

# Two-Level Grammar: A Functional/Logic Query Language for Database and Knowledge-Base Systems

Barrett R. Bryant<sup>1</sup> \* and Aiqin Pan<sup>2</sup>

<sup>1</sup> The University of Alabama at Birmingham, Birmingham, Alabama, U. S. A., 35294

<sup>2</sup> IBM Santa Teresa Laboratory, San Jose, California, U. S. A., 95141

**Abstract.** The Two-Level Grammar specification language is used as a foundation for constructing queries to database and knowledge-base systems. TLG offers a natural language interface with the added advantages of functional and logic programming languages, specifically the capabilities of 1) processing SQL-like queries in the relational model, 2) constructing complex objects of well-defined type and defining queries over those objects, and 3) formulation of deductive database rules.

## 1 Introduction

Two-Level Grammar is a metalanguage for software system specification based upon the functional and logic programming paradigms [2]. In this paper, it is proposed as an implementable specification language for database and knowledge-base systems, with the resulting system supporting not only the basic SQL queries but also allowing the user to formulate more general forms of SQL queries, including queries over universal relations, SQL queries embedded into natural language, pure natural language queries, and deductive rules using both functions and logic. Because both SQL and Datalog (Database Prolog) may be embedded in Two-Level Grammar as a natural subset, existing database systems developed in these languages will remain compatible, yet be greatly extended in their natural language capability and convenience of use. Furthermore, the unique integration of functional and logic programming paradigms in a query language offers considerably more power and flexibility than either singular approach.

In Sect. 2, we introduce Two-Level Grammar. Section 3 shows how TLG may be used for querying of databases and knowledge-bases. The implementation is briefly described in Sect. 4, and we conclude with Sect. 5.

## 2 Two-Level Grammar

Two-Level Grammar (TLG) was originally developed as a specification language for programming languages [13]. The two levels of the grammar are two context-free grammars interacting in a manner such that their combined computing power is

---

\* Part of this research was carried out while this author was a Visiting Scientist at the IBM Palo Alto Scientific Center, Palo Alto, California, U. S. A.

equivalent to that of a Turing machine. Edupuganty and Bryant [5] developed a practical method of using TLG as a programming language by showing that the two levels of the TLG notation could be formulated as a set of domain definitions and the set of function definitions operating on those domains.

The type declarations of a TLG program define the domains of the functions and allow strong typing of identifiers used in the function definitions. Domains may be structured as linear data structures (regular sets), or be configured as tree-structured data objects (context-free languages). The standard structured data types of product domains (tuples), sum domains (discriminated unions), and function domains (lambda abstractions), modelled after the ML functional programming language [7], may be treated as special cases of these. Type declarations have the following form:

```
IDENTIFIER-1, IDENTIFIER-2, ..., IDENTIFIER-m ::  
  data-object-1; data-object-2; ...; data-object-n.
```

where each data-object-*i* is a combination of domain identifiers and constants, which taken together form the type of the left-side identifiers. Type declarations may be optional, in which case types are inferred from variable usage.

The function definitions are the main part of a TLG program. Their syntax allows for the semantics of the function to be expressed using a structured form of natural language. Function definitions take the form:

```
function-predicate : sub-function-1, sub-function-2, ..., sub-function-n.
```

where  $n \geq 0$ . Function predicates are a combination of constants, usually phrases in natural language, and domain identifiers, which correspond to variables in a conventional logic program. Predefined functions in the TLG programming language include arithmetic and logical operations, list operations, control operations to provide function sequencing, and Zermelo Frankel (ZF) set expressions, all of which are themselves directly TLG-definable.

### 3 Two-Level Grammar Query Language

Database query language development has closely followed advances in programming languages [1], resulting in the functional [3], logic [8], and object-oriented [6] models. While there have been successful attempts to use a combined paradigm in programming, such attempts have not been applied to database query language design. The unique aspect of TLG is its combination of functional programming, logic programming, and natural language. The natural language aspect is a significant advance over other functional/logic programming systems (e.g., see [4]), particularly as part of a database query language, which is mainly used by non-computer specialists.

The primary query language enhancements presented are: 1) addition of a well-defined type system to relations, 2) embedding SQL commands into TLG as predefined functions, 3) the ability to define structured objects, 4) the use of TLG rules to define deductive database rules, and 5) the construction of natural language queries using TLG. Each of these extensions is described in the following subsections.

#### 3.1 Database Object Type System

TLG, being a strongly typed language, has the capabilities of defining the types of all objects that might be used in relations. Furthermore, the structuring of the

types allow tuple components to be more complex than ordinary relational database values. We illustrate these concepts with an example based upon [11].

Consider the definition of sets of tuples for the customers of a department store, the orders they place, the items that comprise these orders, and the suppliers of those items. These may be defined at several levels. At the pure relational level of tuple sets, we may define:

```
CUSTOMERS :: {name STRING address STRING balance INTEGER}*.
```

```
ORDERS :: {number INTEGER date STRING customer STRING}*.
```

```
INCLUDES :: {order number INTEGER item STRING quantity INTEGER}*.
```

```
SUPPLIES :: {name STRING item STRING price FLOAT}*.
```

Note that the `{TUPLE_TYPE}*` notation means that there may be zero or more occurrences of objects of type `TUPLE_TYPE` (i.e., a set of `TUPLE_TYPE`). The interrelationships between these items may then be computed by joining the different tuple sets. Alternatively, we may construct these types in a more hierarchical way, such that the ownership relations are explicitly specified.

```
CUSTOMERS ::
```

```
  {name STRING address STRING balance INTEGER orders ORDERS}*.
```

```
ORDERS :: {number INTEGER date STRING includes INCLUDES}*.
```

```
INCLUDES :: {item STRING quantity INTEGER}*.
```

```
ITEMS :: {item STRING price FLOAT}*.
```

```
SUPPLIES :: {name STRING supplies ITEMS}*.
```

To access the components of the hierarchy requires a powerful query mechanism. The functional characteristics of TLG facilitate the construction of such queries.

### 3.2 Embedding SQL Into TLG

TLG may be extended to allow SQL queries by defining an SQL grammar with TLG type SQL and a set of rules to execute the SQL queries, each of the form “sql SQL,” where SQL is some SQL statement defined by our SQL type grammar. The rules of the type grammar verify that the query is syntactically correct. The sql rules will then verify semantic consistency and then execute the query, returning the appropriate set of tuples requested.

An SQL query to find out the items which are ordered by customer John is given as:

```
sql select name from supplies where item in
  (select item from includes where order number in
    (select number from orders where customer = John)).
```

Queries need not be so detailed in the specification of the navigation path between relations, as in the universal relation model [12].

```
sql select name from supplies where customer = John.
```

is substantially simpler and requires no detailed knowledge on the part of the user of the database structure. The sql rules will then deduce the necessary set of relations to be joined to evaluate the query. Unfortunately there are limits on the degree to which queries can be stated in such a general way. For example, if the query requests attributes which are ambiguous, then an exact rule in the TLG database cannot be matched and the user is requested to provide more precise information.

### 3.3 Queries Over Structured Objects

In addition to querying relation tables using SQL, we may also write TLG queries for the components of complex objects. Referring to the object declarations earlier, suppose we wish to find all customers who have ordered caviar. We first define a function to determine if a particular item appears in the INCLUDES set.

- item STRING appears in item STRING quantity INTEGER INCLUDES.
- item STRING1 appears in item STRING2 quantity INTEGER INCLUDES :  
STRING1 /= STRING2, item STRING1 appears in INCLUDES.
- item STRING appears in EMPTY : false.

This function consists of three rules: the first which matches STRING in the item field of the first tuple, the second which does not match the first tuple so recursively checks the remaining tuples, and the third which is invoked if the tuple list is exhausted before a match is found. These rules follow the Datalog declarative style of being order-independent. We may similarly determine if the item appears in the ORDERS set.

- item STRING1 appears in  
number INTEGER date STRING2 includes INCLUDES ORDERS :  
item STRING1 appears in INCLUDES.
- item STRING1 appears in  
number INTEGER date STRING2 includes INCLUDES ORDERS :  
not item STRING1 appears in INCLUDES,  
item STRING1 appears in ORDERS.
- item STRING appears in EMPTY : false.

Now the ZF expression  $\{(s_1, s_2, i, o) \mid (s_1, s_2, i, o) \in c, \text{“caviar” appears in } o\}$  may be used to determine all such customers. It is expressed in TLG as:

- list all name STRING1 address STRING2 balance INTEGER orders ORDERS  
from CUSTOMERS such that item caviar appears in ORDERS.

These rules show the power of combining functional and logic programming notions into a single paradigm for database querying. The natural language aspect of TLG also facilitates the ease with which such queries may be constructed.

### 3.4 Deductive Database Rules

In the previous section, we have seen the capability of TLG to define rules. The same method may be used to define the rules of a knowledge-base or deductive database. This is illustrated with an example, adapted from [11].

- the ancestor of PERSON is ANCESTOR :
  - the parent of PERSON is ANCESTOR.
- the ancestor of PERSON is ANCESTOR :
  - the parent of PERSON is PARENT,
  - the ancestor of PARENT is ANCESTOR.

Ordinarily, PERSON, PARENT and ANCESTOR will be defined as sets of constants representing the individuals of the database (all of the same type). However, since functions may be data types in TLG, we may also allow these to be {the parent of}\* PERSON. This would allow the following query:

- the ancestor of john is the parent of the parent of X.

with X being instantiated to all grandchildren of the ancestors of john. We may also define functions using lambda abstractions and compose them to construct other functions. In addition, because TLG is rule based and uses logical variables, we may specify inverses of functions by reversing the directionality in which variables are used in function calls.

### 3.5 Natural Language Queries

Although TLG queries should generally match the definitions in the rule-base, we may use more flexible natural language queries as well. The method works because the TLG database is stored in a natural language form and we may employ a partial matching algorithm to the stated query to determine which tuples are being requested. For example, we can state the queries of the previous sections as follows:

print the names of suppliers of items ordered by John.

print the customers with orders of item caviar.

The algorithm to answer these queries requires that sufficient information be provided to give a unique match or the user is requested to provide more information. In general, the more specific the data which is stored, the more general the queries may be to match successfully, but as the data becomes more general, match conflicts and ambiguities may also occur more frequently with queries which are not fully specified. The natural language processing algorithm is detailed in [10].

## 4 Implementation

The Two-Level Grammar rule-base and queries may be implemented in three different ways: 1) interpretation using parsing automata, 2) preprocessing into Prolog, and 3) transformation into C. We confine our discussion to the second and third options since they are the most efficient.

TLG programs can generally be simulated by Prolog so it is possible to use TLG as a front-end for Prolog, although all type checking is translated into rules which are executed dynamically instead of statically. All objects are fully type-tagged in the target Prolog program. If no rules are used (i.e., the system is a database system instead of a knowledge-base system), then all queries are translatable into SQL relational operations directly. If the functions used in the TLG rule-base are only first-order functions (i.e., no lambda abstractions), then the rule-base translation into Prolog will be consistent with the Datalog logic model having well-defined semantics in terms of relational algebra (e.g., see [11]). Higher-order functions may also be used. These are implemented by calls to Prolog reduction rules [9].

It is also possible to use a program transformation procedure to produce programs in C which implement the TLG rule-base. All type checking is done statically in the transformation so the resulting C code is quite efficient. Because the TLG rule-base may be defined at three different levels: 1) SQL, 2) logical rules with first-order functions only, and 3) logical rules with higher-order functions, the transformation procedure can take advantage of the most efficient translation for each level. With each being translated into C, TLG rule bases may interface with a variety of other application procedures. For further details of the transformation scheme, see [2].

## 5 Conclusions

We have shown how Two-Level Grammar may be used as a query language for database and knowledge-base systems, greatly extending the capabilities of current query languages. The language provides powerful yet elegant ways of defining rules which may then be queried using a form of natural language. Existing database and knowledge-base systems developed using SQL or Datalog may be easily integrated into Two-Level Grammar. Furthermore, any system designed using TLG may be efficiently implemented through translation into C.

For future research, we expect to integrate object-oriented features into the TLG query language so as to have a combination of functional, logic, and object-oriented paradigms. We also need to address other aspects of a complete database management system besides the query language, such as concurrency control and the ability to distribute the database. Since TLG provides a great deal of inherent parallelism, we expect that these extensions will be very natural.

*Acknowledgement.* The authors are grateful to the referees for their helpful suggestions in clarifying the paper.

## References

1. Bancilhon, F., Buneman, P., eds.: Advances in database programming languages. New York: ACM Press, 1990
2. Bryant, B. R., Pan, A.: Formal specification of software systems using Two-Level Grammar. Proc. COMPSAC '91, 15th Int. Computer Software and Applications Conf., 1991, pp. 155–160.
3. Buneman, P.: Functional programming and databases. in Research topics in functional programming, ed. D. A. Turner. Reading, MA: Addison-Wesley, 1990
4. DeGroot, D., Lindstrom, G.: Logic programming: Functions, relations, and equations. Englewood Cliffs, NJ: Prentice-Hall, 1986
5. Edupuganty, B., Bryant, B. R.: Two-Level Grammar as a functional programming language. Comput. J. **32** (1989) 36–44
6. Kim, W., Lochovsky, F. H., eds.: Object-oriented concepts, databases, and applications. New York: ACM Press, 1989
7. Milner, R., Tofte, M., Harper, R.: The definition of Standard ML. Cambridge, MA: MIT Press, 1990
8. Minker, J., ed.: Foundations of deductive databases and logic programming. Los Altos, CA: Morgan Kaufmann, 1988
9. Pan, A., Bryant, B. R.: Logic programming implementation of functional programming languages. Proc. TENCON '89, 4th IEEE 10th Region Int. Conf., 1989, pp. 174–178.
10. Pan, A., Bryant, B. R.: A natural language database front-end based on Two-Level Grammar. Proc. Int. Conf. for Young Computer Scientists, 1991, pp. 490–494.
11. Ullman, J. D.: Principles of database and knowledge-base systems, Rockville, MD: Computer Science Press, 1988
12. Vardi, M. Y.: The universal relation data model for logical independence, IEEE Software (1988) 80-85.
13. van Wijngaarden, A.: Revised report on the algorithmic language ALGOL 68. Acta Inform. **5** (1974) 1–236

This article was processed using the  $\LaTeX$  macro package with LLNCS style