A Simple Technique for Structured Variable Lookup

Geoffrey W. Gates and David A. Poplawski
Michigan State University

A simple technique for the symbol-table lookup of structured variables based on simple automata theory is presented. The technique offers a deterministic solution to a problem which is currently handled in a nondeterministic manner in PL/I and COBOL compilers.

Key Words and Phrases: symbol table organization, PL/I and COBOL structured variables
CR Categories: 4.12

Introduction

Two programming languages commonly used today, PL/I and COBOL, offer the programmer flexibility in defining his data layout by the use of structured variables. Figures 1 and 2 show a typical three-level structure from a PL/I program. Ordinarily various data attributes such as type and length would be specified for each name; however, for the purpose of this paper, the information is superfluous and will be omitted.

The tree structure in Figure 2 illustrates how nicely the structured variable delineates the hierarchical nature of the data. For instance, each ADDRESS must be made up of three lines. The primary advantage is the multitude of referencing methods for the actual use of the variables. Ordinarily, each item can be referenced by its tree name, that is, the list of all nodes down to and including the referenced node. For example, the city name would be referred to as

ADDRESS.THIRD_LINE.CITY

Making use of the hierarchical nature of the structured variable, the entire street address or second line could be referred to as

ADDRESS.SECOND_LINE

In this last example, ADDRESS is said to qualify SECOND_LINE, thus making the reference specific; and the entire reference is called a qualifier list.

All is not this simple since, in the general case, the entire tree name need not be given. The city name could also be referenced as

ADDRESS.CITY or THIRD_LINE.CITY

or even just CITY. This is included for the benefit of the programmer, but does nothing to aid the compiler writer. To further complicate the situation, each qualifier list is not necessarily unique. In the above example, ADDRESS could be either the entire three line address or simply the street number in the second line. This conflict could be cleared by requiring all names to be unique, however the only requirement which is actually imposed is that any particular qualifier list refer to a unique node. Thus ADDRESS would not be permitted even though the entire structure is legal. The rule is that only enough qualifiers to make a reference unique are required although others are allowed.

One of the basic functions needed in any compiler is the ability to maintain a symbol table and to look up variable name references in this table. In languages such as FORTRAN and ALGOL this is usually accomplished through the use of a sequentially ordered symbol table and some type of hash function for performing the table lookup. This is usually the most efficient solution. Figure 3 illustrates that this straightforward technique is not well suited to PL/I or COBOL. Where in FORTRAN or ALGOL each name appears only once (in a subroutine or block) with a single set of attributes, in this case each name appears twice and may very possibly have different attributes in each place. In other words, the actual symbol table entries cannot be *A* or *B* or *C* or *D*, but their uniquely qualified occurrences, such as *A.B.C*, *A.B.D*, etc.

The problem to be solved, then, is the recognition that while *A.B* is a valid reference because it is unique, *A.C* is not, since it may be either *A.B.C* or *A.C* (where *C* is at level 2).

Current Solution

The solution to this problem used in a typical compiler [1, 2] consists of two parts. First, a standard hash function is applied to the first (left-most) qualifier in the reference. Instead of getting an address in the

Authors' address: Computer Science Department, Michigan State University, East Lansing, MI 48823.

Fig. 1. PL/I structured variable, example 1.

```
DECLARE   1   ADDRESS
              2   FIRST_LINE
                  3   NAME
                  3   TITLE
              2   SECOND_LINE
                  3   ADDRESS
                  3   STREET NAME
              2   THIRD_LINE
                  3   CITY
                  3   STATE
                  3   ZIP_CODE
```

Fig. 2. Tree structure associated with the structured variable in Figure 1.

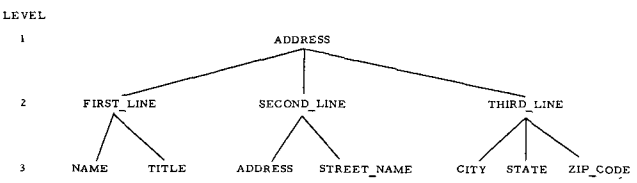


Fig. 3. Example 2.

```
DECLARE   1   A
              2   B
                  3   C
                  3   D
              2   C
                  3   D
          1   B
              2   A
```

symbol table which will give the attributes of the variable, the resulting address is an entry into a complicated structure which is simply a replica of the original structure with convenient links added. Figure 4 shows such a structure for example 2. In this figure, the left-most box in each row (boxes 1, 4, 7, and 10) would contain the actual display code name of the variable to avoid needless duplication. Each of the remaining boxes represents one uniquely qualifiable instance of the name. The dashed lines link different occurrences of the same name, the solid lines actually represent the hierarchical structure of the data, linking each name to one of its successors.

The structure shown in Figure 4 obviously contains all the information in the original structured variable and thus can be used to lookup any desired reference. The following procedure accomplishes just this.

1. Apply the hash function to the first qualifier in the list to find the entry point into the structure. This will point to one of the rows shown in Figure 4.
2. List all paths leaving all occurrences of this name as candidates. The remaining steps are repeated for each candidate.
3. Pick a candidate and delete it. Follow the path to its successor. Find the name. Compare this name with the next qualifier.
4. If the names compare, bypass the qualifier and add all paths out of the box to the list of candidates.
5. If the names do not compare, add all paths out of the box to the list of candidates.
6. A candidate is disqualified if the hierarchy is exhausted before the list of qualifiers is used up.
7. A candidate is added to the list of final candidates if its qualifiers are exhausted before the hierarchy is.

This process is repeated until all candidates are either disqualified or added to the list of final candidates. If there are no final candidates, the item is not defined. If there is one final candidate, the item is uniquely defined and referenced. If there is more than one final candidate, the reference is ambiguous and therefore not allowed.

As an example, the procedure may be applied to the reference *A.C*. After the application of step 2 above the list of candidates is: Box 2 path 1, Box 2 path 2, Box 3. Applying steps 3 to 7 on the first candidate, the new list is: Box 2 path 2, Box 3, Box 5 path 1, Box 5

Fig. 4. Typical symbol table data structure for example 2.

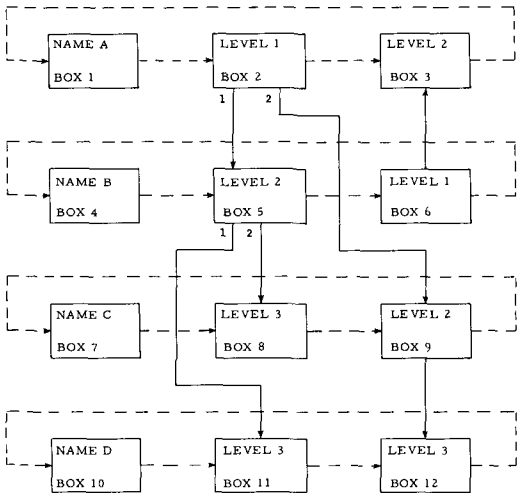


Fig. 5. Application of current method to *A.C*.

Iteration of 3 to 7	Candidates	Bypass qualifier
	Box 2 path 1 Box 2 path 2 Box 3	yes (A)
1	Box 2 path 2 Box 3 Box 5 path 1 Box 5 path 2	no
2	Box 3 Box 5 path 1 Box 5 path 2 Box 2 path 2 (final)	yes (C)
3	Box 5 path 1 Box 5 path 2 Box 2 path 2 (final)	no
4	Box 5 path 2 Box 2 path 2 (final)	no
5	Box 2 path 2 (final) Box 5 path 2 (final)	yes (C)

Ambiguous, either A. C or A. B. C

Fig 7. Application of current method to *B.C*.

Iteration of 3 to 7	Candidates	Bypass qualifier
	Box 5 path 1 Box 5 path 2 Box 6	yes (B)
1	Box 5 path 2 Box 6 Box 11	no
2	Box 6 Box 11 Box 8 (final)	yes (C)
3	Box 11 Box 3 Box 8 (final)	no
4	Box 3 Box 8 (final)	no
5	Box 8 (final)	no

Uniquely defined reference B. C.

Fig. 6. Application of curren method to *A.B.A*.

Iteration of 3 to 7	Candidates	Bypass qualifier
	Box 2 path 1 Box 2 path 2 Box 3	yes (A)
1	Box 2 path 2 Box 3 Box 5 path 1 Box 5 path 2	yes (B)
2	Box 3 Box 5 path 1 Box 5 path 2 Box 9	no
3	Box 5 path 1 Box 5 path 2 Box 9	no
4	Box 5 path 2 Box 9 Box 11	no
5	Box 9 Box 11 Box 8	no
6	Box 11 Box 8 Box 12	no
7	Box 8 Box 12	no
7	Box 8 Box 12	no
8	Box 12	no
9	--	no

Undefined reference

Fig. 8. Application of step 1 in the construction of the nondeterministic finite state automaton for example 2.

STATE \ QUALIFER	A	B	C	D
S	A	B	--	--
A	--	A. B	A. C	--
B	B. A	--	--	--
A. B	--	--	A. B. C	A. B. D
A. C	--	--	--	A. C. D
B. A	--	--	--	--
A. B. C.	--	--	--	--
A. B. D	--	--	--	--
A. C. D	--	--	--	--

Fig. 9. Final nondeterministic finite state automaton for example 2.

STATE \ QUALIFIER	A	B	C	D
S	A B. A	B A. B	A. C A. B. C	A. B. D A. C. D
A	--	A. B	A. C A. B. C	A. B. D A. C. D
B	B. A	--	--	--
A. B	--	--	A. B. C	A. B. D
A. C	--	--	--	--
B. A	--	--	--	--
A. B. C	--	--	--	--
A. B. D	--	--	--	--
A. C. D	--	--	--	--

Fig. 10. Deterministic finite state automaton for example 2.

COMBINATION	STATE \ QUALIFIER	A	B	C	D
S	1	2	3	4	5
A B. A	2	-	6	4	5
B A. B	3	7	-	8	9
A. C A. B. C	4	-	-	-	10
A. B. D A. C. D	5	-	-	-	-
A. B	6	-	-	8	9
B. A	7	-	-	-	-
A. B. C	8	-	-	-	-
A. B. D	9	-	-	-	-
A. C. D	10	-	-	-	-

path 2, and the next qualifier is not bypassed. The entire procedure is carried out in Figure 5. Figures 6 and 7 contain additional examples.

The obvious disadvantage of this method is the requirement for several paths through the qualifier list and the resulting enumeration of all possible paths. The internal symbol table structure directly reflects the original structured variable and is thus easy to construct. However, since the table is constructed only once while several references will be made to it, it would be better to put extra effort into the construction so that each reference requires only a single pass through the qualifier list.

Deterministic Solution

The solution to this is recognition of the fact that the original data structure for example 2 is just another method of describing a regular expression [3] and all possible legal qualifications are simply cases of this regular expression. Furthermore, if the finite state automaton which accepts the regular expression is constructed in a clever way, it will produce all possible final candidates in one pass through the qualifier list.

If we define Λ to be the string of length 0 and $\tilde{X}$ to be the regular expression $\Lambda \cup X$, that is, zero or one occurrence of the symbol X, then we can apply the language of regular expressions to example 2 producing:

$$A(\tilde{B}\ (\tilde{C} \cup \tilde{D}) \cup \tilde{C}\tilde{D}) \cup B(\tilde{C} \cup \tilde{D}) \cup C\tilde{D} \cup C \cup D \cup B\tilde{A} \cup A$$

While this representation does not convey any new information about this particular structure, it does allow the application of some of the basic results of automata theory to the problem. First, since the data structure can be represented by a regular expression [4, Th. 3.10], there is a nondeterministic finite state automaton which accepts the regular expression. This is in essence what the process described above and in Figures 5, 6, and 7 implements. However, due to the nondeterministic nature of the solution, enumeration of all solutions is required. However, by [4, Th. 3.3] if a nondeterministic machine exists, there is an equivalent deterministic machine, thus eliminating the need for enumeration.

The procedure for construction of the nondeterministic machine is as follows:

1. In addition to the unique start state, S, each state is labeled by the longest string of qualifiers which goes from S to the state. The interconnections representing these longest strings are filled in. (See Figure 8.)
2. Form the closure of the machine resulting from step 1. That is, since state S can reach state A, copy row A into row S. Repeat until there are no more copies to be made. (See Figure 9.)

The deterministic machine corresponding to this may be constructed as indicated in the proof of [3, Th. 3.3] and is shown for example 2 in Figure 10.

This machine is used by starting in state S and applying the qualifiers in a left-to-right order. The qualifier list is illegal if a required transition is not present in the machine. Otherwise, when the qualifier list is exhausted, if the deterministic machine is in a state which represents a single state from the nondeterministic machine, the item is correctly defined and referenced. If the final deterministic state represents two or more states from the nondeterministic machine, then the item is ambiguous.

The machine finally reached in Figure 10 may then be applied to various qualifier lists. For example, $A.C$ ends in state 4, $A.C$ and $A.B.C$, and is thus ambiguous.

The list $A.B.A$ gets as far as state 6 and has no transition and is therefore undefined. The list $B.C$ ends in state 8, $A.B.C$ and is uniquely defined.

Application to Cobol

The above discussion has been concerned with structured variables in PL/I. However, COBOL also has structured variables which are similar in every detail except the actual referencing. Where, in PL/I a reference would be made as $A.B.C$, going down the tree, in COBOL the same variable would be referred to as C OF B OF A.

At first, it would appear that the deterministic finite state automaton could be constructed in an analogous way. However, since the list of qualifiers goes from most specific to least specific this cannot be done. A machine can be constructed which will tell whether or not a particular reference is legal but will not give the specifically referenced variable. This is due to the fact that the process moves up the tree ending at the top or broadest reference. The solution to this problem is simply to stack the qualifiers as encountered in a LIFO manner. When the end of the qualifier list is reached, remove the qualifiers one at a time and apply to the machine constructed above.

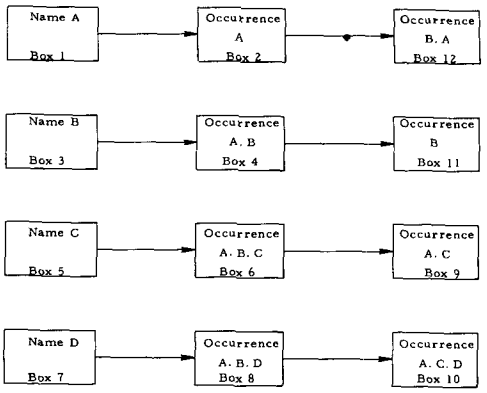
Implementation

The discussion above gives an explanation and motivation of the proposed method. As usual, this form is not always the most efficient for implementation and this is true here, too. In this case, an efficient construction algorithm can be arrived at by considering the form of the transition matrix for the final deterministic machine. The matrix appears to be constructed of several roughly upper triangular structures where the transitions in each row are a subset of those in the row above. Thus, it will be advantageous to construct the machine working up from the bottom of these triangular portions. The following construction method makes use of this information to construct the automaton and the symbol table simultaneously.

This method requires a pushdown stack called CURRENT_STATE, an expandable list called TRANSITIONS and some mechanism for making entries in the symbol table. Notice that the initial entry for each unique name (string of characters) can be made and looked up by use of a hash function or other traditional technique. CURRENT_STATE refers specifically to the top element of the stack and is initialized to 0, the starting state.

1. Get the next name in the structure. (The structure is scanned going from first to last statement in order of appearance.)
2. Look up the name in the symbol table. If it doesn't appear yet, add a header cell containing the display code characters (column one in Figure 11).

Fig. 11. Symbol table structure for deterministic look-up.



3. Add a new occurrence of this name with the appropriate data attributes (columns two and three, Figure 11).
4. Add a transition from CURRENT_STATE to the new occurrence of the particular name. In doing this, the symbol table addresses (box numbers in Figure 11) should be used rather than actual display code strings.
5. If the next entry to be scanned is at a higher level number than the current entry, push the new occurrence on CURRENT_STATE and go to 1.
6. If the next entry to be scanned is at the same level number, go to 1.
7. If the next entry to be scanned is at a lower level number, pop CURRENT_STATE once for each level lower and go to 1.

The results of applying these steps to a structured variable is a symbol table with one entry for each qualifiable occurrence of a name and a list of transitions (nondeterministic) for doing the look-up. Figure 11 shows the symbol table representation for the structure in Figure 3. Notice that this diagram is similar to the one in Figure 4 only without as many links. This link information is given in the transition list in Figure 12. The box numbers represent symbol table addresses. The only restriction on these addresses is that the occurrences of the names must be assigned ascending addresses in the order in which they are encountered.

The set of transitions constructed above must be expanded to form the closure described above and must be made deterministic. These two operations can be accomplished simultaneously as follows:

1. Sort the transitions into descending order by current state.
2. Move all transitions for the next state to be processed (initially the first state) to a work area. Remove the current state and save it.
3. For each "next state" add to the working storage all transitions out of the state. Again, strip off the current state and throw it away. Repeat until no more transitions are added. This gives the closure of this one state. Notice that each "next state" must either appear

Fig. 12. Sorted nondeterministic transitions (each entry refers to a box number in Figure 11).

Current state	Input symbol	Next state
11	1	12
9	7	10
4	7	8
4	5	6
2	5	9
2	3	4
0	3	11
0	1	2

Fig. 13. Construction of deterministic machine before and after processing state 2.

State set

State	Transitions	
11	1	12
9	7	10
4	5	6
	7	8

Working storage

Current state 2
Transitions

3	4	
5	9	
5	6	from state 4
7	8	from state 4
7	10	from state 9

State set (after this step)

State	Transitions	
11	1	12
9	7	10
4	5	6
	7	8
2	3	4
	5	13 combination of states 9 and 6
	7	14 combination of states 8 and 10

Fig. 14. Final deterministic machine state set.

State set

State	Transitions	
11	1	12
9	7	10
4	5	6
	7	8
2	3	4
	5	13 combination of states 6 and 9
	7	14 combinations of states 8 and 10
13	7	10
0	1	15 combination of states 2 and 12
	3	16 combination of states 4 and 11
	5	13 combination of states 6 and 13*
	7	14 combination of states 8 and 10
15	3	4
	5	13
	7	14
16	1	12
	5	6
	7	8

* state 13 already includes state 6

in the state set (initially empty) or not appear at all (a state with no transitions out) because of the order of address assignment.

4. Sort the contents of the working storage on input symbol and make deterministic (one transition per input symbol). Keep track of combined states.
5. Add the result of step 4 to the state set, and empty the working storage.
6. At this point, certain state combinations have been introduced as a result of making the machine deterministic. These may be satisfied one at a time in the following manner. (a) Generate a unique number for this combination. (b) Copy all transitions from any state in the combination to the working storage. All states required must already be in the state set because of the order of address assignment. (c) Perform steps 4 and 5 above. (d) Repeat until all state combinations are satisfied and then return to step 2.

This process can best be illustrated by an example. Figure 13 shows the contents of the state set and working storage when state 2 from Figure 12 is being processed. Figure 14 shows the final state set. This final state set, plus the symbol table, provides all the information needed to perform the deterministic table look-up.

This analysis provides one further piece of information which can be usefully included in a compile-time error diagnostic. Realizing that the machine must always start in state 0, it is obvious that it is unconnected. That is, some states (and therefore occurrences of variables) can never be qualified uniquely. In the example in Figure 14, states 2, 9, 11 cannot be reached, indicating that occurrences A, $A.C$, and B cannot be used. Looking at the initial structure, this is the case, since referring to A could be either the A at level 1 or the A at level 2 and, while the A at level 2 can be uniquely qualified, there is no possibility of qualifying one at level 1.

Conclusion

The discussion presents an application of simple automata theory to symbol table organization and lookup. It clearly demonstrates the trade-off between constructing the symbol table and being able to easily reference items. For most applications the second alternative would seem preferable.

Received June 1972

References

1. Internal Reference Specifications, 64/65/6600, COBOL Compiler, Version 3.0. Control Data Corp., 1967.
2. Gries, David. *Compiler Construction for Digital Computers.* Wiley, New York, 1971, pp. 239–240.
3. Harrison, Michael A. *Introduction to Switching and Automata Theory.* McGraw-Hill, New York, 1965.
4. Hopcroft, John E., and Ullman, Jeffrey D. *Formal Languages and Their Relation to Automata.* Addison-Wesley, Reading, Mass., 1969.