ASTROL—An associative structure-oriented language

by JAMES F. WIRTH
East Carolina University
Greenville, North Carolina

INTRODUCTION

The language ASTROL resulted from the search for a "small" language somewhat like LISP 1.5 which could be easily implemented on a minicomputer with about 32K bytes of store. The LISP cell was considered to be an example of the record—an object whose structure is specified by a set of field descriptors. However, the set of descriptors is often language whose structure was implicitly bound to the facilities provided by an associative memory—an Associative Structure-Oriented Language. This was a conceptual experiment to discover what ideas would arise in the attempt to adapt traditional programming constructs to such an environment.

FUNDAMENTAL CONSTRUCTS

The kind of memory envisioned for ASTROL was one that could associate a value z to a pair x, y where x, y and z were any members of some set A. Perhaps a content addressable memory (CAM) could be used—where each word of the memory is divided into three fields x, y and z. Instead of being addressed, a word (x, y, z) would be accessed by specifying its x and y field contents. The storage of the word (x, y, z) would represent the association of z to the pair (x, y). In ASTROL the instruction xy<–z expresses the storage of such an association in the memory while the expression (xy) denotes the retrieval of the value z last associated to x and y. The association is deterministic in that a subsequent storage operation xy<–w will erase the previous association. The operation (xy) can be thought of as meaning "the yth element of array x," "field y of record x," or "subject x's y-attribute." It is left associative with maximum precedence so that xyz is the same as (xy)z while xyz<–w means (xy)z<–w.

The objects x, y and z from the set A are called atoms and include the signed integers, symbols à la LISP such as BIRZWP, and two other categories—the code atom and anonymous atom—to be described later. A special anonymous atom, nil, is included as the default value for an undefined association (xy).

Beyond the simple notations given above, the evolution of ASTROL was guided by the principle of simplicity, and so much of the time the language has only one data type—the atom from A. The subcategories of A are mostly irrelevant to the language so that no expression is restricted to using a special kind of atom.

Simplicity also required that there be a single statement type and so ASTROL is an "expression" language. Each instruction is an expression with a value and so can appear elsewhere within a larger expression. The value of the association xy<–z is z, whereas the expression x(y<–z) causes the same association but has the value x. The latter form is useful in assigning multiple attributes to the same subject. Thus

\[ x(y_1<–z_1)(y_2<–z_2) \cdots (y_n<–z_n) \]

can be thought of as defining a "record" x whose \( y_i \)th field has the value \( z_i \). As in ALGOL 60, the operation < is right associative so that \( xyz<–zw<–u \) means \( xyz<–(zw<–u) \) and results in two associations \( zw<–u \) and \( xy<–u \). For a general example suppose that the memory has already stored the associations:

\[ MARY \rightarrow BILLY \quad JOHN \rightarrow WIFE \rightarrow MARY \]

Then the expression:

\[ JOHN \rightarrow REL \rightarrow I \rightarrow SON \leftarrow DOG \rightarrow MARY \rightarrow DOG \leftarrow DOG \rightarrow SPOT \rightarrow REL \rightarrow I \leftarrow WIFE \]

has the value JOE and causes the associations MARY DOG<–SPOT, BILLY DOG<–SPOT, and BILLY BUDDY<–JOE to be placed in memory.

In addition to the \((xy)\) and \(xy<–z\) operations, there are seven binary infix operators +, —, *, /, <, <=, and = with the usual meanings. These all have the same precedence as == and are all right-associative. The relational operators yield 1 for true and the nil atom for false; and a missing left operand for = is presumed to be 0.

CONTROL STRUCTURES

Formally speaking ASTROL does not support the concept of a program. However, a series of expressions \( e_i \) can be
combined into a new expression \((e_1; e_2; \ldots e_n)\) by the operator \(;\) which is left-associative, has minimum precedence and operates by deleting its left operand after evaluating both. Clearly this is a thinly-disguised program block.

Conversational languages such as BASIC provide editing flexibility by labeling every instruction in a program. This process can be seen as the association of an instruction code segment to each label. It is natural to adapt the associative mechanism in ASTROL to this purpose, and to that end code segments are allowed to be atoms in \(A\). The series of instructions which evaluate an expression \((e_1; e_2; \ldots e_n)\) can be organized into a single object called a code atom and will then be a member of \(A\). This data object is denoted by \([e_1; e_2; \ldots e_n]\). The operator \(DO\) is used to evaluate the instructions in such a code atom. For example the value of \([2+3]\) is the code atom which can add 2 and 3, but the value of \(DO\ [2+3]\) is 5. The operator \(DO\) has a precedence just above that of the semicolon. When \(DO\) is applied to a non-code atom, it leaves that atom unchanged. Consequently, the value of \(DO\ MARY\) is just \(MARY\). However if the association:

\[
\text{STEP} 1 \leftarrow [\text{MARY DOG} \rightarrow \text{SPOT}; \text{JOHN HAIR} \rightarrow \text{BROWN}]
\]

were placed in memory, then the value of \((\text{STEP} 1)\) would be a single code atom while the expression \(DO\ \text{STEP} 1\) would have the value \(BROWN\) and would cause the two indicated associations.

The code atom is the only device for controlling program flow. It is unnecessary to have a "case" construction, because what might have been written:

\[
\begin{align*}
\text{case} \ X \text{(}) & \text{ of begin} \\
A : & \text{ MARY DOG} \rightarrow \text{SPOT}; \\
B : & \text{ JOHN DOG} \rightarrow \text{SPOT}; \\
Y : & \text{ MARY HAIR} \rightarrow \text{BLOND}
\end{align*}
\]

\text{end case}
\]

can instead be rendered as \(DO\ CS\ (X\ 1)\), provided that the associations:

\[
\begin{align*}
CS\ A & \leftarrow [\text{MARY DOG} \rightarrow \text{SPOT}] \\
CS\ B & \leftarrow [\text{JOHN DOG} \rightarrow \text{SPOT}] \\
CS\ Y & \leftarrow [\text{MARY HAIR} \rightarrow \text{BLOND}]
\end{align*}
\]

are present in memory.

Because the "if-then-else" construction is a special version of the "case" statement, it too becomes unnecessary. However, the resulting ASTROL formulation is awkward and so an \(IF-ELSE\) operator was included in the language:

\[
\text{value}[x IF y ELSE z] = \begin{cases} \text{value} [x] & \text{when \ value} [y] \text{ is not nil} \\ \text{value} [z] & \text{when \ value} [y] \text{ is nil} \end{cases}
\]

This operator does not control program flow as in ALGOL because each operand of the \(IF-ELSE\) is evaluated before the result is selected. Thus the code atom remains the sole vehicle for program control and the ALGOL version of "if \(x\) then \(y\) else \(z\)" must be expressed as \(DO\ [y]\ IF\ x ELSE [z]\), where the parentheses could be omitted since the \(DO\) and \(IF-ELSE\) operators have the same precedence and are associated to the right.

The traditional loop construction can be implemented by recursion, but it was felt that this would be an expensive solution in practice. Consequently the language supports an operator, \(REPEAT\), with the same precedence as \(DO\). Its operand will be repeatedly activated by \(DO\) until it yields a non-nil value, which will then be the result of the \(REPEAT\) operator. Several loop flowcharts are given in Figure 1 along with equivalent expressions in ASTROL.

It should be emphasized that \(REPEAT\ x\) is an expression and does have a value. For example if \(A\) were an array of 20 elements, then the instruction:

\[
\text{STEP FIND} \leftarrow [\text{POS} I = 0; \text{REPEAT}\ \text{DO NONE IF} \ (20 < \text{POS} I = \text{POS} I + 1) \ \text{ELSE} \ [\text{POS} I \text{ IF} A(\text{POS} I) = 3 \ \text{ELSE} \ \text{NIL}]]]
\]

will create a code atom \(STEP\ FIND\) whose activation gives the value of the first position in \(A\) whose element is 3, or the value \(NONE\) in case none are.

**SYSTEM SYMBOLS**

Although generally speaking there are no variables in ASTROL, it does support six operandless operators \#, $, @, ?, ??, and \(NIL\) which are like read-only variables. The result of executing each of these system symbols is described below:

- \# Results in a new anonymous atom for each execution.
- $ Results in an anonymous atom, called the procedural context, which represents the procedure call depth.
- @ Results in the atom representing the function procedure currently executing.
- ? Results in the next atom from the input data stream each time that it is executed.
- ?? Results in the atom previously read by the ? operator.
- \(NIL\) Results in the nil atom.

The operator \# introduces the fourth category of atom—the anonymous atom—which is a data object not in one of the other three categories: integer, spelled symbol or code segment. An anonymous atom is an internal object created by the system when the user needs a new object but does not want to name it. Each occurrence of the operator \# in an expression will represent a new such object. As an example consider the expression:

\[
\text{THE ROOT} \leftarrow \# \ (\text{OP} = \text{ADD}) \\
\ (\text{LEFT} = \# \ (\text{OP} = \text{ADD}) (\text{LEFT} = 3) (\text{RIGHT} = 7)) \\
\ (\text{RIGHT} = \# \ (\text{OP} = \text{SQRT}) (\text{RIGHT} = 9))
\]

which creates the labeled ordered tree shown in Figure 2. The first occurrence of \# creates the root node of the tree, while the second and third occurrences create the left and right daughters of the root. The root node is associated to the pair \(TIE\ \text{ROOT}\). The node labels are considered to be OP-attributes of the anonymous nodes.
The operator $ is related to the procedural treatment of identifiers. In a sense no procedural language has explicit variables but rather uses an identifier to represent a variable in that specific context which is the scope of the variable. Hence the value of the variable is associated with the context/scope and the identifier. This association is explicit in ASTROL so that the context/scope is a concrete atom in $A$ represented by $\$ and not an abstraction of the form of a program. Hence $\$x$ represents the value associated to the identifier $x$ in the current context $\$ without need of a special mechanism. When a procedure is invoked, $\$ acquires a new value to represent the context of that copy of the procedure, while return from the procedure reinstates the previous value of $\$. Hence the action of $\$ simulates the pushing and popping of procedure stack “frames.” In ASTROL the only means of changing context is the procedure call and so contexts can be stacked dynamically (even recursively) but cannot be statically nested as in ALGOL. The use of $\$ is discussed further in the next section.

FUNCTIONS

Since ASTROL is an expression language, the only procedures are functions. There is no special syntax for the
A function invocation has the form:

\[ x[y_1, y_2, \ldots, y_n] \]

where \( x \) is the function name and the \( y_i \) are the actual parameters and are passed by value. The \([ ]\) operation has the same precedence as the elided operator \((xy)\) so that \( xy[z, w]p(q\rightarrow r) \) means the same as \(((xy)[z, w])p(q\rightarrow r)\).

The actual parameter \( y_i \) is passed to the function \( x \) as follows. The expression \((xi)\) should be the corresponding parameter name and so the association \( S(xi)\rightarrow y_i \) is used to pass \( y_i \) under the name \( (xi) \). After the parameters have been passed, the system performs a \( DO(x VAL) \) to execute the code body \( (x VAL) \) associated to the function. As mentioned in the last section, the value of \( S \) is changed just before initiating the function call and is restored when the code atom \( (x VAL) \) has finished execution. Even if \( x \) is not a function, or \((x VAL)\) is nil, all these operations are still legal, although the function value may be nil.

Since a local variable \( n \) used in a function can be referred to as \( \$n \), the associative mechanism and the explicit context atom \( \$ \) make it unnecessary to allocate local variables to the function in any special way.

As one example, consider the expression:

\[ F(1\leftarrow X)(2\leftarrow Y) VAL\leftarrow [(2*\$X)\leftarrow \$Y], \]

which creates the function \( F(x, y) = 2x - y \). A more elaborate example is given by the definition of the function composition operator \( COMPOS \) which creates a function dynamically. This created function is represented by an anonymous atom and its composition factors are stored as the \( F \) and \( G \) attributes of that atom. Since the created function is anonymous, its name is accessed by the operator \( @ \) within its code body:

\[ COMPOS(1\leftarrow F)(2\leftarrow G) VAL\leftarrow [\# (1\leftarrow X)(F\leftarrow \$F) (G\leftarrow \$G) VAL\leftarrow @[F[@ G]\$X])]]. \]

Although parameters are passed by value in ASTROL, the existence of the code atom allows a parameter to be effectively passed by name.

USER INTERACTION

User input is handled by the \( ? \) and \( ?? \) operators covered earlier, while output is done with the operators \( ! \) and \( : \) which have a precedence slightly higher than the semicolon and are left-associative. Both operators tabulate to the position given in the left operand. The \( ! \) then outputs the print record and begins a new one. Finally both operators enter the right operand into the print record and yield the next available record position as their value. For example the expression \(!A:B:C/D\) would produce:

\[ ABC \]

\[ D \{\text{no carriage return}\}, \]

and have 3 as its value. Output of anonymous atoms and \( NIL \) is omitted, while output of a code atom will print it out in source form. This eliminates the need for a program listing command.

There are no special editing or command instructions. The user types an expression followed by \( a . \) and the system evaluates it. For example the instruction \( :1+2. \) would print out 3.

Liberal use of the \( DO \) operator will allow simple instruction editing via the \( \leftarrow \) operator. This also facilitates debugging since individual steps in a task can be tested in isolation.

Creating user program files presents a problem, since a task in ASTROL is performed by a loosely organized system of code segments and there is no program as such for the user to save. Dumping all the associations in memory would be awkward because of the anonymous atoms. Consequently, the primitive function \( SAVE[\text{file-name}, n] \) includes a parameter \( n \) to determine the class of associations \( xy\rightarrow z \) to save. When \( n \) is 0, then \( z \) must be a code atom and \( x \) and \( y \) must be integers or symbols. When \( n \) is 1, then \( z \) may also be an integer or symbol. The associations are dumped in source-readable format so that the primitive function \( \text{READ[\text{file-name}]\} \) can simply substitute the specified text file for the normal teletype input stream.

Actual implementation of the language required simulation of the associative memory. This was done by hash-coding the subject-attribute pairs. For efficiency in execution, code atoms were translated into a postfix polish notation, although this requires de-translation in order to list such an atom. The only garbage collection is to delete unreferenced code atoms. Currently, ASTROL runs on a PDP11/20 with 32K bytes of store under the RT11 operating system.

CONCLUSION

A potential technological advance such as the associative memory might be expected to have some impact on programming concepts. ASTROL was created in anticipation of this advance and illustrates some of its consequences for programming. Although it is goto-less and programs should be carefully organized in a top-down fashion—the code segmentation and automatic data structure allocation impart an unstructured flavor to the language. The use of data associations to control program flow allows easy extension of
ASTROL—An Associative Structure-Oriented Language

conversational programs—often without requiring any alteration of existing code segments.
Currently the language is being used to implement CAI programs for mathematics and computer science and to investigate an alternative to transformational English grammar.

REFERENCES

APPENDIX

BNF grammar for ASTROL

\[\begin{align*}
\text{(digit)} & ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
\text{(letter)} & ::= A \mid B \mid C \mid D \ldots \mid X \mid Y \mid Z \\
\text{(binary operator)} & ::= + \mid - \mid * \mid < \mid \leq \mid = \mid / \\
\text{(io operator)} & ::= : \mid ! \\
\text{(system symbol)} & ::= $ \mid @ \mid \text{NIL} \\
\text{(integer)} & ::= \text{(digit)} \mid \text{(integer)}(\text{digit}) \\
\text{(name)} & ::= \text{(letter)} \mid \text{(name)}(\text{letter}) \mid \text{(name)}(\text{digit}) \\
\text{(instruction)} & ::= \text{(block)}. \\
\text{(block)} & ::= \text{(io)} \mid \text{(block)}; \mid \text{(io)} \\
\text{(io)} & ::= \text{(expression)} \mid \text{(io operator)}(\text{expression}) \mid \text{(io)}(\text{io operator})(\text{expression}) \\
\text{(expression)} & ::= \text{(simple expression)} \mid \text{-(expression)} \mid \text{DO(}\text{expression}) \mid \text{REPEAT(}\text{expression}) \mid \text{IF(}\text{expression})\text{ELSE(}\text{expression}) \\
\text{(simple expression)} & ::= \text{(term)} \mid \text{(term)}(\text{primary}) \leftrightarrow \text{(expression)} \mid \text{(term)}(\text{binary operator})(\text{expression}) \\
\text{(term)} & ::= \text{(primary)} \mid \text{(term)}(\text{primary}) \mid \text{(term)}(\text{primary}) \leftrightarrow \text{(expression)} \mid \text{(term)}[1] \mid \text{(term)}(\text{argument list}) \mid \text{((block}) \\
\text{(primary)} & ::= \text{(integer)} \mid \text{(name)} \mid \text{(system symbol)} \mid \text{RND(}\text{expression}) \mid \text{READ(}\text{expression}) \mid \text{SAVE(}\text{expression}) \mid \text{(expression)} \\
\text{(argument list)} & ::= \text{(expression)} \mid \text{(argument list)}.
\end{align*}\]