JOURNAL OF THE TENNESSEE ACADEMY OF SCIENCE
Volume 39, Number 4, October, 1964

A RECOGNITION TECHNIQUE FOR THE TRANSLATION OF ALGOL 60*

MARJORIE P. LIETZKE
Union Carbide Corporation
Oak Ridge, Tennessee

ABSTRACT

This paper describes a recognition technique based on a tree structure which was originally developed to facilitate implementation of a syntax checking program for ALGOL 60. The same technique may be applied to the design of a complete translator for ALGOL or other languages.

A syntax checking program for ALGOL 60 [1, 4] was developed as a part of the SHARE ALGOL project. In designing and implementing this program, several criteria were kept in mind: the recursive definitions of the ALGOL language must be properly handled; a method of error recovery must be devised which would permit, in so far as possible, complete checking of a source program in one machine run; all declarations must be processed in order to check for correct variable types in expressions; and finally, the processor must be as fast as possible so that syntactically correct programs may be passed without using an undue amount of time in the syntax checking phase. All of these objectives were achieved.

The syntax checker is based on a tree structure which may be considered as being derived from the translation matrix described by Bauer and Samelson [2]. A state-symbol pair controls the action and the next state, as in their method. However, the tree structure permits economy in storage as well as flexibility. The tree structure may conveniently be expressed as a series of tables where each state or node of the tree is represented by a table containing the expected or permissible symbols for that state; these represent the non-null elements of a particular row of a translation matrix. All of the null elements may be lumped together under the single entry "other". Thus, by this method, an entire translation matrix may be reduced to a series of variable length vectors, with no wasted storage space. Now, using a two dimensional translation matrix, it is possible to check only two items directly, i.e., the current state and the current symbol. If additional information concerning what has preceded this pair is necessary for a decision, it must be obtained by contextual examination of the preceding symbols. An alternative would be to increase the number of dimensions of the matrix; an N-dimensional matrix would permit a decision to be made on the basis of N items. However, there would be an extremely large number of null elements in such a matrix, resulting in wasted storage space. Also, the computation involved in selecting the desired matrix element would become longer and more complex with each additional dimension.

It was decided that it was possible to simulate an N-dimensional translation matrix by breaking the tree structure down into syntactic units and using these units on different levels. Each new level would, in effect, add another dimension to the matrix. A block, for example, has basically four states; the **begin,** declarations, statements, and the **end.** Within the declarations there may be blocks in procedure declarations, and a statement may itself be a block. The ALGOL language places no restrictions on how far this recursiveness may be carried, thus it is impossible to define a finite tree structure capable of handling all possible cases. The direct result of this is a set of mutually recursive processing subroutines, one for each syntactic unit of the ALGOL language. Each processor is capable of calling itself or other processors as necessary. This makes it possible to append all or any desired portion of the ALGOL tree structure dynamically, where ever it is required, and still have a relatively short program consisting of a single definition for each element of the language.

The processors are sets of tables which list the possible or expected symbols for each state in a syntactic unit. Each symbol has associated with it an action and a next state. The action depends on the symbol and the state of the syntactic unit being checked and may range from no operation to a fairly complex sequence of operations, including the recursive call of other processors. The next state also depends on the current symbol and state and may depend on whether a processor invoked in the action part has given a normal return or an error return. The next state is always another state in the same processor, a return, or an error eturn.

In designing the processors the official ALGOL Report [3] was followed exactly. In general, each processor can call directly only the processors for the syntactic units which may properly be used within the unit being checked. The block processor, for example, may call the general declaration processor or the general statement processor. It may not call directly an expression processor or a processor for a particular type of statement or declaration. A list of the syntactic units for which processors were designed is given in Table 1, together with the other processors which may properly be called by each processor.

The arithmetic expression processor shown in Table 2 is an example of a typical processor. When this processor is called, the assumption is that the first symbol of an arithmetic expression is under scan and checking begins at state AREX1. The symbol under scan is compared with each symbol in the table until a match is found or until the entry "other", meaning any other symbol, is encountered. At this point, the indicated action is taken and control passes to the next state where comparison is resumed.

* This paper is based on work done at Central Data Processing Facility operated by Union Carbide Corporation for the U. S. Atomic Energy Commission.

Special consideration had to be given to the problem of finding a new starting point to continue checking after an error has been detected. It would be possible to skip to the next semicolon, **end,** or **else** and attempt to go on from there, but this could leave sections of the program unchecked. To minimize the number of runs necessary to obtain a syntactically correct program, a Resume processor was designed to recognize and check any syntactic unit and then return control to the processor which called it. If on a return from Resume the calling processor is unable to continue, it may call Resume again to check the next unit; if the calling processor finds a legal symbol on return it will continue checking in the normal mode. If the Resume processor recognizes the beginning of a statement it gives an error return, indicating to the calling processor that it also should give an error return; normally the beginning of a new statement indicates that the end of the incorrect structure has been reached and processing may continue in the normal mode. In any case, the entire program is checked and most errors detected in one machine run.

In using these processors, it is necessary to keep track of each call of a processor in order to make a proper return when processing has been completed. It is also necessary to save the current values of certain quantities on entering a processor and to retrieve these values on leaving the processor in order to follow the block structure and flow of an algorithm. To do this, a number of pushdown lists are set up for declared variables, labels, formal parameters of procedures, temporary storage of pointers and indicators, and the locations of calls of processors. These lists are manipulated by a set of subroutines which perform the actions requested by the processors and do all the necessary bookkeeping on the pushdown lists.

It was recognized that this syntax checking program was performing most of the functions of a translator, including even rudimentary storage allocation for the declared variables. However, certain short-cuts were permissible here which could not be allowed in a complete translator. For example, Table 3 shows the arithmetic expression processor as it must appear for a complete translator. Note that there are six additional states or nodes in the tree to permit the proper generation of object code. It would, of course, be possible to add more states to permit the tree structure to handle the precedence rules for the binary operators, but it was felt that the forcing value technique used by Bauer and Samelson was more efficient in this case. It is necessary only to compare the current operator with the preceding operator to determine whether code may be generated at a given point.

Preliminary work on the implementation of this technique indicates that it is possible to set an error flag and design the action subroutines so that no more code will be generated after a source program error has occurred. This permits the processor to continue scanning as a syntax checker only, so that the first attempt at translation will give the programmer a relatively complete list of his errors.

This paper has described a recognition technique as it may be used in a translator for ALGOL 60. It should be noted, however, that this technique is applicable to any situation where it is necessary to make a decision on N successive items.

Table 1

List of Syntactic Units and Processors

Syntactic Unit	Processors Called	Syntactic Unit	Processors Called
Block	Declaration, Statement, Comment, Label, Resume	Conditional Statement	Statement, Unconditional Statement, Resume
Declaration	Type declaration, Array declaration, Procedure declaration, Type Procedure declaration, Switch declaration, Resume, Comment	Go To Statement	Designational Expression
		For Statement	Arithmetic Variable, Arithmetic Expression, Boolean Expression, Statement
Statement	Assignment statement, Conditional statement, Go To statement, For statement, Compound statement, Block Label, Procedure statement	Compound Statement	Statement, Resume
		Label	———
		Comment	———
		Type Declaration	———
Assignment Statement	Arithmetic Expression, Subscript Expression, Boolean Expression	Array Declaration	Arithmetic Expression

Syntactic Unit	Processors Called	Syntactic Unit	Processors Called
Procedure Declaration	Specifier, Statement, Resume	Procedure Statement	Parameter List, General Expression
Type Procedure Declaration	Procedure Declaration, Resume	Parameter List	Parameter type, Parameter Delimiter
Switch Declaration	Designational Expression		
Arithmetic Expression	Arithmetic Expression, Boolean Expression, Function call, Subscript Expression, Resume	Parameter Type	General Expression
		Parameter Delimiter	———
Subscript Expression	Arithmetic Expression	Specifier	———
Boolean Expression	General Expression	Comparison Expression	Arithmetic Expression
General Expression	Boolean Expression, Arithmetic Expression, General Expression Subscript Expression, Function Call, Comparison Expression, Resume	Designational Expression	Designational Expression, Boolean Expression, Simple Designational Expression, Resume
Unconditional Statement	Statement	Simple Designational Expression	Designational Expression, Arithmetic Expression
Arithmetic Variable	Subscript Expression	Truth Value	———
Function Call	Parameter List, General Expression	Resume	Any of the above processors

Table 2

Arithmetic Expression Processor

State	Symbol	Action	Next State No Error	Next State Error
AREX 1	+	skip (pass on this symbol, get next symbol)	AREX 2	
	—	skip	AREX 2	
	if	skip, Boolean Expression Processor	AREX 5	AREX 9
AREX 2	ID	get type of identifier	AREX 8	
	(	skip, Arithmetic Expression Processor	AREX 3	AREX 10
	other	———	error return	
AREX 3	)	skip	AREX 4	
	other	Error message	AREX 10	
AREX 4	+	skip	AREX 2	
	—	skip	AREX 2	

State	Symbol	Action	Next State No Error	Error
	×	skip	AREX 2	
	/	Real Exp: = true, skip	AREX 2	
	÷	skip	AREX 2	
	↑	skip	AREX 2	
	other	———	return	
AREX 5	**then**	skip	AREX 6	
	other	———	AREX 9	
AREX 6	**if**	Error message, Insert (, Arithmetic Expression Processor, insert)	AREX 7	AREX 9
	other	Arithmetic Expression Processor	AREX 7	AREX 9
AREX 7	**else**	skip, Arithmetic Expression Processor	return	error return
	other	Error message	AREX 9	
AREX 8	real variable	Real exp: = true, skip	AREX 4	
	real procedure	skip, function processor	AREX 4	error return
	integer variable	skip	AREX 4	
	integer procedure	skip, function processor	AREX 4	error return
	array	Real exp: = true, skip, subscript processor	AREX 4	error return
	real array	Real exp: = true, skip, subscript processor	AREX 4	error return
	integer array	skip, subscript processor	AREX 4	error return
	Boolean variable	Error message, skip	AREX 4	
	Boolean array	Error message, skip, subscript	AREX 4	error return
	Boolean procedure	Error message, skip, function statement processor	AREX 4	error return
	other	Error message	error return	
AREX 9	**then**		AREX 5	
	else		AREX 7	
	other	Resume processor	AREX 9	error return
AREX 10	)		AREX 3	
	other	Resume processor	AREX 10	error return

Table 3

Arithmetic Expression Processor for Translation

State	Smybol	Action	Next State No Error	Error
AREX 1	**if**	if code, Advance, Boolean Expression	AREX 5	AREX 9
	+	Advance	AREX 2	
	−	Push	AREX 2A	
	other		AREX 2	
AREX 2A	ID	Qualify, Unary Code	AREX 2	
	(	Push, Arithmetic Expression	AREX 3A	AREX 10A
	other		error return	
AREX 2	ID	Qualify	AREX 8	
	(	Push, Arithmetic Expression	AREX 3	AREX 10
	other		error return	
AREX 3A	)	) code	AREX 2A	
	other	error message	AREX 10A	
AREX 3	)	) code	AREX 2	
	other	error message	AREX 10	
AREX 4A	]	Array code, Advance	AREX 4	
	)	error, Symbol : =], Array Code, Advance	AREX 4	
	other	error message	error return	
AREX 4B	)	function code, advance	AREX 4	
	]	error, symbol : =), function code, advance	AREX 4	
	other		error return	
AREX 4	+	binary op code, push	AREX 2	
	−	binary op code, push	AREX 2	
	×	binary op code, push	AREX 2	
	/	binary op code, push	AREX 2	
	÷	binary op code, push	AREX 2	
	↑	binary op code, push	AREX 2	
	other	binary op code	return	
AREX 5	**then**	**then** code, advance	AREX 6	
	other		AREX 9	
AREX 6	**if**	error, arithmetic expression	AREX 7	AREX 9
	other	Arithmetic expression	AREX 7	AREX 9

State	Symbol	Action	Next State (No Error)	Next State (Error)
			return	
AREX 7A	other	catch code	AREX 7A	AREX 9
AREX 7	**else**	**else** code, arithmetic expression	AREX 9	
	other	error message	AREX 4	
AREX 8	real variable	Push	AREX 4	
	integer variable	Push	AREX 4	
	Boolean variable	error, push	AREX 4A	error return
	array	push, subscript expression	AREX 4A	error return
	real array	push, subscript expression	AREX 4A	error return
	integer array	push, subscript expression	AREX 4A	error return
	Boolean array	error, push, subscript	AREX 4B	error return
	real procedure	push, parameter list	AREX 4B	error return
	integer procedure	push, parameter list	AREX 4B	error return
	Boolean procedure	error, push, parameter list	error return	
	other	error	AREX 5	
AREX 9	**then**		AREX 7	
	else		AREX 9	error return
	other	Resume	AREX 3A	
AREX 10A	)		AREX 10A	error return
	other	Resume	AREX 3	
AREX 10	)		AREX 10	error return
	other	Resume		

LITERATURE CITED

1) Lietzke, M. P. 1963. An ALGOL 60 Syntax Checker for the IBM 7090 Computer. ORNL 3399.
2) Samelson, K. and Bauer, F. L. 1960. Sequential Formula Translation. *Communications of the ACM*. Vol. 3, No. 2.
3) Naur, Peter, et al 1960. Report on the Algorithmic Language ALGOL 60. *Communications of the ACM*. Vol. 3, No. 5.
4) Lietzke, M. P. 1964. A Method of Syntax Checking ALGOL 60. *Communications of the ACM*. Vol. 7, No. 8.

NEWS OF TENNESSEE SCIENCE

(Continued from Page 124)

The University of Tennessee Department of Psychology has received an $8,989 grant from the U.S. Department of Health, Education, and Welfare to help finance research in cognitive structure and verbal behavior. The program will be directed by Dr. Howard R. Pollio, Assistant Professor of Psychology. According to Dr. Pollio, "The human mind is like a filing cabinet—it stores away the meaning of words. It retrieves the filed meanings for talking and thinking. The purpose of this research is to determine how the mind performs this function."

Dr. Louis A. Rayburn, Professor of Physics at the University of Georgia, has joined the staff of the Oak Ridge Institute of Nuclear Studies for 15 months as head of the University Participation Section in the University Relations Division.

(Continued on Page 136)