A SYNTACTIC DATABASE MODEL

by

Dennis Heimbigner

CU-CS-283-84 December, 1984

Department of Computer Science
University of Colorado
Boulder, Colorado 80309

A Syntactic Database Model

Dennis Heimbigner
Department of Computer Science
University of Colorado
Boulder, Colorado 80303

ABSTRACT

This paper proposes an amalgam of compiler technology with relational database techniques as a way to provide a database model that can easily handle recursive relationships. Such relationships are common in databases for VLSI, software engineering, and text processing. This model is called the Syntactic Database Model. In this model, the schema is represented by an attributed grammar and the database is a parse tree conforming to the grammar. Sets of attributes are attached to the nonterminals of the grammar. The grammar explicitly represents the recursive properties of the data, and the attributes represent all other associated data. The model provides for sharing of subtrees, update of the database, and computed relational attributes. The paper describes the data model, a language for manipulating syntactic databases, and some proposals for efficient storage of the database.

1. Introduction

A major thrust of current database research involves the application of database technology to applications other than the traditional business-oriented ones. Some of these applications include software engineering [Powell 83, Ceri 83], VLSI [Mcleod 83], and text processing [Stonebraker 83]. We claim that all of these applications have a need to represent a strong recursive property. In software engineering, recursion is in the form of program dependencies. In VLSI, cells are composed of smaller cells. In text processing, documents contain paragraphs, which in turn contain sentences.

This paper proposes an amalgam of compiler technology with relational database techniques as a way to provide the structures and operations needed by such applications. The compiler technology (attributed grammars [Knuth 68]) provides the recursion while a Relational database model provides the database manipulations.

2. Representing Recursive Relationships

Most attempts to apply databases to VLSI, software engineering, and text processing applications have relied upon the relational database model as their primary data structuring formalism. A number of deficiencies in the model have become apparent in the process of designing these applications. In particular, relations have difficulty in representing recursive relationships.

Often, one "object" in the database system is known to be a part of another object. The fact that a document contains sections, sections contain paragraphs, paragraphs contain sentences, and sentences contain words, is difficult to capture in a relational system. It requires a relation for each level and also some sort of link between the levels. In the relational model, these links are usually represented by synthetic identifiers that serve as logical pointers These kinds of explicit pointers would seem difficult for users to understand. Object-based database models can provide some help because the links, while present, are not so explicit. Still, we might argue that treating paragraphs, sentences, and words as objects is not much better than thinking about sentence identifiers.

The transitive closure operation is closely related to the recursion property. Given a recursive relationship for documents, it is natural to ask for all the sentences in a document. In such recursive structures as documents this requires a transitive closure, and this in turn requires a join sequence whose length may not be statically bounded. Note that object models also suffer from a similar problem; usually they do not have a built-in transitive closure operation.

The notion of recursive relationships can be used to represent a number of similar properties. The "derives" relationship or "depends on" relationship is used in the software engineering examples to represent the fact that one program is derived from another. The "contains" relationship (for documents) is another recursive relationship

that could be modeled by the syntactic model of this paper.

3. A Syntactic Approach

In light of these problems of the relational models, it seems reasonable to consider some alternative that embeds the recursion property in the model. Some attempts have been made to do this to the relational model [Stonebraker 83, Powell 83], but these attempts are awkward and tend to destroy the clean structure of the relational model.

We propose to construct a database model around the notion of (modified) context-free grammars. In this model, the schema is represented by an attributed grammar and the database is a parse tree conforming to the grammar. Sets of attributes are attached to the non-terminals of the grammar. The grammar explicitly represents the recursion property of the data, and the attributes represent all other associated data.

Our primary example will be a software engineering database of programs (modeled after the Unix Make program [Feldman 79] or the Odin facility [Clemm 84]). In our version of "Make", we assume that programs depend upon other programs, and that these dependencies are modeled by the recursion property of our syntactic model. Suppose that we have a set of programs, A through F, with the dependencies represented in figure 1. This figure shows that program A depends upon programs B and C. Similarly, program F depends on program C. In both cases, program C depends on program D. We would like to represent this and other trees using a grammar. For generality, this grammar must not refer to the actual programs in the productions[1]. To do this, we encode the trees of figure 1 into equivalent linear forms:

A(B C(D)) and F(C(D)).

[1]Actually, something like W-grammars would allow us to do it. W-Grammars, however, are not computationally attractive.

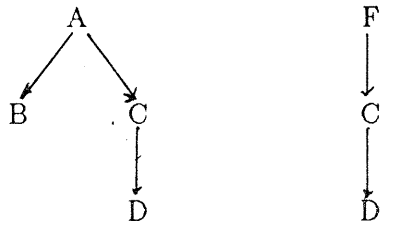

Figure 1. A Program Dependency Tree

Figure 2 shows a grammar for parsing this linear form. It shows that a database is a project, which is in turn a sequence of programs. Each program specifies a (possibly empty) sequence of other programs on which it depends. Figure 3 show the parse tree that results from applying the grammar to the linear form of the database of program dependencies.

3.1. Grammar Form

The particular grammar format used here is a variation on Unix YACC. A typical production in the grammar consists of a left side, which is the name of a nonterminal symbol. The right side of the production is a collection of alternatives

```
project     : progseq

program     : progname '(' progseq ')'
            | progname

progseq     : program
            | program progseq

progname    : /*string of letters*/
```

Figure 2. Software Engineering Grammar.

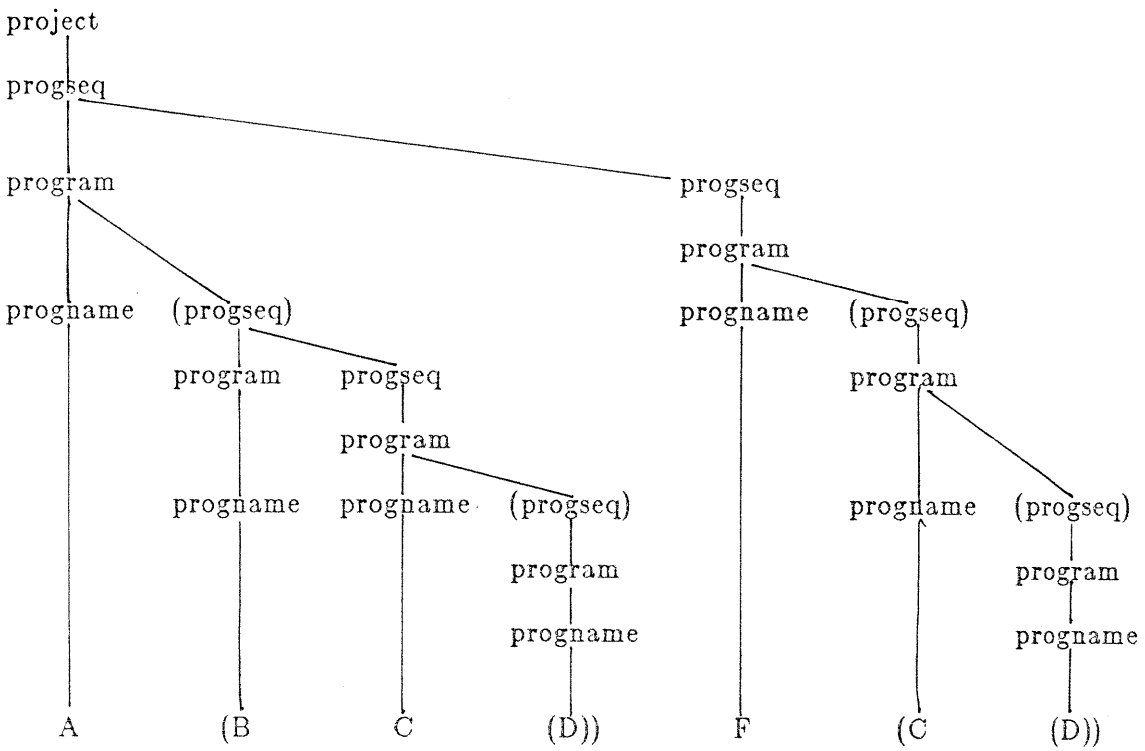

Figure 3. Software Engineering Database tree

separated by the symbol "|". Each alternative is, in turn, a concatenation of non-terminals and constant strings in double quotes.

3.2. Database Form

The database at any moment corresponds to a possible parse tree produced by applying the grammar to an ordered set of primitive objects (the leaves of a tree). For the Make database, the objects are the names of files.

The leaves of the tree may be rather unconventional in that they need not be strings. Rather, they could be any set of nested objects. In the case of the software engineering database, they leaves might be the actual programs. It should be noted, though, that most objects have some form of string identifier, and it is usually most

convenient to work with those identifiers rather than the actual objects. This convention will be followed in this paper.

4. Attributing the Parse Tree

The tree format alone is not powerful enough to easily represent all of the relationships that we might wish to store in a software engineering database. In the Make database, we must store extra information such as the following:

1. the object file corresponding to some source file,
2. the modification and creation dates of source files,
3. the author for a given program,
4. the mail address of the program authors,
5. the result of executing the object file of a particular source file.

Some of these attributes (1-3) can be considered to be direct properties of nodes of the tree. Others (4-5), are either independent of the tree or indirectly dependent upon it[2].

In order to handle attributes, and especially indirect attributes, we will augment our tree database with a relational database system.

The two formats (trees and relations) are connected by "attaching" some of the attributes to the nodes of the tree. This attachment is accomplished by defining some special attributes whose domain is the instances of some non-terminal type in the grammar. Each actual instance of the non-terminal in the parse tree is assumed to have a unique id. Certain relations are defined to include such attributes, and these relations connect the parse tree nodes to the other attributes in such a relation.

As an example, consider the tree in figure 4. It shows a partial parse tree with some of the node identifiers in angle brackets. Figure 5 shows a particular relational schema involving several attributes. Note that the schema contains relations involving nodes (the program attribute), direct attributes (object-file and name), and indirect

[2] The indirect attributes could become more direct at the cost of normalization.

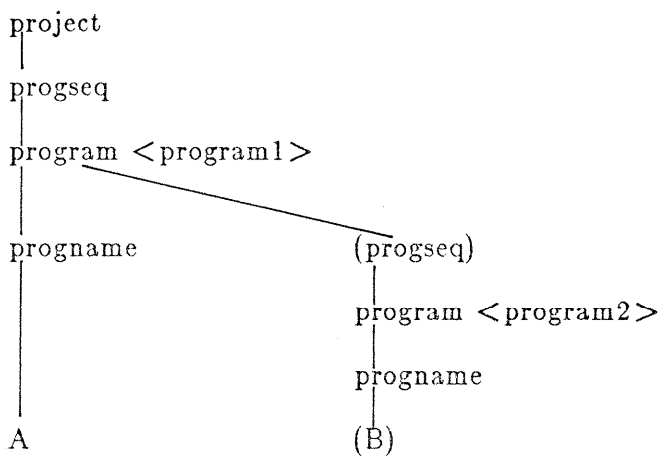

Figure 4. An Attributed Tree

attributes (addr). The OBJECT relation maps a program node (via its unique-id) to the name of its corresponding object file. The AUTHOR relation maps program nodes to author names. The LOCATION relation maps author names to mailing addresses. Figure 6 shows a possible contents of the relations defined by the tree of figure 4 and the schema of figure 5.

5. Querying the Syntactic Model

Given this combination of trees and relations, what kind of queries might one ask? Some possibilities are as follows:

OBJECT(program,object-file)
AUTHOR(program,name)
LOCATION(name,addr)

Figure 5. Software Engineering Relations

AUTHOR	program	name
	program1	Smith
	program2	Smith

OBJECT	program	object-file
	program1	A.obj
	program2	B.obj

LOCATION	name	addr
	Smith	smith@boulder
	Jones	jones@tut

Figure 6. Relational Database Instance.

(1) Queries involving the recursive relationship, such as "retrieve all of the programs depending upon program C," or "retrieve all of the programs upon which program A depends."

(2) Queries involving the direct attributes of various tree nodes, such as "retrieve the author of program A, " or "retrieve the authors of all the programs depending upon program C."

(3) Queries involving indirect attributes, such as "retrieve the mail addresses of all of the authors of all programs depending upon program C."

The first class of queries involves traversals and transitive closures over the tree alone. The other two classes involve not only the nodes of the tree, but also attributes of those nodes. The last case additionally involves a join connection between immediate attributes (the author's name) and an indirect attribute (the author's mailing address) via the LOCATION relation.

5.1. A Query Language

It is fairly easy to define a QUEL-like language to pose queries against the syntactic model. We introduce a star operator ("*") to represent transitive closure along the recursive relationship. In the queries below, the dot operator is overloaded to refer to attributes of relation variables and also to children of node variables (such as p1 or p2). Using this notation, the above queries are translated as follows:

Retrieve the programs depending on program C

```
range of p1 is program
range of p2 is program
Retrieve p1 where
p1.*.p2 &
p2.progname = "C"
```

Retrieve the programs upon which program A depends

```
range of p1 is program
range of p2 is program
Retrieve p1 where
p2.progname = "A" &
p2.*.p1
```

Retrieve the author of program A

```
range of p1 is program
range of a is author
Retrieve a.name where
a.program = p1 &
p1.progname = "A"
```

Retrieve the authors of the programs depending upon program C

```
range of p1 is program
range of a is author
range of p2 is program
Retrieve a.name where
a.program = p1 &
p1.*.p2 &
p2.progname = "C"
```

Retrieve the mail addresses of all of the authors of all programs depending upon program C

```
range of p1 is program
range of a is author
range of l is location
range of p2 is program
Retrieve l.address where
l.name = a.name &
a.program = p1 &
p1.*.p2 &
p2.progname = "C"
```

6. Update in the Syntactic Model

Update in the syntactic data model is performed by specifying modifications to the database tree. A parse tree is formed by expanding non-terminals by one of the alternative definitions for that non-terminal. A modification (insertion or deletion) involves replacing an existing expansion by one of its alternatives.

Consider the problem of modifying the example of figure 4 so that program A depends upon program C as well as program B. In this example, the lowest progseq node was expanded using the first alternative of the grammar. Adding a dependency on program C may be accomplished by changing the expansion choice for progseq to use the second alternative. This leads to the tree of figure 7a. In this new tree, there is room to fit a new program under the extra progseq node. Once the alternate expansion has been made, the user must provide the data (a program name in this case) to complete the tree (figure 7b).

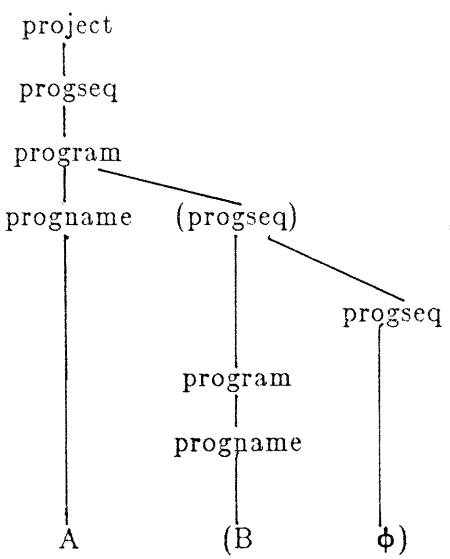

Figure 7a. Initial Stage of Update

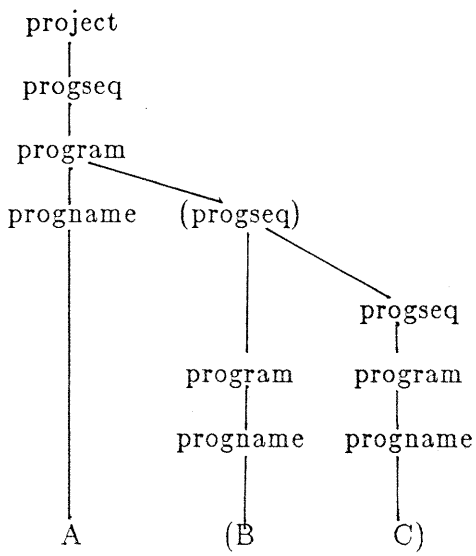

Figure 7b. Completion of Update

A key issue for update is deciding how to map from the old expansion to the new one. There appear to be two choices: total reparsing and semi-automatic mapping. The first method involves collecting the existing data which, when parsed, forms the original subtree. This data is presented to the user for modification (typically by a text editor) to include the new information. This new information is reparsed and the new subtree set into place in the tree. In this case, it is not strictly necessary for the user to specify the new expansion since it is automatically derived from the parsing process.

The second method for handling the change involves providing rules for placing the old subtree into some position in the new expansion and then asking the user to fill in all the other slots. In the previous example, our rule would say that the the subtree under the original program is moved to the program node in the new expansion, and the remaining node (progseq) must be filled from user data. There are a number of problems with this:

(1) There may be no completely consistent rules. In our example, the rule seems obvious. But consider what would happen if the new program was to be placed *before* the existing program (i.e., A(C,B)). In that case, the "obvious" rule would not work.

(2) What if the new expansion provides for two or more new non-terminals and the user only provides data for one of them. We could allow this and just use some form of null data. But, each expansion represents an elementary constraint on the data format, and it would seem unreasonable to allow the user to violate those constraints[3].

7. Extending the Syntactic Model

Two extensions to the basic model are introduced to increase the utility of the system. These extensions are (1) derived attributes, and (2) automatic derivation of joins.

7.1. Derived Attributes

It is often convenient to allow the values of some attributes to be calculated (derived) from other attributes via arbitrarily complex functions. For example in the Make database, it is reasonable to ask for the executable version of a program. This entails compiling it and storing the result in some file. Effectively, the name of the executable file is a derived attribute of the specified program node.

Derived attributes are defined in the system in terms of derived relations. A derived relation is specified to have a functional or multi-valued dependency that involves all of its attributes on one side or the other of its dependency. For example, the relation EXECUTABLE(program, objectfile) has a functional dependency: program

[3] One might speculate on the possibility of *syntactic normalization* to limit updates.

→ objectfile. A function in some language is also attached to the relation. The function takes as input some set of values from the attributes on the left side of the dependency, and calculates the values for the attributes on the right side. Thus, the function attached to the EXECUTABLE relation is designed to take a program attribute as input and calculate the associated object file, whose name it returns as the value of the objectfile attribute.

Tuples may be inserted into a derived relation with a dummy value occupying the derived attribute position. The actual derivation is performed when the derived attribute of a tuple is accessed. At this time, the function is invoked with the appropriate inputs. The result it returns is stored in the derived attribute field as its current value.

Two specialized forms of derived attributes correspond to the inherited and synthesized attributes of grammar theory [Knuth 68]. It should be noted that inheritance here is along the recursive relationship and is quite different from the subtype inheritance of semantic models.

A synthesized attribute is one that is derived for a node in terms of the attributes of the children of that node. The executable attribute (above) is also a synthesized attribute since it is synthesized from the object files for all the files on which it depends.

An inherited attribute is one that is derived for a node from the attributes of its parent. Reference date is an example of inheritance. If a program is referenced, then that may indicate that the files on which it depends are considered referenced also.

7.2. Automatic Join Derivations

It is convenient, though not essential, if a user can avoid specifying the join conditions in his (or her) query. The work on Universal Relations [Maier 83] has shown

that join sequences can be automatically determined. This concept is introduced into this model to more closely mimic one of the capabilities of the real Make and Odin programs. Odin can, for example take a request that says "give me the result of executing the file test.c" and determine that it must actually compile test.c before it can execute it to get the results. In the syntactic model, this corresponds to calculating the following expression:

```
range of c is COMPILED
range of e is EXECUTE
Retrieve e.result where
        c.program.progname = test.c &
        c.objectfile = e.objectfile
```

COMPILE is a derived relation with dependency program → objectfile, and EXECUTE is a derived relation with the dependency objectfile → results).

Using the automatic derivation of joins, the query might be written more simply as

```
Retrieve results where
        program = test.c
```

The only possible path (in this example, anyway) between program and results is through the join of COMPILE and EXECUTE (which also obviates the need for the range variables).

8. Storage Representations

The syntactic data model requires two major storage structures: the parse tree and the attributes. With some modifications for derived relations, the storage for attributes can be performed using a conventional relational database

Efficiently storing the parse tree is another matter. Compilers normally do not have this problem since the tree is usually small enough to fit in memory. The syntactic model system, by contrast, must allow efficient random access, and in particular, the storage structure must support transitive closure both up the tree (from child to

parents) and down the tree (from parent to children). In addition, the structure must allow insertion and deletion of new subtrees.

8.1. Parse Tree Storage

We choose as our basic storage structure a pre-order listing of the nodes of the parse tree[4]. The pre-order is actually stored in a linked list of blocks with multiple nodes per block. New subtrees can be inserted anywhere in the tree by inserting new blocks into the appropriate place in the linked list.

Each node in the tree contains the following information:

(1) The node non-terminal type,

(2) Pointers to its children,

(3) Pointer to its parent,

(4) Pointers to the limiting leaves of the subtree rooted at this node.

Given this structure, there are two obvious alternatives for computing transitive closures in the form of the star operator.

(1) Brute force search of all nodes may be used. This is most effective for downward closure because only the subsection of the tree corresponding to a subtree needs to be searched. This subsection is determined by the values stored in the parent node of the subtree. Note that this is not effective for upward closures because it is hard to separate sibling nodes from parent nodes.

(2) Path following is effective for upward closure. Since each node points to its parents, this sequence may be followed to find all parents of a node. Again, notice that this is less effective for downward closure because all of the descendant paths must be traced, which involves backtracking.

---

[4]We will specify pre-order, but the structure could equally well use post-order.

8.2. Storage for Inherited and Synthesized Attributes

As mentioned above, attributes will be stored as part of a conventional relational database system. Inherited and synthesized attributes are treated specially. For efficiency, it may be reasonable to store the derived value once it has been calculated and recalculate when changes occur. This is handled by storing change bits with the nodes to flag changed attribute values. When a node changes a synthesized value, it sets a bit in its parent node. If, later, the parent's value is to be extracted, the bit serves as a signal that the value must be re-computed. Similar actions occur for inherited attributes.

8.3. Subtree Sharing

In many situations, there will be duplicated subtrees in the parse tree. For example in the Make database, any two programs that use a common third program will have duplicated subtrees in the database. As a storage optimization, we allow the tree to become a DAG by sharing these common subtrees. To do this, certain non-terminals in the grammar are marked as sharable. For a sharable node, the user must specify a key for that that kind of non-terminal. Each alternative of the non-terminal must have a specified key, and in each case, the key must lead to a leaf that is a string or can be converted to one.

When a database user alters the parse tree by expanding a node for a sharable non-terminal, several steps occur.

(1) The user is asked for the key for the sharable node.

(2) The parse tree is searched for an existing node with a matching key.

(3) If the match is found, then the newly expanded node is equated to the matched subtree and the user is informed that sharing is being used.

(4) If the match fails, then the expansion occurs normally.

Going back to the grammar of figure 2, we could mark the non-terminal "program" as sharable. The occurrences of "progname" in the right side of the definition of "program" would be marked as the key for sharing. Looking again at figure 3, we can see that it has a subtree that can be shared ("C(D)") because two of its nodes have a common key value, namely the string "C". Figure 8 shows the result of applying sharing to the tree in figure 3. The dotted line represents the link formed as a result of subtree sharing.

Although sharing may save storage space, it also causes a number of problems in the implementation.

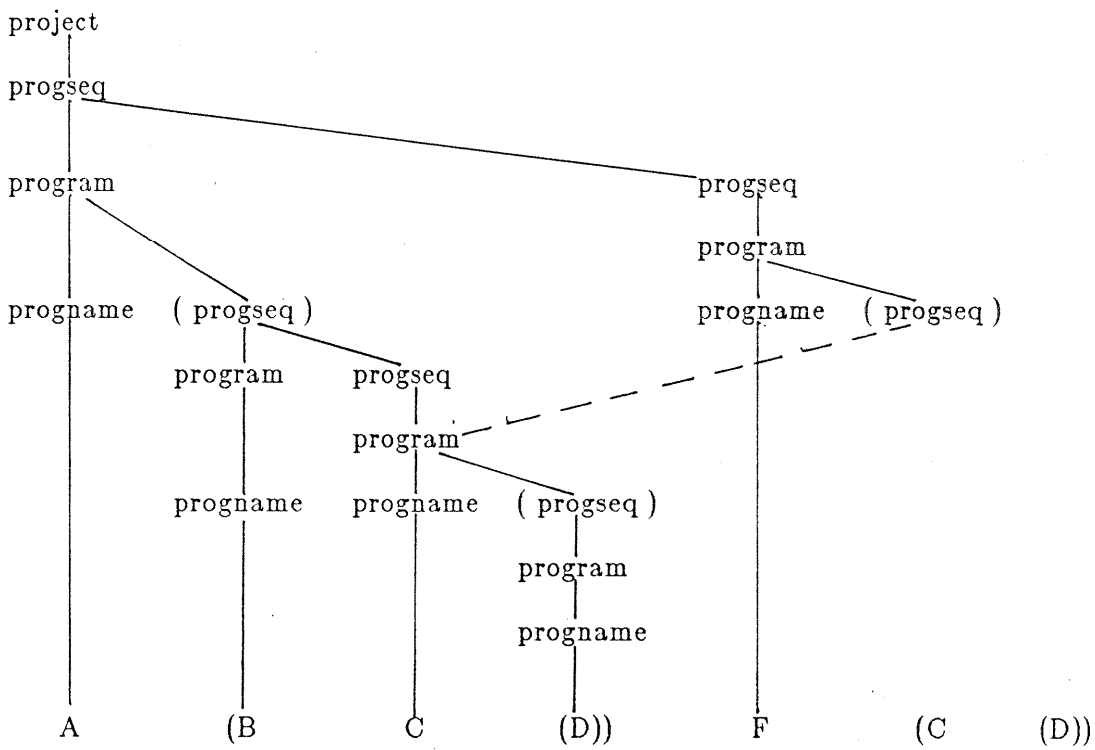

Figure 8. Example of Subtree Sharing

(1) If the key of a shared node is changed, then what should be done? Should the tree be duplicated (i.e., unshared) or should it be left as is? We assume the tree is left as is, and the user is provided with a way to explicitly "unshare" a node.

(2) If the shared node has inherited attributes, then it may inherit conflicting values from its parents. In this case, we need some rule for deciding the inheritance. For example, reference date inheritance could be resolved by choosing the later of the two dates. In general, this must be handled by a user specified decision procedure.

9. Related Work

The principal source for this work is the author's previous work on a syntactic interface to external databases (the system is called Diverse) [Heimbigner 84a] and on syntax-directed databases (SDDB) [Heimbigner 84b]. Among its many features, Diverse attempted to extract data from external databases by parsing their text output. The SDDB attempted to provide a relational interface to text files by parsing the text and storing, into the tuples, appropriate pointers into the text. The syntactic model represents another attempt to combine syntactic structures with database technology.

Most other attempts to handle recursion have tried to force everything into a single structure (relations). Two specific examples are the document database systems [Stonebraker 83], and the software environments [Powell 83, Linton 84, Ceri 83].

In this paper, we argue that relations are not useful for recursive relationships. Rather, we propose the use of two complementary structures: trees and relations. Of course, trees do not have a good reputation as database models. This is principally because they are identified with IMS style databases, which are notoriously complicated. As compiler writers have shown, (parse) trees can be as formal and useful as

relations for structuring information.

Syntax directed editors [Teitelbaum 81] and data structure editors [Fraser 81] were perhaps the first example of the use of syntactic structures to represent information. In this case, it was a partially constructed program. In some sense, the work reported here may be viewed as an attempt to generalize such editors and extend them to a database framework.

Mention should also be made of some attempts to provide nested relations (non first normal form) (see, for example, [Jaeschke 82]). Such systems have concentrated more on multi-valued fields rather than recursion, and have not attempted to use syntactic methods.

Finally, the syntactic model owes a debt to the Odin project for providing a specific example of the utility of recursive relationship and the utility of automatic derivations.

10. Future Work

The first step is to actually implement a syntactic database system. This is being carried out in two steps. First, the interface is being implemented and a standard relational database is being used to provide storage. The second step will be to provide direct storage for the parse trees as discussed in section 7. Beyond the implementation, it will be important to apply the system to some software environments problems to verify its utility. We would expect to draw on the Odin system work and the Toolpack project [Osterweil 83] for examples.

References

[Clemm 84] Clemm, G. M., "ODIN - An Extensible Software Environment," University of Colorado, Department of Computer Science Technical Report CU-CS-262-84, 1984.

[Ceri 83] Ceri, S. and Crespi-Reghizzi, S., "Relational Databases in the Design of Program Construction Systems", SIGPLAN Notices 18(11):34-44 (November 1983).

[Feldman 79] Feldman, S. I., "Make - A Program for Maintaining Computer Programs", *Software - Practice and Experience,* 9(2):255:265 (April 1979).

[Fraser 81] Frase, C. W. and Lopez, A. A., "Editing Data Structures", *ACM Transactions on Programming Languages and Systems* 3(2):115-125 (April 1981).

[Heimbigner 84a] Heimbigner, D. M., "Towards an Integrated Environment for Accessing External Databases", *Proceedings of the Second ACM-SIGOA Conference on Office Information Systems,* Toronto, Canada, 25-27 June 1984.

[Heimbigner 84b] Heimbigner, D. M., "A Syntax-Directed Database System", University of Colorado, Boulder, Department of Computer Science Technical Report CU-CS-289-85, February, 1985.

[Jaeschke 82] Jaeschke, G. and Schek, H. - J., "Remarks on the Algebra of Non First Normal Form Relations", *Proceedings of the ACM Symposium on Principles of Database Systems,* 29-31 March 9182, Los Angeles, CA, pages 124-137.

[Knuth 68] Knuth, D. E., "Semantics of Context-free Languages", *Mathematical Systems Theory* 2(2):127-145.

[Linton 84] Linton, M. A., "Implementing Relational Views of Programs", *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Environments,* 23-25 April 1984. Available as SIGPLAN Notices 19(5):132-140.

[Maier 83] Maier, D. and Ullman, J. D., "Maximal Objects and the Semantics of Universal Relation Databases", *ACM Transactions on Database Systems* 8(1):1-14 (March 1983).

[McLeod 83] McLeod, D. and Narayanaswamy, K., *Database Week: Engineering Design Applications Proceedings of Annual Meeting,* San Jose, California, 23-26 May 1983.

[Osterweil 83] Osterweil, L. J., "Toolpack - An Experimental Software Development Environment Research Project," *IEEE Transactions on Software Engineering,* vol. SE-9, no. 6, pp. 673-685, 1983.

[Powell 83] Powell, M. L. and Linton, M. A., "Database Support for Programming Environments", *Database Week: Engineering Design Applications Proceedings of Annual Meeting,* San Jose, California, 23-26 May 1983, pages 63-70.

[Stonebraker 83] Stonebraker, M., Stettner, H., Lynn, N., Kalash, J., and Guttman, A., "Document Processing in a Relational Database System", *ACM Transactions on Office Information Systems,* 1(2):143-158, April 1983.

[Teitelbaum 81] Teitelbaum, T. and Reps, T., "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", *Communications of the ACM* 24(9):563-573 (September 1981).