a lazy language solves a problem automatically that would otherwise require special programming. The solution one gets for free here is that the user is not asked questions whose answer is not relevant to the result of a query. The paper describes the experience of using the system in real life industrial applications including reports on some of the difficulties encountered.

1 Introduction

NATURAL EXPERT (Software AG, 1990a, b) by Software AG is a product that supports the development of knowledge-based systems. It includes a functional language called NATURAL EXPERT LANGUAGE (NEL), which is the first successful implementation of a lazy functional language in a database-oriented mainframe environment. NATURAL EXPERT is currently installed and used at about 20 customer sites on a variety of platforms, and the applications are in every day use by several hundred people.

Software AG (Darmstadt) decided to implement NATURAL EXPERT for several reasons. Looking at expert systems projects in Germany, they found that about 2% reached production status. The reasons for this seemed to be the following:

- they had primitive I/O, restricted database access support, and were generally incompatible with other products;
- the tools and methodology came from the AI world, but did not use the principles of software engineering;
- production rules look simple, but make it hard to understand an entire application's behaviour in practice. Maintenance of production systems is very difficult.

Expert systems are expected to be good at handling lots of detailed information, both initially and during application maintenance. To do this, they have to avoid adding to the inherent problems of managing the application. It seemed clear that an important part of reducing development and maintenance overhead was to build a tool that can minimize early errors, and support well-structured programs.

Fortunately, Software AG had another product, a sophisticated entity relationship DBMS (database management system) called ADABAS ENTIRE, which was capable of efficiently handling complex queries with the expressive power of set comprehensions. It could easily maintain consistency between different sets of rules and types as a programmer developed an application. All that was needed was a language that supported the development of efficient, clean rule definitions within a logical framework.

The team compared Lisp (McCarthy, 1960; Steele, 1984), Scheme (Abelson and Sussman, 1984), Prolog (Clocksin and Mellish, 1981), ML (Wikstroem, 1987) and lazy functional languages such as Hope (Burstall *et al*., 1980) and Miranda (Turner, 1985; Bird and Wadler, 1988). Their pilot project was based on an existing product, SAVOIR (SAVOIR, 1985). They constructed a new rule maintenance environment with the rule sources and compiled code stored in an ADABAS ENTIRE database. The rules were then combined to make an application which could be interpreted by the SAVOIR inference engine. However, the language used was strict, did not support lists or other structured datatypes, did not have function definitions and it's I/O was poor. They then chose to implement a lazy functional language for the following reasons:

- They wanted a language that was declarative, concise, powerful, extendable and general.
- Lazy evaluation was likely to be especially suitable, because, like depth first backward chaining, it evaluates only those rules directly relevant to satisfying a goal and subgoal.
- Functional programs can be reasoned about.
- Polymorphic type checking makes it much easier to validate an application at compile time and supports the user in the production of correct reusable code.
- Functional languages are the subject of implementation-oriented research which is public domain.

- Functional languages are increasingly being taught in universities.

This paper first describes the programming environment offered by NATURAL EXPERT, and the system that supports it. The programming environment itself is written in NATURAL, Software AG's proprietary fourth generation language.

In the process, it shows what happened to a lazy functional language as it was tailored to fit the requirements of NATURAL EXPERT, giving functional language designers a better insight into the restrictions encountered by industry when using functional languages in practice. We try to give a feel for what it is like to use the system. However, the figures are not actual snapshots of windows because we've omitted some irrelevant fields.

Finally, the paper explores the commercial applications of NATURAL EXPERT and summarizes customer feedback from questionnaires and letters.

2 An introduction to NATURAL EXPERT

As we will see, the NATURAL EXPERT environment is quite complex. For this reason, we introduce the environment by considering a tiny demonstration program – a program to compute the *n*th prime number. Of course, this is not intended to be an applications program, since the goal is to familiarize the user with the basic components of the system, rather than start building an expert system.

A little NATURAL EXPERT terminology: programs are referred to as *functional models*. A function is called a *rule*. A *semantic model* is a hypertext-like graph maintained by the DBMS. Semantic models are used in two ways: first, to maintain the program itself, as discussed in this section; second, to maintain data that is manipulated by an application program, as discussed in the next section.

NATURAL EXPERT provides a menu-driven user interface. The menu fields can be described as follows:

- *function name*: the name of the function.
- *function type*: the type of the function. Lists of type `a` are written as `L(a)`. The integer type is written as `I`, and the *text* or string type as `T`. NEL supports a variety of fixed precision types; for instance, `N7` is the type of a number with seven decimal places.
- *attached-to*: the *attached-to* refers to a node in the semantic model. This node could, for example, contain text describing the behaviour of the function.
- *rule form*: rule forms are categories of rules. Usually the form is *Standard* which shows an editable rule in NEL. It may also be a *Gateway* definition of an interface to NATURAL, or an *Imported* definition from a specific library.
- *stage*: as rules are developed they can acquire one of the following four *stage* labels:
  - — A rule that only exists as a name and a type (created as a forward reference) is labelled as *Incomplete*.
  - — A rule that lacks type declarations for at least one of its components or else cannot be compiled because of syntax, type or semantic errors is given the label *Tokenized*.

https://doi.org/10.1017/S0956796897002657 Published online by Cambridge University Press

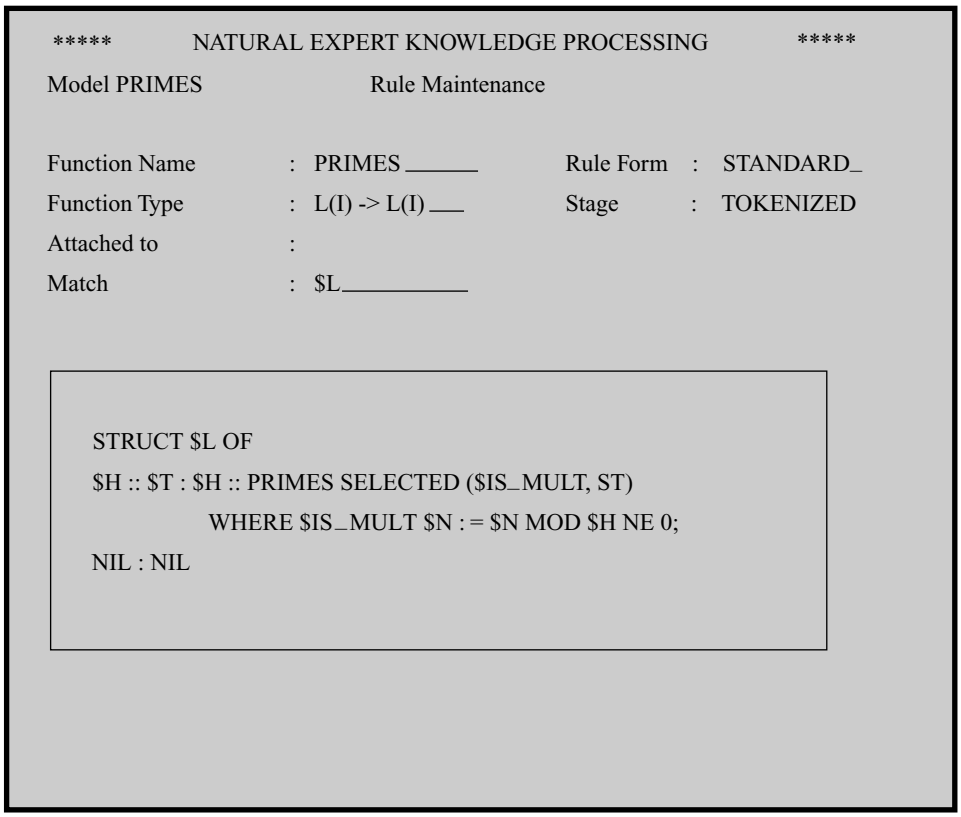

Fig. 1. Definition of the PRIMES rule written in NEL.

```
primes        :: [Int] -> [Int]
primes []     =
              []
primes (h:t)  =
              h : primes (filter (\n -> n mod h /= 0) t)
```

Fig. 2. Haskell version of the PRIMES function.

— A rule that has been successfully compiled is given the status *Production.*
— All rules which call other rules with out-of-date types with respect to the last compilation are labeled *Recompile.*

The programming environment helps the developer to keep track of these stages by maintaining a *workplan* for each model. It automatically adds rules to the workplan (for instance, where forward references are created or rules are set to recompile). The user may also insert or remove rules from the plan.

- *match*: function argument names appear in this field.

https://doi.org/10.1017/S0956796897002657 Published online by Cambridge University Press

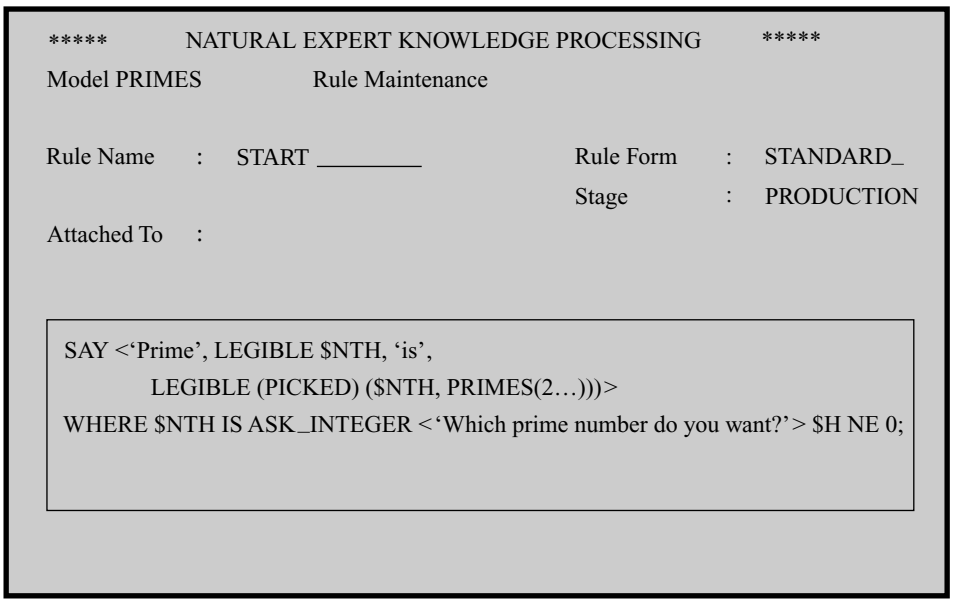

Fig. 3. START rule for the PRIMES model.

```
main =
  appendChan stdout ``Which prime number do you want?'' abort (
  readChan stdin abort (\ n_string ->
  appendChan stdout (``Prime '' ++ n_string ++ `` is '' ++ msg) abort done
      where
      n = (read n_string) :: Int
      msg = show ((primes [2..]) !! n)))
```

Fig. 4. Haskell version of the START rule in the PRIMES model.

Figure 1 shows a NEL rule (that is, a function) that given the infinite list `[2..]` returns the infinite list of primes. This is a standard program – see, for instance, Bird and Wadler (1988). For comparison, figure 2 gives the definition as it appears in Haskell (Hudak *et al.* (1990). Various aspects of the NEL code may appear unfamiliar at first glance, so we explain these aspects below.

A glance at figure 1 shows that NEL resembles a functional language intermediate code (as found in the core syntax of several compilers written in functional languages); and that it has some syntactic idiosyncracies. Note, in particular, that local variables start with a `$` and function arguments are tupled rather than curried. These design decisions were in general taken to make it easier for the user to learn the language.

Lists are constructed with infix 'cons', written `::`, and deconstructed with `STRUCT`, which is equivalent to Haskell `case`. The function `SELECTED` applies a predicate

https://doi.org/10.1017/S0956796897002657 Published online by Cambridge University Press

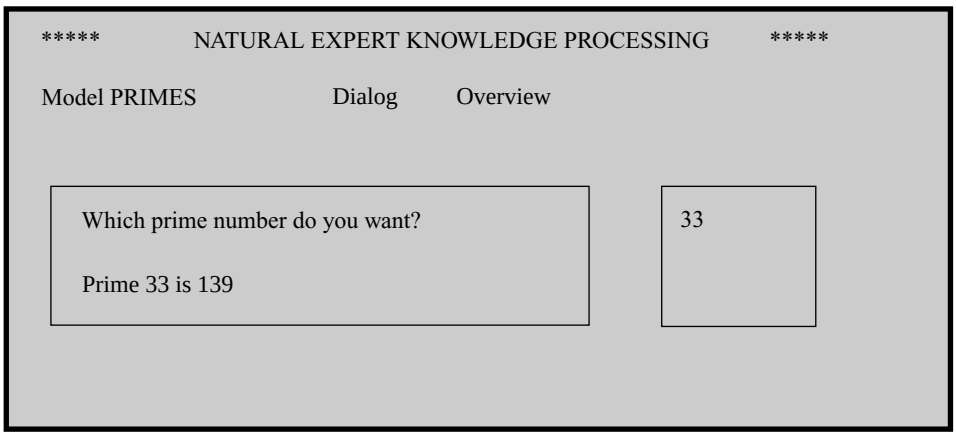

Fig. 5. Interacting with the PRIMES model.

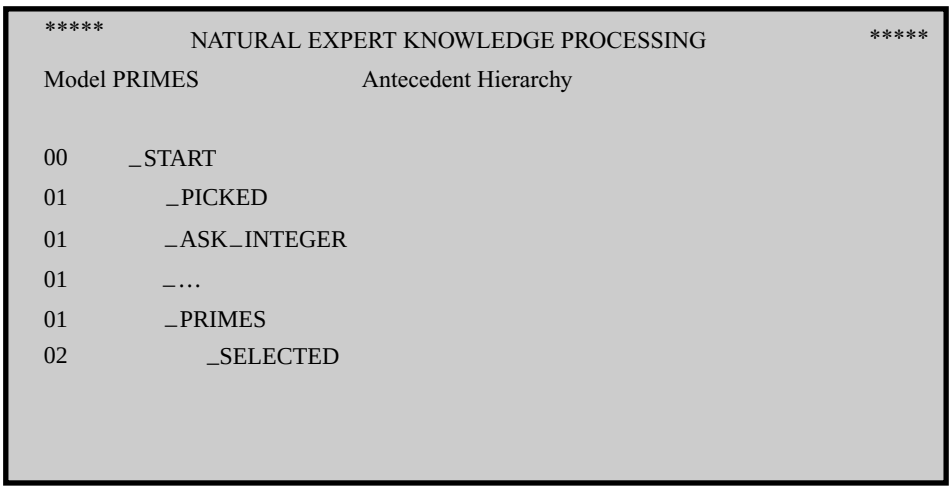

Fig. 6. Antecedent Hierarchy for the PRIMES model.

to a list, returning those elements for which the predicate is true, like the Haskell function `filter`, and `NE` is the infix predicate 'not equal', like the Haskell `/=`. The syntax for function binding is `:=`, compared with `=` in Haskell.

The first rule called in a program is always the `START` rule, similar to the use of `main` in Haskell. Figure 3 gives the `START` rule for the primes example, and its equivalent in Haskell appears in figure 4.

The function `LEGIBLE` converts atoms (numbers, booleans ...) to text, similar to a restricted version of the Haskell `show` function. The function `PICKED` returns the *n*th element of a list, similar to `!!` in Haskell, and the notation `2 ...` returns an infinite list starting with 2, equivalent to the Haskell notation `[2..]`.

Input and output is performed by functions with side effects. The `SAY` outputs its string argument to the screen. The function `ASK_INTEGER` takes a text, prints the text on the screen, waits for the user to type an integer, and returns that integer.

https://doi.org/10.1017/S0956796897002657 Published online by Cambridge University Press

Side effects are intended to be used in NEL in a disciplined manner. Output is usually performed only at the top level, and the use of side effects for input is discussed further in the next section. In contrast, Haskell scrupulously avoids side effects, leading to the referentially transparent but somewhat more complex code using `appendChan` and `readChan`.

Finally, the model is executed by the *runtime session manager*. This facility provides a dialog box in which the user responds to the application, and matches each query to its response, as shown in figure 5.

Navigation through both the semantic and functional models is made easy by the `ZOOM` command, which can follow kind, item and attribute links in the semantic model (much like hypertext) (see section 3. This is often used to support program documentation and information on types. Figure 6 shows a snapshot of the system as `ZOOM` expands the `START` rule, displaying the *consequent hierarchy*, that is, a list of all callers. It can also display the *antecedent hierarchy*, a list of all callees.

3 Interfacing NEL to the world – the *office* example

Our next example matches a team of programmers with the office that best fits its specification. This example will illustrate the way in which input/output is integrated with lazy evaluation, and the interaction between the function model (application program) and semantic model (database).

The semantic model consists of *items*, each of which is an instance of a *kind*. Associated with the kind is the set of *attributes* each item may possess. In our example, the kind `OFFICE` possesses five attributes: office number, area, number of windows, terminals and telephones. In this simple example, each attribute has type `I` (integer). An office instance, the item `OFFICE_M101`, appears in figure 7. This corresponds to an entry in the database.

Notice that one of the attributes, the `TERMINALS` attribute, is assigned the value `UNKNOWN`. All value domains in NEL include this value, which has a variety of uses. Functions that test for this value include the predicate `known`, which returns `False` if its argument is `UNKNOWN`.

Data types specify records, which are built by *constructors* applied to *attributes*. In our example, the type `TEAM` is built by the constructor `C_TEAM` and has three attributes: number of team members, number of terminals, and number of telephones. Again, each attribute has type `I`. Figure 8 shows the declaration of this type. The corresponding Haskell data type definition is

```
data TEAM = C_TEAM Int Int Int
```

The best office for a given team is computed as follows. An office cannot be used by a team unless it has enough space per person and there are at least as many windows as there are people, so the acceptable offices have to be filtered out from all of the possible offices. Then, the best of the acceptable offices can be determined by cost. Calculating the cost takes into account the expense of either taking away or adding new resources such as telephones and terminals, and the cost of any space over the minimum required. One office is better than another if its cost is lower, so

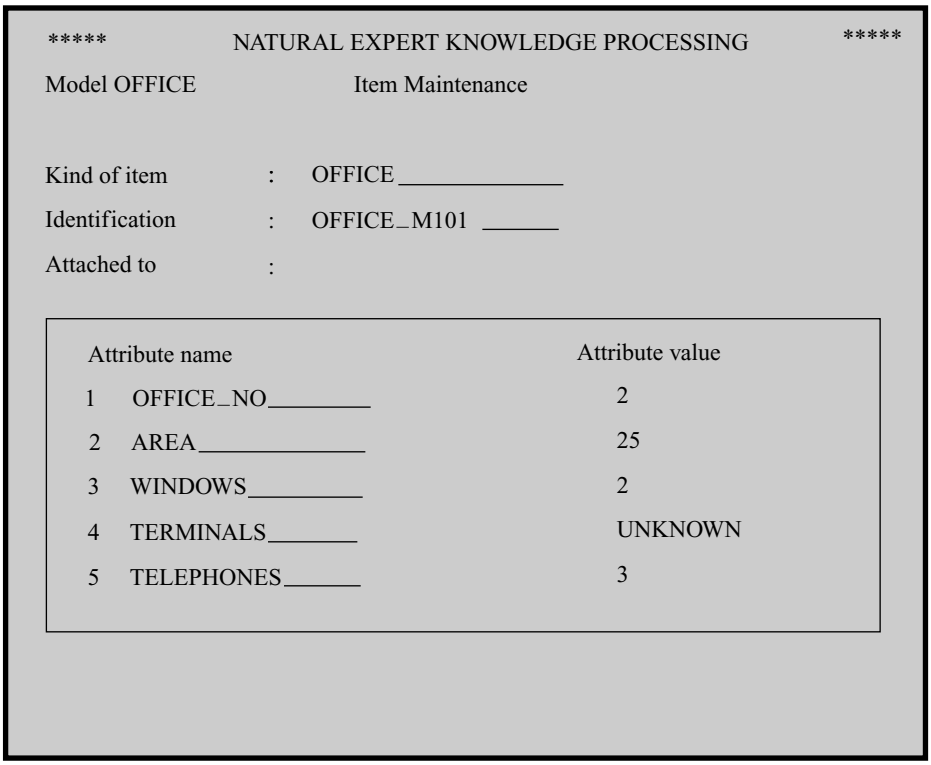

Fig. 7. The `OFFICE` instance `OFFICE_M101`.

the best office can be found by sorting the acceptable offices by cost, then picking the cheapest. This may be expressed straightforwardly using functional concepts like *filter*, *map*, *sort*, and so on. We omit this code as it demonstrates no new features.

3.1 Integrating I/O with lazy evaluation

Recall that `ASK_INTEGER` takes a list of text, prints that text, waits for the user to type an integer, and returns that integer. A convenient aspect of lazy evaluation is that sometimes the user will not be plagued with questions the answer to which is irrelevant. Consider, for instance, the following code, part of the `team_requirements` rule (figure 9):

```
C_TEAM ($TEAM_MEMBERS, $TEAM_TERMINALS, $TEAM_TELEPHONES)
     WHERE
     $TEAM_MEMBERS IS ASK_INTEGER <'How many team members?'>,
     $TEAM_TERMINALS IS (IF ($TEAM_MEMBERS EQ 1)
                            THEN 1
                            ELSE ASK_INTEGER <'How many terminals?'>),
     $TEAM_TELEPHONES IS ASK_INTEGER <'How many telephones?'>
```

https://doi.org/10.1017/S0956796897002657 Published online by Cambridge University Press

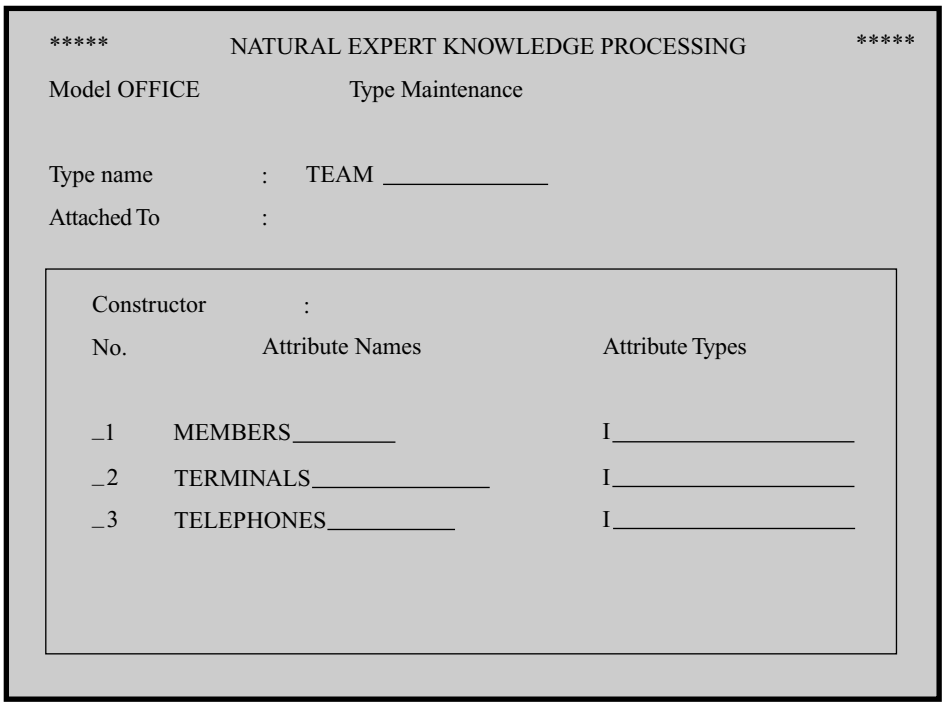

Fig. 8. Defining the TEAM type.

Here EQ tests for equality, similar to == in Haskell. If the answer to the question "How many team members?" is 1 then it is assumed that 1 terminal is required, and the question "How many terminals?" will not be asked.

The order in which questions are asked will depend upon the order of evaluation. It is up to the user to write in a style where this order does not matter, which tends to be easy for the sort of questions asked by expert systems.

Observe that ASK_INTEGER is *not* referentially transparent. For example, the expression

```
($N, $N) WHERE $N IS ASK_INTEGER <'Pick a number'>
```

and the expression

```
(ASK_INTEGER <'Pick a number'>, ASK_INTEGER <'Pick a number'>)
```

are not equivalent: the first will ask the question once and the two components must be the same, while the second will ask the question twice and the two components may differ.

This lack of referential transparency has had a real impact on the compiler design. Some optimizations based on common subexpression elimination or full laziness must be disabled, because they would transform programs like the second into programs like the first.

An alternative is to keep a 'memo table' which pairs each text with the associated answer, and if ASK_INTEGER is called again with the same text, return the same answer

https://doi.org/10.1017/S0956796897002657 Published online by Cambridge University Press

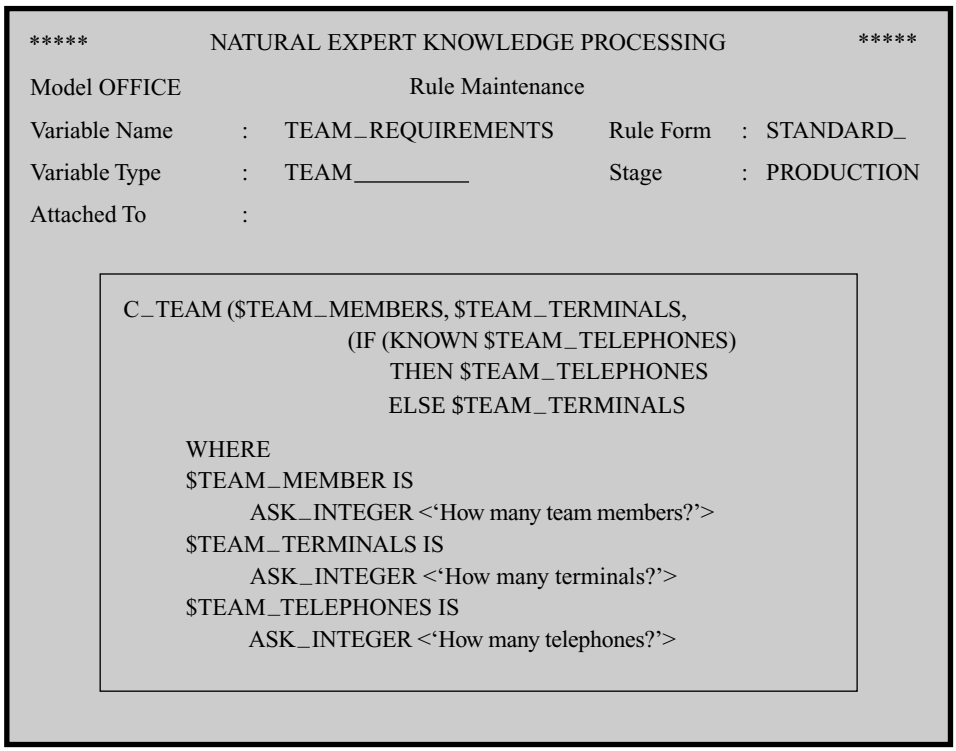

Fig. 9. Defining `team_requirements` in NEL.

without asking the user a second time. This guarantees referential transparency, but requires time and space to maintain the table. Since both are at a premium in NEL applications, this solution was not adopted.

3.2 Access to the DBMS using 'gateway' functions

In our example, the variable `available_offices` is bound to a list of records describing each office. The computation of this list is interesting, as it demonstrates the interface between NEL and its associated DBMS.

This list is computed in two steps. First, the database is queried to find a list of each office instance. Then, for each instance in the list, the database is queried to find attributes associated with that instance. These steps convert the database information on each office into a NATURAL EXPERT datatype.

Two so-called *gateway* functions, `ITEM_FIND` and `ITEM_ATT_NUMBER`, provide the necessary access to the database. One role of the gateway functions is to validate DBMS access. Each gateway function must be passed a DBMS profile, which contains information such as a password. A value of type `DBMS_PROFILE` contains a user-id, a password, and a model name. In the following, we assume the variable `PROFILE` contains a profile that allows access to the office model.

The function `ITEM_FIND` takes the name of a kind and a profile and returns a list

https://doi.org/10.1017/S0956796897002657 Published online by Cambridge University Press

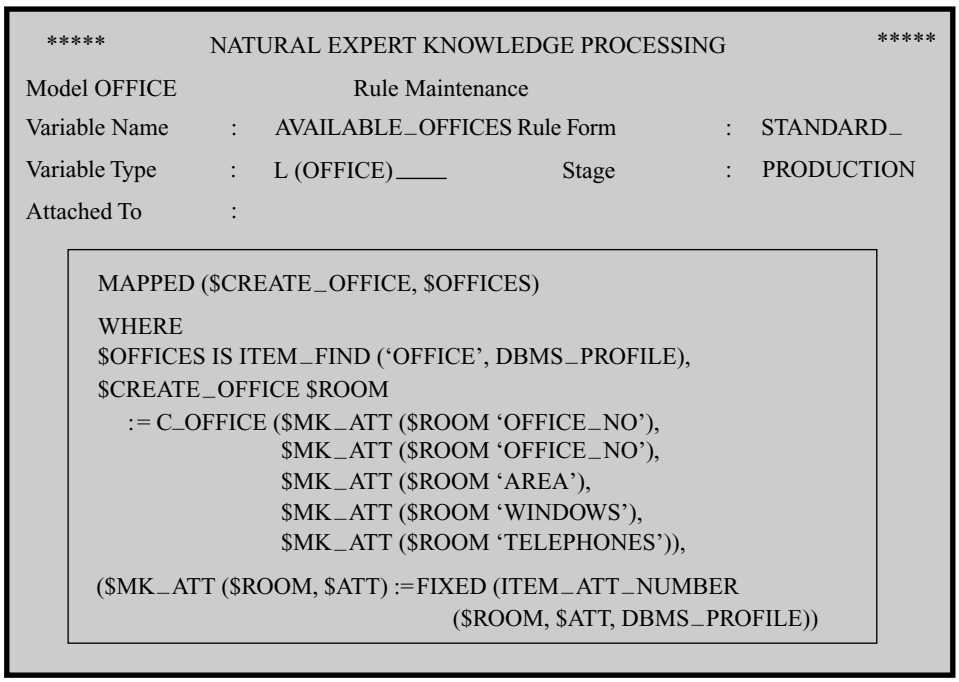

Fig. 10. Available offices.

```
available_offices
          = let  office_rooms = item_find "office" dbms_profile
                 mk_att room att
                        = item_att_number office_room att dbms_profile
                 create_office room att
                        = C_office  (mk_att room "office_no")
                                    (mk_att room "area")
                                    (mk_att room "windows")
                                    (mk_att room "terminals")
                                    (mk_att room "telephones")
          in     map create_office office_rooms
```

Fig. 11. Defining `available_offices`.

of all instance keys of that kind. Kind names are represented by text. The text type is written simply `T`, the profile type is written `DBMS_PROF`, and the instance key type is written `INT_KEY`.

```
type:  (T, DBMS_PROF) -> L(INT_KEY)
syntax:  ITEM_FIND ($KIND, $PROFILE)
```

The function `ITEM_ATT_NUMBER` takes an instance name, an attribute name, and

https://doi.org/10.1017/S0956796897002657 Published online by Cambridge University Press

a profile and returns the value of that attribute, which must be a fixed-precision number. We write N7 for fixed-precision numbers with seven decimal places.

```
type:    (INT_KEY, T, DBMS_PROF) -> N7
syntax:  ITEM_ATT_NUMBER ($INSTANCE, $ATTRIBUTE, $PROFILE)
```

In our example, the variable AVAILABLE_OFFICES contains a list representing all offices in the database. Figure 10 shows the NEL code defining this variable. This code works as follows:

- The variable $OFFICES is a list of office instance names. It is computed using ITEM_FIND.
- The function $MK_ATT takes an instance name and an attribute name and returns the integer representing the instance attribute. It has type

  ```
  (T, T) -> I
  ```

  It uses ITEM_ATT_NUMBER to perform the lookup, and FIXED to convert a fixed-precision number (type N7) to an integer (type I).
- The function $CREATE_OFFICE takes an office instance name and returns a record of type OFFICE representing the corresponding attributes. A value of type OFFICE is built with the constructor C_OFFICE and has five fields, all of type integer, representing the office number, area, number of windows, number of terminals and number of telephones.
- Finally, mapping function $CREATE_OFFICE down the list $OFFICES computes the appropriate list. The function MAPPED applies a function to each element of a list, similar to map in Haskell.

The equivalent Haskell code appears in figure 11.

3.2.1 Gateway functions – what they are and how they are defined?

Gateway functions call routines from NATURAL. Since NATURAL and NATURAL EXPERT use different language paradigms, these functions must impose restrictions on the data they transmit. Gateway functions must be first-order and monomorphic, and so cannot take advantage of polymorphism. More significantly, gateway functions are hyperstrict, that is, they entirely evaluate all arguments and results (Buneman *et al.*, 1982), and so cannot take advantage of lazy evaluation. In the office example, the list $OFFICES returned by the gateway will be evaluated in full, although the list AVAILABLE_OFFICES may be computed from this incrementally via lazy evaluation.

NATURAL EXPERT eases the creation of gateways by generating templates, so the user only needs to write part of the code. The user first specifies the argument and result data types for the gateway. The system uses this information to automatically create a template that converts arguments from the representation used by NEL to NATURAL, and converts results back again. The template also calls servers to transmit data between the two language systems. The user is responsible for writing a NATURAL program that performs the actual database lookup.

https://doi.org/10.1017/S0956796897002657 Published online by Cambridge University Press

Gateway functions may update as well as read the database. The order in which these side effects occur is determined by the order in which the gateway functions are evaluated during program execution. Application programmers are taught to handle side effects such as these carefully, usually by performing them at top level. There is a special class of rules, called *action rules*, that are helpful for such sequencing. For instance, one action rule causes its argument to be evaluated in full. The NEL team has found this approach to be awkward, and hopes to eventually develop a better means by which I/O can be controlled.

3.2.2 The role of the DBMS in supporting NATURAL EXPERT

One of the main features of NATURAL EXPERT is the way in which its programming environment uses the underlying DBMS, ADABAS ENTIRE, to make software development easier. Both the functional model (program) and semantic model (data) are maintained by the DBMS. As we have seen, the DBMS aids in tasks such as labeling functions with their stage (for instance, whether they need to be recompiled), and generating lists of the antecedents (callees) and consequents (callers) of a given function.

The reason that ADABAS ENTIRE is efficient enough to be used in this way is that it is based on the *entity relationship* model. This is a structured, object-oriented model (Chen76) that allows entities and their relationships to be represented and manipulated directly, using a powerful query language. Such a model is closely related to the structure of semantic and functional models, and so will naturally be efficient in representing them. In contrast, relational databases store data in tables that represent relations. This representation is simple and easy to understand, and it supports data independence, but it is too inefficient for the DBMS queries typical for NATURAL EXPERT.

4 Integrating NEL with commercial DP-environments

NEL is written in both NATURAL and C, as is the runtime system. It is designed to give access to a variety of databases. In addition, NEL was designed to be used on a mainframe with a transaction processing (TP) monitor. In this section, we describe the restrictions these decisions impose, and show how NEL was implemented to handle them successfully.

4.1 Restrictions on the design of NEL

TP-monitors A TP-monitor can be viewed as an operating system within an operating system. It maximizes the number of possible users within the constraint of acceptable response times, making efficient usage of resources by sharing them as much as possible. Application programs that run within a TP-monitor have access to operating system resources only through interfaces supplied by the monitor. This includes database accesses as well as terminal communication.

These requirements imply that the resources available per end-user are rather

https://doi.org/10.1017/S0956796897002657 Published online by Cambridge University Press

restricted. For example, at most one copy of an executable program or module is in main storage, and there is always a limit on the maximum CPU-time for a single user-request. Storage per user session thread seldom exceeds 500 KB. This means that programs and static data are best shared, while dynamic data is allocated by individual users.

4.2 How NEL *was designed to handle these restrictions*

NATURAL EXPERT was developed with TP monitors in mind, so it respects their resource limitations. Typically, a user is allocated a heap size of 80K. This is sufficient for running applications and compares favourably with the megabytes required by other functional languages on workstations. NEL's implementation was constrained by this space restriction, which is why some features normally supported by a functional language were omitted. For example, local recursion is not supported because it requires allocating 'holes' for recursive references, and because garbage collection is done using reference counting. Dynamic data was reduced by adding a read-only heap. Maximal structures which do not contain redexes are identified at compile time and generated into this heap. A small range of integers is supported, again to save space.

It should be noted that many companies cannot use NATURAL EXPERT because it currently takes up too much space, even given these compromises. The architectural limits of many commonly used mainframes demand that each user be given at most 80K. Thus, reducing space consumption is an important goal in developing industrial lazy functional languages.

Complex functional language features, such as list comprehensions, have been omitted from NEL to reduce problems in training. However, Software AG has had success in teaching customers functional programming through a company training course. Customers learn functional programming techniques using higher order functions, and find polymorphism very useful.

5 NATURAL EXPERT in industry

This section contains reports on a variety of applications in locations around the world. Development times include research, design, implementation, testing and documentation: times for multi-phase projects include maintenance times for earlier phases. Some customers used other projects implemented with NATURAL EXPERT by Software AG for the company, and simply did the new work themselves. This suggests that the implementation language, NEL, was in fact easy enough to use. It also makes it harder to keep track of the applications developed in NATURAL EXPERT.

The following table summarizes the person days, number of Natural Expert models and rules, and Natural modules for some of the applications described below:

https://doi.org/10.1017/S0956796897002657 Published online by Cambridge University Press

Name	days	models	rules	modules
Satellite	2400	10	1400	
Siade	150	2	240	100+
Arian names	300	1	46	120
Arian addresses	180	1	65	120
Dietas	900	4	200	500
Sades	550	1	250	420
Consumo	150	1	30	194

The applications themselves are described below.

- SATELLITE**, Telefonica de Espana, Madrid, Spain**
  The aim of the SATELLITE system was to create a tool for the user which could interpret a question in normal Spanish, and retrieve the appropriate information. Telefonica plans to package the nucleus of the application and to launch it as a commercial product.
- SIADE**, Telefonica de Espana, Madrid, Spain**
  SIADE is a system for automating the generation and maintenance of cost/investment records, designed to reduce management workloads. It implements a complex standard, defining 50 distinct types of decisions which may be taken about expenditures or investments, so that the standard can be managed by personnel without computer experience. SIADE is used 20,000 times a month.
- ARIAN**, Software AG Espana, Spain**
  ARIAN identifies Hispanic names and addresses, a task which can be difficult because both the first and the second name can have more than one part *and* these two parts of the name can occur in reverse. It matches names against a dictionary which can be augmented by local operators more familiar with unusual local names.
- DIETAS**, Software AG Espana, Spain**
  DIETAS works out menus for hospital patients.
- SADES**, Software AG Espana, Spain**
  SADES looks at a road network and works out where to put gas stations on it.
- **Teacher Retirement System of Texas, USA**
  NATURAL EXPERT was used to create two applications for TRS:
  — The Claims system selected the forms required to process a particular death claim. Previously, only a few of the 50 possible forms were selected (and not always correctly) because many factors could affect a possible selection and forms frequently changed, requiring the rules to change as well.
  — NATURAL EXPERT also helps select which paragraphs are needed for a particular claims letter. This allows clerks to create a complete letter with the push of a button.
- CONSUMO**, Banco Exterior, Spain**
  The CONSUMO project was a decision support system for personal loans, constructed for one of the largest banks in Spain. Initial processing was done by using a statistics package to handle 60% of the cases. However, this package could not make any recommendations if it rejected an applications, and this

https://doi.org/10.1017/S0956796897002657 Published online by Cambridge University Press

could cost money when these applications were reasonable. CONSUMO salvages more than half of these, and can offer alternatives to the borrower when it does reject an application.

Three other interesting applications, for which no figures are given, are:

- **Banco Mercantil, Caracas, Venezuela**
  Banco Mercantil in Caracas, Venezuela, recently began to issue VISA and Mastercard credit cards. It was essential that a system for processing these be implemented as soon as possible, since 100,000 applications were anticipated, and there were only 20 people to process them. Knowledge engineers at Banco Mercantil had just developed a successful expert system for processing loan applications using NATURAL EXPERT, so it was decided to take advantage of the similarities in the problems and adapt it.
  The system was developed in three weeks. Many functions from another NEL application (a Personal Loan Expert system) were reused without any changes.
- COLISAGE GROS VOLUME**, Polygram, France**
  Polygram is a subsidiary of Phillips which controls 33% of the European market for records and cassettes. They initially solved their gross volume distribution problem by minimizing two variables: the empty space within a parcel, and the number of stations it stops at. The existing system was written in GURU, a production rule system, one of the few cases in which NATURAL EXPERT was compared directly against another form of expert system.
  GURU started having problems when the ratio of cassettes to records changed. An initial system was then developed by two people at Polygram and two from Software AG, and finished by Polygram, which commented that the new system was shorter and better integrated the existing environment.
- IVANHOE**, Paris, France**
  IVANHOE invoices airlines for services provided when an airplane is on the ground at an airport, generating simple invoices and explanations. After intensive discussion, the customer built their own system. Today, each flight passing by Orly or Roissy is supported by IVANHOE.

5.1 Customer opinions

Software AG asks their customers to fill out a questionnaire on NATURAL EXPERT, and maintains a 'Likes and Dislikes' file for applications. Here are some of their comments:

- **Inland Steel, East Chicago, Indiana**
  NATURAL EXPERT was the easiest expert system product to integrate with our current production mainframe environment. Coding the Surface Defect expert system (over 100 rules in NATURAL EXPERT) took much less time than coding the others, despite its size.
- IVANHOE **project, Paris, France**
  Development in NEL is rapid and sure. It is necessary to have a bit more training than that required by 4GLs because the programmer needs to learn

functional programming, recursion, list and tree manipulation. For a problem of average complexity, we estimate productivity gains at a ratio of 1:10, which is likely to be much greater for very complex problems.

- SATELLITE**, Telefonica de Espana, Madrid, Spain**
  The system is powerful and efficient, admitting comparison with typical systems of menus and query by example. In fact, in addition to the advantages of functionality and control, which the other systems (the pieces of the original software before SATELLITE was installed) do not have, the retrieval time and the total time taken to produce data are even better in SATELLITE.
- COLISAGE GROS VOLUME, Polygram, France
  The NATURAL EXPERT system was much shorter and easier to understand than the existing solution. Integration with the existing environment was much better.

One customer noted that it was necessary to rethink problems to use recursion and functional programming, but that the effort paid off.

On the subject of production rule systems, another commented "One problem in our application is making sure that all rules are disjoint. This is not necessary in NATURAL EXPERT. Some areas of our expertise do not lend themselves well to being written as production rules: the limits of the product are reached quickly."

The negative comments tended to center around two areas. Functional programming often required training, which took time, and functional programs behaved rather differently from imperative programs that caused one customer to worry about resource consumption for large problems. He decided not to use NATURAL EXPERT for the reason that he couldn't predict its behaviour.

One interesting question asked was "Do you see any barriers to NATURAL EXPERT's use in production applications? If so, what would it take to remove those barriers? (Consider both technical and organizational/educational barriers)". A customer responded that "functional programming is not taught in many US universities", and for this reason training programs needed to take the steeper learning curve required into account.

6 New applications

Recently, NATURAL EXPERT has been used to build two new tools. First, NATURAL EXPERT has been used to build re-engineering tools for the analysis and transformation of conventional source programs. The nucleus of these tools is a reusable set of modules for source manipulation.

Customers applications consisting of many hundreds of thousands of lines and tens of thousands of source objects have been successfully processed with these tools. Typically, one tool scans an entire application looking for certain features and builds a knowledge base. Other tools read the knowledge base, select the programs which need attention and then transform them, recording their progress in the knowledge base.

Experience has shown that particular processing requirements can often be met

with very little human effort compared with error prone manual correction of such large applications.

The second is a configurator for complex machinery. This application, comprised of several NATURAL EXPERT models, processes customer requirements using a large database of rules and constraints. The application is integrated with Computer Aided Design and a Bill of Materials system. The system is now online to users in several countries, and presents mission critical services to a large international company.

7 Conclusions

NATURAL EXPERT is marketed as a component of a broad range of Sofware AG products and solutions designed to meet the requirements of commercial companies. NATURAL EXPERT applications now run on PC, Digital and various UNIX platforms.

7.1 What advantages can Software AG customers obtain from NATURAL EXPERT*?*

It is clear that increasingly customers require special application components with a high internal logical complexity to secure a competitive advantage . These must be able to interface with existing commercial databases.

7.2 What needs to be done to support industry in using lazy functional languages?

These are some important problems that need to be addressed. Space consumption of lazy functional languages is not well understood and users find this unsettling. Methods to analyse, present, and reason about space requirements are needed.

Absolute space requirements in functional languages tend to be too high. To make NATURAL EXPERT acceptable to mainframe users a great deal of effort had to be invested in keeping this within the bounds of what is normally available. The 'Space is cheap' motto applies to the chips and the boards but doesn't translate through to commercial platforms very quickly.

Not enough programmers are familiar with functional languages, and some experienced programmers find it hard to learn concepts such as recursion and polymorphism. Wider use of functional langauges in computer science courses would filter down to industry – indeed, the effect is beginning to be apparent. Though there are now several excellent books on functional programming techniques, the examples used are sometimes difficult to transfer to real life programming.

It should be noted that many multi-user mainframe installations find NATURAL EXPERT difficult to accommodate in their TP monitors because it requires considerably more space from the user threads compared with conventional software. A thread of 450K is still considered quite large, and of this perhaps 80K will be available for the NATURAL EXPERT runtime stack and heap. Many users currently use threads even smaller than this. Reconfiguring the environment for larger threads impacts on the dynamics of the system, and is not popular with administrators. Having said that, experiences with NATURAL EXPERT applications on Unix and

PC platforms indicate that the NATURAL EXPERT runtime space requirements seem relatively modest compared with what is commonly required by other packages.

If we were to build this environment again, it would not be called NATURAL EXPERT. Instead, we would market the tool as something to create software components which otherwise would be too complex and uneconomic to build. The name aroused expectations based on the current naive view of what expert systems could achieve.

We were not courageous enough in presenting the power of functional languages. Early experiences gave the impression that the language was difficult to learn unless one restricted oneself to a subset of the functionality. We ignored counter evidence from experiences in Spain – in fact, we realize now that we were unlucky in the choice of people we presented NEL to at first. We have found that if we present NATURAL EXPERT and a superior programming language and give students lots of examples to do, then functional languages do very well in comparison. At this point, we believe we can include high order combinators in the week induction course.

NATURAL EXPERT is now sold as a part of the entire product range of Software AG, because companies want to have the option of using it even if they currently are buying another Software AG product. Current plans include expanding the customer base by porting the system to Unix, OS/2 and Windows.

7.3 Why do customers buy NATURAL EXPERT*?*

One customer thought that NATURAL EXPERT increased their productivity by a factor of ten. There is good reason to suppose that NEL can be learned reasonably quickly and that it makes maintenance of applications easier, since customers buy NATURAL EXPERT for one project, maintain it themselves and create other applications from it. Several applications are robust and reliable (e.g. SIADE), which is used 20,000 times per month, and IVANHOE, which regularly services air traffic at two busy French airports.

References

Atkinson, M. P., Buneman, P. and Morrison, R. (1988) *Data Types and Persistence*. Springer-Verlag.

Abelson, H. and Sussman, G. J. (1984) *The Structure and Interpretation of Computer Programs*. MIT Press.

Augustsson, L. and Johnsson, T. (1989) The Chalmers lazy-ML compiler. *The Computer Journal*, **32**(2), 127–141.

Brachman, R. and Schmolze, J. (1985) An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 171–216.

Burstall, R. M., McQueen, D. B. and Sanella, D. T. (1980) Hope: an experimental applicative language. *Proc. ACM Symposium on Lisp and Functional Programming*, pp. 136–143.

Backus, J. (1978) Can programming be liberated from the von Neumann style. *Communications of the ACM*, **21**(8), 613–644.

Bird, R. and Wadler, P. (1988) *Introduction to Functional Programming*. Prentice Hall.

Buneman, O. P, Frankel, R. E and Nikhil, R. (1982) An implementation technique for database query languages. *ACM Transactions on Database Systems*, **7**(2), June, 164–186.

Chen P. P. (1976) The Entity-Relationship model: towards a unified view of data. *ACM Transactions on Database Systems*, **1**(1), March.

Clocksin, W. F. and Mellish, C. S. (1981) *Programming in Prolog*. Springer-Verlag.

Frost, R. and Launchbury, J. (1989) Constructing natural language interpreters in a lazy functional language. *The Computer Journal*, **32**(2), 108–121.

Hudak, P., Wadler, P. *et al.* (1990) Report on the programming language Haskell, *Technical Report YALEU/DCS/RR-777*, Yale University, Department of Computer Science.

McCarthy, J. (1960) Recursive function of symbolic expressions and their computation by machine. Part I. *Communications of the ACM*, **3**(4),184–195.

Milner, R. (1978) A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, (17), 348–375.

Milner, R. (1984) The standard ML core language. *Internal report CSR-168-84*, Edinburgh University.

Turner, D. (1985) Miranda: a non-strict functional language with polymorphic types. *Proc. IFIP International Conference on Functional Programming Languages and Computer Architecture*, Nancy, France. *Lecture Notes in Computer Sciences 201*. Springer-Verlag.

Montague, R. (1973) The proper treatment of quantification in ordinary English. In Hintakka, K. J. J. (ed.), *Approaches to Natural Languages*. Kluwer.

Peyton Jones, S. L. (1987) *The Implementation of Functional Languages*. Prentice-Hall.

Poulovassilis, A. and King, P. (1990) Extending the functional data model to computational completeness. *Advance in Database Technology (EDBT 90). Lecture Notes in Computer Science 416*. Springer-Verlag.

Software AG (1990a) NATURAL EXPERT *Reference Manual 1.1.2*.

Software AG (1990b) NATURAL EXPERT *Reference Manual 1.1.3*.

Software AG (1990c) NATURAL *Reference Manual 2.1*.

Sastre, J. and Frias, J. (1990) Satelite, "Lenguaje natural en exploitacion". *Bulletin de AEPIA (Associaon Espanola para la Intelligencia Artificial)*, 12–13.

SAVOIR Expert System Package (1985) *Users Manual 1.2*. ISI Limited, 11 Oakdene Road, Redhill, Surrey RH1 6BT, UK.

Steele G. L. (1984) *Common Lisp: The Language*. Digital Press.

Trinder, P. (1989) A functional database. *PhD Thesis*, Department of Computer Science, University of Glasgow.

Wikstroem, A. (1987) *Functional Programming using Standard ML*. Prentice-Hall.