APRIL EUCLID TO PASCAL TRANSLATOR

by

DAVID PAUL ROESENER

B. S., Kansas State University, 1979

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1980

Approved by:
## TABLE OF CONTENTS

1.0 Introduction ........................................................................... 1
    1.1 Purpose ........................................................................... 1
    1.2 Euclid ............................................................................ 1
    1.3 Bootstrapping process ..................................................... 3
    1.4 Organization of this paper ............................................... 8

2.0 General Overview ................................................................. 9
    2.1 Overall Translator Organization ...................................... 9
    2.2 Specific Examples .......................................................... 10
        2.2.1 IF statement .......................................................... 11
        2.2.2 CASE statement ................................................... 13

3.0 Problems of translation .......................................................... 15
    3.1 Declarations in any order ............................................... 15
    3.2 Modules ....................................................................... 16
    3.3 Arbitrarily long definitions ............................................ 19
    3.4 Function declarations ................................................... 22
    3.5 Other problems ............................................................ 24

4.0 Compiler modifications necessary ......................................... 27

5.0 Unsolved problems ............................................................... 28

6.0 Data Structures ................................................................... 31
    6.1 Symbol table ............................................................... 31
    6.2 Stack ......................................................................... 34
    6.3 Procedure initialization queue ....................................... 34

7.0 Conclusions and final remarks ............................................. 36

8.0 References ......................................................................... 38

9.0 Appendices ......................................................................... 39
    9.1 Translation syntax ....................................................... 39
    9.2 Translator code ........................................................... 47
ACKNOWLEDGEMENTS

I would like to thank my major professor, Dr. Rodney Bates, for all the input he has made to this project. He has cleared up my misunderstandings, given me ideas about the translation, read this paper over too many times, and in general has put up with me graciously over the past six months.
CHAPTER 1

INTRODUCTION

1.1 PURPOSE

The purpose of this project is to install a version of Euclid, a relatively new programming language, in the Kansas State University Computer Science computer laboratory. The machine selected for this installation project is the Interdata 8/32, a minicomputer with a functioning Pascal compiler. The Euclid available for this project is an April Euclid compiler, written in April Euclid. The compiler creates PDP-11 object code as its target language designed to run under the UNIX or 11/UNIX operating system.

One solution to the installation problem would be to design a translator from April Euclid to the compilable Pascal. This course of action was selected and the translator created is discussed in this paper.

Before discussing the translator, however, this paper will briefly discuss the Euclid programming language, and the overall installation process.

1.2 EUCLID

April Euclid is a subset of Euclid, developed by B. W. Lampson, J. J. Horning, R. L. Landon, J. G. Mitchell, and G. J. Popek. Its
description appears in the "Report On The Programming Language Euclid" in the February 1977 issue of SIGPLAN Notices [1]. The compiler acquired for this project was developed by the joint effort of the Computer Systems Research Group of the University of Toronto and I. P. Sharp Associates Ltd., both of Toronto, Canada. Its description appears in the documentation accompanying the compiler [2]. Euclid is a language similar to Pascal [3], but carries the philosophy of Pascal further. Euclid has strict type checking, as does Pascal, but also has parameterized type declarations. That is, one type can be used to declare many variables of types differing only by a constant. For example, all variables whose type is a character array can be declared using:

```plaintext
   type CHARARRAY (lower, upper) =
     lower .. upper of char
```

Euclid's variables and types also have implied characteristics which can be referenced in the program. For example, THIS_VARIABLE.Size returns the number of bytes the variable uses in machine code. Euclid also includes ASSERT statements that aid in run-time program proving. The compiler generates a run-time check that produces an error if the ASSERT statement is false. Other features of Euclid are mentioned later in this paper in the discussions of translation techniques.

April Euclid has fewer features than Euclid. In Euclid, modules have both an initialization procedure and a finalization procedure. April Euclid has only the initialization procedure. It also does not allow for parameterized types. Euclid also permits variables to contain sets of objects. April Euclid does not allow this.
According to the Euclid Report, the April Euclid compiler should not be called a compiler, but a translator. The Euclid Report specifies that a Euclid compiler should perform certain functions that this April Euclid translator does not [2]. For example, a Euclid compiler should generate code from ASSERT statements, but the April Euclid translator ignores them. However, this paper will continue to refer to the April Euclid translator as a compiler, to avoid confusion with the April Euclid to Pascal translator.

Hereafter, "Euclid" will be used to mean April Euclid, and "Full Euclid" to denote the language described in the Euclid Report.

1.3 BOOTSTRAPPING PROCESS

The Pascal used as the object language of the translator is a dialect of Pascal as defined by Wirth [3]. The definition of the dialect is in PASCAL/32 LANGUAGE DEFINITION [4]. Hereafter, in this paper, "Pascal" refers to this local dialect.

The first step in bootstrapping Euclid onto the Interdata 8/32 is to write a program, in Pascal, which translates from Euclid to Pascal. The Euclid compiler is then used as input to the translator, producing a Euclid compiler, which is written in Pascal and which generates PDP-11 object code. This Pascal version of the Euclid compiler is then compiled by the Pascal compiler, producing a Euclid compiler written in Interdata 8/32 machine code. This compiler can then be used to compile Euclid source programs to unusable PDP-11 object code. This can be used to eliminate compile-time errors in Euclid source programs. These source programs are then used as input
THIS BOOK CONTAINS NUMEROUS PAGES WITH DIAGRAMS THAT ARE CROOKED COMPARED TO THE REST OF THE INFORMATION ON THE PAGE. THIS IS AS RECEIVED FROM CUSTOMER.
to the translator, producing Pascal programs which can be compiled and run on the Interdata machine. The input/output routines of the Euclid compiler have to be modified before it is translated to Pascal, because the target machines' operating systems are different.

The translator from Euclid to Pascal can assume error-free Euclid programs, because the first code to be translated is the Euclid compiler. This translated compiler is then used to insure that future Euclid programs, used as input to the translator, do not have any errors.

To make bootstrapping processes clearer, T-shaped diagrams are used. Each "T" represents a translator. The left part of the cross bar of the "T" is the source language. The right part of the cross bar is the object language. The bottom of the "T" is the language the translator is written in. The diagram is shown as:

```
   translator  
      /        
     /         
source       object
     
   translator's language
```

If the translator in a diagram is itself translated, the following "T" diagram identity applies. This example shows a translator that is written in language "D". The translator is itself translated by a second translator to language "C". Therefore, the total translation is as if the first translator were written in language "C". The second translator is written in any language that can be interpreted.
If a program is translated to an object language, and this object language is further translated, the following "T" diagram identity applies. The language of the two translators has to be the same.

The following diagrams illustrate the first step of the bootstrapping process. The Euclid compiler is illustrated by the following diagram:
The Euclid compiler is translated to Pascal:

The Euclid compiler is then compiled by the Pascal compiler:
This last "T" diagram shows the point at which Euclid source programs can be compiled to unusable PDP-11 object code.

Finally, the Euclid source programs are used as input to the translator as follows:

The second step of the bootstrapping process is to combine the first five passes of the Euclid compiler with the last four passes of the Pascal compiler. This requires small modifications to both, and the addition of a completely new pass. The result of this final step is a Euclid compiler that translates Euclid to Interdata 8/32 object code. This compiler is written in both Euclid and Pascal.

There are not many alternatives to the described bootstrapping process. The translator from Euclid to Pascal is necessary; otherwise, hand compilation of the Euclid compiler would have been required. An alternative to the second step of interfacing the Euclid compiler to the code generating passes of the Pascal compiler would be to write or rewrite the code generating passes of the Euclid compiler. The intermediate code of the two compilers is similar enough, however, that interfacing the two compilers was chosen as the best alternative.
1.4 ORGANIZATION OF THIS PAPER

This paper assumes some familiarity of the Pascal and Euclid languages and of recursive parsing techniques.

This paper is organized into six additional chapters. Chapter 2 provides the reader with a general overview of the translator, and gives examples of simple syntactic translation. Chapter 3 discusses problems that require semantic processing. Chapter 4 explains compiler modifications necessary to handle some untranslated constructs. Chapter 5 explains problems of translation that are not solved. Chapter 6 discusses the data structures used in the translator. Chapter 7 is then the conclusion.

Finally, appendix A contains the translation syntax, and appendix B contains the code of the translator.
CHAPTER 2
GENERAL OVERVIEW

2.1 OVERALL TRANSLATOR ORGANIZATION

The translator uses recursive descent parsing techniques. Each production rule of the Euclid syntax charts is implemented as a procedure with approximately the same name.

The translator is a one-pass program with a callable lexical scanner. The scanner constructs tokens in two forms: "TEXT" and "TOK". The first form consists of the literal text of the input token, with some translation already performed. For example, Euclid accepts number literals as octal values. The lexical scanner changes these octal values into their decimal equivalents. For another example, Euclid accepts one-character literal strings as the character preceded by a dollar sign. The lexical scanner translates the dollar sign constructs into characters surrounded by single quotes before returning to the main translator. This form is for the purpose of writing the token to the output stream.

The other token form produced by the scanner is a value of an enumeration type. This enumeration type contains a value for each possible token that affects the syntax of the translation. For instance, "CASESY" is the token representing the keyword "case", but "RELOPSY" represents "<", ">", "<="", and ">=". This form of the token is for the purpose of making decisions within the translator.

One more issue of the overall approach of the translator design is discussed here. Pascal programs do not only consist of the standard
definition of Pascal, but they also contain a prefix in the beginning to interface the input/output with the operating system. The translator has to output the prefix to the object Pascal code before it translates the Euclid source. The method chosen for this is simply to copy the prefix from another file. So, there are two input files to the translator. One is the Euclid source, and the other is the file containing the prefix to be appended to the output code.

2.2 SPECIFIC EXAMPLES

Given this particular organization of the translator, the output grammar has to be determined. Most of the output is a simple syntactic translation of the input. Some examples of this straightforward translation follow.

In the syntax diagrams that are used, nonterminals are delimited by "<" and ">". Braces ("{" and "}") around a construct indicate that the construct can be used zero or more times, and square brackets ("[" and "]") around a construct indicate that the construct is optional. Hyphenated identifiers are neither terminal symbols nor productions, but are elements to be explained in the context of the syntax diagram. Double quotes enclose comments about the translation. The symbol "|" indicates different alternatives within the production rules.

One example of straightforward syntactic translation has to do with the precedence of operators in Euclid. Euclid evaluates operators by a different precedence rule than does Pascal. In Euclid, the precedence of operators is:
highest: *, div, mod

+, -

=, not =, <, <=, >, >=

not

and

lowest: or

The order of precedence in Pascal, however, is:

highest: not

*, /, DIV, MOD, AND

+, -, OR

lowest: =, <>, <, <=, >, >=

The symbol "<>" means "not equal".

An example of this difference that occurs frequently in programming is "A = B OR C = D". This expression causes an error in Pascal, but is legal in Euclid. The translation method is simply to rewrite the Euclid grammar to be right recursive instead of left recursive, and to write matching parentheses at each precedence level while parsing the input. This consists simply in writing a left parenthesis at the beginning of all the recursive descent procedures that parse the expressions, and a right parenthesis at the end of these procedures.

2.2.1 IF STATEMENT
The translation of the IF statement is a good example of direct translation. The syntax of the IF statement in Euclid is:

\[
\text{IFST} \rightarrow \text{if} \ (<\text{expn}> \ \text{then} \ \{\ <\text{stmt}> \}) \\
\hspace{1cm} \{\ \text{elseif} \ (<\text{expn}> \ \text{then} \ \{\ <\text{stmt}> \}) \} \\
\hspace{2cm} [\ \text{else} \ \{\ <\text{stmt}> \} \] 
\]

The corresponding output production is:

\[
\text{IFST} \rightarrow \text{IF} \ (<\text{expn}> \ \text{THEN} \ <\text{STMTLIST}> \\
\hspace{1cm} [\ \text{ELSE IF} \ (<\text{expn}> \ \text{THEN} \ <\text{STMTLIST}> \) \\
\hspace{2cm} \text{ELSE} \ <\text{STMTLIST}> 
\]

Note that the else clause is always included in the output grammar.

The resulting pseudo code for this translation is:

\[
\text{OUTPUT CURRENT TOKEN "IF"} \\
\text{NEXT "consume IF"} \\
\text{CALL EXPN} \\
\text{OUTPUT CURRENT TOKEN "THEN"} \\
\text{NEXT "consume THEN"} \\
\text{CALL STMTLIST "which outputs BEGIN...END"} \\
\text{DO WHILE TOKEN = 'ELSEIF'} \\
\hspace{1cm} \text{OUTPUT 'ELSE IF'} \\
\hspace{2cm} \text{NEXT "consume 'ELSEIF'"} \\
\hspace{2cm} \text{CALL EXPN} \\
\hspace{1cm} \text{OUTPUT TOKEN "THEN"} 
\]
NEXT "consume THEN"
CALL STMTLIST
OUTPUT 'ELSE'
IF TOKEN = 'ELSE' THEN
    NEXT "simply consume token"
CALL STMTLIST "which can be null"
NEXT NEXT "consume 'END IF'"

2.2.2 CASE STATEMENT

One final example of straightforward translation is the CASE statement. The syntax for the CASE statement is:

CASEST--> case <expn> of
    { <label> {, <label> } = {<stmt>} }
    end <label>}
[otherWise = {<stmt>}]}
end case

The Pascal output syntax is:

CASEST--> CASE <EXPN> OF
    { <LABEL> {, <LABEL>} : <STMTLIST> }
    [ ELSE <STMTLIST> ]
END

The translator changes "=" to ",", changes "OtherWise" into
"ELSE", deletes the "case" after the last "end" of the syntax graph, and encloses all the statements of each case designator in a BEGIN...END block. The translator also has to modify the labels as it does all identifiers in a process as described later.
CHAPTER 3
PROBLEMS WITH TRANSLATION

There are, however, some problems that cannot be handled in a straight-forward manner. They require semantic processing.

One problem is that Euclid permits variables to be initialized in their declarations. The object code has to have these variables initialized in assignment statements in the beginning of the block in which they are declared. The translator makes use of a stack to store the variables and their initialization values, as described later.

3.1 DECLARATIONS IN ANY ORDER

A major problem is that, in Euclid, identifiers can be declared in any order, as long as they are declared before they are used. The only exception to the rule that they have to be declared before their use is the declaration of recursive routines. In Pascal, however, constants and types have to be declared first, then variables, then routines, and then the main block of the outermost structure. The obvious solution to this problem is to output the structures to different files as they are declared. Then at the end of the source code, all the files are read in, in their proper order, and written to the object code file. But, the Pascal compiler enforces the rule about the order of declarations in only one procedure. Therefore, it is reasonable to change the definition of Pascal and modify the
compiler to accept declarations in any order.

3.2 MODULES

Another problem arises from the fact that Euclid allows modularization of structures. These structures are grouped together within a structure called a module. These modules are very similar to Pascal classes, with the exception that any identifier declared within the module can be referenced outside the module if the module specifically states that ability as one of the identifier's attributes. The way the module does this is to include the identifiers in its export list.

Since object Pascal has classes as part of its definition, an obvious solution to the translation of modules is to translate them directly to classes. The problem with this solution is that Pascal classes allow only its procedures and functions to be referenced outside the class. To implement this solution would require moving all type, constant and variable declarations outside the class. Furthermore, the declarations would have to be moved ahead of the class. This would involve the same process as the one required if the object Pascal code followed standard Pascal rules pertaining to the order of declarations. The class could not be written to the output stream, but to a temporary file, until the entire class had been processed. Then, the class could be read in and written to the regular output stream.

Instead, another option to solve the problem of modules is used. It was decided to eliminate the modules entirely, and simply
output the constructs within the module as if they were not within the module. This would not have been an acceptable solution if the translator were not assuming error free Euclid source programs. It would have defeated the entire purpose of Euclid to keep constructs strictly modularized. But, the translator can assume the modularization has already been enforced.

There remain problems with that solution. Since the modules constitute a new scope within Euclid, eliminating them eliminates the new scope, and thus naming conflicts can easily arise. To avoid these naming conflicts, the names of all the modules that an identifier is nested inside are appended to each use of that identifier. So, if procedure "A" is declared inside of module "B" which is inside of module "C", the identifier appears as "C_B_A" in the object program. Furthermore, if the procedure "A" is used within module "B" in Euclid, it is called simply by it's name "A". The translator appends "C" and "B" to it, producing "C_B_A", because the declaration of "A" is within those two modules. Also, if the procedure is called within module "C", but outside of module "B", Euclid semantics require it to be called with the qualifier "B.", so it is called by "B.C". The translator locates the identifier "B" in the symbol table, and appends "A" to it. Then, since the identifier "B" is a module, it has to be followed by the delimiter ".". The translator changes the "." to "_.", and looks for the identifier "A" in module B's own symbol table, which is described later. Since "A" is not a module, if there were any following ".", they would not be changed, because "A" would be a record identifier. Therefore, "B.A" would be translated to C_B_A, and the result is consistent with the declaration of procedure "A".

To illustrate this translation further, an example of a "Euclid"
program and the rough "Pascal" translation follows:

The "Euclid" source is:

```
MODULE OUT
    PROCEDURE AA
    END AA

MODULE MID
    PROCEDURE BB
    CALL AA
    END BB
    END MODULE MID
    CALL MID_BB
    CALL AA
    END MODULE OUT
    CALL OUT.MID_BB
    CALL OUT.AA
```

The resulting "Pascal" is:

```
(* MODULE OUT *)
    PROCEDURE OUT_AA
    END (* AA *)

(* MODULE MID *)
    PROCEDURE OUT_MID_BB
    CALL OUT_AA
    END (* BB *)
    (* END MODULE MID *)
    CALL OUT_MID_BB
    CALL OUT_AA
```
3.3 ARBITRARILY LONG DEFINITIONS

Some problems arise because Euclid allows some declarations to contain arbitrarily long definitions in some places where Pascal allows only identifiers or simple types or simple constants.

One such place is in subranges. Euclid permits a subrange to be expressed as "3+4..9" whereas this is illegal in Pascal. One solution to this problem would be to evaluate all compile time expressions in the translator, and replace the expressions with their values. So, "3+4..9" would be changed to "7..9".

This technique would require the use of a symbol table. The following example would require that the value for "C" be stored in the symbol table:

\[
\text{const } c := 4 \\
\text{type } a = 3+c .. 9
\]

But, another alternative was chosen, because the same technique described is used to solve a similar problem. This alternative is to convert these compile-time expressions imbedded within type declarations to generated constant declarations. Object Pascal allows constants to be declared as expressions, and not only as simple constant values. The translator has to declare these generated
constants before it outputs anything else, so it suppresses outputting anything but generated identifier declarations until it is sure it will not encounter any more arbitrarily long definitions. It then uses the generated constants in the place of the compile time expressions.

For example,

\texttt{TYPE A = 1+2+3..8*10 would be translated to}

\texttt{CONST SYSID\_1 = 1+2+3;}

\texttt{CONST SYSID\_2 = 8*10;}

\texttt{TYPE A = SYSID\_1..SYSID\_2;}

Another example of the possibility of arbitrarily long definitions where Pascal's syntax does not allow immediate output of tokens is the declaration of structured constants. Pascal's structured constant declarations have this syntax:

\texttt{CONST <ID> = ( <LIST-OF-CONSTANTS> ) :}

\texttt{ARRAY [ <RANGE> ] OF <TYPEID>}

But, the syntax for structured constants in Euclid is:

\texttt{const <id> : array <range> of <type>}

\texttt{( <list-of-constants> )}

Not only does Pascal require a type identifier where Euclid allows a type declaration, but also, Euclid's type comes before the constants to be loaded into the structure and Pascal's type comes after the constants. The translator does not output any part of the
structured constant until it reaches the list of constants to be loaded into the structure. But, it does output many intermediate generated types, until one generated type declares the structure constant type. The translator then outputs "const ID =" and the list of constants to be loaded. It finally can output the generated type that declares the type of the Euclid generated constant at the end as the type of Pascal's structured constant type.

For example, the following Euclid source is translated into the following Pascal:

```
const Test : array 1..4 of Integer :=
(1,3,5,7)
```

The Pascal is:

```
Const Sysid_1 = 1;
"This starts to process the range."
Const Sysid_2 = 4;
Type Sysid_3 = Sysid_1 .. Sysid_2;
"This declares the range."
Type Sysid_4 = Integer;
"'Integer' could be a multi-token
 type definition. So, this processes
 that possibility."
Type Sysid_5 = Array [Sysid_3]
of Sysid_4;
"We are finally ready to process
the structured part:" Const ID = (1,3,5,7) of Sysid_5;
```
The above example shows almost a complete example of the necessity to save information inside the translator to make legal output text. The translator saves the information in the form of generated identifiers and declares these identifiers before it processes the structure being translated.

3.4 FUNCTION DECLARATIONS

There are several problems that arise from function declarations. Euclid returns values in two ways: one is by accompanying the return statement by the expression that is to be returned, and the other is by assigning a value to a locally declared variable that is declared as the function return variable. Pascal returns values by assignment to the function name, and does not have a return statement.

To translate values returned by way of Euclid's local return variable, the translator treats that local variable as a regular local variable, but it omits an assignment statement of that local variable to the function name as the last statement of the function definition.

This assignment statement has a statement label of "1" for all functions, so a return statement is translated into a "goto 1" statement. If the return has an expression associated with it, the return statement is translated into a compound statement, which is composed of two statements. The first of the two statements is an assignment statement of the expression into Euclid's local return
variable. The second statement then is the "goto 1" statement. This method translates the second way that Euclid returns function values.

One problem still has to be considered. The return value's type is declared only once in the Euclid's source. This one time is as the type of the return variable. But, to use this described method of translation, the return value's type has to appear twice in the object code. It has to appear as the type of the function itself, and it has to appear as the type of the local variable that Euclid uses as the return variable.

Assuming that the type is a one-token type, the type that the single token translates into is simply saved and used in the two places. An example of a normal function declaration follows:

```plaintext
function f (oneparm: integer, secparm: real)
  returns ret : char =
  begin
    ret := 'F'
    return ('G')
    "some more stmts"
  end f
```

This function is translated as follows:

```plaintext
FUNCTION F (ONEPARM: INTEGER; SECPARM: REAL) :
    CHAR;
VAR RET : CHAR; "The return type appears twice"
BEGIN
  RET := 'F';
  "This BEGIN...END block translates
  'return ('G')': "
```
BEGIN
RET:="G";
GOTO 1
END;
"some more stmts"
1: F:=RET
END;

But, the problem is further compounded by the fact that Euclid's return variable's type does not have to be either a simple type or a type identifier. In Pascal, the type of the function does have to be either a simple type or a type identifier. This problem is similar to the class of problems that was just described in the section "ARBITRARILY LONG DEFINITIONS". But, to output generated identifiers to declare the return value's type as described previously, the output of the entire function declaration would have to be suppressed until that point. That is not an acceptable alternative because the function declaration up to that point is also arbitrarily long. So, the translator does not handle the case if Euclid's return value's type is defined as more than a simple type or a type identifier. If the return value's type is not a one-token type, the translator outputs an error message, and the source has to be modified.

There are several more similar cases that the translator does not handle, and this class of problems is discussed later, in the section "UNSOLVED PROBLEMS".

3.5 OTHER PROBLEMS
Another problem is generated by the method Euclid uses by which a programmer may bypass the strict type checking rules. Euclid contains a construct called a converter. If a programmer wishes to consider a variable of one type as having a different type, he must declare a converter in the block he is going to breach the strict type checking rules. The declaration of the converter consists of its name, the type of the variable, and the new type of the variable. Then, to breach the strict type checking rules, he gives the name of the converter followed by the variable name in parentheses. For example, if "I" is an integer and he wants it considered as a character variable, the section of code would be as follows:

```
converter Inttochar (integer) returns char
Some_Routine(first-parm, Inttochar(I), third-parm)
```

This section of code passes "I" to the routine "Some_Routine" as a character, instead of as an integer.

Declaring formal parameters as universal is Pascal's method of breaching type compatibility rules. An actual parameter of any type can be passed to a universal formal parameter if their lengths are equal.

To translate this, the translator simply makes all formal parameters universal, thereby making most types compatible. This would, as before, not be an acceptable solution, if the translator was not assuming error-free input. Furthermore, when the translator is processing an identifier, if that identifier is a converter, as indicated in the symbol table, then the translator deletes it and the parentheses around its parameter. The converter declarations are also
deleted.

One final problem that is not trivially solved pertains to Euclid loops. The only way a loop is terminated in Euclid is to execute an EXIT statement. The EXIT statement essentially means that the flow of control is to be given to the first statement after the loop. The method of translation of LOOP statements and EXIT statements is as follows:

Each loop is translated to "WHILE TRUE DO <STMTLIST>" and is assigned a loop number. Then this loop number is written after the loop is processed as the statement label of the first statement following the loop. The loop number is also passed to the "STMTLIST" procedure so that any EXIT statements in that list can use it in their translation. The EXIT statements, then, translate simply to "Goto loop-number".
CHAPTER 4
PASCAL COMPILER MODIFICATIONS

Some of the translation would have been difficult assuming the existing Pascal, but redefining Pascal and modifying the compiler simplifies matters greatly. The compiler changes are as follows:

Some words are keywords in Pascal, but not in Euclid. These words can be used as regular identifiers in a Euclid source program. These words have to be changed within the translator or deleted from the keyword list in the lexical scanner in the Pascal compiler. The alternative chosen is to delete them from the keyword list of the Pascal compiler. At the time of writing, only one word has been discovered which requires this treatment. That word is the keyword "Entry".

As mentioned previously, the order of declarations has to be changed to allow constructs to be declared in any order.

Another problem arises having to do with the declaration of structured constants, and the translation technique of writing layers of parentheses to insure that the order of evaluation is correct. The only way Pascal distinguishes between a structured constant and a regular constant is that a structured constant's definition begins with a left parenthesis. But, the translation technique for expressions writes a left parenthesis for a regular constant, also. So, the compiler is modified to allow a left parenthesis to introduce the definition of a regular constant. It is also modified so that structured constants are declared with a new keyword "STRUCTCON". Therefore, let the reader note that some previous examples are not strictly correct.
CHAPTER 5
UNSOLVED PROBLEMS

There are, naturally, constructs in Euclid that would be very difficult to translate into Pascal. As many problems are handled as possible, but some constructs are considered unreasonable to translate. For these constructs, the Euclid source should be modified. The translator prints out error messages in most cases where the construct is not translated.

One unsolved problem is the definition of type compatibility of the two languages. Euclid defines two types to be compatible if they look "the same" after recursive replacement of constant and type identifiers by their definitions. Pascal defines two complex types to be compatible only if they are declared in the same type definition.

For instance, the following is legal in Euclid, but not in Pascal:

\begin{verbatim}
  type a = (blue, green);
  type b = (blue, green);
  var aa : a;
  bb : b;
  begin bb := aa end
\end{verbatim}

Another problem already alluded to has to do with Pascal's restriction on declarations of types and ranges. Euclid allows expressions in ranges but Pascal does not. Euclid allows a complex type definition to be used in places where Pascal requires that a
single type identifier be used. As stated previously, many of these cases are translated by declaring the complex type or expression as generated identifiers previous to their use.

But, there are some cases where generated identifiers can not be used to solve this problem. These are the cases that an arbitrarily long string would not be allowed to be output because the translator has to check the entire construct for these special cases, before it can output anything of the construct. These cases are the return value of a function, as was mentioned previously, labels of case statements, ranges in record declarations, and complex type definitions inside of formal parameter lists.

For instance, the expression "3+4" can not be replaced by a generated constant in the following example:

```
Type a_rec =
  record
    id1,id2,id3,id4,etc: array [1..4] of real;
    awholebunchofstuff: thisisalongtype;
    this_is_the_important_part:
      array [3+4..9] of integer;
  end;
```

A difficult problem arises having to do with recursive procedures. As mentioned previously, a recursive procedure is the only construct that can be referenced before it is declared. It needs only to be imported to the scope where it is being used. Pascal does not permit any identifier to be referenced before it is declared. It forces a recursive procedure to be declared by a "forward" declaration before its use, although its body can appear later. The translator
does not output any forward declarations, so this problem remains unsolved.

One other problem that the translator does not take care of has to do with the built in attributes that variables and types have, that were mentioned in the introduction. These attributes make the identifier look like a record when used. Therefore they produce compile-time errors in Pascal.

Finally, there is a possibility that appended identifiers or generated identifiers could conflict with identifiers in the source. However, the Euclid compiler is written using a naming convention that does not allow "_" in identifiers. Therefore, this problem will not occur in the Euclid compiler.
CHAPTER 6
DATA STRUCTURES

Now the data structures used in the various techniques of translation shall be described. The most interesting one is the symbol table.

6.1 SYMBOL TABLE

There are several uses of the symbol table, but the principle one is in appending the identifiers with the names of enclosing modules. To accomplish the appending, the following structure is used. The structure is the conglomerate symbol table.
An overview of the general procedure is as follows: Each time a new scope is entered, another display record is pushed on the display stack. This display record has the text to be appended to any identifiers declared inside the scope, and it also has a pointer to the scope's own individual symbol table. So, when an identifier is being processed, it is searched for in the conglomerate symbol table. When it is found, all the append texts are output, starting from the bottom (i.e. the root) of the display and continuing to the display record pointing to the symbol table where the identifier is found. These append texts are separated by "_". Then the identifier itself is output, and the result is consistent with the declaration of the identifier. Furthermore, if the identifier is a module, the next token is assured to be ".". This "." is changed to a "_", and the next identifier is found in the module's own symbol table. If that identifier is another module, the process is repeated with it. That is, the next token after it is assured to be a ".", and it needs to be changed to "_". When an identifier is encountered in the process that is not a module, if any more "." are encountered, the identifier is assured to be a record and the "." are not to be translated to "_".

When the scope is left, the display record is popped off the display stack, and its identifiers then become inaccessible directly from the display. If the scope that is left is a module, the identifiers are accessible, but only through a pointer in the symbol table containing the module's declaration.

The method of searching for an identifier is to start at the top of the display and its symbol table, and then continue with the other symbol tables going towards the bottom of the display. This method assures that if an identifier is found in the conglomerate symbol table, it is the correct one. It is the innermost accessible
identifier whose scope is accessible at that point.

When the new scope is a procedure, its append text is simply a blank field. Any identifier declared within that field has no need of modification, since it would be a local variable to the procedure. Therefore, when an identifier is encountered whose display's append text is a blank field, the identifier is not appended.

There are two other cases that the symbol table is used in translation. One case is that, syntactically, it is impossible to determine if an identifier is a function call or an array in Euclid. Both these structures' arguments are delimited by parentheses. In Pascal, however, "[" and "]" delimit the indices of an array, and "(" and ")" delimit the parameters of a function call. So, the symbol table contains the information as to the type of the identifier, also. If an identifier is not indicated as being a procedure, then the translator changes any following parentheses to brackets. Otherwise, the translator leaves the parentheses as parentheses. The second translation technique in which the symbol table is used for is the converter constructs already mentioned. If an identifier is encountered that is a converter, it simply is deleted from the output text. Its parentheses are also deleted.

There are some identifiers that have to be declared in the symbol table at the bottom of the display, because these identifiers are not declared, but they are used. All the identifiers declared in the prefix have to be declared in the bottom symbol table, the boolean constants "true" and "false" and other predefined identifiers have to be declared there, and the parameters of the program declaration itself have to be there. This insures that when these identifiers are encountered, they are accounted for in the conglomerate symbol table.

The examples in this paper show that each individual symbol
table is a linear list, but each table, in actuality, is an unbalanced, inordered binary tree.

6.2 STACK

Another stack is also used whose only purpose is to deal with the variable initializations in their declarations. There are three distinct places where variables can be initialized in their declarations: module variables, local procedure variables, and the local variables of module initialization routines. When either a procedure or module declaration is entered, the stack is marked. Each variable that is initialized in that structure and its initialization value are pushed on the stack. Then, when the code body is encountered, the stack is popped until the stack marker is encountered, and the records that are popped are used to output execution time assignment statements.

Since the module initialization procedure has to initialize both its own local variables, and the module's variables, the stack is actually popped twice. Therefore the stack must be marked an extra time when a regular procedure is entered, so that the synchronization is correct.

6.3 PROCEDURE INITIALIZATION QUEUE

The final data structure that is mentioned in this section is
the procedure initialization queue. Each initialization procedure for
the modules is declared as the module name followed by "_INITIAL" in
the translation. The module name is then stored in the procedure
initialization queue, so that the main body of the Pascal object
consists entirely of the invocation of these procedures. When the
Euclid source is exhausted, each module name is withdrawn from the
queue. The module name is then written, followed by "_INITIAL". This
is consistent with these procedures' declarations.
CHAPTER 7
CONCLUSIONS AND FINAL COMMENTS

To date, the translator has been completed. The first pass of the Euclid compiler has been modified and translated to Pascal. This Pascal object has been compiled, producing only those compile-time error messages that are a result of untranslated constructs in the source. These are all expected errors. The future work involves translating and compiling the rest of the Euclid passes. Then these passes need to be modified so that they produce an intermediate code to be used as input to the code-generating passes of the Pascal compiler. These code-generating passes of Pascal may need to be modified, also. The modified Euclid passes will need to be debugged and translated. When this process is completed, Euclid will be installed on the computer science department's Interdata 8/32 in its final form.

Several modifications of the translator could be made. All of the unsolved problems mentioned in this report are solvable. More elaborate data structures could be developed which would allow reordering of declarations, as well as translating all types in parameters and function results. The data structure would also solve the problem of expressions used in ranges in record declarations, and in case labels.

The translator outputs too many generated identifiers. It could be modified to check to see if there is really a need for a bound on a range to be declared as a system constant, or if the bound is acceptable as it already is.
For instance,

    Type a = 3..6 is translated as:

    Const Sysid_1 = 3;
    Const Sysid_2 = 6;
    Type Sysid_3 = Sysid_1 .. Sysid_2;
    Type a = Sysid_3;

This translation is not necessary in this case, and the
translator could be modified to let a simple case like this remain as
it already is.
REFERENCES


4] Young, Robert and Wallentine, Virgil. PASCAL/32 LANGUAGE DEFINITION. Kansas State University, Manhattan, Kansas, 1978
APPENDIX A

TRANSLATION SYNTAX

In the translation syntax diagrams, the Euclid production is given, then the Pascal translation production, then comments about the translation. Each Euclid production is numbered consecutively. The name of the Pascal production is the same as the Euclid production, but the name given the procedure that parses that production in the translator follows in parentheses before the actual definition of the Pascal production.

In the definitions, square brackets ("[" and "]") enclose optional constructs. Braces ("{" and "}") enclose constructs that can be used zero or more times. The symbols "<" and ">" surround names of other productions. The symbol "!" is used to indicate a choice between two paths in the production. Double quotes surround comments about the translation. When "!" is used, it is assumed that the choices that "!" delimit are obvious, and do not need further clarification. Semicolons are generally left out of the translation syntax.

1) variableDeclaration --> var <id> : <variableType>
   [ : = <expn> ]
   | var <id> : {array <range> of} <variableType>

Pascal:
variableDeclaration (MOD_OR_VAR_DCL) -->
   [ CONST SYS-GEN-ID = <expn>; ]
   <APPENDED-ID> : SYS-GEN-ID-2
   "passed by production typeDefinition"

SYS-GEN-ID-2 is passed to variableDeclaration by typeDefinition after it has been used to declare the variable's type. SYS-GEN-ID is used to declare the initialization value of the variable. It is then used in the run-time assignment statement. "<id>" is explanatory enough and will not be further developed.

2) variableType --> SignedInt
   | <range>
   | Boolean
   | ( <id> [ , <id> ] )
   | Char
   | [ <moduleId> . ] <typeId>

Pascal:
variableType (VAR_TYPE) --> TYPE SYS-GEN-ID =
   INTEGER
   | SYS-GEN-ID-2 "passed by production range"
   | BOOLEAN
   | ( <APPENDED-ID> [ , <APPENDED-ID> ] )
   | CHAR
   | <APPENDED-ID> [ _ <id> ]

APPENDED-ID is produced by appending all module names, inside of which the identifier is nested, to the identifier. "moduleId" and "typeId" are explanatory enough, and will not be further defined.
3) range --> <expn> .. <expn> | typeId

Pascal:

\[
\text{range (RANGE) --> CONST SYS-GEN-ID = <expn>;}
\]
\[
\text{CONST SYS-GEN-ID-2 = <expn>;
\text{TYPE SYS-GEN-ID-3 = SYS-GEN-ID .. SYS-GEN-ID-1;}
\]
\[
\text{TYPE SYS-GEN-ID-3 = <APPENDED-ID>}
\]

SYS-GEN-ID-3 is then passed to the "calling" production as the range's type.

4) typeDeclaration --> [pervasive ]
   type <id> = typeDefinition

Pascal:

\[
\text{typeDeclaration (TYPE_DCL) --> [ (* pervasive *) ]
\]
\[
\text{TYPE <APPENDED-ID> = SYS-GEN-ID}
\]

SYS-GEN-ID is generated because of the possibility of the existence of expressions within ranges in its definition.

5) typeDefinition --> {array <range> of} variableType
   | record
   |   [ <variableDeclaration> ]
   end <id>
   "The id is the type id inside of which the record definition appears."

Pascal:

\[
\text{typeDefinition (TYPE_DEF) --> TYPE SYS-GEN-ID =
\]
\[
\text{[ ARRAY [<range>] OF ]
\]
\[
\text{SYS-GEN-ID "This generated id is passed by typeDefinition
\text{or variableType, depending on if there are more than one dimension of arrays."
\]
\[
\text{TYPE SYS-GEN-ID = RECORD
\text{<RECORD-VAR-DCL> \{; <RECORD-VAR-DCL>\}
\text{END (* <id> *)}
\]

"RECORD-VAR-DCL" is not parsed by the translator. Each token is processed individually. It is just like "variableDeclaration" except that it is inside of a construct where no system identifiers can be produced and defined. Therefore, if expressions appear inside of ranges, they generate an error in the object. Also, "var" inside of RECORD-VAR-DCL has to be deleted.

6) structuredConstantDecl --> [ pervasive ]
   const <id> : array <range> of variableType := ( <expn> \{, <expn>\} )

Pascal:

\[
\text{structuredConstantDecl (CONST_DCL) --> [ (* pervasive *) ]
\]
\[
\text{CONST <APPENDED-ID> = STRUCTCON ( <expn> \{, <expn>\} )
\]
\[
\text{: SYS-GEN-ID "passed by typeDefinition"}
\]
"STRUCTCON" is a new keyword developed for the modified Pascal compiler so that it can distinguish between a structured constant whose definition has to begin with a left parenthesis in standard Pascal, and a simple constant, whose definition always starts with a left parenthesis by the translator.

7) constantDeclaration (CONST_DCL) -->  [ pervasive ]
   const <id> := <expn>

Pascal:
   constantDeclaration (CONST_DCL) -->
   CONST <APPENDED-IDENTIFIER> = <expn>

8) expn --> <disjunction> [ -> <disjunction> ]

Pascal:
   expn (EXPN) --> ( <disjunction> | error-condition )

"->", which means "implies" in the logical sense, is not available in Pascal.

9) disjunction --> <conjunction> | <disjunction>
   or <conjunction>

Pascal:
   disjunction (DISJUNCTION) --> ( <conjunction>
   {OR <conjunction>} )

10) conjunction --> <negation> |
   <conjunction> and <negation>

Pascal:
   conjunction (CONJUNCTION) --> ( <negation>
   {AND <negation>} )

11) negation --> <relation> | not <relation>

Pascal:
   negation (NEGATION) --> ( [NOT] <relation> )

12) relation --> <sum> | <sum> <relationOperator>
   <sum>

Pascal:
   relation (RELATION) --> ( <sum>
   [ relationOperator <sum> ] )

"relationOperator" consists of ":="", "not =", "<", "<=", ">", and ">=".
13) sum -- <term> | <sum> addingOperator <term>

Pascal:
    sum (SUM) -- ( <term> {addingOperator <term> } )

"addingOperator" consists of "+" and "-".

14) term -- <factor>
    | <term> multiplyingOperator <term>

Pascal:
    term (TERM) -- ( factor
        { multiplyingOperator <factor> } )

"multiplyingOperator" consists of "*", "div", and "mod".

15) factor -- <variable> | literal-constant |
    constantIdentifier | functionDesignator |
        ( <expn> ) | - <factor>

Pascal:
    factor (FACTOR) -- { - } <variable>
    | { - } ( <expn> )

The difference between function calls and arrays is handled semantically by manipulation of the symbol table. The production "variable" handled all the rest of the cases. It is defined here:

    variable -- <APPENDED-ID> { . <id> | (<expn> {, <expn> } ) ) }  

16) statement -- <variable> := <expn>
    | { moduleId . } procedureId
        [ ( <expn> {, <expn> } ) ]
    | exit [ when <expn> ]
    | return [ ( <expn> ) ]
    | assert [ ( <expn> ) ]
    | begin [statement] end
    | loop [statement] end loop
    | if <expn> then [statement]
        { elseif <expn> then [statement] } 
    | else [statement] ] end if
    | case <expn> of
        {<expn> {, <expn>} =>
            [statement] end <expn> }
    | otherwise => [statement] ] end case

Pascal:
    statement (STMT) -- <variable> := <expn>
        [ <APPENDED-ID> { . <id> } ]
        [ ( <expn> {, <expn> } ) ]
    | [ IF <expn> THEN ] GOTO <STATEMENT-LABEL>
    | BEGIN [ <FUNCTION-LOCAL-VAR-NAME> := <expn> ] GOTO 1 END
    "1" is the label of the last statement in the function body."
    | (* ASSERT [ ( <expn> ) ] *)
    | <STATEMENT-LIST>
\[ \text{WHILE TRUE DO} \ <\text{STATEMENT-LIST}> \ ; \ \\
<\text{STATEMENT-LABEL}> : \\
\text{IF} \ <\text{expn}> \ \text{THEN} \ <\text{STATEMENT-LIST}> \\
\text{ELSE IF} \ <\text{expn}> \ \\
\quad \text{THEN} \ <\text{STATEMENT-LIST}> \} \\
\text{ELSE} \ <\text{STATEMENT-LIST}> \\
\text{CASE} \ <\text{expn}> \ \text{OF} \\
\quad \{ \text{LABEL [, Label]} : <\text{STATEMENT-LIST}> \} \\
\text{ELSE} : <\text{STATEMENT-LIST}> \} \\
\text{"STMT can be null in translation."} \]

In the CASE statement, Euclid allows expressions whereas Pascal allows only one-token labels. If an expression occurs, an error in the object Pascal will result. The labels are not parsed. In the LOOP statement, the label is unique for that procedure. That label is passed to STATEMENT-LIST so that the EXIT statements can be translated with the correct statement labels. In the RETURN statement, the method of returning function values in Euclid is completely different than the method in Pascal. Please refer to the report body in the chapter on "PROBLEMS" in the section on "FUNCTIONS". In the LOOP statement, the only way to exit a loop in Euclid is by executing an EXIT statement. CASE and IF statements are further explained in the main body of this paper.

The production "STATEMENT-LIST" does not appear in the definition of Euclid. It is a production used only in this translation syntax. It is defined as:

\[
\text{STATEMENT-LIST} \rightarrow \text{BEGIN} \{ \ <\text{statement}> \} \ \text{END}
\]

17) \text{routineDeclaration} \rightarrow \{ \ \text{pervasive} \}

\begin{align*}
\text{procedure} & \ <\text{id}> \ \text{paramList} = \\
& \ <\text{routineDefinition}> \\
\text{function} & \ <\text{id}> \ \text{paramList} \\
\quad \text{returns} & \ <\text{id2}> : \ \text{variableType} = \\
& \ <\text{routineDefinition}> \\
\end{align*}

Pascal:

\[
\text{routineDeclaration} \ (\text{PROC\_DCL}) \rightarrow \{ \ (* \ \text{pervasive} \ *) \} \\
\quad \text{PROCEDURE} \ <\text{APPENDED-ID}> \ <\text{paramList}> ; \\
\quad \ <\text{routineDefinition}> \ \text{END} \\
\{ \ (* \ \text{pervasive} \ *) \} \\
\quad \text{FUNCTION} \ <\text{APPENDED-ID}> \ <\text{paramList}> : \ <\text{ONE-ID-OF-VAR-TYPE}> \\
\quad \ (* \ \text{RETURNS ID2 : *} \) \\
\quad \text{VAR} \ <\text{id2}> : \ <\text{ONE-ID-OF-VAR-TYPE}> ; \\
\quad \ <\text{routineDefinition}> \\
1: \ <\text{APPENDED-ID}> := \text{id2} \ \text{END}
\]

If the return value type of "id2" is not a simple type or a type identifier, an error in the object Pascal occurs. For this reason, the translation syntax outputs the first token of the type (called ONE-ID-OF-VAR-TYPE in the syntax above) as the type of both the return value of Euclid’s local variable and of Pascal's function type. Please see the main body of this paper for more information.

18) \text{paramList} \rightarrow \{ \ ( \ \text{formalSection} [, \ \text{formalSection} ] \} \} \]
Pascal:

paramList (PARM_LIST) -->
  [ ( <formalSection> { ; <formalSection> } ) ]

19) formalSection --> [ pervasive ]
   var | readOnly | const | "null"
   <id> : { array <range> of }
   <varType>

Pascal:

formalSection (FORMAL_SECT) --> [ (# pervasive #) ]
   var | (* readOnly *) | (* const *) | "null"
   APPENDED_ID : UNIX "to bypass type checking"
   error-condition | one-token-type

In Pascal, the type of a formal parameter has to be a simple type or a type identifier. If there are arrays or enumeration constants in the source's formal parameter list, there will be an error in the Pascal object.

20) routineDefinition --> [ imports ( importItem { , importItem } ) ]
   [ pre [ ( <expn> ) ] ]
   [ post [ ( <expn> ) ] ]
   <routineBody>

Pascal:

routineDefinition (ROUT_DEF) -->
  (# everything-until "begin" or "code" #)
  <declarationInRoutine>
  BEGIN
  "Here the variable initializations are written as assignment statements."
  <STATEMENT-LIST>
  | BEGIN "Here the input is copied literally."

This Pascal production definition includes not only "routineDefinition", but also "routineBody", which is defined here:

routineBody --> begin
  <declarationInRoutine> }
  {<statement>}
  end
  | code
  | literal-Pascal-text
  | end

The code body is supposed to be PDP-11 assembly language statements, but for translation purposes it is Pascal source.

21) declarationInRoutine --> <typeDeclaration>
    | <constantDeclaration>
    | <variableDeclaration>

Pascal:
declarationInRoutine --> <typeDeclaration>  
| <constantDeclaration>  
| <variableDeclaration>  

This is actually incorporated into "ROUT_DEF" in the translator.

22) moduleDeclaration --> var  
    <id> : <moduleType>  

Pascal:  
    moduleDeclaration (MOD_OR_VAR_DCL) --> (* var  
    <id> : #) <moduleType>  

    The identifier is added to the symbol table for appending  
    purposes.

23) moduleType --> [ machine dependent ]  
    module  
    [ imports ( <importItem> {, <importItem> } ) ]  
    [ exports ( <exportItem> {, <exportItem> } ) ]  
    [ [ not ] checked ]  
    {dclInModule}  
    [ initially  
      <routineDefinition> ]  
    end module <id> "The id is the module's name."

Pascal:  
    moduleType (MODULE_TYPE) --> [ (* machine dependent *) ]  
    (* module *)  
    { (* imports inside-import-list *)  
      { (* exports inside-export-list *)  
      | dclInModule }  
    <moduleId> _INITIAL  
    <routineDefinition>  
    (* END MODULE <id> *)

24) dclInModule --> <moduleDeclaration>  
    | <routineDeclaration>  
    | <structuredConstantDcl>  
    | <constantDeclaration>  
    | <typeDeclaration>  
    | <variableDeclaration>  

Pascal: dclInModule --> <moduleDeclaration>  
    | <routineDeclaration>  
    | <structuredConstantDcl>  
    | <constantDeclaration>  
    | <typeDeclaration>  
    | <variableDeclaration>  

This production is incorporated within the "moduleType"  
production.
25) program --> type <id> = <moduleType>

Pascal:
program (main body of translator) --> PREFIX
"Here, all the necessary house-keeping constructs
are written."
(* type <id> = *)
<moduleType>
BEGIN
"Now all the module initialization procedures
are output appended with "_INITIAL"
END .

For the reader who is interested in Euclid's syntax, the
following productions have no significance to the translation, but
they are part of Euclid's syntax.

importItem --> [ pervasive ] [ <bindCondition> ] <id>

exportItem --> [ <bindCondition> ] <id>
[ with ( <exportItem> , <exportItem> ) ]
| :=
| =

bindCondition --> const | readOnly | var
APPENDIX B

TRANSLATOR CODE

(******************************
APRIL EUCLID TO PASCAL TRANSLATOR
DAVID ROBESNER, SUMMER 1980

THIS PROGRAM TRANSLATES APRIL EUCLID (A SUBSET
OF PASCAL) INTO KSU INTERDATA 8/32 PASCAL.

DR. ROD BATES IS RESPONSIBLE FOR THE TABLE LOOK UP
OF IDENTIFIERS IN THE LEXICAL SCANNER. HE IS
ALSO RESPONSIBLE FOR THE PROGRAM "PASCAL PRETTY
PRINTER" THAT FORMATTED THIS SOURCE LISTING.

HE IS ALSO RESPONSIBLE FOR THE FOLLOWING CHANGES TO
THE ORIGINAL VERSION:
1. SHORTEN SYSTEM ID'S TO 10 CHAR SO XREF CAN DISTINGUISH
   THEM
2. PRINT UNRECOGNIZED CHARACTERS EXACTLY AND IN HEXADECIMAL
3. PRINT ERROR MESSAGE COMMENTS AT THE FAR LEFT, WHILE
   PUTTING EVERYTHING ELSE IN 5 SPACES
4. CHANGE EUCLID RESERVED WORDS TO ALL CAPS IN THE OUTPUT
5. CHANGE CONVERTER DECLARATIONS TO COMMENTS AND DELETE
   CONVERTER CALLS (LEAVING ONLY THEIR PARAMETERS)
6. REMOVE 'PACKED' FROM TYPE DEFINITIONS
7. REMOVE 'INLINE' FROM PROCEDURE DECLARATIONS
8. REMOVE 'CHECKED' AND 'NOT CHECKED'
9. DECLARE ALL FORMAL PARAMETERS 'UNIV'
10. PASS CODE BODIES UNMODIFIED, EXCEPT FOR REMOVAL OF '?'
11. APPEND PREFIX (COPY FROM LU 3)
12. ALTER PASCAL STRUCTURED CONSTANT SYNTAX
13. DELETE EXTRA '+' IN EXPRESSIONS
14. FIX DISJUNCTION TO CALL CONJUNCTION AGAIN
15. FIX MODULE_TYPE TO SKIP IMPORT LISTS CONTAINING 'VAR'
16. FIX TO OUTPUT ENTIRE LITERAL STRINGS

******************************)
"ROBERT YOUNG"
"KANSAS STATE UNIVERSITY"
"DEPARTMENT OF COMPUTER SCIENCE"

CONST COPYRIGHT = 'COPYRIGHT ROBERT YOUNG 1978'

"************
PREFIX
************"

; CONST NL = '(:10:)' 
; FF = '(:12:)' 
; CR = '(:13:)'
; EM = '(:25:')'

; CONST PAGELength = 512

; TYPE PAGE = ARRAY (. 1 .. PAGELength .) OF CHAR

; CONST LIneLengtH = 132

; TYPE LINE = ARRAY (. 1 .. LIneLengtH .) OF CHAR

; CONST IDLengtH = 12

; TYPE IDENTIFIER = ARRAY (. 1 .. IDLengtH .) OF CHAR

; TYPE FILE = 1 .. 2

; TYPE FILEKIND = ( EMPTY, SCRATCH, ASCII, SEQCODE,

, CONCODE )

; TYPE FILEATTR

  = RECORD

    KIND : FILEKIND

    ADDR : INTEGER

    ; PROTECTED : BOOLEAN

    ; NOTUSED : ARRAY (. 1 .. 5 .) OF INTEGER

    END

; TYPE IODEVICE

  = ( TYPEDEVICE, DISKDEVICE, TAPEDEVICE, PRINTDEVICE,

    , CARDDEVICE )

; TYPE IOOPERATION = ( INPUT, OUTPUT, MOVE, CONTROL )

; TYPE IOARG = ( WRITEEOF, REWIND, UPSPACE, BACKSPACE )

; TYPE IORESULT

  = ( COMPLETE, INTERVENTION, TRANSMISSION, FAILURE,

    , ENDFILE, ENDMEDIUM, STARTMEDIUM )

; TYPE IOPARAM

  = RECORD

    OPERATION : IOOPERATION

    ; STATUS : IORESULT

    ; ARG : IOARG

    END

; TYPE TASKKIND = ( INPUTTASK, JOBTASK, OUTPUTTASK )

; TYPE ARGTAG = ( NILTYPE, BOOLTYPE, INTTYPE, IDTYPE,

    , PTRTYPE )

; TYPE POINTER = @ BOOLEAN

; TYPE PASSPTR = @ PASSLINK

; TYPE PASSLINK

  = RECORD

    OPTIONS : SET OF CHAR
; FILLER1 : ARRAY (.1 .. 7.) OF INTEGER
; FILLER2 : BOOLEAN
; RESET_POINT : INTEGER
; FILLER3 : ARRAY (.1 .. 11.) OF POINTER
END

; TYPE ARGTYP = RECORD
  CASE TAG : ARGTAG
    OF NILTYPE , BOOLTYPE : ( BOOL : BOOLEAN )
    ; INTTYPE : ( INT : INTEGER )
    ; IDTYPE : ( ID : IDENTIFIER )
    ; PTRTYPE : ( PTR : PASSPTR )
  END

; CONST MAXARG = 10

; TYPE ARGLIST = ARRAY (.1 .. MAXARG.) OF ARGTYP

; TYPE ARGSEQ = ( INP, OUT )

; TYPE PROGRE = ( TERMINATED, OVERFLOW, POINTERERROR, RANGEERROR
                   , VARIANTERROR, HEAPLIMIT, STACKLIMIT, CODELIMIT
                   , TIMELIMIT, CALLERROR )

; PROCEDURE READ ( VAR C : CHAR )

; PROCEDURE WRITE ( C : CHAR )

; PROCEDURE OPEN ( F : FILE ; ID : IDENTIFIER
                   ; VAR FOUND : BOOLEAN )

; PROCEDURE CLOSE ( F : FILE )

; PROCEDURE GET ( F : FILE ; P : INTEGER
                 ; VAR BLOCK : UNIV PAGE )

; PROCEDURE PUT ( F : FILE ; P : INTEGER
                 ; VAR BLOCK : UNIV PAGE )

; FUNCTION LENGTH ( F : FILE ) : INTEGER

; PROCEDURE MARK ( VAR TOP : INTEGER )

; PROCEDURE RELEASE ( TOP : INTEGER )

; PROCEDURE IDENTIFY ( HEADER : LINE )

; PROCEDURE ACCEPT ( VAR C : CHAR )

; PROCEDURE DISPLAY ( C : CHAR )

; PROCEDURE NOTUSED

; PROCEDURE NOTUSED2

; PROCEDURE NOTUSED3
; PROCEDURE NOTUSED4

; PROCEDURE NOTUSED5

; PROCEDURE NOTUSED6

; PROCEDURE NOTUSED7

; PROCEDURE NOTUSED8

; PROCEDURE NOTUSED9

; PROCEDURE NOTUSED10

; PROCEDURE RUN

    ( ID : IDENTIFIER
    ; VAR PARAM : ARGLIST
    ; VAR LINE : INTEGER
    ; VAR RESULT : PROGRRESULT
    )

; PROGRAM P ( PARAM : LINE )

; TYPE BUFFER = ARRAY [ 1 .. 70 ] OF CHAR

; CONST RWMAX = 80

; TYPE STRING10 = ARRAY [ 1 .. 10 ] OF CHAR

; SYMBOLTY

    = ( IDENTSY , CONSTANTSY , CONSTSY , EQUALSSY , ANDSY
    , RELOPSY
    , ADDOPSY , ORSY , MULOPSY , ELIPSISSY , LPARS
    , RPARYS
    , NOTSY , COMMASY , DOTSY , CASESY , COLONSY , OFSY
    , ENDSSY
    , ARRAYSY , BEGINSY , TYPESY , PROCSY , FUNCSY , VARYS
    , COMESSY , IPSY , THENSY , ELSESY , RECORDSY
    , INITSY
    , EOFSY , LITSTRSY , IMPLSY , CASESEPSY , ASSERTSY
    , CODESY
    , ELSEIFSY , EXITSY , EXPORTSSY , IMPORTSSY
    , INCLUDESY
    , LOOPSY , MACHINESY , MODULESY , OTHERWISESY
    , PERSYVAYS
    , POSTSY , PRESY , READONLYSY , RETURNSY , RETURNSSY
    , WHENSY
    , BOOLSY , CHARSY , SIGINTSY , CONVETERSY , PACKEDSY
    , LINELSY , CHECKEDSY )

; CONST BLANKS

    =

; TYPE KINDOFID

    = ( PROCID , MODULEID , TYPEID , CONVERTERID , OTHERID )

    ,
; TYPE PTR_TO_SYM_TAB = ^ SYM_TABLE
; DISPLAY_PTR = ^ DISPLAY

(* THIS TYPE IS OF A STACK-LIKE STRUCTURE WHOSE
ELEMENTS CONTAIN POINTERS TO SYMBOL TABLES
AND THE TEXT THAT SHOULD BE APPENDED TO THE
IDENTIFIERS WHEN PROCESSED. *)
; TYPE DISPLAY
  = RECORD
    EARLIER_PTR : DISPLAY_PTR
    ; APPEND_ID : BUFFER
    ; LATER_PTR : DISPLAY_PTR
    ; SYM_TAB_PTR : PTR_TO_SYM_TAB
END

(* THIS IS THE TYPE OF THE NODE OF THE SYMBOL TABLE
IN BINARY TREE FORM. *)
; TYPE SYM_TABLE
  = RECORD
    LEFT_SON , RT_SON : PTR_TO_SYM_TAB
    ; ID : BUFFER
    ; TYPE_OF_ID : KINDOFID
    ; ITS_OWNS_TAB : PTR_TO_SYM_TAB
    ; BACK_TO_DISP : DISPLAY_PTR
END

; CONST STACK_MARKER = NIL

; TYPE PTR_TO_STACK = ^ STACK

(* THE STACK IS USED FOR "COMPILE TIME" VARIABLE
INITIALIZATIONS TO BE REALLY INITIALIZED AT
RUN-TIME INSTEAD. *)
; TYPE STACK
  = RECORD
    PTR_TO_ID : PTR_TO_SYM_TAB
    ; INIT_VAL : BUFFER
    ; NEXT_PTR : PTR_TO_STACK
END

(* THIS DATA STRUCTURE JUST KEEPS ALL THE NAMES
OF THE MODULE INITIALIZATION ROUTINES, TO BE
OUTPUT AS THE MAIN BODY OF THE PASCAL OBJECT. *)
; TYPE INIT_PTR = ^ PROC_INIT_QUEUE

; TYPE PROC_INIT_QUEUE
  = RECORD
    INIT_ID : BUFFER
    ; NEXT_QUEUE : INIT_PTR
END

; VAR TOP_OF_DISP : DISPLAY_PTR
; BOTTOM_OF_DISP : DISPLAY_PTR
; PTR_D : DISPLAY_PTR
; PTR_T : PTR_TO_SYM_TAB
; FREE_STACK : PTR_TO_STACK
; FREE_DISP : DISPLAY_PTR
; FREE_TABLE : PTR_TO_SYM_TAB
; GARBAGE : BOOLEAN
; STACK_TOP , PTR_S : PTR_TO_STACK
; FRONT_OF_QUEUE , ADD_ON_QUEUE : INIT_PTR
; SYSTEM_ID : BUFFER
; NUMB_OUTPUT : INTEGER
; INPUT_CHAR : CHAR
; TEXT : BUFFER
; TOK : SYMBOOLTYPE
; FILL_UP_PTR : INTEGER
; I : INTEGER
; LETTER_DIG , DIGITS : SET OF CHAR
; NUMB_ERR : INTEGER
; INSIDE_COMMENT : BOOLEAN
; QFIRSTSTMT , QCASE , QDCL , QFIRSTMODDCL , QGOODRECORD
; : SET OF SYMBOOLTYPE
; RWPTR : ARRAY [ 1 .. 10 ] OF 1 .. RWMAX
(* THIS ARRAY IS USED IN THE LEXICAL SCANNER TO
HOLD ALL THE EUCLID KEYWORDS AND THEIR TOKEN
SYMBOL: *)
; RW
; ARRAY [ 1 .. RWMAX ]
OF RECORD
STRING : STRING10
; SYMBOL : SYMBOOLTYPE
END
; RWCT : INTEGER

; PROCEDURE BREAKPNT
; EXTERN

; PROCEDURE NLINDENT
(* "NEW LINE INDENT" *)

; VAR I : INTEGER

; BEGIN
; WRITE ( NL )
; FOR I := 1 TO 5 DO WRITE ( ' ' )
; NUMB_OUTPUT := 5
END

; PROCEDURE OUT_WOUT_BL_LIT ( PRINT : BUFFER )
(* "OUT WITHOUT OUTPUTTING A BLANK THE LITERAL" *)

; VAR I : INTEGER

; BEGIN
I := 1
; WHILE PRINT [ I ] <> '&'
DO BEGIN
WRITE ( PRINT [ I ] )
; NUMB_OUTPUT := SUCC ( NUMB_OUTPUT )
; I := I + 1
END
END

; PROCEDURE OUT_LITERALLY ( OUT : BUFFER )
; BEGIN
  WRITE ( ' ' )
  ; NUMB_OUTPUT := SUCC ( NUMB_OUTPUT )
  ; IF NUMB_OUTPUT >= 55
  THEN NLINDENT
  ; OUTFOUT_BL_LIT ( OUT )
END

; PROCEDURE OUT_WOUT_BL ( OUTPUT_STR : BUFFER )
(* "OUT WITHOUT A BLANK THE INPUT TEXT" *)

; VAR I : INTEGER

; BEGIN
  I := 1
  ; WHILE OUTPUT_STR [ I ] <> ' '
  DO BEGIN
    WRITE ( OUTPUT_STR [ I ] )
    ; I := SUCC ( I )
    ; NUMB_OUTPUT := SUCC ( NUMB_OUTPUT )
  END
END

; PROCEDURE OUT_After_BL ( OUT_STR : BUFFER )

; BEGIN
  NUMB_OUTPUT := SUCC ( NUMB_OUTPUT )
  ; WRITE ( ' ' )
  ; IF NUMB_OUTPUT >= 55
  THEN NLINDENT
  ; OUTFOUT_BL_LIT ( OUT_STR )
END

; PROCEDURE OUT_ERROR ( OUT : BUFFER )

; BEGIN
  WRITE ( NL )
  ; NUMB_OUTPUT := 0
  ; OUTFITERLITALLY ( OUT )
END

; PROCEDURE OUT_UNREC_CH ( CH : CHAR )
(* "OUT UNRECOGNIZED CHARACTER" *)

; VAR HEDIGITS : ARRAY [ 1 .. 2 ] OF INTEGER
; I : INTEGER

; BEGIN
  IF INSIDE_COMMENT
  THEN OUT_ERROR ( '** UNRECOGNIZED CHARACTER **&' )
  ELSE OUT_ERROR ( '(* UNRECOGNIZED CHARACTER **&' )
  ; WRITE ( CH )
  ; OUTFITERLITALLY ( '*** CODE &' )
  ; HEDIGITS [ 1 ] := ORD ( CH ) DIV 16
  ; HEDIGITS [ 2 ] := ORD ( CH ) MOD 16
  ; FOR I := 1 TO 2
  DO IF HEDIGITS [ I ] < 10
  THEN WRITE ( CHR ( HEDIGITS [ I ] + ORD ( '0' ) ) )
ELSE WRITE (CHR(HEXDIGITS[I] - 10 + ORD('A')))  

; IF INSIDE_COMMENT  
THEN OUT_LITERALLY("##&")  
ELSE OUT_LITERALLY("#&")  
END

; PROCEDURE NEXT  
; FORWARD

; PROCEDURE OUT_AND_GET  
(* "OUTPUT THIS TOKEN AND GET NEXT ONE" *)

; VAR I : INTEGER

; BEGIN  
IF TOK = LITSTRSY  
THEN BEGIN  
WRITE("*")  
END

BEGIN  
OUT_LITERALLY("(*)")  
OUT_AFTER_BL(TEXT)  
OUT_LITERALLY("(*)&")  
END

; PROCEDURE OUT_TOK_AS_COMM  
(* "OUTPUT THIS TOKEN AS A COMMENT AND GET NEXT ONE" *)

; BEGIN

; PROCEDURE POP  
(VAR TOP_ID_PTR : PTR_TO_SYM_TAB  
VAR INIT_VAL : BUFFER)

; VAR TEMP_PTR : PTR_TO_STACK

; BEGIN

; PROCEDURE PUSH (NEW_ID_PTR : PTR_TO_SYM_TAB  
; INIT_VAL : BUFFER)

; BEGIN

IF FREE_STACK = NIL
THEN NEW ( PTR_S )
ELSE BEGIN
  PTR_S := FREE_STACK
  ; FREE_STACK := FREE_STACK ^ . NEXT_PTR
END
; PTR_S ^ PTR_TO_ID := NEW_ID_PTR
; PTR_S ^ INIT_VAL := INIT_VAL
; PTR_S ^ NEXT_PTR := STACK_TOP
; STACK_TOP := PTR_S
END

; PROCEDURE MARK_STACK
; BEGIN PUSH ( STACK_MARKER , BLANKS ) END

(* THIS PROCEDURE STORES THE NAME OF ALL MODULE
INITIALIZATION PROCEDURES, SO THEY CAN BE DUMPED
AS THE MAIN BODY OF THE PASCAL OBJECT: *)
; PROCEDURE QUEUE_INIT_PROC ( ID_INIT : BUFFER )

; VAR PTR_Q : INIT_PTR
BEGIN
  NEW ( PTR_Q )
  ; ADD_ON_QUEUE ^ NEXT_QUEUE := PTR_Q
  ; PTR_Q ^ INIT_ID := ID_INIT
  ; PTR_Q ^ NEXT_QUEUE := NIL
  ; ADD_ON_QUEUE := PTR_Q
END

(* THIS PROCEDURE GENERATES A UNIQUE IDENTIFIER: *)
; PROCEDURE GET_SYS_ID ( VAR OUT_SYS_ID : BUFFER )
BEGIN
  VAR I : INTEGER
BEGIN
  I := 10
  ; SYSTEM_ID [ I ] := SUCC ( SYSTEM_ID [ I ] )
  ; WHILE SYSTEM_ID [ I ] > '9'
    DO BEGIN
      SYSTEM_ID [ I ] := '0'
      ; I := I - 1
    DO END
  ; SYSTEM_ID [ I ] := SUCC ( SYSTEM_ID [ I ] )
END
; OUT_SYS_ID := SYSTEM_ID
END

; PROCEDURE TRAVERSE
( WHAT_LOOK_FOR : BUFFER ; VAR WALKER : PTR_TO_SYM_TAB )
; FORWARD

(* THIS PROCEDURE SHOULD BE THE CALLED THE FIRST TIME
WITH "WALKER" POINTING TO THE ROOT OF A SYMBOL
TABLE.
IF THE DESIRED ID ("WHAT LOOK FOR") IS IN THE
SYMBOL TABLE, "WALKER" WILL POINT TO IT.
IF THE DESIRED ID IS NOT IN THE SYMBOL TABLE,
"WALKER" WILL POINT TO THE NODE THAT THE ID SHOULD
BE INSERTED AFTER. IT IS UP TO THE CALLING PROC
TO FIND OUT WHICH WAY (LEFT OR RIGHT) IF IT
WANTS TO ADD THE DESIRED ID TO THIS TABLE: *)

; PROCEDURE TRAVERSE

; BEGIN
    IF ( WHAT_LOOK_FOR > WALKER ^. ID )
       AND ( WALKER ^. RT_SON <> NIL )
    THEN BEGIN
        WALKER := WALKER ^. RT_SON
        ; TRAVERSE ( WHAT_LOOK_FOR , WALKER )
    END
    ELSE IF ( WHAT_LOOK_FOR < WALKER ^. ID )
       AND ( WALKER ^. LEFT_SON <> NIL )
    THEN BEGIN
        WALKER := WALKER ^. LEFT_SON
        ; TRAVERSE ( WHAT_LOOK_FOR , WALKER )
    END

(* THIS PROCEDURE RETURNS A SYMBOL TABLE TO THE
FREE LIST WHEN ITS SCOPE IS LEFT, AND ITS ID'S
ARE NO LONGER NEEDED: *)

; PROCEDURE FREE_UP_TREE ( VAR PTR : PTR_TO_SYM_TAB )
; FORWARD

; PROCEDURE FREE_UP_TREE

; BEGIN
    IF PTR ^. LEFT_SON <> NIL
       THEN FREE_UP_TREE ( PTR ^. LEFT_SON )
    IF PTR ^. RT_SON <> NIL
       THEN FREE_UP_TREE ( PTR ^. RT_SON )
    PTR ^. RT_SON := FREE_TABLE
    FREE_TABLE := PTR

; PROCEDURE START_NEW_TAB ( APP_TEXT : BUFFER )

; BEGIN
    IF FREE_DISP = NIL
       THEN NEW ( PTR_D )
    ELSE BEGIN
        PTR_D := FREE_DISP
        FREE_DISP := FREE_DISP ^. LATER_PTR
    END
    PTR_D ^. EARLIER_PTR := TOP_OF_DISP
    PTR_D ^. LATER_PTR := NIL
    PTR_D ^. APPEND_ID := APP_TEXT
    TOP_OF_DISP ^. LATER_PTR := PTR_D
    TOP_OF_DISP := PTR_D
    IF FREE_TABLE = NIL
       THEN NEW ( PTR_T )
    ELSE BEGIN
        PTR_T := FREE_TABLE
        FREE_TABLE := FREE_TABLE ^. RT_SON
    END
    TOP_OF_DISP ^. SYM_TAB_PTR := PTR_T
; PTR_T * ID := BLANKS
; PTR_T * LEFT_SON := NIL
; PTR_T * RT_SON := NIL
; PTR_T * BACK_TO_DISP := TOP_OF_DISP
END

; PROCEDURE GET_RID_OF_TAB ( FREE_UP_TABLE : BOOLEAN )

; VAR TEMP_DISP : DISPLAY_PTR

; BEGIN
    TEMP_DISP := TOP_OF_DISP
    IF FREE_UP_TABLE THEN FREE_UP_TREE ( TEMP_DISP ^ . SYM_TAB_PTR )
    TOP_OF_DISP := TOP_OF_DISP ^ . EARLIER_PTR
    TOP_OF_DISP ^ . LATER_PTR := NIL
    TEMP_DISP ^ . LATER_PTR := FREE_DISP
    FREE_DISP := TEMP_DISP
END

(* THIS PROCEDURE ADDS THE CURRENT ID TO THE CURRENT
SYMBOL TABLE. IT THEN PASSES BACK A POINTER
TO THE ID. *)

; PROCEDURE ADD_TO_SYM_TAB
( WHICH_ID : BUFFER
; WHAT_KIND : KINDOFID
; VAR PTR : PTR_TO_SYM_TAB )

; VAR WALKER : PTR_TO_SYM_TAB

; BEGIN
    IF FREE_TABLE = NIL THEN NEW ( PTR )
    ELSE BEGIN
        PTR := FREE_TABLE
        FREE_TABLE := FREE_TABLE ^ . RT_SON
    END
    WITH PTR ^ DO BEGIN
        LEFT_SON := NIL
        RT_SON := NIL
        TYPE_OF_ID := WHAT_KIND
        ID := WHICH_ID
        BACK_TO_DISP := TOP_OF_DISP
    END
    WALKER := TOP_OF_DISP ^ . SYM_TAB_PTR
    Traverse ( WHICH_ID , WALKER )
    IF WHICH_ID > WALKER ^ . ID THEN WALKER ^ . RT_SON := PTR
    ELSE WALKER ^ . LEFT_SON := PTR
END

(* IN THIS PROC, WHICH_ID SHOULD POINT TO THE ROOT NODE
OF THE SYMBOL TABLE TO BE SEARCHED. IF THE ID
IS NOT CONTAINED IN THIS TABLE, WHICH_ID THEN
HAS NO SIGNIFICANCE. *)

; PROCEDURE SEARCH_SYM_TAB
( WHICH_ID : BUFFER
; VAR PTR : PTR_TO_SYM_TAB
; VAR NOT_LOCATED : BOOLEAN
)

; BEGIN
    TRAVERSE ( WHICH_ID , PTR )
    ; IF PTR ` . ID = WHICH_ID
    THEN NOT_LOCATED := FALSE
    ELSE NOT_LOCATED := TRUE
    END

; PROCEDURE SEARCH_FOR_ID
( WHICH_ID : BUFFER ; VAR PTR : PTR_TO_SYM_TAB )

; VAR OUT_WALKER : DISPLAY_PTR
; ID_NOT_FOUND : BOOLEAN

; BEGIN
    OUT_WALKER := TOP_OF_DISP
    ; ID_NOT_FOUND := TRUE
    ; WHILE ( OUT_WALKER < NIL ) AND ( ID_NOT_FOUND )
    DO BEGIN
        PTR := OUT_WALKER ` . SYM_TAB_PTR
        ; SEARCH_SYM_TAB ( WHICH_ID , PTR , ID_NOT_FOUND )
        ; OUT_WALKER := OUT_WALKER ` . EARLIER_PTR
        END
    ; IF ID_NOT_FOUND
    THEN PTR := NIL
    END

(* GIVEN A PTR TO A PARTICULAR ID, THIS PROC OUTPUTS
THAT ID WITH ALL APPROPRIATE APPEND TEXTS
APPENDED TO IT. *)

; PROCEDURE APPEND ( PTR : PTR_TO_SYM_TAB )

; VAR WALKER : DISPLAY_PTR

; BEGIN
    IF PTR ` . BACK_TO_DISP ` . APPEND_ID = BLANKS
    THEN OUT_AFTER_BL ( PTR ` . ID )
    ELSE BEGIN
        WALKER := BOTTOM_OF_DISP
        ; OUT_AFTER_BL ( WALKER ` . APPEND_ID )
        ; WHILE WALKER < PTR ` . BACK_TO_DISP
        DO BEGIN
            WALKER := WALKER ` . LATER_PTR
            ; OUT_WOUT_BL ( WALKER ` . APPEND_ID )
            ; OUT_WOUT_BL_LIT ('&')
        END
    END
    ; OUT_WOUT_BL ( PTR ` . ID )
    END

(* THIS CONVERTS THE CURRENT "CONGLOMERATE ID" (OF THE
FORM "ID.ID,ID") INTO THE FORM "ID_ID_ID.ID".
I.E., IT APPENDS THE FIRST ID, THEN CHANGES ALL THE
APPRIOPR AT ".
 TO "._" FOR ALL THE SUCCEEDING MODULE
IDS. IT ALSO RETURNS THE TYPE OF THE CONGLOMERATE ID,
WHICH IS THE TYPE OF THE ID THAT IS THE FIRST ONE AFTER
THE LAST "._" IS OUTPUT. *)

; PROCEDURE OUT_FIXED_ID ( VAR WHAT_KIND : KINDOFID )

; VAR WHERE : PTR_TO_SYM_TAB
; GARBAGE : BOOLEAN

; BEGIN
SEARCH_FOR_ID ( TEXT, WHERE )
IF WHERE = NIL
THEN BEGIN
OUT_ERROR ( '*ID NOT DECLARED YET- RECURSIVE PROC? *' & ' )
NUMB_ERR := SUCC ( NUMB_ERR )
WHAT_KIND := PROCID
OUT_AND_GET
END
ELSE BEGIN
APPEND ( WHERE )
WHILE WHERE ^ . TYPE_OF_ID = MODULEID
(* THE ID IS A MODULE, SO WE KNOW THAT "." AND
ANOTHER ID FOLLOW AND THE "." NEEDS TO BE
CHANGED TO "._" *)
DO BEGIN
NEXT
OUT_WOUT_BL_LIT ( '._' )
NEXT
OUT_WOUT_BL ( TEXT )
WHERE := WHERE ^ . ITSOWN_TAB
SEARCH_SYM_TAB ( TEXT, WHERE, GARBAGE )
IF GARBAGE
THEN BEGIN
OUT_ERROR ( '*LOGIC ERROR - APPEND *' & ' )
NUMB_ERR := SUCC ( NUMB_ERR )
END
WHAT_KIND := WHERE ^ . TYPE_OF_ID
(* FIRST ID AFTER ALL MODULES DETERMINES THE KIND. *)
NEXT
END
END
END
END
END
END
END
END
END
END
END

(* THIS IS DR. BATES' PART. THIS PROC ADDS A KEYWORD
TO THE KEY WORD LIST. *)

; PROCEDURE INITRW ( PSTRING : STRING10
; FSsymbol : SYMBO-L-TYP )
VAR LEN, I, J : INTEGER

BEGIN
 IF RWCT < RWMAX THEN BEGIN
   LEN := 10
   WHILE ( LEN >= 2 ) AND ( FSTRING [ LEN ] = ' ' )
     DO LEN := PRED ( LEN )
   I := RWPTR [ LEN ]
   J := RWPTR [ LEN + 1 ]
   WHILE ( I < J ) AND ( RW [ I ]. STRING < FSTRING )
     DO I := SUCC ( I )
   IF RW [ I ]. STRING <> FSTRING THEN BEGIN
     J := RWPTR [ 10 ] - 1
     WHILE J >= I
     DO BEGIN
       RW [ J + 1 ] := RW [ J ]
       J := PRED ( J )
     END
   RW [ I ]. STRING := FSTRING
   FOR J := LEN + 1 TO 10
     DO RWPTR [ J ] := SUCC ( RWPTR [ J ] )
 END
 RWCT := SUCC ( RWCT )
 END (* INITRW *)

(* BUILD KEY WORD LIST *)
PROCEDURE INITRWTABLE

BEGIN
 RWCT := 0
 FOR I := 1 TO 10 DO RWPTR [ I ] := 1
 INITRW ( 'if' , IFSY )
 INITRW ( 'of' , OFSY )
 INITRW ( 'or' , ORSY )
 INITRW ( 'and' , ANDSY )
 INITRW ( 'div' , MULOPSY )
 INITRW ( 'end' , ENDSY )
 INITRW ( 'mod' , MULOPSY )
 INITRW ( 'not' , NOTSY )
 INITRW ( 'var' , VARSY )
 INITRW ( 'pre' , PRESY )
 INITRW ( 'case' , CASESY )
 INITRW ( 'else' , ELSESY )
 INITRW ( 'initially' , INITSY )
 INITRW ( 'then' , THENSY )
 INITRW ( 'type' , TYPESY )
 INITRW ( 'code' , CODESY )
 INITRW ( 'exit' , EXITSY )
 INITRW ( 'loop' , LOOPSY )
 INITRW ( 'post' , POSTSY )
 INITRW ( 'when' , WHENSY )
 INITRW ( 'Char' , CHARSY )
 INITRW ( 'array' , ARAYSY )
; INITRW ( 'begin' , BEGINSY )
; INITRW ( 'const' , CONSTSY )
; INITRW ( 'record' , RECORDSY )
; INITRW ( 'assert' , ASSERTSY )
; INITRW ( 'elseif' , ELSEIFSY )
; INITRW ( 'inline' , INLINESY )
; INITRW ( 'module' , MODULESY )
; INITRW ( 'packed' , PACKEDSY )
; INITRW ( 'return' , RETURNSY )
; INITRW ( 'checked' , CHECKEDSY )
; INITRW ( 'exports' , EXPORTSSY )
; INITRW ( 'imports' , IMPORTSSY )
; INITRW ( 'include' , INCLUDESY )
; INITRW ( 'machine' , MACHINESY )
; INITRW ( 'returns' , RETURNSSY )
; INITRW ( 'function' , FUNCSY )
; INITRW ( 'readonly' , READONLYSY )
; INITRW ( 'Boolean' , BOOLSY )
; INITRW ( 'otherwise' , OTHERWISESY )
; INITRW ( 'pervasive' , PERVASIVESY )
; INITRW ( 'procedure' , PROCESY )
; INITRW ( 'SignedInt' , SIGINTSY )
; INITRW ( 'converter' , CONVERTERSY )
END (* INITRWTABLE *)

(* THIS PROCEDURE EATS UP SEPARATORS (";" ARE UNNEEDED
SEPARATORS IN EUCLID) AND EATS UP AND PRINTS OUT COMMENTS,
SO THAT "INPUT_CHAR" HAS THE FIRST CHAR OF A VALID TOKEN *)

; PROCEDURE EAT_UP_GARBAGE
VAR SEPERATOR_OR_SEMI : BOOLEAN

BEGIN
  SEPERATOR_OR_SEMI := TRUE
  WHILE SEPERATOR_OR_SEMI
    DO BEGIN
      SEPERATOR_OR_SEMI := FALSE
      WHILE ( INPUT_CHAR = ' ' )
        OR ( INPUT_CHAR = NL )
        OR ( INPUT_CHAR = ';' )
        DO BEGIN SEPERATOR_OR_SEMI := TRUE
                  READ ( INPUT_CHAR ) END
      IF INPUT_CHAR = '{'
        THEN BEGIN
          NLINDENT
          SEPERATOR_OR_SEMI := TRUE
          IF INSIDE_COMMENT
            THEN WRITE ( '"' )
            ELSE WRITE ( '(' )
          WRITE ( '"' )
          READ ( INPUT_CHAR )
          WHILE NOT ( INPUT_CHAR = '}' )
          DO BEGIN
            WRITE ( INPUT_CHAR )
          END
          NUMB_OUTPUT := SUCC ( NUMB_OUTPUT )
          READ ( INPUT_CHAR )
          END
END
; BEGIN
; EAT_UP_GARBAGE
; WHILE INPUT_CHAR = '?'
DO BEGIN
READ ( INPUT_CHAR )
WRITE ( NL )
WHILE NOT ( INPUT_CHAR IN [ '?' , ' ' ] )
DO BEGIN
WRITE ( INPUT_CHAR )
READ ( INPUT_CHAR )
END
IF INPUT_CHAR = '?'
THEN READ ( INPUT_CHAR )
END
WRITE ( NL )
END

(* THIS IS THE LEXICAL SCANNER - IT RETURNS THE NEXT LOGICAL

; PROCEDURE NEXT
VAR I : INTEGER
; PROCEDURE OCTAL_CONVERSION
VAR NUMBER , BEGIN_PTR , I : INTEGER
; TEMP_TEXT : BUFFER
BEGIN
READ ( INPUT_CHAR )
READ ( INPUT_CHAR )
NUMBER := 0
FOR I := 1 TO FILL_UP_PTR
DO BEGIN
NUMBER := NUMBER * 8
NUMBER := NUMBER + ORD ( TEXT [ I ] )
- ORD ( '0' )
END
TEMP_TEXT := BLANKS
BEGIN_PTR := 71
WHILE NUMBER <> 0
DO BEGIN
    BEGIN_PTR := BEGIN_PTR - 1
    ; TEMP_TEXT [ BEGIN_PTR ]
    := CHR ( ( NUMBER MOD 10 ) + ORD ( '0' ) )
    ; NUMBER := NUMBER DIV 10
    END
    ; TEXT := BLANKS
    ; FOR I := BEGIN_PTR TO 70
    DO TEXT [ I - BEGIN_PTR + 1 ] := TEMP_TEXT [ I ]
    ;
    END
    (* THIS PROCEDURE CONCATENATES A CHARACTER TO THE END OF
    THE TOKEN TO BUILD IT. *)
    ; PROCEDURE ADD_TO_OUT ( CONCAT_TO_STR : CHAR )
    ; BEGIN
    IF FILL_UP_PTR >= 71
    THEN BEGIN
    IF INSIDE_COMMENT
    THEN OUT_ERROR
    (**THIS TOKEN TRUNCATED TO 70 CHAR**) &'
    ELSE OUT_ERROR
    (' (* THIS TOKEN TRUNCATED TO 70 CHAR *) &'
    ; NUMB_ERR := NUMB_ERR + 1
    END
    ELSE BEGIN
    FILL_UP_PTR := FILL_UP_PTR + 1
    ; TEXT [ FILL_UP_PTR ] := CONCAT_TO_STR
    END
    END (* ADD_TO_OUT *)
    ; PROCEDURE ADD_ADVANCE_CHAR ( CONCAT_TO_STR : CHAR )
    ; BEGIN
    ADD_TO_OUT ( CONCAT_TO_STR )
    ; READ ( INPUT_CHAR )
    END (* ADD_ADVANCE_CHAR *)
    (* THIS IS ALSO DR. BATES' PART. IT THE TOKEN IS AN
    IDENTIFIER, IT LOOKS IT UP IN THE KEYWORD
    LIST AND DETERMINES THE TOKEN VALUE. *)
    ; PROCEDURE LOOKUPIDENT
    ; BEGIN
    VAR LSTRING : STRING10
    ; J, LO, HI, PROBE : INTEGER
    ; EXITLOOP : BOOLEAN
    ; PROCEDURE ALLCAPS ( VAR STRING : BUFFER )
    ; BEGIN
    FOR I := 1 TO 70
    DO IF ( 'a' <= STRING [ I ] )
    AND ( STRING [ I ] <= 'z' )
    THEN STRING [ I ]
:= CHR
   ( ORD ( STRING [ I ] )
   - ORD ( 'a' )
   + ORD ( 'A' )
   )
END

BEGIN
   IF FILL_UP_PTR > 9
      THEN TOK := IDENTSY
   ELSE BEGIN
      LSTRING := ' ';
      FOR J := 1 TO FILL_UP_PTR
         DO LSTRING [ J ] := TEXT [ J ]
      ; LO := RWPTR [ FILL_UP_PTR ] - 1
      ; HI := RWPTR [ FILL_UP_PTR + 1 ]
      ; EXITLOOP := FALSE
      ; REPEAT IF LO + 1 >= HI
      THEN BEGIN TOK := IDENTSY ; EXITLOOP := TRUE END
   ELSE BEGIN
      PROBE := ( LO + HI ) DIV 2
      ; IF RW [ PROBE ] . STRING = LSTRING
      THEN BEGIN
         TOK := RW [ PROBE ] . SYMBOL
      ; ALLCAPS ( TEXT )
      ; EXITLOOP := TRUE
      END
      ELSE IF RW [ PROBE ] . STRING < LSTRING
      THEN LO := PROBE
   ELSE HI := PROBE
   UNTIL EXITLOOP
END
(* LOOKUPIDENT *)

(* EUCLID USES "#$" WHERE PASCAL USES "(....)" - FURTHERMORE
EUCLID HAS SPECIAL CONTROL CHARACTERS, SUCH AS "NEWLINE",
"ENDOFMEDIUM", ETC ALSO INDICATED BY "#$" (FOLLOWED BY A
CHARACTER). EUCLID ALSO SHOWS A ONE-CHARACTER STRING BY
PRECEDING THE CHARACTER BY "#$".
FOR EXAMPLES,
   $B = 'B' PASC
   $N = 'N' PASC
   $S = '$S$' = NL PASC
   '$023' = "$023 = '023:' PASC
   THIS PROCEDURE IS CALLED AFTER ONE DOLLAR SIGN HAS BEEN
FOUND IF THE SCAN IS NOT INSIDE A LITERAL, AND IS CALLED FOR
EVERY CHARACTER IF THE SCAN IS INSIDE LITERAL.
IF CALLED INSIDE LITERAL AND IF THE NEXT CHAR IS NOT A
DOLLAR SIGN, THIS PROC SIMPLY OUTPUTS IT. IF IT ENCOUNTERS
A DOLLAR SIGN, THAT MEANS A SPECIAL CHAR IS TO BE PROCESSED
IF CALLED OUTSIDE LITERAL, AND ONLY ONE DOLLAR SIGN IS
FOUND, THIS PROC SIMPLY OUTPUTS THE CHAR (THE CALLING
ROUTINE WILL OUTPUT THE TWO "$". OTHERWISE, A SPECIAL
CHARACTER NEEDS TO BE PROCESSED
ONE CAUTION - THE SYNCHRONIZATION OF THIS ROUTINE IS VERY
WEIRD, DEPENDING ON IF IT IS CALLED FROM INSIDE OR OUTSIDE
A LITERAL. *)
; PROCEDURE PROCESS_DOLLAR
; VAR I : INTEGER

; BEGIN
  IF INPUT_CHAR = '¥'
  THEN BEGIN
    READ ( INPUT_CHAR )
    CASE INPUT_CHAR
      OF 'E', 'F', 'T', 'N', '(:39:):'
        BEGIN
          ADD_TO_OUT ( '(' )
          ; ADD_TO_OUT ( ':' )
          (* DOUBLE CASE STMT WAS TO
            AVOID HAVING TO WRITE OUT
            "ADD_TO_OUT('(');ADD_TO_OUT(':')" A LOT *)
          CASE INPUT_CHAR
            OF 'E'
              BEGIN
                ADD_TO_OUT ( '"' )
                ; ADD_TO_OUT ( '"' )
                END
              ; 'F'
                BEGIN
                  ADD_TO_OUT ( "1" )
                  ; ADD_TO_OUT ( "2" )
                  END
              ; 'T'
                BEGIN
                  ADD_TO_OUT ( "2" )
                  ; ADD_TO_OUT ( "3" )
                  END
              ; 'N'
                BEGIN
                  ADD_TO_OUT ( "1" )
                  ; ADD_TO_OUT ( "0" )
                  END
              ; '(:39:):'
                BEGIN
                  ADD_TO_OUT ( "3" )
                  ; ADD_TO_OUT ( "9" )
                  END
              END
          END
        ; ADD_TO_OUT ( ':' )
      ; ADD_ADVANCE_CHAR ( '"' )
  END
; 'S': ADD_ADVANCE_CHAR ( '"' )
; '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
; BEGIN
  ADD_TO_OUT ( '(' )
  ; ADD_TO_OUT ( ':' )
  FOR I := 1 TO 3
  DO ADD_ADVANCE_CHAR ( INPUT_CHAR )
  ; ADD_TO_OUT ( ':' )
  ; ADD_TO_OUT ( ')' )
  END
; ELSE : ADD_ADVANCE_CHAR ( 'S' )
END
END
ELSE ADD_ADVANCE_CHAR ( INPUT_CHAR )
END (* PROCESS DOLLAR *)

; BEGIN (* NEXT *)
TEXT := BLANKS
; FILL_UP_PTR := 0
; EAT_UP_GARBAGE
; CASE INPUT_CHAR
OF 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I',
 'J',
 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R',
 'S',
 'T',
 'U', 'V', 'W', 'X', 'Y', 'Z', '_', 'a',
 'b',
 'c',
 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
 'l',
 'm',
 'n', 'o', 'p', 'q', 'r', 's', 't', 'u',
 'v',
 'w', 'x', 'y', 'z'
(* TOKEN IS AN ID OR KEYWORD *)
BEGIN
ADD_ADVANCE_CHAR ( INPUT_CHAR )
WHILE INPUT_CHAR IN LETTER_DIG
DO ADD_ADVANCE_CHAR ( INPUT_CHAR )
LOOKUPIDENT
IF TOK = INCLUDESYS
THEN BEGIN
OUT_ERROR ( 'NEED TO INCLUDE A FILE - &' )
; INSIDE_COMMENT := TRUE
; NUMB_ERR := SUCC ( NUMB_ERR )
; NEXT
; OUT_AFTHER_BL ( TEXT )
; OUT_LITERALLY ( ' ' & ' ' )
; INSIDE_COMMENT := FALSE
; NEXT
END
ELSE IF ( TOK = CHASY ) AND ( INPUT_CHAR = '.' )
BEGIN
TOK := IDENTSY
FOR I := 1 TO 4 DO READ ( INPUT_CHAR )
; TEXT := BLANKS
; TEXT [ 1 ] := 'O'
; TEXT [ 2 ] := 'R'
; TEXT [ 3 ] := 'D'
END
; '.' (* TOKEN IS EITHER '.' OR '..' *)
BEGIN
TOK := DOTS
ADD_ADVANCE_CHAR ( INPUT_CHAR )
IF INPUT_CHAR = '.'
THEN BEGIN
ADD_ADVANCE_CHAR ( INPUT_CHAR )
; TOK := ELIPSIS
END

; '-' (* TOKEN IS "-", "->" *)
BEGIN
TOK := ADDOPSY
; ADD_ADVANCE_CHAR ( INPUT_CHAR )
; IF INPUT_CHAR = '>'
THEN BEGIN
ADD_ADVANCE_CHAR ( INPUT_CHAR )
; TOK := IMPLYSY
END

; '=' (* TOKEN IS "=", "=>" *)
BEGIN
TOK := EQUALSSY
; ADD_ADVANCE_CHAR ( INPUT_CHAR )
; IF INPUT_CHAR = '='
THEN BEGIN
ADD_ADVANCE_CHAR ( INPUT_CHAR )
; TOK := CASESEPSY
END

; ':' (* TOKEN IS ":", ":=" *)
BEGIN
TOK := COLONSY
; ADD_ADVANCE_CHAR ( INPUT_CHAR )
; IF INPUT_CHAR = ':'
THEN BEGIN
ADD_ADVANCE_CHAR ( INPUT_CHAR )
; TOK := BECOMESSY
END

; '>'; '<'
(* TOKEN IS EITHER ">=", "<=", "=>", "<=" *)
BEGIN
TOK := ROLOPSY
; ADD_ADVANCE_CHAR ( INPUT_CHAR )
; IF INPUT_CHAR = '='
THEN ADD_ADVANCE_CHAR ( INPUT_CHAR )
END

; '$' (* TOKEN IS A ONE-CHARACTER STRING *)
BEGIN
TOK := LITSTRSY
; ADD_ADVANCE_CHAR ( '(:39:)')
; PROCESS_DOLLAR
; ADD_TO_OUT ( '(:39:)')
END

; '(:39:)' (* TOKEN IS A LITERAL *)
BEGIN
TOK := LITSTRSY
; WHILE INPUT_CHAR = '(:39:)' DO BEGIN
ADD_ADVANCE_CHAR ( INPUT_CHAR )
; WHILE NOT ( INPUT_CHAR = '(:39:)') DO PROCESS_DOLLAR
; ADD_ADVANCE_CHAR ( "(39:)"
END
END
'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'

(* TOKEN IS A NUMBER *)
BEGIN
TOK := CONSTANTSY
; ADD_ADVANCE_CHAR ( INPUT_CHAR )
WHILE ( INPUT_CHAR IN DIGITS )
DO ADD_ADVANCE_CHAR ( INPUT_CHAR )
IF INPUT_CHAR = '#' THEN OCTAL_CONVERSION
END
EM : TOK := EOFSY
ELSE (* TOKEN IS ANY OTHER ONE-CHARACTER TOKEN *)
BEGIN
CASE INPUT_CHAR
OF '+'
BEGIN
TOK := ADDOsy
; ADD_ADVANCE_CHAR ( INPUT_CHAR )
END
'
BEGIN
TOK := LPARSY
; ADD_ADVANCE_CHAR ( INPUT_CHAR )
END
'
BEGIN
TOK := RPARSY
; ADD_ADVANCE_CHAR ( INPUT_CHAR )
END
',
BEGIN
TOK := COMMASY
; ADD_ADVANCE_CHAR ( INPUT_CHAR )
END
'**'
BEGIN
TOK := MULOsy
; ADD_ADVANCE_CHAR ( INPUT_CHAR )
END
ELSE
BEGIN
OUT_UNREC_CH ( INPUT_CHAR )
READ ( INPUT_CHAR )
NEXT
END
END (* CASE OF ONE-CHAR TOK *)
END (* ELSE BIG CASE *)
END (* CASE *)
END (* NEXT *)

; PROCEDURE EXPN
; FORWARD

(* THIS PROC SIMPLY DELETES AN ID FROM THE CODE BODY
OF EUCLID IF IT IS A CONVERTER CALL. IT ALSO DELETES
THE ACCOMPANYING PARENTHESSES. *)

; PROCEDURE DELETE_CONVERTER_CALL

; BEGIN
INSIDE_COMMENT := TRUE
; OUT_LITERALLY ( "(* &" )
; OUT_AND_GET (* NAME *)
; OUT_AND_GET (* LPARSY *)
; OUT_LITERALLY ( " *)&" )
INSIDE_COMMENT := FALSE
EXPN
; INSIDE_COMMENT := TRUE
; OUT_LITERALLY ( "(* &" )
; OUT_AND_GET (* RPARSY *)
; OUT_LITERALLY ( " *)&" )
INSIDE_COMMENT := FALSE
END

(* THIS PROCEDURETranslates A VARIABLE. IT ASSUMES
THAT THE SYNTAX FOR A VARIABLE USED IN EUCLID IS:
VAR---> { "." <ID> ! "(" <EXPN> ( "." <EXPN>) )
*}

; PROCEDURE PROC_AND_OUT_VAR

; VAR KIND_OF_ID : KINDOFID
; PROCESS_VAR : BOOLEAN

; BEGIN
IF TOK = INLINESY
THEN OUT_TOK_AS_COMM
; OUT_FIXED_ID ( KIND_OF_ID )
IF TOK = LPARSY
THEN
(* IS EITHER A PROCEDURE OR ARRAY *)
BEGIN
IF KIND_OF_ID = PROCID
THEN OUT_LITERALLY ( "(&" )
ELSE OUT_LITERALLY ( "[&" )
; NEXT
; EXPN
; WHILE TOK = COMMASY DO BEGIN OUT_AND_GET ; EXPN END
IF KIND_OF_ID = PROCID
THEN OUT_LITERALLY ( "[" &" )
ELSE OUT_LITERALLY ( "[" &" )
; NEXT
(* AT THIS STAGE OF THE GAME, IF THERE ARE ANY
* "." OR "(" LEFT, WE ARE INSIDE ARRAYS AND RECORDS,
* NOT PROCEDURES OR FUNCTIONS.
* )
; PROCESS_VAR := TRUE
; WHILE PROCESS_VAR
DO BEGIN
PROCESS_VAR := FALSE
; WHILE TOK = DOTSY
DO BEGIN
PROCESS_VAR := TRUE
1324 ; OUT_WOUT_BL ( TEXT )
1325 ; NEXT
1326 ; OUT_WOUT_BL ( TEXT )
1327 ; NEXT
1328 END
1329 ; IF TOK = LPARSY
1330 THEN BEGIN
1331 OUT_LITERALLY ( "[ &" )
1332 ; NEXT
1333 ; EXPN
1334 ; OUT_LITERALLY ( "] &" )
1335 ; NEXT
1336 PROCESS_VAR := TRUE
1337 END
1338 END
1339 END
1340 END
1341
1342 (* THE NEXT BUNCH OF PROCEDURES ARE TO PARSE EXPRESSIONS.
1343 THE SYNTAX IS QUITE OBVIOUS FROM THE CODE. THE SYNTAX
1344 WAS REWRITTEN FROM THE EUCLID SYNTAX, BECAUSE THE EUCLID
1345 SYNTAX WAS LEFT RECURSIVE. *)
1346 ; PROCEDURE FACTOR
1347
1348 ; VAR WHERE : PTR_TO_SYM_TAB
1349
1350 ; BEGIN
1351 WHILE TOK = ADDOPSY DO OUT_AND_GET
1352 ; CASE TOK
1353 OF LPARSY : BEGIN OUT_AND_GET ; EXPN ; OUT_AND_GET END
1354 ; CONSTANTSY : OUT_AND_GET
1355 ; LITSTRSY : OUT_AND_GET
1356 ; IDENTSY
1357 ; BEGIN
1358 SEARCH_FOR_ID ( TEXT , WHERE )
1359 ; IF ( WHERE <> NIL )
1360 AND ( WHERE ^.TYPE_OF_ID = CONVERTERID )
1361 THEN DELETE_CONVERTER_CALL
1362 ELSE PROC_AND_OUT_VAR
1363 END
1364 ; ELSE
1365 ; BEGIN
1366 OUT_ERROR ( "(* LOGIC ERR = FACT *) & " )
1367 ; NUMB_ERR := SUCC ( NUMB_ERR )
1368 END
1369 END
1370 END
1371
1372 ; PROCEDURE TERM
1373
1374 ; BEGIN
1375 OUT_LITERALLY ( "(&" )
1376 ; FACTOR
1377 ; WHILE TOK = MULOPSY DO BEGIN OUT_AND_GET ; FACTOR END
1378 ; OUT_LITERALLY ( ")&" )
1379 END
1380
1381 ; PROCEDURE SUM
; BEGIN
    OUT_LITERALLY ( '(&' )
; TERM
; WHILE TOK = ADDOPSY DO BEGIN OUT_AND_GET ; TERM END
; OUT_LITERALLY ( ')'&' )
END

; PROCEDURE RELATION
; BEGIN
    OUT_LITERALLY ( '(&' )
    ; SUM
    ; IF ( TOK = RELOPSY ) OR ( TOK = EQualSSY )
    THEN BEGIN OUT_AND_GET ; SUM END
    ELSE IF TOK = NOTSY
    THEN BEGIN OUT_LITERALLY ( '<>' &' )
    ; NEXT ; NEXT ; SUM END
    ; OUT_LITERALLY ( ')'&' )
END

; PROCEDURE NEGATION
; BEGIN
    OUT_LITERALLY ( '(&' )
    ; IF TOK = NOTSY
    THEN OUT_AND_GET
    ; RELATION
    ; OUT_LITERALLY ( ')'&' )
END

; PROCEDURE CONJUNCTION
; BEGIN
    OUT_LITERALLY ( '(&' )
    ; NEGATION
    ; WHILE TOK = ANDSY DO BEGIN OUT_AND_GET ; NEGATION END
    ; OUT_LITERALLY ( ')'&' )
END

; PROCEDURE DISJUNCTION
; BEGIN
    OUT_LITERALLY ( '(&' )
    ; CONJUNCTION
    ; WHILE TOK = ORSY DO BEGIN OUT_AND_GET
    ; CONJUNCTION END
    ; OUT_LITERALLY ( ')'&' )
END

; PROCEDURE EXPN
; BEGIN
    OUT_LITERALLY ( '(' &' )
    ; DISJUNCTION
    ; IF TOK = IMPLYSY
    THEN BEGIN
    OUT_ERROR ( '(*( &' )

; OUT_LITERALLY ( ' THIS CAN NOT BE TRANSLATED &' )
; OUT_LITERALLY ( ' *' &' )
; NUMB_ERR := SUCC ( NUMB_ERR )
; OUT_LITERALLY ( ' *( &' )
; INSIDE_COMMENT := TRUE
; OUT_AND_GET
; DISJUNCTION
; INSIDE_COMMENT := FALSE
; OUT_LITERALLY ( '*' &' )
END
; OUT_LITERALLY ( ' )&' )
END

(* THIS PARSES A COMPILE TIME EXPRESSION, ASSIGNS IT
TO A GENERATED CONSTANT ID, AND PASSES THAT ID
TO THE CALLING PROC.
THIS PROC IS THE FIRST ONE IN THIS LISTING
THAT USES THE TECHNIQUES OF DECLARING
INPUT CONSTRUCTS AS GENERATED SYSTEM IDS, AND PASSING
THOSE DECLARED GENERATED IDS TO THE CALLING PROC. *)

; PROCEDURE COMP_EXP ( VAR SYS_ID : BUFFER )

; BEGIN
  GET_SYS_ID ( SYS_ID )
  ; OUT_LITERALLY ( 'CONST &' )
  ; OUT_AFTER_BL ( SYS_ID )
  ; OUT_LITERALLY ( ' = &' )
  ; SUM
  ; OUT_LITERALLY ( ';' &' )
END

; PROCEDURE RANGE ( VAR SYS_TYPE : BUFFER )

; VAR SYS_CONST_1 , SYS_CONST_2 : BUFFER
; GARbage : KINDOFID
; PTR : PTR_TO_SYM_TAB

; BEGIN
  GET_SYS_ID ( SYS_TYPE )
  ; SEARCH_FOR_ID ( TEXT , PTR )
  ; IF ( PTR = NIL ) OR ( PTR " . TYPE_OF_ID = OTHERID )
  THEN
    (* RANGE HAS ELIPSIS *)
    BEGIN
      COMP_EXP ( SYS_CONST_1 )
      ; NEXT
      COMP_EXP ( SYS_CONST_2 )
      ; OUT_LITERALLY ( 'TYPE &' )
      OUT_AFTER_BL ( SYS_TYPE )
      ; OUT_LITERALLY ( ' = &' )
      OUT_AFTER_BL ( SYS_CONST_1 )
      ; OUT_LITERALLY ( ' .. &' )
      OUT_AFTER_BL ( SYS_CONST_2 )
      ; OUT_LITERALLY ( ';' &' )
END

(* RANGE IS TYPE *)
ELSE BEGIN
OUT_LITERALLY ( 'TYPE &' )
PROCEDURE VAR_TYPE ( VAR SYS_ID : BUFFER )

VAR WHERE : PTR_TO_SYM_TAB
  SYS_TYPE : BUFFER
  GARBAGE_KIND : KINDOFID

BEGIN
  GET_SYS_ID ( SYS_ID )
  CASE TOK
    OF SIGINTSY , BOOLSY , CHARSY , LPARYS
      BEGIN
        OUT_LITERALLY ( 'TYPE &' )
        OUT_AFTER_BL ( SYS_ID )
        OUT_LITERALLY ( '=' &' )
        CASE TOK
          OF SIGINTSY : OUT_LITERALLY ( 'INTEGER &' )
          BOOLSY , CHARSY : OUT_AFTER_BL ( TEXT )
          LPARYS
            BEGIN
              OUT_AFTER_BL ( TEXT )
              REPEAT NEXT
              IF TOK = IDENTSY
                THEN BEGIN
                  ADD_TO_SYM_TAB ( TEXT , OTHERID , WHERE )
                  APPEND ( WHERE )
                END
                ELSE OUT_AFTER_BL ( TEXT )
              UNTIL TOK = RPARYS
            END
          END
        END
    NEXT
  END
  NEXT
  IDENTSY
  BEGIN
    SEARCH_FOR_ID ( TEXT , WHERE )
    IF WHERE ^ . TYPE_OF_ID < OTHERID
      THEN BEGIN
        OUT_LITERALLY ( 'TYPE &' )
        OUT_AFTER_BL ( SYS_ID )
        OUT_LITERALLY ( '=' &' )
        OUT_FIXED_ID ( GARBAGE_KIND )
        OUT_LITERALLY ( ';' &' )
      END
      ELSE BEGIN
        RANGE ( SYS_TYPE )
        ( * RANGE CAN BEGIN WITH AN ID * )
        OUT_LITERALLY ( 'TYPE &' )
        OUT_AFTER_BL ( SYS_ID )
        OUT_LITERALLY ( '=' &' )
        OUT_AFTER_BL ( SYS_TYPE )
; OUT_LITERALLY ( ';' & ' )
END
END

; CODESY
(# THIS IS AN ADDITION BY DR. BATES.
THIS WAY IT IS POSSIBLE TO DECLARE
VARIANT RECORDS. #)
BEGIN
OUT_LITERALLY ( 'TYPE & ' )
OUT_AFTER_BL ( SYS_ID )
OUT_LITERALLY ( ' = & ' )
COPY_VERBATUM
OUT_LITERALLY ( ';' & ' )
NEXT
END
ELSE
(* ANY OTHER POSSIBILITY MEANS THE ID IS
THE FIRST OF A RANGE *)
BEGIN
RANGE ( SYS_TYPE )
OUT_LITERALLY ( 'TYPE & ' )
OUT_AFTER_BL ( SYS_ID )
OUT_LITERALLY ( ' = & ' )
OUT_AFTER_BL ( SYS_TYPE )
OUT_LITERALLY ( ';' & ' )
END

(* THIS PROCEDURE PROCESSES ONE FIELD INSIDE OF A
RECORD DECLARATION. IT DOES NOT EXACTLY PARSE THE
FIELD, BUT IT TAKES CORRECT ACTION. I.E., THE
SYNTAX IS NOT CHECKED, BUT ASSUMING CORRECT SYNTAX,
THE TRANSLATOR TAKES CORRECT ACTION. #)
PROCEDURE REC_VAR_DCL
BEGIN
VAR_GARBAGE : KINDOFID
PTR : PTR_TO_SYM_TAB
BEGIN
IF TOK = VARSY
THEN BEGIN
OUT_TOK_AS_COMM
OUT_AND_GET
OUT_AND_GET
WHILE NOT ( ( TOK = VARSY ) OR ( TOK = ENDSY ) )
DO CASE TOK
OF SIGINTSY
BEGIN
OUT_LITERALLY ( 'INTEGER & ' )
NEXT
END
IDENTSY : OUT_FIXED_ID ( GARBAGE )
LPARSY
BEGIN
REPEAT OUT_AND_GET
ADD_TO_SYM_TAB ( TEXT, OTHERID, PTR )
END
APPEND ( PTR )
; NEXT (* ID *)
UNTIL TOK = RPARSY
; OUT_AND_GET
END
; ARRAYS
BEGIN
OUT_AND_GET
; OUT_LITERALLY ( ' & ' )
END
; OFSY : BEGIN OUT_LITERALLY ( '[' & ' )
; OUT_AND_GET END
; BECOMESSY
BEGIN
OUT_AND_GET
; OUT_ERROR ( '(* ERROR - INIT IN RECORD *) & ' )
; NUMB_ERR := SUCCE ( NUMB_ERR )
END
; PACKEDSY : OUT_TOK_AS_COMM
; ELSE
BEGIN
; BEGIN
IF NOT ( TOK IN QGOODRECORD )
THEN OUT_ERROR ( '(* RANGE HAS EXPN *) & ' )
; NUMB_ERR := SUCCE ( NUMB_ERR )
; OUT_AND_GET
END
END
END

(* THIS IS THE FIRST OF SEVERAL STRAIGHT FORWARD PARSING PROCEDURES. THEY FOLLOW THE SYNTAX DIAGRAMS PRETTY DIRECTLY, SO THEY ARE NOT COMMENTED. *)
; PROCEDURE TYPE_DEF ( VAR SYS_TYPE_OUT : BUFFER )
; FORWARD
; PROCEDURE TYPE_DEF

; VAR SYS_TYPE, SYS_TYPE_2 : BUFFER
; PROCEDURE PUT_OUT_TYPE_PRE
BEGIN
OUT_LITERALLY ( 'TYPE & ' )
; OUT_AFTER_BL ( SYS_TYPE_OUT )
; OUT_LITERALLY ( ' = & ' )
END
BEGIN
GET_SYS_ID ( SYS_TYPE_OUT )
; IF TOK = PACKEDSY
THEN OUT_TOK_AS_COMM
; IF TOK = RECORDSY
THEN BEGIN
PUT_OUT_TYPE_PRE
; OUT_AND_GET
; REC_VAR_DCL
OUT_LITERALLY ( '; &' )
; REC_VAR_DCL
END
; OUT_AND_GET
; OUT_LITERALLY ( '; &' )
; OUT_TOK_AS_COMM
END
ELSE IF TOK = ARAYS
THEN BEGIN
    NEXT
    ; RANGE ( SYS_TYPE )
    ; NEXT
    IF TOK = ARAYS
    THEN TYPE_DEF ( SYS_TYPE_2 )
    ELSE VAR_TYPE ( SYS_TYPE_2 )
    ; PUT_OUT_TYPE_PRE
    ; OUT_LITERALLY ( 'ARRAY [ &' )
    ; OUT_AFTER_BL ( SYS_TYPE )
    ; OUT_LITERALLY ( ' ] OF &' )
    ; OUT_AFTER_BL ( SYS_TYPE_2 )
    ; OUT_LITERALLY ( '; &' )
END
ELSE BEGIN
    VAR_TYPE ( SYS_TYPE_2 )
    ; PUT_OUT_TYPE_PRE
    ; OUT_AFTER_BL ( SYS_TYPE_2 )
    ; OUT_LITERALLY ( '; &' )
END
PROCEDURE CONST_DCL
; VAR SYS_ID : BUFFER
; WHERE : PTR_TO_SYM_TAB
BEGIN
NEXT
; ADD_TO_SYM_TAB ( TEXT, OTHERID, WHERE )
NEXT
IF TOK = BECOMESY
THEN
(* IS SIMPLE CONST *)
BEGIN
    OUT_LITERALLY ( 'CONST &' )
    ; APPEND ( WHERE )
    ; OUT_LITERALLY ( ' = &' )
    ; NEXT
    ; EXPN
END
ELSE
(* IS STRUCTURED CONST *)
BEGIN
NEXT
; TYPE_DEF ( SYS_ID )
NEXT
; OUT_LITERALLY ( 'CONST &' )
; APPEND ( WHERE )
; OUT_LITERALLY ( ' = STRUCTCON &' )
; OUT_AND_GET
1731 ; EXPN
1732 ; WHILE TOK = COMMASY DO BEGIN OUT_AND_GET ; EXPN END
1733 ; OUT_AND_GET
1734 ; OUT_LITERALLY ( ' ; & ' )
1735 ; OUT_AFTER_BL ( SYS_ID )
1736 END
1737 ; OUT_LITERALLY ( ' ; & ' )
1738 END
1739
; PROCEDURE FORMAL_SECT
1740
; VAR WHERE : PTR_TO_SYM_TAB
1743 ; GARBAGE : KINDOFID
1744
; BEGIN
1745 IF TOK = Pervasivesy
1747 THEN OUT_TOK_AS_COMM
1748 ; CASE TOK
1749 OF CONS, READONLYS : OUT_TOK_AS_COMM
1750 ; VARSY : OUT_AND_GET
1751 ; ELSE :
1752 END
1753 ; ADD_TO_SYM_TAB ( TEXT , OTHERID , WHERE )
1754 ; OUT_AND_GET
1755 ; OUT_AND_GET
1756 ; OUT_LITERALLY ( 'UNIV & ' )
1757 ; WHILE TOK = ARRAYSY
1758 DO BEGIN
1759 OUT_ERROR ( '(* NOT LEGAL PARM DCL CHECK RANGE, TOO *) & ' )
1761 ; NUMB_ERR := SUC ( NUMB_ERR )
1762 ; OUT_AND_GET ( "ARRAY" )
1763 ; OUT_LITERALLY ( ' [ & ' )
1764 ; REPEAT IF TOK = IDENTSY
1765 THEN OUT_FIXED_ID ( GARBAGE )
1766 ELSE OUT_AND_GET
1767 UNTIL TOK = OPSY
1768 ; OUT_LITERALLY ( [ ] & ' )
1769 ; OUT_AND_GET
1770 END
1771 ; CASE TOK
1772 OF SIGINTSY : BEGIN OUT_LITERALLY ( 'INTEGER & ' )
1773 ; NEXT END
1774 ; BOOLSY , CHARSY : OUT_AND_GET
1775 ; LPARSY
1776 ; BEGIN
1777 OUT_ERROR ( '(* NOT LEGAL PARM DCL *) & ' )
1778 ; NUMB_ERR := SUC ( NUMB_ERR )
1779 ; OUT_AND_GET
1780 ; OUT_AND_GET ( FIRST ID )
1781 ; WHILE TOK = COMMASY
1782 DO BEGIN
1783 OUT_AND_GET
1784 ; ADD_TO_SYM_TAB ( TEXT , OTHERID , WHERE )
1785 ; OUT_AND_GET
1786 END
1787 ; OUT_AND_GET
END

; IDENTSY
BEGIN

OUT_FIXED_ID ( GARBAGE )

; IF NOT ( ( TOK = COMMASY ) OR ( TOK = RPARYS ) )
THEN BEGIN

OUT_ERROR ( '(* ILLEGAL IN PARM TYPE *)&' )

; NUMB_ERR := SUCC ( NUMB_ERR )
; WHILE NOT ( ( TOK = COMMASY )
OR ( TOK = RPARYS ) )
DO IF TOK = IDENTSY
THEN OUT_FIXED_ID ( GARBAGE )
ELSE OUT_AND_GET
END

END

ELSE
BEGIN

OUT_ERROR ( '(* NOT LEGAL IN PARM TYPE *)&' )

; NUMB_ERR := SUCC ( NUMB_ERR )
; WHILE NOT ( ( TOK = COMMASY )
OR ( TOK = RPARYS ) )
DO IF TOK = IDENTSY
THEN OUT_FIXED_ID ( GARBAGE )
ELSE OUT_AND_GET
END

END

PROCEDURE PARM_LIST
BEGIN

IF TOK = LPARYS
THEN BEGIN

OUT_AND_GET
FORMAL_SECT
; WHILE TOK = COMMASY
DO BEGIN

NEXT
; OUT_LITERALLY ( ';'&' )
FORMAL_SECT
END

OUT_AND_GET
END

PROCEDURE TYPE_DCL
BEGIN

FORWARD

PROCEDURE MOD_OR_VAR_DCL
BEGIN

FORWARD

(* THIS IS THE PROCEDURE THAT OUTS ALL THE "COMPILE-TIME" INITIALIZATIONS. IT POPS THE STACK TWICE
(TWICE FOR SYNCRONIZATION) AND USES THE INFO
TO GENERATE ASSIGNMENT STATEMENTS FOR THE INITS. *)

PROCEDURE OUT_VAR_INIT
BEGIN

VAR INIT_VAR : PTR_TO_SYM_TAB
; INIT_CONST : BUFFER

BEGIN
FOR I := 1 TO 2
DO BEGIN
POP ( INIT_VAR , INIT_CONST )
; WHILE INIT_VAR <> STACK_MARKER
DO BEGIN
APPEND ( INIT_VAR )
; OUT_LITERALLY ( '=' &' '
; OUT_AFTER_BL ( INIT_CONST )
; OUT_LITERALLY ( ';' &' '
; POP ( INIT_VAR , INIT_CONST )
END
END
END

(* THIS OUTPUTS THE TOKENS "CHECKED" AND "NOT CHECKED"
AS TOKENS IF THEY OCCUR IN THE SOURCE. *)

; PROCEDURE CHECK_CHECKED

BEGIN
IF TOK = NOTSY
THEN OUT_TOK_AS_COMM
; IF TOK = CHECKEDSY
THEN OUT_TOK_AS_COMM
END

; PROCEDURE CONVERTER_DCL

; FORWARD

; PROCEDURE ROUT_DEF ( WHERE2 : PTR_TO_SYM_TAB )

VAR THIS_LOOP_LABEL : CHAR

; PROCEDURE STMT ( WHICH_LAB : CHAR )

; PROCEDURE STMT_LIST ( WHICH_LAB : CHAR )

BEGIN
OUT_LITERALLY ( 'BEGIN &' )
; STMT ( WHICH_LAB )
; WHILE TOK IN QFIRST_STMT
DO BEGIN
OUT_LITERALLY ( ';' &' '
; STMT ( WHICH_LAB )
END
; OUT_LITERALLY ( 'END &' )
END

(* PLEASE REFER TO THE ACCOMPANYING DOCUMENTATION FOR
THE EXPLANATION OF THE TRANSLATION OF THE RETURN,
LOOP, ETC. STATEMENTS. *)

; PROCEDURE STMT "WHICH_LAB: CHAR"

VAR GARBAGE_KIND : KINDOFID
; SAVE_LABEL : CHAR
BEGIN

CASE TOK

OF RETURNSY

BEGIN

OUT_LITERALLY ( 'BEGIN &' )

NEXT

IF TOK = LPARSY

THEN BEGIN

APPEND ( WHERE2 )

OUT_LITERALLY ( ' := &' )

NEXT

EXPN

OUT_LITERALLY ( '; &' )

NEXT

END

OUT_LITERALLY ( 'GOTO 1 END &' )

END

IDENTSY, INLINESY

BEGIN

PROC_AND_OUT_VAR

IF TOK = BECOMESY

THEN BEGIN OUT_AND_GET ; EXPN END

END

ASSERTSY

BEGIN

OUT_LITERALLY ( '(' &' )

INSIDE_COMMENT := TRUE

OUT_AND_GET

IF TOK = LPARSY

THEN BEGIN OUT_AND_GET ; EXPN ; OUT_AND_GET END

INSIDE_COMMENT := FALSE

OUT_LITERALLY ( ' *) &' )

END

BEGINSY

BEGIN

NEXT

CHECK_CHECHED

STM_LIST ( WHICH_LAB )

NEXT

END

LOOPSY

BEGIN

THIS_LOOP_LABEL := SUCC ( THIS_LOOP_LABEL )

SAVE_LABEL := THIS_LOOP_LABEL

OUT_LITERALLY ( 'WHILE TRUE DO &' )

NEXT

STM_LIST ( THIS_LOOP_LABEL )

OUT_LITERALLY ( '; &' )

WRITE ( SAVE_LABEL )

NUMB_OUTPUT := SUCC ( NUMB_OUTPUT )

OUT_LITERALLY ( '; &' )

OUT_TOK_AS_COMM

OUT_TOK_AS_COMM

END

CASESY

BEGIN

OUT_AND_GET
; EXPN
1963 ; OUT_AND_GET
1964 ; WHILE NOT ( TOK IN QCASE )
1965 DO BEGIN
1966 (* TOKEN = FIRST OF CASE DESIGNATOR *)
1967 REPEAT IF TOK = IDENTSY
1968 THEN OUT_FIXED_ID ( GARBAGE_KIND )
1969 ELSE OUT_AND_GET
1970 UNTIL TOK = CASESEPSY
1971 ; OUT_LITERALLY ( ': &' )
1972 ; NEXT
1973 ; STMT_LIST ( WHICH_LAB )
1974 ; OUT_TOK_AS_COMM
1975 ; OUT_TOK_AS_COMM
1976 ; IF TOK <> ENDSY
1977 THEN OUT_LITERALLY ( '; &' )
1978 END
1979 ; IF TOK = OTHERWISESY
1980 THEN BEGIN
1981 NEXT
1982 ; NEXT
1983 ; OUT_LITERALLY ( 'ELSE: &' )
1984 ; STMT_LIST ( WHICH_LAB )
1985 END
1986 ; OUT_AND_GET
1987 ; OUT_TOK_AS_COMM
1988 END
1989 ; IPSY
1990 ; BEGIN
1991 OUT_AND_GET
1992 ; EXPN
1993 ; OUT_AND_GET
1994 ; STMT_LIST ( WHICH_LAB )
1995 ; WHILE TOK = ELSEIFSY
1996 DO BEGIN
1997 ; OUT_LITERALLY ( 'ELSE IF &' )
1998 ; NEXT
1999 ; EXPN
2000 ; OUT_AND_GET
2001 ; STMT_LIST ( WHICH_LAB )
2002 END
2003 ; OUT_LITERALLY ( 'ELSE &' )
2004 ; IF TOK = ELSESY
2005 THEN NEXT
2006 ; STMT_LIST ( WHICH_LAB )
2007 ; OUT_TOK_AS_COMM
2008 ; OUT_TOK_AS_COMM
2009 END
2010 ; EXITSY
2011 ; BEGIN
2012 NEXT
2013 ; IF TOK = WHENSY
2014 THEN BEGIN
2015 NEXT
2016 ; OUT_LITERALLY ( 'IF &' )
2017 ; EXPN
2018 ; OUT_LITERALLY ( 'THEN &' )
2019 END
; OUT_LITERALLY ( 'GOTO &' )
; WRITE ( WHICH_LAB )
; NUMB_OUTPUT := SUCC ( NUMB_OUTPUT )
END
; ELSE :
END
END

; BEGIN
OUT_LITERALLY ( '(* &' )
; INSIDE_COMMENT := TRUE
; WHILE NOT ( ( TOK = BEGINSY ) OR ( TOK = CODESY ) )
DO OUT_AND_GET
; INSIDE_COMMENT := FALSE
; OUT_LITERALLY ( ' *') &' )
; MARK_STACK
; IF TOK = CODESY
THEN BEGIN
OUT_LITERALLY ( 'BEGIN &' )
; OUT_VAR_INIT
; COPY_VERBATUM
; NEXT
END
ELSE BEGIN
; CHECK_CHECKED
; NEXT
; CHECKED
; WHILE TOK IN QDCL
DO CASE TOK
OF TYPESY : TYPE_DCL
; CONSY : CONST_DCL
; VARSY : MOD OR VAR_DCL
; CONVERTERSY : CONVERTER_DCL
ELSE
; BEGIN
OUT_ERROR ( '(*LOGIC EROR DCL-IN-ROUT *) &' )
; NUMB_ERR := SUCC ( NUMB_ERR )
END
END
; OUT_LITERALLY ( 'BEGIN &' )
; OUT_VAR_INIT
; THIS_LOOP_LABEL := '1'
; STMNT_LIST ( '1' )
END
END

; PROCEDURE PROC_DCL

VAR WHERE3 , WHERE , WHERE2 : PTR_TO_SYM_TAB
; SAVE : BUFFER
; GARBAGE : KINDOFID
; FUNCT_IND : BOOLEAN
; LAB : CHAR
; BEGIN
IF TOK = FUNCXY
THEN FUNCT_IND := TRUE
ELSE FUNCT_IND := FALSE
; OUT_AND_GET
2079    ; ADD_TO_SYM_TAB ( TEXT , PROCID , WHERE )
2080    ; MARK_STACK
2081    ; APPEND ( WHERE )
2082    ; START_NEW_TAB ( BLANKS )
2083    ; NEXT
2084    ; PARM_LIST
2085    ; IF FUNCT_END
2086    THEN BEGIN
2087    OUT_TOK_AS_COMM ( "RETURNS" )
2088    ; ADD_TO_SYM_TAB ( TEXT , OTHERID , WHERE2 )
2089    ; OUT_TOK_AS_COMM ( ID )
2090    ; OUT_AND_GET ( "":"" )
2091    ; CASE TOK
2092    OF SIGINTSY
2093        BEGIN
2094        OUT_LITERALLY ( 'INTEGER '&' )
2095        ; SAVE := 'INTEGER
2096        ?
2097    'INTEGER
2098        ; NEXT
2099    END
2100    ; BOOLSY , CHARSY : BEGIN SAVE := TEXT ; OUT_AND_GET
2101    END
2102    ; LPARSY
2103    BEGIN
2104    OUT_ERROR
2105    ( )
2106    (* ILLEGAL FUNCTION DCL-WILL CAUSE MULTIPL ERR-CHANGE SOURCE *)&'
2107    ; NUMB_ERR := NUMB_ERR + 1
2108    ; OUT_AND_GET ( "(" "")
2109    ; SAVE := TEXT
2110    ; ADD_TO_SYM_TAB ( TEXT , OTHERID , WHERE3 )
2111    ; OUT_AND_GET
2112    ; WHILE TOK = COMMASY
2113    DO BEGIN
2114        OUT_AND_GET
2115    ; ADD_TO_SYM_TAB ( TEXT , OTHERID , WHERE3 )
2116    ; OUT_AND_GET
2117    END
2118    ; OUT_AND_GET ( "" )
2119    END
2120    ; IDENTSY
2121    BEGIN
2122    SAVE := TEXT
2123    ; OUT_FIXED_ID ( GARBAGE )
2124    ; IF TOK <> EQUALSSY
2125    THEN (* TYPE IS RANGE *) BEGIN
2126        OUT_ERROR ( "(* ILLEGAL FUNCTION TYPE *)&' )
2127    ; NUMB_ERR := NUMB_ERR + 1
2128    ; WHILE TOK <> EQUALSSY
2129    DO IF TOK = IDENTSY
2130    THEN OUT_FIXED_ID ( GARBAGE )
2131    ELSE OUT_AND_GET
2132    END
END
2134 ; ELSE (* TYPE IS RANGE *)
2135 BEGIN
2136 SAVE := TEXT
2137 ; OUT_ERROR ( "(* ILLEGAL FUNCTION TYPE *)"&' )
2138 ; NUMB_ERR := NUMB_ERR + 1
2139 ; WHILE TOK <> EQUALSSY
2140 DO IF TOK = IDENTSY
2141 THEN OUT_FIXED_ID ( GARBAGE )
2142 ELSE OUT_AND_GET
2143 END
2144 END
2145 ;
2146 ; OUT_LITERALLY ( ' ; VAR &' )
2147 ; OUT_AFTER_BL ( WHERE2 ^ . ID )
2148 ; OUT_LITERALLY ( ':' &' )
2149 ; OUT AFTER BL ( SAVE )
2150 ; OUT_LITERALLY
2151 ( "(* BE SURE THIS MATCHES FUNCT TYPE!!!@!!!*)"&' )
2152 END
2153 ;
2154 ; OUT_LITERALLY ( ' ; &' )
2155 NEXT
2156 ; LAB := '1'
2157 ; OUT_LITERALLY ( 'LABEL &' )
2158 ; NUMB_OUTPUT := SUCC ( NUMB_OUTPUT )
2159 ; WRITE ( LAB )
2160 ; REPEAT OUT_LITERALLY ( ' , &' )
2161 ; LAB := SUCC ( LAB )
2162 ; NUMB_OUTPUT := SUCC ( NUMB_OUTPUT )
2163 ; WRITE ( LAB )
2164 UNTIL LAB = '9' .
2165 ; OUT_LITERALLY ( ' ; &' )
2166 ; ROUT_DEF ( WHERE2 )
2167 ; OUT_LITERALLY ( ' ; &' )
2168 ; OUT_LITERALLY ( '1 ; &' )
2169 ; IF FUNCT_IND
2170 THEN BEGIN
2171 Append ( WHERE )
2172 ; OUT_LITERALLY ( ' := &' )
2173 ; APPEND ( WHERE2 )
2174 END
2175 ; OUT_AND_GET
2176 ; OUT_LITERALLY ( ' ; &' )
2177 ; GET_RID_OF_TAB ( TRUE )
2178 ; OUT_TOK_AS_COMM
2179 ;
2180 (* THIS PROC ADDS A CONVERTER ID TO THE SYMBOL TABLE,
2181 AND DELETES ITS DECLARATION. *)
2182 ; PROCEDURE CONVERTER_DCL
2183 ; VAR WHERE : PTR_TO_SYM_TAB
2184 ; BEGIN
2185 INSIDE_COMMENT := TRUE
2186 ; OUT_LITERALLY ( '(* &' )
2187 ; OUT_AND_GET (* CONVERTER SY *)
2191  ; ADD_TO_SYM_TAB ( TEXT , CONVERTERID , WHERE )
2192  ; WHILE TOK <> RETURNSSY DO OUT_AND_GET
2193  ; OUT_AND_GET ( * RETURNSSY * )
2194  ; OUT_AND_GET ( * RESULT TYPE * )
2195  ; OUT_LITERALLY ( ' * ' )
2196  ; INSIDE_COMMENT := FALSE
2197  END
2198
2199  ; PROCEDURE TYPE_DCL (* TOK = "TYPE" *)
2200
2201  ; VAR SYS_TYPE : BUFFER
2202  ; WHERE_ADDED : PTR_TO_SYM_TAB
2203
2204  ; BEGIN
2205   NEXT
2206  ; ADD_TO_SYM_TAB ( TEXT , TYPEID , WHERE_ADDED )
2207  ; NEXT
2208  ; NEXT
2209  ; TYPE_DEF ( SYS_TYPE )
2210  ; OUT_LITERALLY ( 'TYPE &' )
2211  ; APPEND ( WHERE_ADDED )
2212  ; OUT_LITERALLY ( ' = &' )
2213  ; OUT_AFTER_BL ( SYS_TYPE )
2214  ; OUT_LITERALLY ( ';' &' )
2215  END
2216
2217  ; PROCEDURE MODULE_TYPE ( VAR INITIALLY_PART : BOOLEAN )
2218  ; FORWARD
2219
2220  ; PROCEDURE MOD_OR_VAR_DCL
2221
2222  ; VAR WHERE_ADDED : PTR_TO_SYM_TAB
2223  ; INITIALLY_PART : BOOLEAN
2224  ; SYS_TYPE , SYS_CONST : BUFFER
2225
2226  ; BEGIN
2227   NEXT
2228  ; ADD_TO_SYM_TAB ( TEXT , MODULEID , WHERE_ADDED )
2229  ; NEXT
2230  ; NEXT
2231  ; IF ( TOK = MACHINESY ) OR ( TOK = MODULESY )
2232  THEN BEGIN
2233  START_NEW_TAB ( WHERE_ADDED " . ID )
2234  ; WHERE_ADDED " . ITS_OWN_TAB
2235  := TOP_OF_DISP . SYM_TAB_PTR
2236  ; MODULE_TYPE ( INITIALLY_PART )
2237  (* "INITIALLY_PART" IS TO TELL IF THE MODULE HAD
2238  HAD AN INITIALIZATION PROC WITH IT. IF IT DID,
2239  THE PROCEDURE IS ALREADY TAKEN CARE OF. IF
2240  IT DID NOT, WE NEED TO HAVE ONE ANYWAY SO THE
2241  MODULE VARIABLES CAN GET INITIALIZED. SO, THIS
2242  IS WHERE THE INIT ROUTINE IS GENERATED IF THE
2243  MODULE DID NOT HAVE AN INIT ROUTINE. *)
2244  ; IF NOT INITIALLY_PART
2245  THEN BEGIN
2246  OUT_LITERALLY ( 'PROCEDURE &' )
2247  ; MARK_STACK
2248  ; OUT_AFTER_BL ( TOP_OF_DISP " . APPEND_ID )
; OUT_WOUT_BL_LIT ( '_INITIAL ;&' )
; QUEUE_INIT_PROC ( TOP_OF_DISP ^ . APPEND_ID )
; OUT_LITERALLY ( 'BEGIN &' )
; OUT_VAR_INIT
; OUT_LITERALLY ( 'END; &' )
END
; GET_RID_OF_TAB ( FALSE )
END
ELSE
(* IT WAS A VAR DCL *)
BEGIN
WHERE_ADDED ^ . TYPE_OF_ID := OTHERID
; TYPE_DEF ( SYS_TYPE )
; OUT_LITERALLY ( 'VAR &' )
; APPEND ( WHERE_ADDED )
; OUT_LITERALLY ( ';' &' )
; OUT_AFTER_BL ( SYS_TYPE )
; OUT_LITERALLY ( ';' &' )
IF TOK = BECOMESY
THEN BEGIN
    NEXT
; GET_SYS_ID ( SYS_CONST )
; PUSH ( WHERE_ADDED , SYS_CONST )
; OUT_LITERALLY ( 'CONST &' )
; OUT_AFTER_BL ( SYS_CONST )
; OUT_LITERALLY ( '=' &' )
EXPN
; OUT_LITERALLY ( ';' &' )
END
END
PROCEDURE MODULE_TYPE "VAR INITIALLY_PART: BOOLEAN"
VAR GARbage : PTR_TO_SYM_TAB
; LAB : CHAR
BEGIN
    INITIALLY_PART := FALSE
; OUT_LITERALLY ( '*' &' )
; INSIDE_COMMENT := TRUE
; WHILE NOT ( TOK IN QFIRSTMODDCL ) DO OUT_AND_GET
; OUT_LITERALLY ( ' ' &' )
; INSIDE_COMMENT := FALSE
; MARK_STACK
; WHILE TOK <> ENDSY
DO CASE TOK
OF PERVERSIVESY : OUT_TOK_AS_COMM
; IMPORTSSY , EXPORTSSY
    BEGIN
    OUT_LITERALLY ( '*' &' )
; INSIDE_COMMENT := TRUE
; OUT_AND_GET
; WHILE TOK <> RPARYS DO OUT_AND_GET
; OUT_LITERALLY ( ' ' &' )
; INSIDE_COMMENT := FALSE
; OUT_TOK_AS_COMM
END
TYPESY : TYPE_DCL
CONSTSY : CONST_DCL
FUNCSY, PROCSY : PROC_DCL
CONVERTERSY : CONVERTER_DCL
VARSY : MOD_OR_VAR_DCL
INLINESY : OUT_TOK_AS_COMM
INITSY
BEGIN
INITIALLY_PART := TRUE
OUT_LITERALLY ( 'PROCEDURE &' )
OUT_AFTER_BL ( TOP_OF_DISP "&" APPEND_ID )
OUT_WOUT_BL_LIT ( '_INITIAL; &' )
QUEUE_INIT_PROC ( TOP_OF_DISP "&" APPEND_ID )
START_NEW_TAB ( BLANKS )
LAB := '1'
OUT_LITERALLY ( 'LABEL &' )
NUMB_OUTPUT := SUCC ( NUMB_OUTPUT )
WRITE ( LAB )
REPEAT OUT_LITERALLY ( ' &
LAB := SUCC ( LAB )
NUMB_OUTPUT := SUCC ( NUMB_OUTPUT )
WRITE ( LAB )
UNTIL LAB = '9'
OUT_LITERALLY ( '; &'
ROUT_DEF ( GARBAGE )
OUT_AND_GET
OUT_LITERALLY ( ';' &
GET_RID_OF_TAB ( TRUE )
END
ELSE
BEGIN
OUT_ERROR ( ' (* LOGIC ERROR IN MOD DCL *) &' )
NUMB_ERR := SUCC ( NUMB_ERR )
OUT_TOK_AS_COMM
END
OUT_TOK_AS_COMM
OUT_TOK_AS_COMM
END
(* THIS GENERATES THE MAIN BODY OF THE PASCAL OBJECT.
THIS GENERATES PROCEDURE CALLS ON ALL THE
MODULE INITIALIZATION PROCEDURES. *)
PROCEDURE INITIALIZE_ALL_ROUT
VAR PTR_Q : INIT_PTR
BEGIN
PTR_Q := FRONT_OF_QUEUE
REPEAT PTR_Q := PTR_Q \ NEXT_QUEUE
OUT_AFTER_BL ( PTR_Q \ INIT_ID )
OUT_WOUT_BL_LIT ( '_INITIAL&' )
OUT_LITERALLY ( ';' &
UNTIL PTR_Q \ NEXT_QUEUE = NIL
END
(* THIS PROC READS IN AND OUTPUTS THE PROPER PREFIX *)
PROCEDURE OUT_PREFIX
; CONST PREFIXFILE = 1

; VAR BLOCK : PAGE
; NEXT_CHAR_NO , NEXT_BLOCK_NO : INTEGER

; BEGIN
; GET ( PREFIXFILE , 1 , BLOCK )
; NEXT_BLOCK_NO := 2
; NEXT_CHAR_NO := 1
; WHILE BLOCK [ NEXT_CHAR_NO ] <> EM
; DO BEGIN
; WRITE ( BLOCK [ NEXT_CHAR_NO ] )
; IF NEXT_CHAR_NO = PAGELENGTH
; THEN BEGIN
; GET ( PREFIXFILE , NEXT_BLOCK_NO , BLOCK )
; NEXT_BLOCK_NO := SUCC ( NEXT_BLOCK_NO )
; NEXT_CHAR_NO := 1
; END
; ELSE NEXT_CHAR_NO := SUCC ( NEXT_CHAR_NO )
; END

; PROCEDURE INITIALIZE

; VAR I : INTEGER
; BIG_TEMP : BUFFER
; TEMP : ARRAY [ 1 .. 18 ] OF CHAR

(* THIS PROCEDURE LOADS THE SYMBOL TABLE WITH WHATEVER
ID AND KIND_OF_ID IT IS PASSED. IT'S PURPOSE
IS TO SAVE WORK, BECAUSE THERE ARE MANY SUCH
IDENTIFIERS. *)

; PROCEDURE INIT_RSVD_WDS
( WHAT_RSV_WRD : BUFFER ; WHAT_KIND : KINDOFID )

; VAR PASS_TO_TAB : BUFFER
; I : INTEGER
; PTR : PTR_TO_SYM TAB

; BEGIN
; PASS_TO_TAB := BLANKS
; I := 1
; REPEAT PASS_TO_TAB [ I ] := WHAT_RSV_WRD [ I ]
; I := SUCC ( I )
; UNTIL WHAT_RSV_WRD [ I ] = '0'
; ADD_TO_SYM_TAB ( PASS_TO_TAB , WHAT_KIND , PTR )
; END

; BEGIN
; DIGITS
; := [ '0' , '1' , '2' , '3' , '4' , '5' , '6' , '7' ,
; '8' , '9'
; ]
; LETTER_DIG
; := DIGITS
; + [ 'A' , 'B' , 'C' , 'D' , 'E' , 'F' , 'G' , 'H' ,
; 'I' ]
; QFIRSTSTMT
; QCASE := [ OTHERWISESY, ENDSY ]
; QIRSTMODCL
; QDCL
; QGOODRECORD := [ CONSTANTSY, ELIPSISSY, CHARSY
; BOOLSY ]
; INSIDE_COMMENT := FALSE
; INITWTABLE
; FREE_STACK := NIL
; FREE_DISP := NIL
; FREE_TABLE := NIL
; READ ( INPUT_CHAR )
; NEXT
; NUMB_ERR := 0
; NUMB_OUTPUT := 0
(* INITIALIZATION OF PROCEDURES QUEUE *)
; NEW ( FRONT_OF_QUEUE )
; ADD_ON_QUEUE := FRONT_OF_QUEUE
; ADD_ON_QUEUE ^ . NEXT_QUEUE := NIL
(* STACK INIT *)
; STACK_TOP := NIL
(* SYMBOL TABLE INIT *)
; NEW ( BOTTOM_OF_DISP )
; TOP_OF_DISP := BOTTOM_OF_DISP
; NEW ( PTR_T )
; WITH TOP_OF_DISP ^
DO BEGIN
    EARLIER_PTR := NIL
; LATER_PTR := NIL
; APPEND_ID := BLANKS
; SYM_TAB_PTR := PTR_T
END
; WITH PTR_T ^
DO BEGIN
    LEFT_SON := NIL
; RT_SON := NIL
; ID := BLANKS
; BACK_TO_DISP := TOP_OF_DISP
END
; OUT_LITERALLY ( 'TYPE StorageUnit = INTEGER;' )
; INIT_RSVD_WDS ( 'StorageUnit:', TYPEID )
; INIT_RSVD_WDS ( 'false:', OTHERID )
2481 ; INIT_RSVD_WDS ( 'true', OTHERID )
2482 ; INIT_RSVD_WDS ( 'Chr', PROCID )
2483 ; INIT_RSVD_WDS ( 'Ord', PROCID )
2484 ; INIT_RSVD_WDS ( 'optionsArray', OTHERID )
2485 ; INIT_RSVD_WDS ( 'NL', OTHERID )
2486 ; INIT_RSVD_WDS ( 'FP', OTHERID )
2487 ; INIT_RSVD_WDS ( 'CH', OTHERID )
2488 ; INIT_RSVD_WDS ( 'EH', OTHERID )
2489 ; INIT_RSVD_WDS ( 'PAGELENGTH', OTHERID )
2490 ; INIT_RSVD_WDS ( 'PAGE', TYPEID )
2491 ; INIT_RSVD_WDS ( 'LINELENGTH', OTHERID )
2492 ; INIT_RSVD_WDS ( 'LINE', TYPEID )
2493 ; INIT_RSVD_WDS ( 'IDLENGTH', OTHERID )
2494 ; INIT_RSVD_WDS ( 'IDENTIFIER', TYPEID )
2495 ; INIT_RSVD_WDS ( 'FILE', TYPEID )
2496 ; INIT_RSVD_WDS ( 'FILEKIND', TYPEID )
2497 ; INIT_RSVD_WDS ( 'EMPTY', OTHERID )
2498 ; INIT_RSVD_WDS ( 'SCRATCH', OTHERID )
2499 ; INIT_RSVD_WDS ( 'ASCII', OTHERID )
2500 ; INIT_RSVD_WDS ( 'SEQCODE', OTHERID )
2501 ; INIT_RSVD_WDS ( 'CONCODE', OTHERID )
2502 ; INIT_RSVD_WDS ( 'FILEATTR', OTHERID )
2503 ; INIT_RSVD_WDS ( 'IODEVICE', TYPEID )
2504 ; INIT_RSVD_WDS ( 'TYPEDEVICE', OTHERID )
2505 ; INIT_RSVD_WDS ( 'DISKDEVICE', OTHERID )
2506 ; INIT_RSVD_WDS ( 'TAPEDEVICE', OTHERID )
2507 ; INIT_RSVD_WDS ( 'PRINTDEVICE', OTHERID )
2508 ; INIT_RSVD_WDS ( 'CARDDEVICE', OTHERID )
2509 ; INIT_RSVD_WDS ( 'IOPERATION', TYPEID )
2510 ; INIT_RSVD_WDS ( 'INPUT', OTHERID )
2511 ; INIT_RSVD_WDS ( 'OUTPUT', OTHERID )
2512 ; INIT_RSVD_WDS ( 'MOVE', OTHERID )
2513 ; INIT_RSVD_WDS ( 'CONTROL', OTHERID )
2514 ; INIT_RSVD_WDS ( 'IOARG', TYPEID )
2515 ; INIT_RSVD_WDS ( 'WRITEOF', OTHERID )
2516 ; INIT_RSVD_WDS ( 'REWIND', OTHERID )
2517 ; INIT_RSVD_WDS ( 'UPSPACE', OTHERID )
2518 ; INIT_RSVD_WDS ( 'BACKSPACE', OTHERID )
2519 ; INIT_RSVD_WDS ( 'IRESULT', TYPEID )
2520 ; INIT_RSVD_WDS ( 'COMPLETE', OTHERID )
2521 ; INIT_RSVD_WDS ( 'INTERVENTION', OTHERID )
2522 ; INIT_RSVD_WDS ( 'TRANSMISSION', OTHERID )
2523 ; INIT_RSVD_WDS ( 'FAILURE', OTHERID )
2524 ; INIT_RSVD_WDS ( 'ENDFILE', OTHERID )
2525 ; INIT_RSVD_WDS ( 'ENDMEDIUM', OTHERID )
2526 ; INIT_RSVD_WDS ( 'STARTMEDIUM', OTHERID )
2527 ; INIT_RSVD_WDS ( 'IOPARAM', TYPEID )
2528 ; INIT_RSVD_WDS ( 'TASKKIND', TYPEID )
2529 ; INIT_RSVD_WDS ( 'INPUTTASK', OTHERID )
2530 ; INIT_RSVD_WDS ( 'JOBTASK', OTHERID )
2531 ; INIT_RSVD_WDS ( 'OUTPUTTASK', OTHERID )
2532 ; INIT_RSVD_WDS ( 'ARGETAG', TYPEID )
2533 ; INIT_RSVD_WDS ( 'NILTYPE', OTHERID )
2534 ; INIT_RSVD_WDS ( 'BOOLTYPE', OTHERID )
2535 ; INIT_RSVD_WDS ( 'INTTYPE', OTHERID )
2536 ; INIT_RSVD_WDS ( 'IDTYPE', OTHERID )
2537 ; INIT_RSVD_WDS ( 'PTRTYPE', OTHERID )
2538 ; INIT_RSVD_WDS ( 'POINTER', TYPEID )
; INIT_RSVRD_WDS ('PASSPTR@', TYPEID )
; INIT_RSVRD_WDS ('PASSLINK@', TYPEID )
; INIT_RSVRD_WDS ('ARGTYPE@', TYPEID )
; INIT_RSVRD_WDS ('MAXARG@', OTHERID )
; INIT_RSVRD_WDS ('ARGLIST@', OTHERID )
; INIT_RSVRD_WDS ('ARGS@', TYPEID )
; INIT_RSVRD_WDS ('INFE', OTHERID )
; INIT_RSVRD_WDS ('OUTE', OTHERID )
; INIT_RSVRD_WDS ('PROGRESS@', TYPEID )
; INIT_RSVRD_WDS ('TERMINATED@', OTHERID )
; INIT_RSVRD_WDS ('OVERFLOW@', OTHERID )
; INIT_RSVRD_WDS ('POINTERERROR@', OTHERID )
; INIT_RSVRD_WDS ('RANGEERROR@', OTHERID )
; INIT_RSVRD_WDS ('VARIANTERROR@', OTHERID )
; INIT_RSVRD_WDS ('HEAPLIMIT@', OTHERID )
; INIT_RSVRD_WDS ('STACKLIMIT@', OTHERID )
; INIT_RSVRD_WDS ('CODELIMIT@', OTHERID )
; INIT_RSVRD_WDS ('TIMELIMIT@', OTHERID )
; INIT_RSVRD_WDS ('CALLERR@', OTHERID )
; INIT_RSVRD_WDS ('READ@', PROCID )
; INIT_RSVRD_WDS ('WRITE@', PROCID )
; INIT_RSVRD_WDS ('PfxOpen@', PROCID )
; INIT_RSVRD_WDS ('PfxClose@', PROCID )
; INIT_RSVRD_WDS ('PfxGet@', PROCID )
; INIT_RSVRD_WDS ('PfxPut@', PROCID )
; INIT_RSVRD_WDS ('PfxLength@', PROCID )
; INIT_RSVRD_WDS ('MARK@', PROCID )
; INIT_RSVRD_WDS ('RELEASE@', PROCID )
; INIT_RSVRD_WDS ('IDENTIFY@', PROCID )
; INIT_RSVRD_WDS ('PfxAccept@', PROCID )
; INIT_RSVRD_WDS ('PfxDisplay@', PROCID )
; INIT_RSVRD_WDS ('RUN@', PROCID )
; INIT_RSVRD_WDS ('P@', PROCID )
; INIT_RSVRD_WDS ('IOMode@', TYPEID )
; INIT_RSVRD_WDS ('inFile@', OTHERID )
; INIT_RSVRD_WDS ('out@', OTHERID )
; INIT_RSVRD_WDS ('inOutFile@', OTHERID )
; INIT_RSVRD_WDS ('DIRECTION@', OTHERID )
; INIT_RSVRD_WDS ('size@', PROCID )

(* SYSTEM_ID INIT *)
; SYSTEM_ID := BLANKS
; TEMP := 'SYSID0000000 '
FOR I := 1 TO 18 DO SYSTEM_ID [ I ] := TEMP [ I ]
END

; BEGIN
BREAKPNT
; OUT_PREFIX
; INITIALIZE
; OUT_LITERALLY ( '(* &)' )
; INSIDE_COMMENT := TRUE
; OUT_AFTER_BL ( TEXT )
; WHILE TOK <> EQUALSSY
DO BEGIN NEXT ; OUT_AFTER_BL ( TEXT ) END
; OUT_LITERALLY ( ' *' &' )
; INSIDE_COMMENT := FALSE
2596 ; NEXT
2597 ; MODULE_TYPE ( GARBAGE )
2598 ; OUT_LITERALLY ( ' BEGIN &' )
2599 ; INITIALIZE_ALL_ROUT
2600 ; OUT_LITERALLY ( 'END. &' )
2601 ; WRITE ( NL )
2602 ; WRITE ( EM )
2603 END
2604 .
APRIL EUCLID TO PASCAL TRANSLATOR

by

DAVID PAUL ROESENER

B. S., Kansas State University, 1979

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1980
April Euclid is a computer language which extends the concepts of Pascal to ideas of modularization, strict type checking, and run-time program proving constructs.

Dr. Rod Bates of the computer science faculty of Kansas State University bought an April Euclid compiler, written in April Euclid, that he wanted bootstrapped on the department's machine. The first part of the bootstrapping was composed of writing a translator from April Euclid to Pascal, since the department's Perkin-Elmer 8-32 has Pascal on it, but not Euclid.

Several problems were encountered in the translation - de-modularizing nested structures, variable initializations in these structures, and Euclid's precedence of operators.

The translator was completed. To date, pass 1 of the Euclid compiler is translated and the object code is compiled.

This bootstrapping process is the subject of this paper.