ABSTRACT

The objective of this project is to write a Turbo Pascal program which will serve as a preprocessor that will take any valid Turbo Pascal program containing one or more recursive procedures as input and produce as output an equivalent source program in which all recursive procedures have been converted to iterative procedures. An input of a program with no recursive subprograms will be output in its original form.

The Pascal programming environment uses a pushdown stack to allow recursion. Each time a procedure call is made, the values of all the variables are pushed onto the stack. Each time a procedure returns, the stack is popped. The project program, ITERATOR, implements an explicit stack as a linked list in which a push is implemented by linking a new node onto the front of the list and a pop by removing the first node on the list. 2 Source code declaring the stack is inserted at the beginning of each


recursive subprogram. This stack holds the values of all call-by-value parameters, local variables, and the next label to be used for each call. Also, a label is affixed to the appropriate executable statement in the subprogram. When a return is encountered, code is inserted to pop the stack. All values are assigned to the appropriate variables. Also, a unconditional transfer of control to the beginning of the subprogram is added.

The end of a recursive subprogram is altered by inserting code to check the stack to see if it is empty. If the stack is empty, a normal return is executed. Also, for the case in which the stack is not empty, code is inserted which removes the index of the return address from the stack and assigns it to some unused variable, and removes the values of all local variables and call-by-value parameters from the stack and assigns them to their corresponding variables. If the subprogram is a function, instructions are inserted to evaluate the expression and store the result in the stack. At this point, the index of the label of the return address is used to execute a branch to that label.


INTRODUCTION

This project is implemented on an IBM Personal System/2 Model 30 in Turbo Pascal. It is composed of the primary file, ITERATOR.PAS and six included files - INFORM.PAS, MODIFYCA.PAS, REMOVERE.PAS, RETRIEVE.PAS, MULTIPLE.PAS, and RETURNS.PAS. Two data files, STKPROC.DAT and STACKDEC.DAT, which each contain information to be inserted into the output file, must also be present on the diskette. ITERATOR has been compiled into an executable command file so that it may be run from DOS by entering ITERATOR at the drive prompt.

A series of Turbo Pascal programs are used for input - some with recursive procedures and some with nonrecursive procedures. ITERATOR expects one program at a time as input and produces as output a Turbo Pascal program with all recursion removed. The output is written to a text file under the same file name as the input file but with the extension OUT.

In a recursive procedure, each time the procedure is called, a new activation record is created. This record is placed on top of a stack containing all activation records of all routines with an execution in progress. Each of these activation records contains the return address of the call along with the values associated with all local variables, parameters, and temporaries. When control is returned to an activation of a procedure, that activation record is on top of the stack. When the routine is
completed, the activation record associated with that execution is popped from the stack.\textsuperscript{5}

The project program, ITERATOR, is a preprocessor that converts a program containing one or more recursive procedures to an iterative procedure by inserting source code at appropriate points within the input program to simulate these same functions as described above, instead of utilizing the Pascal recursive stack system. The stack in the iterative version, however, contains the minimum values required by the subprogram call and is therefore generally much smaller than would be used in the recursion system supported by the language.

In ITERATOR, the statements of the iterative procedure which will be the output, are stored in a linked list to facilitate any possible insertions, deletions, or substitutions. Each node of the linked list consists of a single Pascal statement, the length of the statement, and a link to the next node. The format of the statements remains essentially the same as that of the input statements, as ITERATOR reproduces all blank spaces, blank lines, and comments in the output linked list.

ITERATOR scans each line of the input file for the keywords FUNCTION or PROCEDURE. If either is detected, the entire subprogram is put into a linked list of records containing a packed array of characters and the length of the line. Each

entry in the linked list is scanned for recursion by checking for the subprogram identifier. If recursion is not detected, the entire subprogram linked list is written to the output linked list and the main process is repeated beginning with the next line of input. If recursion is detected, the recursive procedure is converted to an iterative procedure by replacing all recursive calls and returns with equivalent nonrecursive source code.

Code which declares a stack and initializes it to empty is inserted at the beginning of the recursive procedure. The stack is used to hold the values of call-by-value parameters, local variables, function values, and the return address for each recursive call. A label L1 is attached to the first executable statement.6

Each recursive call is replaced by a set of instructions. First, code is inserted to cause the values of all call-by-value parameters and local variables to be stored in the stack. Then, code is inserted to create the ith new label, Li, and store it in the stack. It will later be used to compute the return address. Code is inserted to evaluate the arguments of the call and assign values to the appropriate formal parameters. Then, an unconditional branch to the beginning of the procedure is inserted into the subprogram. If the subprogram is a function, the next label is attached to a statement which retrieves the function value from the top of the stack. The code to use this

value as described in the recursive procedure is added. If the subprogram is not a function, the next label is affixed to the statement immediately following the unconditional branch.

At this point, all recursive calls have been removed from the subprogram. The return statements are now altered. If the stack is empty, control is returned to the initial call. Otherwise, code is inserted to cause the current values of all output parameters to be assigned to the corresponding variables on the top of the stack. Code is inserted to remove the index of the return address from the stack and assign it to an unused variable. If the subprogram is a function, code is inserted to evaluate the expression and store this value in the top of the stack immediately following the return. Code is inserted that used the index of the label of the return address to execute a branch to that label. 

Therefore, each recursive call is replaced by a stack push and each return by a stack pop, mimicking what the Pascal system automatically does to implement recursion.

After all of the recursive calls and returns are translated into iterative code, the subprogram is written to the output linked list. Control then returns to the beginning of the process and the next line of input is scanned.

---

7 Horowitz and Sahni, pp. 17-24.

8 Sedgewick, p. 275.
When end-of-file is detected, the entire output linked list is written to an output text file. This text file is a Turbo Pascal program in which recursion is simulated by a program manipulated stack to insure that necessary locations and values can be reclaimed when needed. The program produced is a syntactically correct iterative program equivalent to the recursive input program. It may be run independently as an iterative, nonrecursive program.

The object of removing recursion is to produce a computationally equivalent iterative program. This conversion of the recursive routines in the program can also provide some insight into developing a more efficient nonrecursive version of the original program. Therefore, removing recursion can be a very useful and worthwhile task. Recursion may be removed by translating the recursive algorithm into an equivalent one which uses only iteration. On many compilers, the resulting program will be more efficient than its recursive version.

9 Ellzey, p. 487.

10 Sedgewick, pp. 109-111.

11 Ellzey, p. 490.

PROCEDURE

At the beginning of ITERATOR, a boolean function, DISPLAY_INFORMATION_ANS, is called. This function clears the screen and selects the character and background colors. It also includes cursor positioning to display an introductory message as it asks the user for the name of the program to be converted. At this point, the user is given a second chance to exit the program by entering an "N" in response to the "Do you wish to continue?" query. This causes FALSE to be returned as the function value, thus terminating the program. If the user chooses to continue, the response "Y" causes the function EXISTS to be called to determine if the program is stored on disk. If the program is not found, a message is displayed, FALSE is returned for the function value, and ITERATOR asks the user if he wishes to re-enter the file name. If not, the program terminates. Otherwise, the file is assigned and opened for input and a procedure is called to generate a name for the resulting iterative output program. This procedure, GENERATE_OUTPUT_FILENAME, produces an output file name consisting of the first eight input file characters concatenated with .OUT. For example, the input of the file INFILE.PAS produces the output file INFILE.OUT. The output file is assigned and control
transfers to the main program, returning TRUE in DISPLAY_INFORMATION_ANS.

In the main program, if DISPLAY_INFORMATION_ANS returns TRUE, execution continues. Several variables are initialized and the data files STACKDEC.DAT and STKPROC.DAT are assigned to mnemonic names. At this point, a WHILE loop checking for the end of the input file is entered. If end of file is FALSE, the body of the WHILE loop is executed. Otherwise, the input file is closed and the subprogram is written to the output file.

Until the end of the input file has been detected, the input statements are manipulated based on their contents. First, the procedure SKIPBLANKS is called. This procedure scans the current line until it detects a non-blank character. The number of blanks along with the character itself are returned to the main program. At this point, the procedure CHECK_FOR_FUNC_PROC is called to determine whether the current line is a function or procedure declaration. CHECK_FOR_FUNC_PROC stores the first nine non-blank characters in a packed array. Then, it compares this array with the constants FUNCTION and PROCEDURE by calling the function UPPERCASE to convert any lowercase characters to uppercase for comparison purposes. The packed array and a flag indicating whether the array held a function or procedure declaration are returned to the main program. If the returned flag contains a zero, indicating that the current line is not a function or procedure declaration, the entire line is
written to an output linked list in the procedure WRITE_LINE_TO_OUTPUT_LIST.

If the returned flag contains a one, a function has been detected. A returned value of two indicates a procedure. If the flag holds a one or a two, the subprogram is stored in a linked list in the procedure STORE_SUBPROG. STORE_SUBPROG stores the entire subprogram in a linked list by counting the number of BEGINS and ENDS. As each line is stored, it is scanned for recursive calls by the procedure CHECK_FOR_RECURSION. If a recursive call is detected, the procedure CHECK_FOR_MULTIPLE_STATEMENTS is called to determine whether there are multiple statements on the procedure call line. If extra statements are detected, any statements prior to the call are inserted into the node before the call. Any statements after the call are appended in the node following the call. This is done to eliminate any problems caused by the substitution of stack manipulation statements to replace the recursive call. Since any statements before or after the call which are located on the same line as the call are moved to a different node, they will not be interspersed with the inserted stack manipulation statements.

Also, within the STORE_SUBPROG routine, the procedure CHECK_FOR_BEGIN_END_INSERTION is called. This procedure scans the line of the call and the line immediately before the call for the keywords THEN, ELSE, and DO. These keywords are indications that the subprogram call may be a single statement FOR, WHILE,
THEN, or ELSE. If the call is located immediately after one of these keywords, the characters between the keyword and the call are scanned for a BEGIN. If no BEGIN is found, a BEGIN and END are inserted around the call since the stack manipulation statements which replace the call are multiple whereas the recursive call is a single statement. A flag is passed back to STORE_SUBPROG to indicate that an insertion has taken place. The pointers to the beginning and the end of the subprogram, subprogram name, subprogram length, and the number of recursive calls detected within the subprogram are all returned to the main program.

If recursion is detected, a flag is checked to determine whether this is the first recursive call. If such is the case, the procedure INSERT_STACK_PROC is called. This procedure resets the file STKPROC.DAT and reads it into a linked list before closing the file. This data file contains the PUSH and POP procedures which are included in the iterative version of the input program. These procedures are inserted into the main output program to facilitate the manipulation of the stack by the resulting program.

At this point, the procedure REMOVE_RECURSION is called. This procedure is one of the most important procedures in the entire program as this is where the actual manipulation of the stack is inserted into the program to replace the recursive calls and returns.
The first procedure called by REMOVE_RECURSION is the procedure INSERT_DECLARATIONS. This procedure inserts the appropriate declarations into the subprogram linked list such as the stack pointer declaration, and the declaration of the variable in which the address of the return location of the call is stored. The dummy variant record declaration for pushing and popping data of different types into the same stack, and a function result declaration, if needed, are also inserted into the subprogram.

The location for the declaration insertion is found in the procedure FIND_LOCATION_FOR_STACK_DECLARATION. This procedure searches the subprogram for the keywords TYPE, VAR, and BEGIN. Since TYPE and VAR declarations may not be present in the input file, the declarations are inserted before the BEGIN if neither exist. If a TYPE declaration is found, a flag is set and the procedure continues to search for a VAR or BEGIN. A flag is sent to the procedure to indicate whether the search is of the main program or the subprogram since this procedure is used in another location to search for these same values. If the search is of a subprogram, the list is first searched for the subprogram name. If an open parentheses is detected after the name, the characters between the open and closed parenthesis are omitted from the search. The reason for this is that the formal variable list could possibly contain one or more call by reference arguments declared by a VAR which, if not omitted, would be interpreted as a local variable declaration.
After the location for insertion has been found, control passes back to INSERT_DECLARATIONS returning flags indicating whether TYPE and VAR were found. If there is no VAR declaration section in the input program, one must be added in order to declare the variables needed for the output file. At this point, the extra variables and their declarations are inserted into the list. If the subprogram is a function, the procedure DETERMINE_FUNCTION_TYPE is called to store the type of the function and the length of the declaration. When control is returned to INSERT_DECLARATIONS, the function type is inserted into the subprogram as the type of the "dummy" function variable which must be used to PUSH and POP stack values which use the function. Pointers are adjusted and control returns to REMOVE_RECURSION.

At this point, a statement is inserted to initialize the stack to NIL. This is done within the procedure INITIALIZE_TO_EMPTY which creates a new node, storing the initialization statement in the node. The node is then linked to the subprogram in the appropriate location before control returns to REMOVE_RECURSION.

Next, a statement is inserted to initialize the stack to NIL. This is done within the procedure INITIALIZE_TO_EMPTY which creates a new node, storing the initialization statement in the node. The node is then linked to the subprogram in the appropriate location before control returns to REMOVE_RECURSION.
The label L1 is attached to the first statement in the recursive subprogram. This allows control to be transferred to the beginning of the subprogram each time a recursive call is executed in the iterative version. The label is attached in the procedure ATTACH_LABEL_TO_FIRST_STATEMENT. Because of the possibility of label attachment, the maximum line length of a statement in the input program is reduced by five to 127 characters instead of 132. ATTACH_LABEL_TO_FIRST_STATEMENT checks to make sure that the current line is not blank, since a labeled blank line produces a syntax error. If the line is blank, the label is attached to the next non-blank line. The label is attached by moving each character in the line over five spaces to allow room for an L followed by a one or two-digit number, a colon, and a space.

When control returns to REMOVE_RECURSION, the procedure MODIFY_RECURSIVE_CALLS is called. This procedure inserts the set of instructions which replace the recursive call with iterative statements. MODIFY_RECURSIVE_CALLS first checks to see if the subprogram is a function. If it is, the procedure STORE_PARAM_TYPE is called.

The purpose of STORE_PARAM_TYPE is to set up two arrays - one holding all of the different parameter and function types, and the other holding the position of each parameter's type in the type array. These arrays are used at the end of the program to set up the variant record declarations. Since each parameter or function may be one of an infinite number of possible system
and user-defined types, the stack component to hold each of these parameters must be a variant record. This record must contain a field for every type of parameter or function in any recursive subprogram in the input file. Since the return address of the call is also pushed onto the stack, an integer field must be included in the variant record description if it is not already present. When STORE_SUBPROG is called, the type array is scanned to determine if the type of the current parameter is already present in the array. The parameter type and each array element are converted to capitol letters to check for equality. If the parameter type is not found in the type array, it is added to the array and the number of types is incremented. There is an element in the parameter code array for each parameter, function result, and address. This element is the location of the corresponding type in the type array. For example, if the third parameter is of the type stored in the first position of the type array, the third element of the parameter code array would hold the value one. This value will also be used in generating the field name of the variant record. The parameter type array, the parameter code array, the number of types and the total number of parameters are passed back to MODIFY_RECURSIVE_CALLS.

The next procedure called by MODIFY_RECURSIVE_CALLS is FIND_FORMAL_PARAMS. This procedure scans the formal parameter list, storing the name and length of all formal parameters. Since it is unnecessary to push any call by reference parameters into the stack, the number of call by value parameters and an
array containing the position of all call by value parameters in
the formal list are also determined. If the subprogram does not
have any formal parameters, the number of formal parameters and
call-by-value parameters are both initialized to zero. To
determine whether a parameter is call-by-reference or call-by-
value, each parameter is checked for a VAR declaration. When a
colon is detected, the type following the colon is associated
with each of the previous parameters of that type by calling the
procedure STORE_SUBPROG. If the parameter type is not already in
the type array, it is inserted. The parameter code is set for
each parameter of the current type. This process continues until
a closing parentheses is found which causes control to return to
MODIFY_RECURSIVE_CALLS.

Control is then transferred to FIND_LOCAL_VARIABLES. This
procedure scans the local variable declaration of the subprogram
to find all of the local variables. If there are no local
variables, a zero is returned for the number of local variables.
When a colon is detected, STORE_PARAM_TYPE is called for each of
the local variables of that type to set up the parameter code and
type arrays. When the pointer to the local variables is equal to
the pointer to the stack declarations, control is passed back to
MODIFY_RECURSIVE_CALLS, returning an array of the local variables
and their lengths.

Also in MODIFY_RECURSIVE_CALLS, the subprogram is searched
for each recursive call. When the name of the subprogram is
found, it is checked to make sure that it is a valid recursive
call and not a function result assignment, a comment, or a literal within a WRITE statement. When the call has been determined to be valid, the call line is stored in a temporary location. The procedure FIND_VALUE_OR_PARAMETER is then called to return the actual parameters, their lengths, and the number of actual parameters.

At this point, code to cause the parameters to be pushed onto the stack is inserted into the subprogram. This is done in the procedure PUSH_PARAMS. PUSH_PARAMS creates a new node for each argument to be pushed onto the stack. A flag is sent to the procedure to indicate if code to push the formal parameters is being inserted. If code is inserted, the index of the call-by-value parameters is used by this code to push the parameters since it is unnecessary to save the value of a call-by-reference variable which will be reinitialized upon return from the call. PUSH_PARAMS sets up each node to be inserted with a statement storing the type code of each parameter along with a field name into the tag field variable for the variant record to be placed on the stack. An assignment statement, storing the value of each parameter into the variant record field of corresponding type is also inserted into the node. Another node is created to push the variant record which has been filled by the previous node onto the stack. PUSH_PARAMS is first called to insert the statements to push any formal call by value parameters onto the stack. It is then called again to insert the code to push all of the local variables onto the stack. The pointers are then adjusted to
merge these insertions into the subprogram. At this point, STORE_PARAM_TYPE is called to set up the type and code for the return address. Then, a new node is created to hold the statements that push the return address of the call onto the stack. First, the type code is appended to a variable. ITERATOR then inserts the code to assign this field to the tag field of the variant record. Then, the return address of the call is stored in the corresponding field of the variant record and the variant record is pushed onto the stack. This node is attached to the previous nodes that cause the stack to be pushed in the output program. Now, a flag is checked to see if the call has any actual parameters. If so, the procedure EVALUATE_ARGS_OR_CALL is called. In this procedure, each formal parameter is compared with the corresponding actual parameter in CHECK_FOR_DIFFERENCE. If the formal and actual parameters are not the same, a flag is returned indicating that the actual parameter must be evaluated and assigned to the appropriate formal parameter. When control passes back to EVALUATE_ARGS_OF_CALL from CHECK_FOR_DIFFERENCE, the flag is checked. If the actual parameter is the same as the corresponding formal parameter, no assignment is made. Otherwise, the procedure CHECK_FOR_SWAP is called. CHECK_FOR_SWAP compares the current formal parameter with the rest of the actual parameters to determine if the formal parameter must be saved in a temporary storage location. If the formal parameter is used in a later actual parameter, the value
currently stored in the formal parameter must be saved in a
temporary storage location to prevent it being overwritten by the
current actual parameter. This is performed by SUBSTITUTE_TEMP.
SUBSTITUTE_TEMP substitutes the name of the temporary storage
location for the formal parameter in each actual parameter
containing the formal parameter. The actual parameter list and
the new lengths of the actual parameters are returned to
CHECK_FOR_SWAP. CHECK_FOR_SWAP continues checking all of the
actual parameters to determine whether any more occurrences of
the current formal parameter are found. The process of checking
and calling SUBSTITUTE_TEMP if the parameter is found is
continued until the entire actual parameter list has been
searched. At this point, the new actual parameter list, the
length list, and a flag indicating whether a substitution took
place are returned to EVALUATE_ARGS_OF_CALL.

If the flag indicates that a substitution was made, code is
inserted to store the value of the current formal parameter into
the temporary storage location before assigning the value of the
corresponding actual parameter to the formal parameter. Also,
the type code of the substituted variable is stored in an array
consisting of all of the type codes of the temporary storage
locations. This array will be used later in declaring the type
of each storage location. Otherwise, the actual parameter is
assigned to the formal parameter, overwriting any previous value
stored in the formal parameter. The pointers are then adjusted