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:

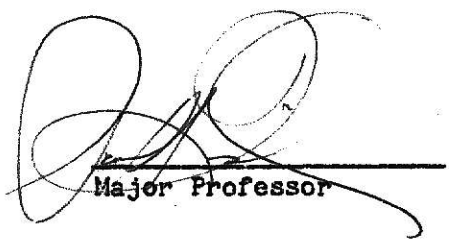


TABLE OF CONTENTS

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:

```
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

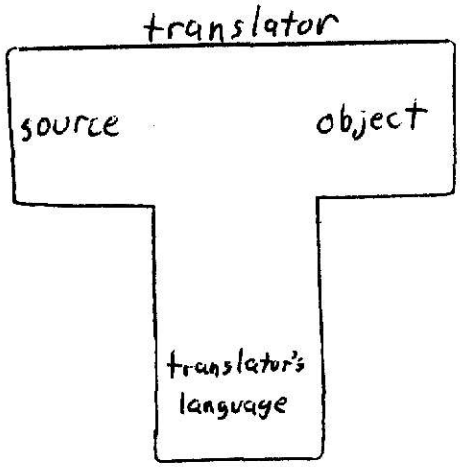
THIS BOOK CONTAINS NUMEROUS PAGES WITH DIAGRAMS THAT ARE CROOKED COMPARED TO THE REST OF THE INFORMATION ON THE PAGE.

THIS IS AS RECEIVED FROM CUSTOMER.

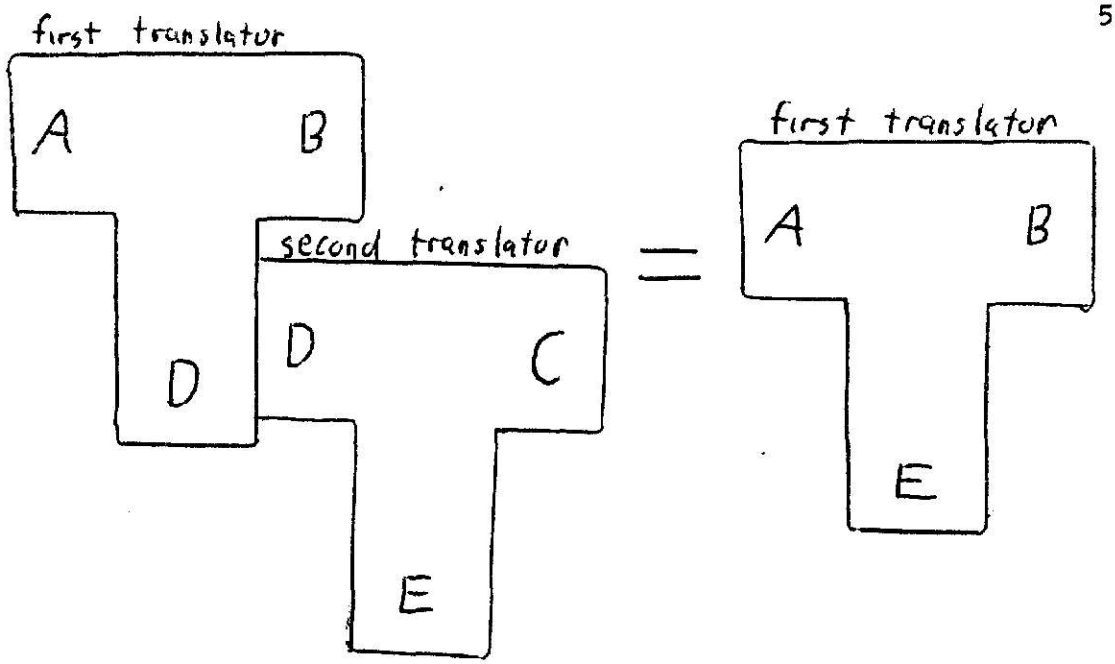
to the translator, producing Pascal programs which can be compiled and run on the Interdata machine. The input/output routines of the Euclid compiler have to be modified before it is translated to Pascal, because the target machines' operating systems are different.

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.

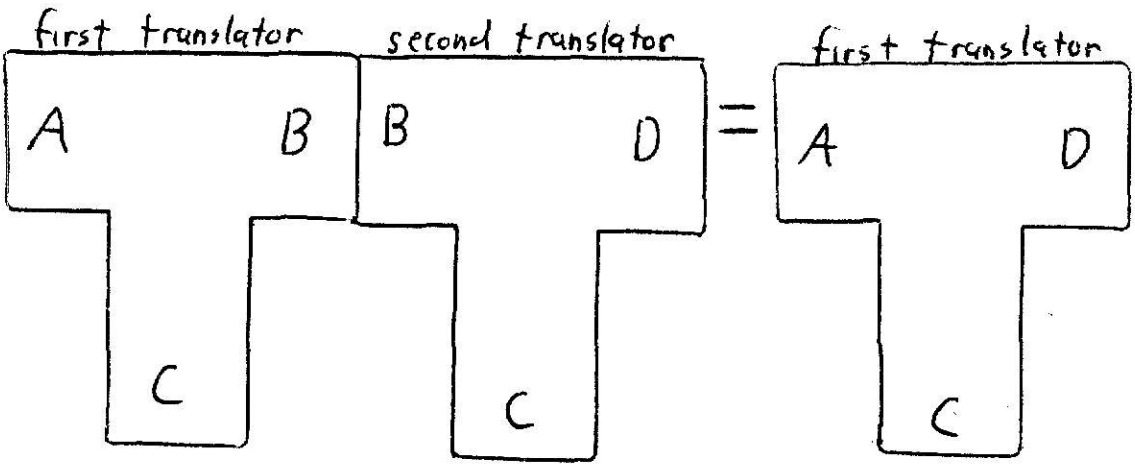
To make bootstrapping processes clearer, T-shaped diagrams are used. Each "T" represents a translator. The left part of the cross bar of the "T" is the source language. The right part of the cross bar is the object language. The bottom of the "T" is the language the translator is written in. The diagram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



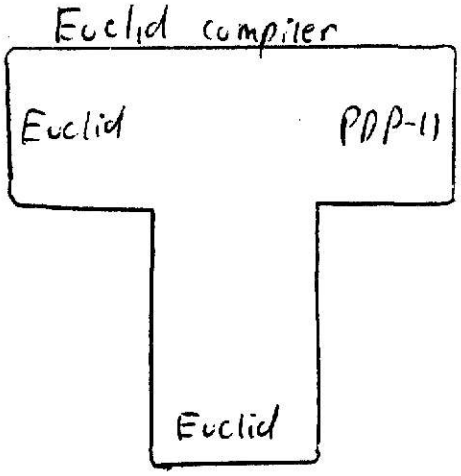
If the translator in a diagram is itself translated, the following "T" diagram identity applies. This example shows a translator that is written in language "D". The translator is itself translated by a second translator to language "C". Therefore, the total translation is as if the first translator were written in language "C". The second translator is written in any language that can be interpreted.



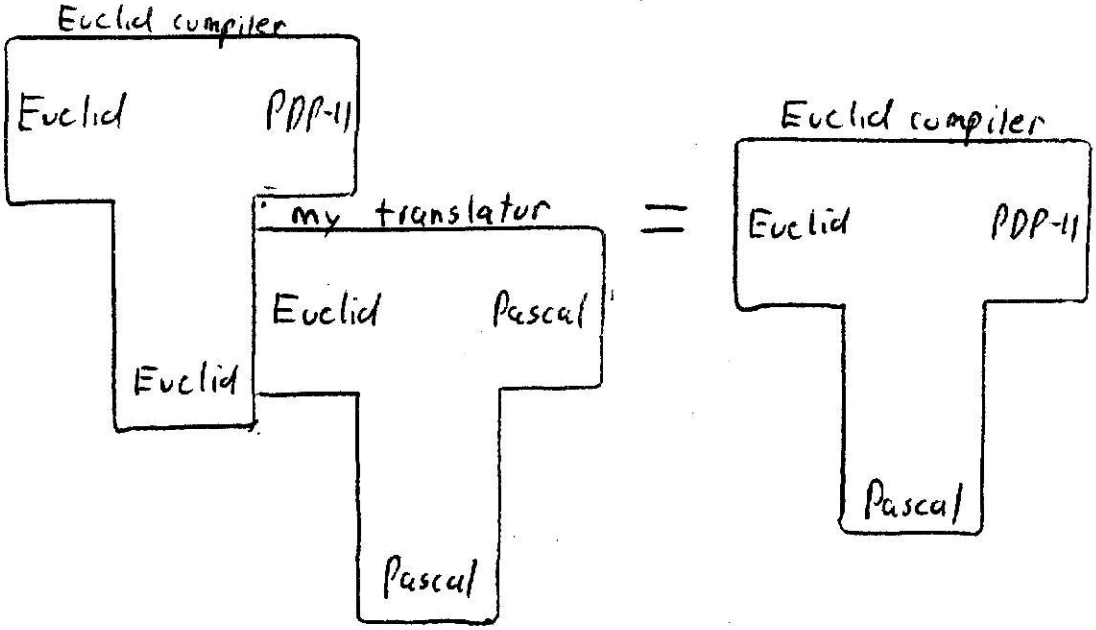
If a program is translated to an object language, and this object language is further translated, the following "T" diagram identity applies. The language of the two translators has to be the same.



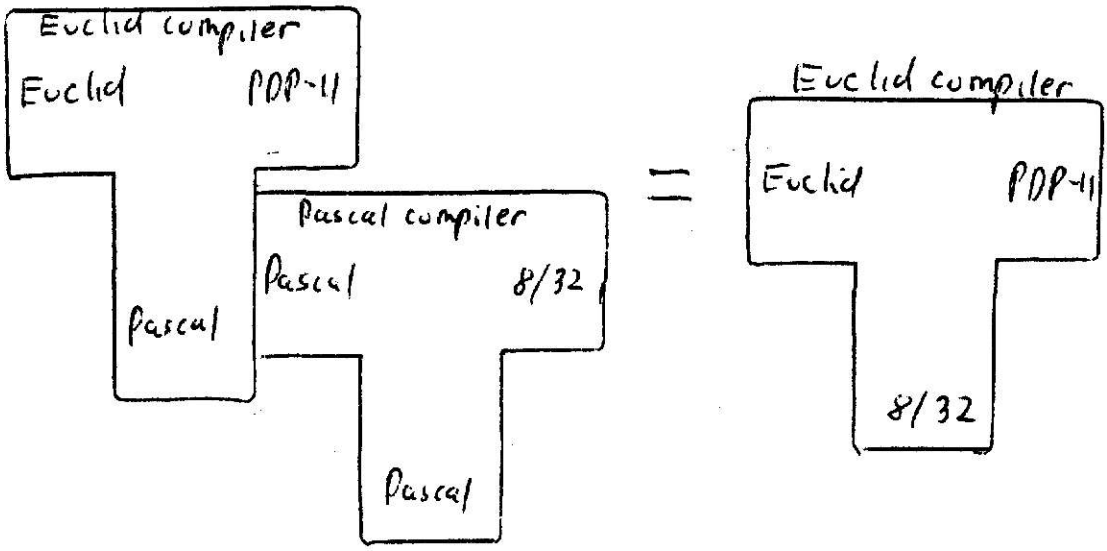
The following diagrams illustrate the first step of the bootstrapping process. The Euclid compiler is illustrated by the following diagram:



The Euclid compiler is translated to Pascal:

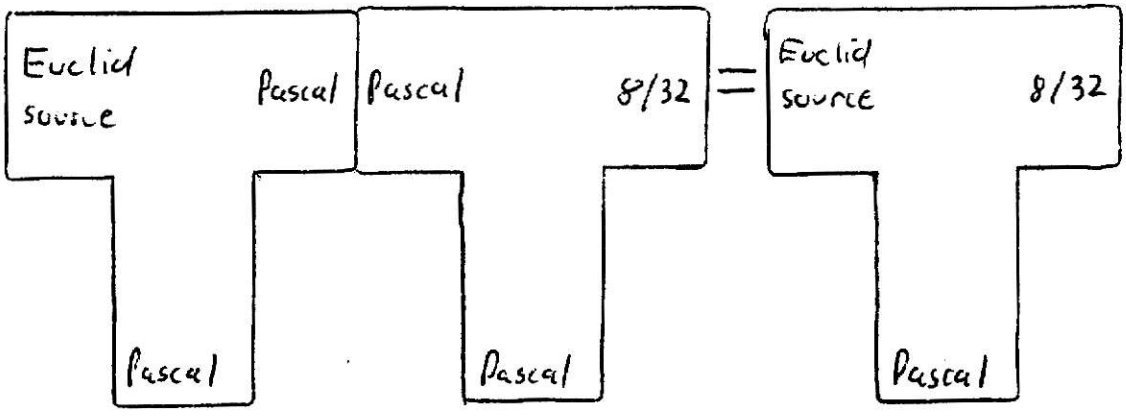


The Euclid compiler is then compiled by the Pascal compiler:



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 straight-forward 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 straight forward 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:

```
IFST --> if <expn> then {<stmt>}
         {elseIf <expn> then {<stmt>} }
         [else {stmt} ]
```

The corresponding output production is:

```
IFST --> IF <EXPN> THEN <STMTLIST>
         [ELSE IF <EXPN> THEN <STMTLIST> ]
         ELSE <STMTLIST>
```

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

The resulting pseudo code for this translation is:

```
OUTPUT CURRENT TOKEN "IF"
NEXT "consume IF"
CALL EXPN
OUTPUT CURRENT TOKEN "THEN"
NEXT "consume THEN"
CALL STMTLIST "which outputs BEGIN...END"
DO WHILE TOKEN = 'ELSEIF'
  OUTPUT 'ELSE  IF'
  NEXT "consume 'ELSEIF'"
  CALL EXPN
  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 straight forward 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 can not 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 DECLARAIONS 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
```

```
    (* END MODULE OUT *)
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:

```
const c := 4
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,

TYPE A = 1+2+3..8*10 would be translated to

```
CONST SYSID_1 = 1+2+3;
CONST SYSID_2 = 8*10;
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:

```
CONST <ID> = ( <LIST-OF-CONSTANTS> ) :
ARRAY [ <RANGE> ] OF <TYPEID>
```

But, the syntax for structured constants in Euclid is:

```
const <id> : array <range> of <type>
      ( <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 emits 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:

```
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:

```
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 DEFINTIONS". 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:

```
type a = (blue, green);
type b = (blue, green);
var aa : a;
    bb : b;
begin bb:=aa end
```

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.

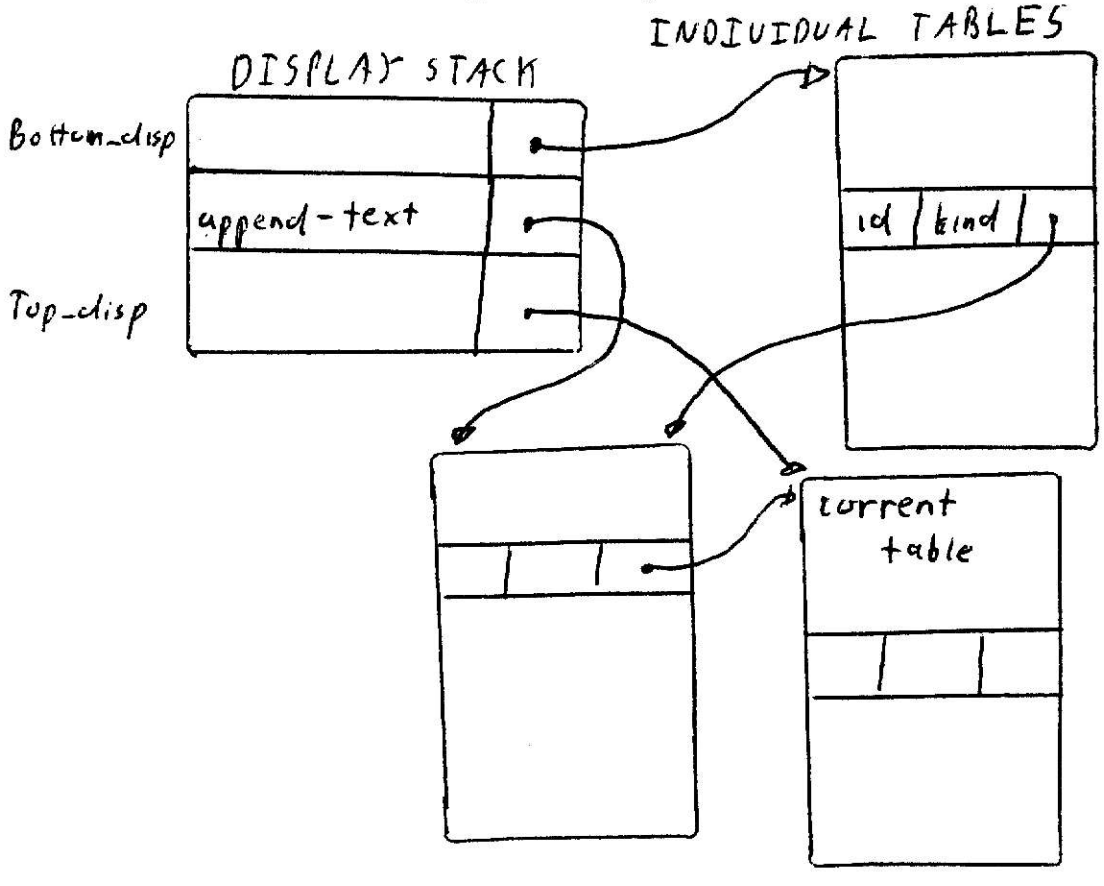
CHAPTER 6

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 accesible, 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 indeces 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 syncronization 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

1] Lampson, B.W., Horning, J.J., London, R.L., Mitchell, J.G., and Popek, G.J. "Report On The Programming Language Euclid", SIGPLAN Notices. vol 12-2 (Feb 1977)

2] Computer Systems Research Group of the University of Toronto, and Special Systems Division of I.P. Sharp Associates Limited. "APRIL EUCLID TRANSLATOR". Toronto, Canada, June 1979.

3] Wirth, N., and Jensen, K. Pascal User Manual and Report. 2nd ed, Springer-Verlag, 1975

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:
   range (RANGE) --> CONST SYS-GEN-ID = <expn>;
        CONST SYS-GEN-ID-2 = <expn>;
        TYPE SYS-GEN-ID-3 = SYS-GEN-ID .. SYS-GEN-ID-1;
      | 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:
   typeDeclaration (TYPE_DCL) --> [ (* pervasive *) ]
        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:
   typeDefinition (TYPE_DEF) --> TYPE SYS-GEN-ID =
        [ ARRAY [<range>] OF ]
           SYS-GEN-ID "This generated id is passed by typeDefinition
                       or variableType, depending on if there are
                       more than one dimension of arrays."
      | TYPE SYS-GEN-ID = RECORD
           <RECORD-VAR-DCL> {; <RECORD-VAR-DCL>}
           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) structuredConstantDcl --> [ pervasive ]
        const <id> : array <range> of
        variableType := ( <expn> { , <expn>} )

Pascal:
   structuredConstantDcl (CONST_DCL) --> [ (* pervasive *) ]
        CONST <APPENDED-ID> = STRUCTCON ( <expn> {, <expn>} )
        : 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-ID> = <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 --> <conjuntion> | <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>
```

```
    | WHILE TRUE DO <STATEMENT-LIST> ;
        <STATEMENT-LABEL> :
    | IF <expn> THEN <STATEMENT-LIST>
        { ELSE IF <expn>
              THEN <STATEMENT-LIST> }
        ELSE <STATEMENT-LIST>
    | CASE <expn> OF
        {LABEL {, LABEL} : <STATEMENT-LIST> }
        [ ELSE : <STATEMENT-LIST> ]
    | "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:

```
STATEMENT-LIST --> BEGIN { <statement> } END
```

```
17) routineDeclaration --> [ pervasive ]
        procedure <id> paramList =
            <routineDefinition>
      | [ pervasive ]
        function <id> paramList
           returns <id2> : variableType =
           <routineDefinition>

Pascal:
    routineDeclaration (PROC_DCL) --> [ (* pervasive *) ]
        PROCEDURE <APPENDED-ID> <paramList> ;
           <routineDefinition> END
      | [ (* pervasive *) ]
        FUNCTION <APPENDED-ID> <paramList> : <ONE-ID-OF-VAR-TYPE>
           (* RETURNS ID2 : *)
           VAR <id2> : <ONE-ID-OF-VAR-TYPE> ;
           <routineDefinition>
           1: <APPENDED-ID> := id2  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) paramList --> [ ( formalSection {, 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 : UNIV "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 initialiation 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

```
1
2
3     (***********************************************
4        APRIL EUCLID TO PASCAL TRANSLATOR
5        DAVID ROESENER, SUMMER 1980
6
7        THIS PROGRAM TRANSLATES APRIL EUCLID (A SUBSET
8        OF PASCAL) INTO K.S.U. INTERDATA 8/32 PASCAL.
9
10
11       DR. ROD BATES IS RESPONSIBLE FOR THE TABLE LOOK UP
12       OF IDENTIFIERS IN THE LEXICAL SCANNER. HE IS
13       ALSO RESPONSIBLE FOR THE PROGRAM "PASCAL PRETTY
14       PRINTER" THAT FORMATTED THIS SOURCE LISTING.
15
16
17       HE IS ALSO RESPONSIBLE FOR THE FOLLOWING CHANGES TO
18       THE ORIGINAL VERSION:
19       1. SHORTEN SYSTEM ID-S TO 10 CHARS SO XREF CAN DISTINGUISH
20             THEM
21       2. PRINT UNRECOGNIZED CHARACTERS EXACTLY AND IN HEXADECIMAL
22       3. PRINT ERROR MESSAGE COMMENTS AT THE FAR LEFT, WHILE
23          PUTTING EVERYTHING ELSE IN 5 SPACES
24       4. CHANGE EUCLID RESERVED WORDS TO ALL CAPS IN THE OUTPUT
25       5. CHANGE CONVERTER DECLARATIONS TO COMMENTS AND DELETE
26          CONVERTER CALLS (LEAVING ONLY THEIR PARAMETERS)
27       6. REMOVE 'PACKED' FROM TYPE DEFINITIONS
28       7. REMOVE 'INLINE' FROM PROCEDURE DECLARATIONS
29       8. REMOVE 'CHECKED' AND 'NOT CHECKED'
30       9. DECLARE ALL FORMAL PARAMETERS 'UNIV'
31      10. PASS CODE BODIES UNMODIFIED, EXCEPT FOR REMOVAL OF '?'
32      11. APPEND PREFIX (COPY FROM LU 3)
33      12. ALTER PASCAL STRUCTURED CONSTANT SYNTAX
34      13. DELETE EXTRA '+' IN EXPRESSIONS
35      14. FIX DISJUNCTION TO CALL CONJUNCTION AGAIN
36      15. FIX MODULE_TYPE TO SKIP IMPORT LISTS CONTAINING 'VAR'
37      16. FIX TO OUTPUT ENTIRE LITERAL STRINGS
38
39     ***********************************************)
40    "ROBERT YOUNG"
41    "KANSAS STATE UNIVERSITY"
42    "DEPARTMENT OF COMPUTER SCIENCE"
43
44      CONST COPYRIGHT = 'COPYRIGHT ROBERT YOUNG 1978'
45
46    "###########
47    #  PREFIX  #
48    ###########"
49
50    ; CONST NL = '(:10:)'
51      ; FF = '(:12:)'
52      ; CR = '(:13:)'
```

```
53      ; EM = '(:25:)'
54
55    ; CONST PAGELENGTH = 512
56
57    ; TYPE PAGE = ARRAY (. 1 .. PAGELENGTH .) OF CHAR
58
59    ; CONST LINELENGTH = 132
60
61    ; TYPE LINE = ARRAY (. 1 .. LINELENGTH .) OF CHAR
62
63    ; CONST IDLENGTH = 12
64
65    ; TYPE IDENTIFIER = ARRAY (. 1 .. IDLENGTH .) OF CHAR
66
67    ; TYPE FILE = 1 .. 2
68
69    ; TYPE FILEKIND = ( EMPTY , SCRATCH , ASCII , SEQCODE
70                       , CONCODE )
71
72    ; TYPE FILEATTR
73        = RECORD
74            KIND : FILEKIND
75          ; ADDR : INTEGER
76          ; PROTECTED : BOOLEAN
77          ; NOTUSED : ARRAY (. 1 .. 5 .) OF INTEGER
78          END
79
80    ; TYPE IODEVICE
81        = ( TYPEDEVICE , DISKDEVICE , TAPEDEVICE , PRINTDEVICE
82          , CARDDEVICE )
83
84    ; TYPE IOOPERATION = ( INPUT , OUTPUT , MOVE , CONTROL )
85
86    ; TYPE IOARG = ( WRITEEOF , REWIND , UPSPACE , BACKSPACE )
87
88    ; TYPE IORESULT
89        = ( COMPLETE , INTERVENTION , TRANSMISSION , FAILURE
90          , ENDFILE , ENDMEDIUM , STARTMEDIUM )
91
92    ; TYPE IOPARAM
93        = RECORD
94            OPERATION : IOOPERATION
95          ; STATUS : IORESULT
96          ; ARG : IOARG
97          END
98
99    ; TYPE TASKKIND = ( INPUTTASK , JOBTASK , OUTPUTTASK )
100
101   ; TYPE ARGTAG = ( NILTYPE , BOOLTYPE , INTTYPE , IDTYPE
102                   , PTRTYPE )
103
104   ; TYPE POINTER = @ BOOLEAN
105
106   ; TYPE PASSPTR = @ PASSLINK
107
108   ; TYPE PASSLINK
109       = RECORD
110           OPTIONS : SET OF CHAR
```

```
111          ; FILLER1 : ARRAY (. 1 .. 7 .) OF INTEGER
112          ; FILLER2 : BOOLEAN
113          ; RESET_POINT : INTEGER
114          ; FILLER3 : ARRAY (. 1 .. 11 .) OF POINTER
115          END
116
117    ; TYPE ARGTYPE
118        = RECORD
119            CASE TAG : ARGTAG
120            OF NILTYPE , BOOLTYPE : ( BOOL : BOOLEAN )
121            ; INTTYPE : ( INT : INTEGER )
122            ; IDTYPE : ( ID : IDENTIFIER )
123            ; PTRTYPE : ( PTR : PASSPTR )
124          END
125
126    ; CONST MAXARG = 10
127
128    ; TYPE ARGLIST = ARRAY (. 1 .. MAXARG .) OF ARGTYPE
129
130    ; TYPE ARGSEQ = ( INP , OUT )
131
132    ; TYPE PROGRESULT
133        = ( TERMINATED , OVERFLOW , POINTERERROR , RANGEERROR
134          , VARIANTERROR , HEAPLIMIT , STACKLIMIT , CODELIMIT
135          , TIMELIMIT , CALLERROR )
136
137    ; PROCEDURE READ ( VAR C : CHAR )
138
139    ; PROCEDURE WRITE ( C : CHAR )
140
141    ; PROCEDURE OPEN ( F : FILE ; ID : IDENTIFIER
142                       ; VAR FOUND : BOOLEAN )
143
144    ; PROCEDURE CLOSE ( F : FILE )
145
146    ; PROCEDURE GET ( F : FILE ; P : INTEGER
147                    ; VAR BLOCK : UNIV PAGE )
148
149    ; PROCEDURE PUT ( F : FILE ; P : INTEGER
150                    ; VAR BLOCK : UNIV PAGE )
151
152    ; FUNCTION LENGTH ( F : FILE ) : INTEGER
153
154    ; PROCEDURE MARK ( VAR TOP : INTEGER )
155
156    ; PROCEDURE RELEASE ( TOP : INTEGER )
157
158    ; PROCEDURE IDENTIFY ( HEADER : LINE )
159
160    ; PROCEDURE ACCEPT ( VAR C : CHAR )
161
162    ; PROCEDURE DISPLAY ( C : CHAR )
163
164    ; PROCEDURE NOTUSED
165
166    ; PROCEDURE NOTUSED2
167
168    ; PROCEDURE NOTUSED3
```

```
169
170     ; PROCEDURE NOTUSED4
171
172     ; PROCEDURE NOTUSED5
173
174     ; PROCEDURE NOTUSED6
175
176     ; PROCEDURE NOTUSED7
177
178     ; PROCEDURE NOTUSED8
179
180     ; PROCEDURE NOTUUED9
181
182     ; PROCEDURE NOTUSED10
183
184     ; PROCEDURE RUN
185         ( ID : IDENTIFIER
186         ; VAR PARAM : ARGLIST
187         ; VAR LINE : INTEGER
188         ; VAR RESULT : PROGRESULT
189         )
190
191     ; PROGRAM P ( PARAM : LINE )
192
193       ; TYPE BUFFER = ARRAY [ 1 .. 70 ] OF CHAR
194
195       ; CONST RWMAX = 80
196
197       ; TYPE STRING10 = ARRAY [ 1 .. 10 ] OF CHAR
198         ; SYMBOLTYP
199           = ( IDENTSY , CONSTANTSY , CONSTSY , EQUALSSY , ANDSY
200             , RELOPSY
201             , ADDOPSY , ORSY , MULOPSY , ELIPSISSY , LPARSY
202             , RPARSY
203             , NOTSY , COMMASY , DOTSY , CASESY , COLONSY , OFSY
204             , ENDSY
205             , ARRAYSY , BEGINSY , TYPESY , PROCSY , FUNCSY , VARSY
206             , BECOMESSY , IFSY , THENSY , ELSESY , RECORDSY
207             , INITSY
208             , EOFSY , LITSTRSY , IMPLYSY , CASESEPSY , ASSERTSY
209             , CODESY
210             , ELSEIFSY , EXITSY , EXPORTSSY , IMPORTSSY
211             , INCLUDESY
212             , LOOPSY , MACHINESY , MODULESY , OTHERWISESY
213             , PERVASIVESY
214             , POSTSY , PRESY , READONLYSY , RETURNSY , RETURNSSY
215             , WHENSY
216             , BOOLSY , CHARSY , SIGINTSY , CONVERTERSY , PACKEDSY
217             , INLINESY , CHECKEDSY )
218
219       ; CONST BLANKS
220           =
221
'                                                                       '
222
223       ; TYPE KINDOFID
224           = ( PROCID , MODULEID , TYPEID , CONVERTERID , OTHERID )
225
```

```
226       ; TYPE PTR_TO_SYM_TAB = ^ SYM_TABLE
227         ; DISPLAY_PTR = ^ DSPLAY
228
229       (* THIS TYPE IS OF A STACK-LIKE STRUCTURE WHOSE
230          ELEMENTS CONTAIN POINTERS TO SYMBOL TABLES
231          AND THE TEXT THAT SHOULD BE APPENDED TO THE
232          IDENTIFIERS WHEN PROCESSED. *)
233       ; TYPE DSPLAY
234           = RECORD
235               EARLIER_PTR : DISPLAY_PTR
236             ; APPEND_ID : BUFFER
237             ; LATER_PTR : DISPLAY_PTR
238             ; SYM_TAB_PTR : PTR_TO_SYM_TAB
239             END
240
241       (* THIS IS THE TYPE OF THE NODE OF THE SYMBOL TABLE
242          IN BINARY TREE FORM. *)
243       ; TYPE SYM_TABLE
244           = RECORD
245               LEFT_SON , RT_SON : PTR_TO_SYM_TAB
246             ; ID : BUFFER
247             ; TYPE_OF_ID : KINDOFID
248             ; ITS_OWN_TAB : PTR_TO_SYM_TAB
249             ; BACK_TO_DISP : DISPLAY_PTR
250             END
251
252       ; CONST STACK_MARKER = NIL
253
254       ; TYPE PTR_TO_STACK = ^ STACK
255
256       (* THE STACK IS USED FOR "COMPILE TIME" VARIABLE
257          INITIALIZATIONS TO BE REALLY INITIALIZED AT
258          RUN-TIME INSTEAD. *)
259       ; TYPE STACK
260           = RECORD
261               PTR_TO_ID : PTR_TO_SYM_TAB
262             ; INIT_VAL : BUFFER
263             ; NEXT_PTR : PTR_TO_STACK
264             END
265
266       (* THIS DATA STRUCTURE JUST KEEPS ALL THE NAMES
267          OF THE MODULE INITIALIZATION ROUTINES, TO BE
268          OUTPUT AS THE MAIN BODY OF THE PASCAL OBJECT. *)
269       ; TYPE INIT_PTR = ^ PROC_INIT_QUEUE
270
271       ; TYPE PROC_INIT_QUEUE
272           = RECORD
273               INIT_ID : BUFFER
274             ; NEXT_QUEUE : INIT_PTR
275             END
276
277       ; VAR TOP_OF_DISP : DISPLAY_PTR
278         ; BOTTOM_OF_DISP : DISPLAY_PTR
279         ; PTR_D : DISPLAY_PTR
280         ; PTR_T : PTR_TO_SYM_TAB
281         ; FREE_STACK : PTR_TO_STACK
282         ; FREE_DISP : DISPLAY_PTR
283         ; FREE_TABLE : PTR_TO_SYM_TAB
```

```
284         ; GARBAGE : BOOLEAN
285         ; STACK_TOP , PTR_S : PTR_TO_STACK
286         ; FRONT_OF_QUEUE , ADD_ON_QUEUE : INIT_PTR
287         ; SYSTEM_ID : BUFFER
288         ; NUMB_OUTPUT : INTEGER
289         ; INPUT_CHAR : CHAR
290         ; TEXT : BUFFER
291         ; TOK : SYMBOLTYP
292         ; FILL_UP_PTR : INTEGER
293         ; I : INTEGER
294         ; LETTER_DIG , DIGITS : SET OF CHAR
295         ; NUMB_ERR : INTEGER
296         ; INSIDE_COMMENT : BOOLEAN
297         ; QFIRSTSTMT , QCASE , QDCL , QFIRSTMODDCL , QGOODRECORD
298           : SET OF SYMBOLTYP
299         ; RWPTR : ARRAY [ 1 .. 10 ] OF 1 .. RWMAX
300         (* THIS ARRAY IS USED  IN THE LEXICAL SCANNER TO
301            HOLD ALL THE EUCLID KEYWORDS AND THEIR TOKEN
302            SYMBOL: *)
303         ; RW
304           : ARRAY [ 1 .. RWMAX ]
305             OF RECORD
306                  STRING : STRING10
307                ; SYMBOL : SYMBOLTYP
308                END
309         ; RWCT : INTEGER
310
311       ; PROCEDURE BREAKPNT
312         ; EXTERN
313
314       ; PROCEDURE NLINDENT
315       (* "NEW LINE INDENT" *)
316
317         ; VAR I : INTEGER
318
319         ; BEGIN
320             WRITE ( NL )
321           ; FOR I := 1 TO 5 DO WRITE ( ' ' )
322           ; NUMB_OUTPUT := 5
323           END
324
325       ; PROCEDURE OUT_WOUT_BL_LIT ( PRINT : BUFFER )
326       (* "OUT WITHOUT OUTPUTTING A BLANK THE LITERAL" *)
327
328         ; VAR I : INTEGER
329
330         ; BEGIN
331             I := 1
332           ; WHILE PRINT [ I ] <> '&'
333             DO BEGIN
334               WRITE ( PRINT [ I ] )
335             ; NUMB_OUTPUT := SUCC ( NUMB_OUTPUT )
336             ; I := I + 1
337             END
338           END
339
340       ; PROCEDURE OUT_LITERALLY ( OUT : BUFFER )
341
```

```
342         ; BEGIN
343             WRITE ( ' ' )
344           ; NUMB_OUTPUT := SUCC ( NUMB_OUTPUT )
345           ; IF NUMB_OUTPUT >= 55
346             THEN NLINDENT
347           ; OUT_WOUT_BL_LIT ( OUT )
348           END
349
350       ; PROCEDURE OUT_WOUT_BL ( OUTPUT_STR : BUFFER )
351       (* "OUT WITHOUT A BLANK THE INPUT TEXT" *)
352
353         ; VAR I : INTEGER
354
355         ; BEGIN
356             I := 1
357           ; WHILE OUTPUT_STR [ I ] <> ' '
358             DO BEGIN
359               WRITE ( OUTPUT_STR [ I ] )
360             ; I := SUCC ( I )
361             ; NUMB_OUTPUT := SUCC ( NUMB_OUTPUT )
362             END
363           END
364
365       ; PROCEDURE OUT_AFTER_BL ( OUT_STR : BUFFER )
366
367         ; BEGIN
368             NUMB_OUTPUT := SUCC ( NUMB_OUTPUT )
369           ; WRITE ( ' ' )
370           ; IF NUMB_OUTPUT >= 55
371             THEN NLINDENT
372           ; OUT_WOUT_BL ( OUT_STR )
373           END
374
375       ; PROCEDURE OUT_ERROR ( OUT : BUFFER )
376
377         ; BEGIN
378             WRITE ( NL )
379           ; NUMB_OUTPUT := 0
380           ; OUT_LITERALLY ( OUT )
381           END
382
383       ; PROCEDURE OUT_UNREC_CH ( CH : CHAR )
384       (* "OUT UNRECOGNIZED CHARACTER" *)
385
386         ; VAR HEXDIGITS : ARRAY [ 1 .. 2 ] OF INTEGER
387           ; I : INTEGER
388
389         ; BEGIN
390             IF INSIDE_COMMENT
391             THEN OUT_ERROR ( '** UNRECOGNIZED CHARACTER **&' )
392             ELSE OUT_ERROR ( '(* UNRECOGNIZED CHARACTER **&' )
393           ; WRITE ( CH )
394           ; OUT_LITERALLY ( '** CODE &' )
395           ; HEXDIGITS [ 1 ] := ORD ( CH ) DIV 16
396           ; HEXDIGITS [ 2 ] := ORD ( CH ) MOD 16
397           ; FOR I := 1 TO 2
398             DO IF HEXDIGITS [ I ] < 10
399               THEN WRITE ( CHR ( HEXDIGITS [ I ] + ORD ( '0' ) ) )
```

```
400             ELSE WRITE ( CHR ( HEXDIGITS [ I ] - 10
401                                + ORD ( 'A' ) ) )
402           ; IF INSIDE_COMMENT
403             THEN OUT_LITERALLY ( ' **&' )
404             ELSE OUT_LITERALLY ( ' *)&' )
405           END
406
407       ; PROCEDURE NEXT
408         ; FORWARD
409
410       ; PROCEDURE OUT_AND_GET
411       (* "OUTPUT THIS TOKEN AND GET NEXT ONE" *)
412
413         ; VAR I : INTEGER
414
415         ; BEGIN
416             IF TOK = LITSTRSY
417             THEN BEGIN
418               WRITE ( ' ' )
419             ; IF NUMB_OUTPUT + FILL_UP_PTR >= 72
420               THEN NLINDENT
421             ; FOR I := 1 TO FILL_UP_PTR DO WRITE ( TEXT [ I ] )
422             END
423             ELSE OUT_AFTER_BL ( TEXT )
424           ; NEXT
425           END
426
427       ; PROCEDURE OUT_TOK_AS_COMM
428       (* "OUTPUT THIS TOKEN AS A COMMENT AND GET NEXT ONE" *)
429
430         ; BEGIN
431             OUT_LITERALLY ( '(* &' )
432           ; OUT_AFTER_BL ( TEXT )
433           ; OUT_LITERALLY ( '*) &' )
434           ; NEXT
435           ;
436           END
437
438       ; PROCEDURE POP
439           ( VAR TOP_ID_PTR : PTR_TO_SYM_TAB
440             ; VAR INIT_VAL : BUFFER )
441
442         ; VAR TEMP_PTR : PTR_TO_STACK
443
444         ; BEGIN
445             TOP_ID_PTR := STACK_TOP ^ . PTR_TO_ID
446           ; INIT_VAL := STACK_TOP ^ . INIT_VAL
447           ; TEMP_PTR := STACK_TOP
448           ; STACK_TOP := STACK_TOP ^ . NEXT_PTR
449           ; TEMP_PTR ^ . NEXT_PTR := FREE_STACK
450           ; FREE_STACK := TEMP_PTR
451           END
452
453       ; PROCEDURE PUSH ( NEW_ID_PTR : PTR_TO_SYM_TAB
454                        ; INIT_VAL : BUFFER )
455
456         ; BEGIN
457             IF FREE_STACK = NIL
```

```
458           THEN NEW ( PTR_S )
459           ELSE BEGIN
460             PTR_S := FREE_STACK
461           ; FREE_STACK := FREE_STACK ^ . NEXT_PTR
462           END
463         ; PTR_S ^ . PTR_TO_ID := NEW_ID_PTR
464         ; PTR_S ^ . INIT_VAL := INIT_VAL
465         ; PTR_S ^ . NEXT_PTR := STACK_TOP
466         ; STACK_TOP := PTR_S
467         END
468
469     ; PROCEDURE MARK_STACK
470
471       ; BEGIN PUSH ( STACK_MARKER , BLANKS ) END
472
473     (* THIS PROCEDURE STORES THE NAME OF ALL MODULE
474        INITIALIZATION PROCEDURES, SO THEY CAN BE DUMPED
475        AS THE MAIN BODY OF THE PASCAL OBJECT: *)
476     ; PROCEDURE QUEUE_INIT_PROC ( ID_INIT : BUFFER )
477
478       ; VAR PTR_Q : INIT_PTR
479
480       ; BEGIN
481           NEW ( PTR_Q )
482         ; ADD_ON_QUEUE ^ . NEXT_QUEUE := PTR_Q
483         ; PTR_Q ^ . INIT_ID := ID_INIT
484         ; PTR_Q ^ . NEXT_QUEUE := NIL
485         ; ADD_ON_QUEUE := PTR_Q
486         END
487
488     (* THIS PROC GENERATES A UNIQUE IDENTIFIER: *)
489     ; PROCEDURE GET_SYS_ID ( VAR OUT_SYS_ID : BUFFER )
490
491       ; VAR I : INTEGER
492
493       ; BEGIN
494           I := 10
495         ; SYSTEM_ID [ I ] := SUCC ( SYSTEM_ID [ I ] )
496         ; WHILE SYSTEM_ID [ I ] > '9'
497           DO BEGIN
498             SYSTEM_ID [ I ] := '0'
499           ; I := I - 1
500           ; SYSTEM_ID [ I ] := SUCC ( SYSTEM_ID [ I ] )
501           END
502         ; OUT_SYS_ID := SYSTEM_ID
503         END
504
505     ; PROCEDURE TRAVERSE
506         ( WHAT_LOOK_FOR : BUFFER ; VAR WALKER : PTR_TO_SYM_TAB )
507       ; FORWARD
508
509     (* THIS PROCEDURE SHOULD BE THE CALLED THE FIRST TIME
510        WITH "WALKER" POINTING TO THE ROOT OF A SYMBOL
511        TABLE.
512        IF THE DESIRED ID ("WHAT LOOK FOR") IS IN THE
513        SYMBOL TABLE, "WALKER" WILL POINT TO IT.
514        IF THE DESIRED ID IS NOT IN THE SYMBOL TABLE,
515        "WALKER" WILL POINT TO THE NODE THAT THE ID SHOULD
```

```
516          BE INSERTED AFTER. IT IS UP TO THE CALLING PROC
517          TO FIND OUT WHICH WAY (LEFT OR RIGHT) IF IT
518          WANTS TO ADD THE DESIRED ID TO THIS TABLE: *)
519        ; PROCEDURE TRAVERSE
520
521          ; BEGIN
522              IF ( WHAT_LOOK_FOR > WALKER ^ . ID )
523                 AND ( WALKER ^ . RT_SON <> NIL )
524              THEN BEGIN
525                WALKER := WALKER ^ . RT_SON
526              ; TRAVERSE ( WHAT_LOOK_FOR , WALKER )
527              END
528              ELSE IF ( WHAT_LOOK_FOR < WALKER ^ . ID )
529                      AND ( WALKER ^ . LEFT_SON <> NIL )
530                THEN BEGIN
531                  WALKER := WALKER ^ . LEFT_SON
532                ; TRAVERSE ( WHAT_LOOK_FOR , WALKER )
533                END
534           END
535
536        (* THIS PROCEDURE RETURNS A SYMBOL TABLE TO THE
537           FREE LIST WHEN ITS SCOPE IS LEFT,A AND ITS ID'S
538           ARE NO LONGER NEEDED: *)
539        ; PROCEDURE FREE_UP_TREE ( VAR PTR : PTR_TO_SYM_TAB )
540          ; FORWARD
541
542        ; PROCEDURE FREE_UP_TREE
543
544          ; BEGIN
545              IF PTR ^ . LEFT_SON <> NIL
546              THEN FREE_UP_TREE ( PTR ^ . LEFT_SON )
547            ; IF PTR ^ . RT_SON <> NIL
548              THEN FREE_UP_TREE ( PTR ^ . RT_SON )
549            ; PTR ^ . RT_SON := FREE_TABLE
550            ; FREE_TABLE := PTR
551            END
552
553        ; PROCEDURE START_NEW_TAB ( APP_TEXT : BUFFER )
554
555          ; BEGIN
556              IF FREE_DISP = NIL
557              THEN NEW ( PTR_D )
558              ELSE BEGIN
559                PTR_D := FREE_DISP
560              ; FREE_DISP := FREE_DISP ^ . LATER_PTR
561              END
562            ; PTR_D ^ . EARLIER_PTR := TOP_OF_DISP
563            ; PTR_D ^ . LATER_PTR := NIL
564            ; PTR_D ^ . APPEND_ID := APP_TEXT
565            ; TOP_OF_DISP ^ . LATER_PTR := PTR_D
566            ; TOP_OF_DISP := PTR_D
567            ; IF FREE_TABLE = NIL
568              THEN NEW ( PTR_T )
569              ELSE BEGIN
570                PTR_T := FREE_TABLE
571              ; FREE_TABLE := FREE_TABLE ^ . RT_SON
572              END
573            ; TOP_OF_DISP ^ . SYM_TAB_PTR := PTR_T
```

```
574           ; PTR_T ^ . ID := BLANKS
575           ; PTR_T ^ . LEFT_SON := NIL
576           ; PTR_T ^ . RT_SON := NIL
577           ; PTR_T ^ . BACK_TO_DISP := TOP_OF_DISP
578           END
579
580       ; PROCEDURE GET_RID_OF_TAB ( FREE_UP_TABLE : BOOLEAN )
581
582         ; VAR TEMP_DISP : DISPLAY_PTR
583
584         ; BEGIN
585             TEMP_DISP := TOP_OF_DISP
586           ; IF FREE_UP_TABLE
587             THEN FREE_UP_TREE ( TEMP_DISP ^ . SYM_TAB_PTR )
588           ; TOP_OF_DISP := TOP_OF_DISP ^ . EARLIER_PTR
589           ; TOP_OF_DISP ^ . LATER_PTR := NIL
590           ; TEMP_DISP ^ . LATER_PTR := FREE_DISP
591           ; FREE_DISP := TEMP_DISP
592           END
593
594       (* THIS PROCEDURE ADDS THE CURRENT ID TO THE CURRENT
595          SYMBOL TABLE. IT THEN PASSES BACK A POINTER
596          TO THE ID. *)
597       ; PROCEDURE ADD_TO_SYM_TAB
598           ( WHICH_ID : BUFFER
599           ; WHAT_KIND : KINDOFID
600           ; VAR PTR : PTR_TO_SYM_TAB
601           )
602
603         ; VAR WALKER : PTR_TO_SYM_TAB
604
605         ; BEGIN
606             IF FREE_TABLE = NIL
607             THEN NEW ( PTR )
608             ELSE BEGIN
609               PTR := FREE_TABLE
610             ; FREE_TABLE := FREE_TABLE ^ . RT_SON
611             END
612           ; WITH PTR ^
613             DO BEGIN
614               LEFT_SON := NIL
615             ; RT_SON := NIL
616             ; TYPE_OF_ID := WHAT_KIND
617             ; ID := WHICH_ID
618             ; BACK_TO_DISP := TOP_OF_DISP
619             END
620           ; WALKER := TOP_OF_DISP ^ . SYM_TAB_PTR
621           ; TRAVERSE ( WHICH_ID , WALKER )
622           ; IF WHICH_ID > WALKER ^ . ID
623             THEN WALKER ^ . RT_SON := PTR
624             ELSE WALKER ^ . LEFT_SON := PTR
625           END
626
627       (* IN THIS PROC, WHICH_ID SHOULD POINT TO THE ROOT NODE
628          OF THE SYMBOL TABLE TO BE SEARCHED. IF THE ID
629          IS NOT CONTAINED IN THIS TABLE, WHICH_ID THEN
630          HAS NO SIGNIFICANCE. *)
631       ; PROCEDURE SEARCH_SYM_TAB
```

```
632           ( WHICH_ID : BUFFER
633           ; VAR PTR : PTR_TO_SYM_TAB
634           ; VAR NOT_LOCATED : BOOLEAN
635           )
636
637         ; BEGIN
638             TRAVERSE ( WHICH_ID , PTR )
639           ; IF PTR ^ . ID = WHICH_ID
640             THEN NOT_LOCATED := FALSE
641             ELSE NOT_LOCATED := TRUE
642           END
643
644       ; PROCEDURE SEARCH_FOR_ID
645           ( WHICH_ID : BUFFER ; VAR PTR : PTR_TO_SYM_TAB )
646
647         ; VAR OUT_WALKER : DISPLAY_PTR
648           ; ID_NOT_FOUND : BOOLEAN
649
650         ; BEGIN
651             OUT_WALKER := TOP_OF_DISP
652           ; ID_NOT_FOUND := TRUE
653           ; WHILE ( OUT_WALKER <> NIL ) AND ( ID_NOT_FOUND )
654             DO BEGIN
655               PTR := OUT_WALKER ^ . SYM_TAB_PTR
656             ; SEARCH_SYM_TAB ( WHICH_ID , PTR , ID_NOT_FOUND )
657             ; OUT_WALKER := OUT_WALKER ^ . EARLIER_PTR
658             END
659           ; IF ID_NOT_FOUND
660             THEN PTR := NIL
661           END
662
663       (* GIVEN A PTR TO A PARTICULAR ID, THIS PROC OUTPUTS
664          THAT ID WITH ALL APPROPRIATE APPEND TEXTS
665          APPENDED TO IT. *)
666       ; PROCEDURE APPEND ( PTR : PTR_TO_SYM_TAB )
667
668         ; VAR WALKER : DISPLAY_PTR
669
670         ; BEGIN
671             IF PTR ^ . BACK_TO_DISP ^ . APPEND_ID = BLANKS
672             THEN OUT_AFTER_BL ( PTR ^ . ID )
673             ELSE BEGIN
674               WALKER := BOTTOM_OF_DISP
675             ; OUT_AFTER_BL ( WALKER ^ . APPEND_ID )
676             ; WHILE WALKER <> PTR ^ . BACK_TO_DISP
677               DO BEGIN
678                 WALKER := WALKER ^ . LATER_PTR
679               ; OUT_WOUT_BL ( WALKER ^ . APPEND_ID )
680               ; OUT_WOUT_BL_LIT ( '_&' )
681               ;
682               END
683             ; OUT_WOUT_BL ( PTR ^ . ID )
684             END
685           END
686
687       (* THIS CONVERTS THE CURRENT "CONGLOMERATE ID" (OF THE
688          FORM "ID.ID.ID") INTO THE FORM "ID_ID_ID.ID".
689          I.E., IT APPENDS THE FIRST ID, THEN CHANGES ALL THE
```

```
690          APPROPRIATE "." TO "_" FOR ALL THE SUCCEEDING MODULE
691          IDS. IT ALSO RETURNS THE TYPE OF THE CONGLOMERATE ID,
692          WHICH IS THE TYPE OF THE ID THAT IS THE FIRST ONE AFTER
693          THE LAST "_" IS OUTPUT. *)
694        ; PROCEDURE OUT_FIXED_ID ( VAR WHAT_KIND : KINDOFID )
695
696          ; VAR WHERE : PTR_TO_SYM_TAB
697            ; GARBAGE : BOOLEAN
698
699          ; BEGIN
700              SEARCH_FOR_ID ( TEXT , WHERE )
701            ; IF WHERE = NIL
702              THEN BEGIN
703                OUT_ERROR (
704                '(* ID NOT DECLARED YET- RECURSIVE PROC? *) &' )
705              ; NUMB_ERR := SUCC ( NUMB_ERR )
706              ; WHAT_KIND := PROCID
707              ; OUT_AND_GET
708              END
709              ELSE BEGIN
710                APPEND ( WHERE )
711              ; WHILE WHERE ^ . TYPE_OF_ID = MODULEID
712              (* THE ID IS A MODULE, SO WE KNOW THAT "." AND
713                 ANOTHER ID FOLLOW AND THE "." NEEDS TO BE
714                 CHANGED TO "_" *)
715                DO BEGIN
716                  NEXT
717                ; OUT_WOUT_BL_LIT ( '_&' )
718                ; NEXT
719                ; OUT_WOUT_BL ( TEXT )
720                ; WHERE := WHERE ^ . ITS_OWN_TAB
721                ; SEARCH_SYM_TAB ( TEXT , WHERE , GARBAGE )
722                ; IF GARBAGE
723                  THEN BEGIN
724                    OUT_ERROR ( '(* LOGIC ERROR - APPEND *) &' )
725                  ; NUMB_ERR := SUCC ( NUMB_ERR )
726                  END
727                END
728              ; WHAT_KIND := WHERE ^ . TYPE_OF_ID
729              (* FIRST ID AFTER ALL MODULES DETERMINES THE KIND. *)
730              ; NEXT
731              END
732            ; WHILE TOK = DOTSY
733             (* WE KNOW THAT ALL "." AT THIS TIME DENOTE RECORD
734                COMPONENTS. *)
735            ; WHILE TOK = DOTSY
736              DO BEGIN
737                OUT_WOUT_BL ( TEXT )
738              ; NEXT
739              ; OUT_WOUT_BL ( TEXT )
740              ; NEXT
741              END
742            END
743
744         (* THIS IS DR. BATES' PART. THIS PROC ADDS A KEYWORD
745            TO THE KEY WORD LIST. *)
746        ; PROCEDURE INITRW ( FSTRING : STRING10
747                           ; FSYMBOL : SYMBOLTYP )
```

```
748
749       ; VAR LEN , I , J : INTEGER
750
751       ; BEGIN
752           IF RWCT < RWMAX
753           THEN BEGIN
754             LEN := 10
755           ; WHILE ( LEN >= 2 ) AND ( FSTRING [ LEN ] = ' ' )
756             DO LEN := PRED ( LEN )
757           ; I := RWPTR [ LEN ]
758           ; J := RWPTR [ LEN + 1 ]
759           ; WHILE ( I < J ) AND ( RW [ I ] . STRING < FSTRING )
760             DO I := SUCC ( I )
761           ; IF RW [ I ] . STRING <> FSTRING
762             THEN BEGIN
763               J := RWPTR [ 10 ] - 1
764             ; WHILE J >= I
765               DO BEGIN
766                 RW [ J + 1 ] := RW [ J ]
767               ; J := PRED ( J )
768               END
769             ; RW [ I ] . STRING := FSTRING
770             ; FOR J := LEN + 1 TO 10
771               DO RWPTR [ J ] := SUCC ( RWPTR [ J ] )
772             END
773           ; RW [ I ] . SYMBOL := FSYMBOL
774           ; RWCT := SUCC ( RWCT )
775           END
776         END (* INITRW *)
777
778     (* BUILD KEY WORD LIST *)
779     ; PROCEDURE INITRWTABLE
780
781       ; BEGIN
782           RWCT := 0
783         ; FOR I := 1 TO 10 DO RWPTR [ I ] := 1
784         ; INITRW ( 'if         ' , IFSY )
785         ; INITRW ( 'of         ' , OFSY )
786         ; INITRW ( 'or         ' , ORSY )
787         ; INITRW ( 'and        ' , ANDSY )
788         ; INITRW ( 'div        ' , MULOPSY )
789         ; INITRW ( 'end        ' , ENDSY )
790         ; INITRW ( 'mod        ' , MULOPSY )
791         ; INITRW ( 'not        ' , NOTSY )
792         ; INITRW ( 'var        ' , VARSY )
793         ; INITRW ( 'pre        ' , PRESY )
794         ; INITRW ( 'case       ' , CASESY )
795         ; INITRW ( 'else       ' , ELSESY )
796         ; INITRW ( 'initially  ' , INITSY )
797         ; INITRW ( 'then       ' , THENSY )
798         ; INITRW ( 'type       ' , TYPESY )
799         ; INITRW ( 'code       ' , CODESY )
800         ; INITRW ( 'exit       ' , EXITSY )
801         ; INITRW ( 'loop       ' , LOOPSY )
802         ; INITRW ( 'post       ' , POSTSY )
803         ; INITRW ( 'when       ' , WHENSY )
804         ; INITRW ( 'Char       ' , CHARSY )
805         ; INITRW ( 'array      ' , ARRAYSY )
```

```
806         ; INITRW ( 'begin     ' , BEGINSY )
807         ; INITRW ( 'const     ' , CONSTSY )
808         ; INITRW ( 'record    ' , RECORDSY )
809         ; INITRW ( 'assert    ' , ASSERTSY )
810         ; INITRW ( 'elseif    ' , ELSEIFSY )
811         ; INITRW ( 'inline    ' , INLINESY )
812         ; INITRW ( 'module    ' , MODULESY )
813         ; INITRW ( 'packed    ' , PACKEDSY )
814         ; INITRW ( 'return    ' , RETURNSY )
815         ; INITRW ( 'checked   ' , CHECKEDSY )
816         ; INITRW ( 'exports   ' , EXPORTSSY )
817         ; INITRW ( 'imports   ' , IMPORTSSY )
818         ; INITRW ( 'include   ' , INCLUDESY )
819         ; INITRW ( 'machine   ' , MACHINESY )
820         ; INITRW ( 'returns   ' , RETURNSSY )
821         ; INITRW ( 'function  ' , FUNCSY )
822         ; INITRW ( 'readonly  ' , READONLYSY )
823         ; INITRW ( 'Boolean   ' , BOOLSY )
824         ; INITRW ( 'otherwise ' , OTHERWISESY )
825         ; INITRW ( 'pervasive ' , PERVASIVESY )
826         ; INITRW ( 'procedure ' , PROCSY )
827         ; INITRW ( 'SignedInt ' , SIGINTSY )
828         ; INITRW ( 'converter ' , CONVERTERSY )
829         END (* INITRWTABLE *)
830
831   (*  THIS PROCEDURE EATS UP SEPARATORS (";" ARE UNNEEDED
832   SEPARATORS IN EUCLID) AND EATS UP AND PRINTS OUT COMMENTS,
833   SO THAT "INPUT_CHAR" HAS THE FIRST CHAR OF A VALID TOKEN *)
834
835     ; PROCEDURE EAT_UP_GARBAGE
836
837       ; VAR SEPERATOR_OR_SEMI : BOOLEAN
838
839       ; BEGIN
840           SEPERATOR_OR_SEMI := TRUE
841         ; WHILE SEPERATOR_OR_SEMI
842           DO BEGIN
843             SEPERATOR_OR_SEMI := FALSE
844           ; WHILE ( INPUT_CHAR = ' ' )
845                   OR ( INPUT_CHAR = NL )
846                   OR ( INPUT_CHAR = ';' )
847             DO BEGIN SEPERATOR_OR_SEMI := TRUE
848                     ; READ ( INPUT_CHAR ) END
849           ; IF INPUT_CHAR = '{'
850             THEN BEGIN
851               NLINDENT
852             ; SEPERATOR_OR_SEMI := TRUE
853             ; IF INSIDE_COMMENT
854               THEN WRITE ( '*' )
855               ELSE WRITE ( '(' )
856             ; WRITE ( '*' )
857             ; READ ( INPUT_CHAR )
858             ; WHILE NOT ( INPUT_CHAR = '}' )
859               DO BEGIN
860                 WRITE ( INPUT_CHAR )
861               ; NUMB_OUTPUT := SUCC ( NUMB_OUTPUT )
862               ; READ ( INPUT_CHAR )
863               END
```

```
            ; READ ( INPUT_CHAR )
            ; WRITE ( '*' )
            ; IF INSIDE_COMMENT
              THEN WRITE ( '*' )
              ELSE WRITE ( ')' )
            ; NLINDENT
            END
          END
        END  (* EAT_UP_GARBAGE *)

    (* CODE BODIES ARE TO BE PRINTED EXACTLY AS IS.
       THEY HAVE PASCAL TEXT INSIDE.
       THE KEYWORD CODE SHOULD BE FOLLOWED BY "?", THEN THE
       LITERAL PASCAL TEXT, THEN ANOTHER "?". *)
    ; PROCEDURE COPY_VERBATUM

      ; BEGIN
          EAT_UP_GARBAGE
        ; WHILE INPUT_CHAR = '?'
          DO BEGIN
            READ ( INPUT_CHAR )
          ; WRITE ( NL )
          ; WHILE NOT ( INPUT_CHAR IN [ '?' , EM ] )
            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
  TOKEN *)

    ; 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
```

```
922             DO BEGIN
923               BEGIN_PTR := BEGIN_PTR - 1
924             ; TEMP_TEXT [ BEGIN_PTR ]
925               := CHR ( ( NUMBER MOD 10 ) + ORD ( '0' ) )
926             ; NUMBER := NUMBER DIV 10
927             END
928           ; TEXT := BLANKS
929           ; FOR I := BEGIN_PTR TO 70
930             DO TEXT [ I - BEGIN_PTR + 1 ] := TEMP_TEXT [ I ]
931           ;
932           END
933
934   (*   THIS PROCEDURE CONCATENATES A CHARACTER TO THE END OF
935   THE TOKEN TO BUILD IT. *)
936
937       ; PROCEDURE ADD_TO_OUT ( CONCAT_TO_STR : CHAR )
938
939         ; BEGIN
940             IF FILL_UP_PTR >= 71
941             THEN BEGIN
942               IF INSIDE_COMMENT
943               THEN OUT_ERROR
944                   ( '**THIS TOKEN TRUNCATED TO 70 CHAR ** &' )
945               ELSE OUT_ERROR
946                   ( ' (* THIS TOKEN TRUNCATED TO 70 CHAR *) &' )
947             ; NUMB_ERR := NUMB_ERR + 1
948             END
949             ELSE BEGIN
950               FILL_UP_PTR := FILL_UP_PTR + 1
951             ; TEXT [ FILL_UP_PTR ] := CONCAT_TO_STR
952             END
953           END (*  ADD_TO_OUT *)
954
955       ; PROCEDURE ADD_ADVANCE_CHAR ( CONCAT_TO_STR : CHAR )
956
957         ; BEGIN
958             ADD_TO_OUT ( CONCAT_TO_STR )
959           ; READ ( INPUT_CHAR )
960           END (* ADD_ADVANCE_CHAR *)
961
962     (* THIS IS ALSO DR. BATES' PART. IT THE TOKEN IS AN
963        IDENTIFIER, IT LOOKS IT UP IN THE KEYWORD
964        LIST AND DETERMINES THE TOKEN VALUE. *)
965       ; PROCEDURE LOOKUPIDENT
966
967         ; VAR LSTRING : STRING10
968           ; J , LO , HI , PROBE : INTEGER
969           ; EXITLOOP : BOOLEAN
970
971         ; PROCEDURE ALLCAPS ( VAR STRING : BUFFER )
972
973           ; VAR I : INTEGER
974
975           ; BEGIN
976               FOR I := 1 TO 70
977               DO IF ( 'a' <= STRING [ I ] )
978                       AND ( STRING [ I ] <= 'z' )
979                 THEN STRING [ I ]
```

```
980                  := CHR
981                       ( ORD ( STRING [ I ] )
982                         - ORD ( 'a' )
983                         + ORD ( 'A' )
984                       )
985              END
986
987          ; BEGIN
988              IF FILL_UP_PTR > 9
989              THEN TOK := IDENTSY
990              ELSE BEGIN
991                LSTRING := '          '
992              ; FOR J := 1 TO FILL_UP_PTR
993                    DO LSTRING [ J ] := TEXT [ J ]
994              ; LO := RWPTR [ FILL_UP_PTR ] - 1
995              ; HI := RWPTR [ FILL_UP_PTR + 1 ]
996              ; EXITLOOP := FALSE
997              ; REPEAT IF LO + 1 >= HI
998                  THEN BEGIN TOK := IDENTSY ; EXITLOOP := TRUE END
999                  ELSE BEGIN
1000                   PROBE := ( LO + HI ) DIV 2
1001                 ; IF RW [ PROBE ] . STRING = LSTRING
1002                   THEN BEGIN
1003                     TOK := RW [ PROBE ] . SYMBOL
1004                   ; ALLCAPS ( TEXT )
1005                   ; EXITLOOP := TRUE
1006                   END
1007                   ELSE IF RW [ PROBE ] . STRING < LSTRING
1008                     THEN LO := PROBE
1009                     ELSE HI := PROBE
1010                 END
1011               UNTIL EXITLOOP
1012             END
1013           END (* LOOKUPIDENT *)
1014
1015   (* EUCLID USES "$" WHERE PASCAL USES "(:...:)" - FURTHERMORE
1016    EUCLID HAS SPECIAL CONTROL CHARACTERS, SUCH AS "NEWLINE",
1017    "ENDOFMEDIUM", ETC ALSO INDICATED BY "$" (FOLLOWED BY A
1018    CHARACTER).  EUCLID ALSO SHOWS A ONE-CHARACTER STRING BY
1019    PRECEDING THE CHARACTER BY "$".
1020       FOR EXAMPLES,
1021          $B = 'B' PASC
1022          $N = 'N' PASC
1023          $$N = '$N' = NL PASC
1024          '$023' = $$023 = '(:023:)' PASC
1025        THIS PROCEDURE IS CALLED AFTER ONE DOLLAR SIGN HAS BEEN
1026    FOUND IF THE SCAN IS NOT INSIDE A LITERAL, AND IS CALLED FOR
1027    EVERY CHARACTER IF THE SCAN IS INSIDE LITERAL.
1028        IF CALLED INSIDE LITERAL AND IF THE NEXT CHAR IS NOT A
1029    DOLLAR SIGN, THIS PROC SIMPLY OUTPUTS IT. IF IT ENCOUNTERS
1030    A DOLLAR SIGN, THAT MEANS A SPECIAL CHAR IS TO BE PROCESSED
1031        IF CALLED OUTSIDE LITERAL, AND ONLY ONE DOLLAR SIGN IS
1032    FOUND, THIS PROC SIMPLY OUTPUTS THE CHAR (THE CALLING
1033    ROUTINE WILL OUTPUT THE TWO "'". OTHERWISE, A SPECIAL
1034    CHARACTER NEEDS TO BE PROCESSED
1035        ONE CAUTION - THE SYNCRONIZATION OF THIS ROUTINE IS VERY
1036    WEIRD, DEPENDING ON IF IT IS CALLED FROM INSIDE OR OUTSIDE
1037    A LITERAL. *)
```

```
1038
1039        ; PROCEDURE PROCESS_DOLLAR
1040
1041          ; VAR I : INTEGER
1042
1043          ; BEGIN
1044              IF INPUT_CHAR = '$'
1045              THEN BEGIN
1046                READ ( INPUT_CHAR )
1047              ; CASE INPUT_CHAR
1048                OF 'E' , 'F' , 'T' , 'N' , '(:39:)'
1049                  : BEGIN
1050                    ADD_TO_OUT ( '(' )
1051                  ; ADD_TO_OUT ( ':' )
1052                 (* DOUBLE CASE STMT WAS TO
1053                    AVOID HAVING TO WRITE OUT
1054                    "ADD_TO_OUT('(');ADD_TO_OUT(':')"  A LOT  *)
1055                    ; CASE INPUT_CHAR
1056                      OF 'E'
1057                        : BEGIN
1058                          ADD_TO_OUT ( '2' )
1059                        ; ADD_TO_OUT ( '5' )
1060                        END
1061                      ; 'F'
1062                        : BEGIN
1063                          ADD_TO_OUT ( '1' )
1064                        ; ADD_TO_OUT ( '2' )
1065                        END
1066                      ; 'T'
1067                        : BEGIN
1068                          ADD_TO_OUT ( '2' )
1069                        ; ADD_TO_OUT ( '3' )
1070                        END
1071                      ; 'N'
1072                        : BEGIN
1073                          ADD_TO_OUT ( '1' )
1074                        ; ADD_TO_OUT ( '0' )
1075                        END
1076                      ; '(:39:)'
1077                        : BEGIN
1078                          ADD_TO_OUT ( '3' )
1079                        ; ADD_TO_OUT ( '9' )
1080                        END
1081                      END
1082                    ; ADD_TO_OUT ( ':' )
1083                    ; ADD_ADVANCE_CHAR ( ')' )
1084                    END
1085                  ; 'S' : ADD_ADVANCE_CHAR ( ' ' )
1086                  ; '0' , '1' , '2' , '3' , '4' , '5' , '6' , '7'
1087                  , '8 , '9'
1088                    : BEGIN
1089                      ADD_TO_OUT ( '(' )
1090                    ; ADD_TO_OUT ( ':' )
1091                    ; FOR I := 1 TO 3
1092                          DO ADD_ADVANCE_CHAR ( INPUT_CHAR )
1093                    ; ADD_TO_OUT ( ':' )
1094                    ; ADD_TO_OUT ( ')' )
1095                    END
```

```
1096              ; ELSE : ADD_ADVANCE_CHAR ( '$' )
1097                END
1098            END
1099            ELSE ADD_ADVANCE_CHAR ( INPUT_CHAR )
1100          END (* PROCESS DOLLAR *)
1101
1102        ; BEGIN (* NEXT *)
1103            TEXT := BLANKS
1104          ; FILL_UP_PTR := 0
1105          ; EAT_UP_GARBAGE
1106          ; CASE INPUT_CHAR
1107            OF 'A' , 'B' , 'C' , 'D' , 'E' , 'F' , 'G' , 'H' , 'I'
1108              , 'J'
1109              , 'K' , 'L' , 'M' , 'N' , 'O' , 'P' , 'Q' , 'R' ,
'S'
1110              , 'T'
1111              , 'U' , 'V' , 'W' , 'X' , 'Y' , 'Z' , '_' , 'a' ,
'b'
1112              , 'c'
1113              , 'd' , 'e' , 'f' , 'g' , 'h' , 'i' , 'j' , 'k' ,
'l'
1114              , 'm'
1115              , 'n' , 'o' , 'p' , 'q' , 'r' , 's' , 't' , 'u' ,
'v'
1116              , 'w' , 'x' , 'y' , 'z'
1117                 (* TOKEN IS AN ID OR KEYWORD *)
1118              : BEGIN
1119                ADD_ADVANCE_CHAR ( INPUT_CHAR )
1120              ; WHILE INPUT_CHAR IN LETTER_DIG
1121                DO ADD_ADVANCE_CHAR ( INPUT_CHAR )
1122              ; LOOKUPIDENT
1123              ; IF TOK = INCLUDESY
1124                THEN BEGIN
1125                  OUT_ERROR ( '(* NEED TO INCLUDE A FILE - &' )
1126                ; INSIDE_COMMENT := TRUE
1127                ; NUMB_ERR := SUCC ( NUMB_ERR )
1128                ; NEXT
1129                ; OUT_AFTER_BL ( TEXT )
1130                ; OUT_LITERALLY ( ' *) &' )
1131                ; INSIDE_COMMENT := FALSE
1132                ; NEXT
1133                END
1134                ELSE IF ( TOK = CHARSY ) AND ( INPUT_CHAR = '.' )
1135                  THEN BEGIN
1136                    TOK := IDENTSY
1137                  ; FOR I := 1 TO 4 DO READ ( INPUT_CHAR )
1138                  ; TEXT := BLANKS
1139                  ; TEXT [ 1 ] := 'O'
1140                  ; TEXT [ 2 ] := 'r'
1141                  ; TEXT [ 3 ] := 'd'
1142                  END
1143              END
1144            ; '.' (* TOKEN IS EITHER "." OR ".."  *)
1145              : BEGIN
1146                TOK := DOTSY
1147              ; ADD_ADVANCE_CHAR ( INPUT_CHAR )
1148              ; IF INPUT_CHAR = '.'
1149                THEN BEGIN
```

```
1150                ADD_ADVANCE_CHAR ( INPUT_CHAR )
1151              ; TOK := ELIPSISSY
1152              END
1153            END
1154          ; '-' (* TOKEN IS "-","->" *)
1155            : BEGIN
1156              TOK := ADDOPSY
1157            ; ADD_ADVANCE_CHAR ( INPUT_CHAR )
1158            ; IF INPUT_CHAR = '>'
1159              THEN BEGIN
1160                ADD_ADVANCE_CHAR ( INPUT_CHAR )
1161              ; TOK := IMPLYSY
1162              END
1163            END
1164          ; '=' (* TOKEN IS "=" OR "=>" *)
1165            : BEGIN
1166              TOK := EQUALSSY
1167            ; ADD_ADVANCE_CHAR ( INPUT_CHAR )
1168            ; IF INPUT_CHAR = '>'
1169              THEN BEGIN
1170                ADD_ADVANCE_CHAR ( INPUT_CHAR )
1171              ; TOK := CASESEPSY
1172              END
1173            END
1174          ; ':' (* TOKEN IS ":" OR ":="  *)
1175            : BEGIN
1176              TOK := COLONSY
1177            ; ADD_ADVANCE_CHAR ( INPUT_CHAR )
1178            ; IF INPUT_CHAR = '='
1179              THEN BEGIN
1180                ADD_ADVANCE_CHAR ( INPUT_CHAR )
1181              ; TOK := BECOMESSY
1182              END
1183            END
1184          ; '>' , '<'
1185               (*  TOKEN IS EITHER ">=", "<=",">","<"  *)
1186            : BEGIN
1187              TOK := RELOPSY
1188            ; ADD_ADVANCE_CHAR ( INPUT_CHAR )
1189            ; IF INPUT_CHAR = '='
1190              THEN ADD_ADVANCE_CHAR ( INPUT_CHAR )
1191            END
1192          ; '$' (*  TOKEN IS A ONE-CHARACTER STRING  *)
1193            : BEGIN
1194              TOK := LITSTRSY
1195            ; ADD_ADVANCE_CHAR ( '(:39:)' )
1196            ; PROCESS_DOLLAR
1197            ; ADD_TO_OUT ( '(:39:)' )
1198            END
1199          ; '(:39:)'
1200               (*  TOKEN IS A LITERAL  *)
1201            : BEGIN                    '
1202              TOK := LITSTRSY
1203            ; WHILE INPUT_CHAR = '(:39:)'
1204              DO BEGIN
1205              ; ADD_ADVANCE_CHAR ( INPUT_CHAR )
1206              ; WHILE NOT ( INPUT_CHAR = '(:39:)' )
1207                    DO PROCESS_DOLLAR
```

```
1208              ; ADD_ADVANCE_CHAR ( '(:39:)' )
1209              END
1210            END
1211          ; '0' , '1' , '2' , '3' , '4' , '5' , '6' , '7' , '8'
1212          , '9'
1213              (*  TOKEN IS A NUMBER  *)
1214            : BEGIN
1215              TOK := CONSTANTSY
1216            ; ADD_ADVANCE_CHAR ( INPUT_CHAR )
1217            ; WHILE ( INPUT_CHAR IN DIGITS )
1218              DO ADD_ADVANCE_CHAR ( INPUT_CHAR )
1219            ; IF INPUT_CHAR = '#'
1220              THEN OCTAL_CONVERSION
1221            END
1222          ; EM : TOK := EOFSY
1223          ; ELSE (*  TOKEN IS ANY OTHER ONE-CHARACTER TOKEN  *)
1224            : BEGIN
1225              CASE INPUT_CHAR
1226              OF '+'
1227                : BEGIN
1228                  TOK := ADDOPSY
1229                ; ADD_ADVANCE_CHAR ( INPUT_CHAR )
1230                END
1231              ; '('
1232                : BEGIN
1233                  TOK := LPARSY
1234                ; ADD_ADVANCE_CHAR ( INPUT_CHAR )
1235                END
1236              ; ')'
1237                : BEGIN
1238                  TOK := RPARSY
1239                ; ADD_ADVANCE_CHAR ( INPUT_CHAR )
1240                END
1241              ; ','
1242                : BEGIN
1243                  TOK := COMMASY
1244                ; ADD_ADVANCE_CHAR ( INPUT_CHAR )
1245                END
1246              ; '*'
1247                : BEGIN
1248                  TOK := MULOPSY
1249                ; ADD_ADVANCE_CHAR ( INPUT_CHAR )
1250                END
1251              ; ELSE
1252                : BEGIN
1253                  OUT_UNREC_CH ( INPUT_CHAR )
1254                ; READ ( INPUT_CHAR )
1255                ; NEXT
1256                END
1257              END (* CASE OF ONE-CHAR TOK *)
1258            END  (* ELSE BIG CASE *)
1259          END (* CASE *)
1260        END (* NEXT *)
1261
1262      ; PROCEDURE EXPN
1263        ; FORWARD
1264
1265      (* THIS PROC SIMPLY DELETES AN ID FROM THE CODE BODY
```

```
1266          OF EUCLID IF IT IS A CONVERTER CALL. IT ALSO DELETES
1267          THE ACCOMPANYING PARENTHESES. *)
1268
1269        ; PROCEDURE DELETE_CONVERTER_CALL
1270
1271          ; BEGIN
1272              INSIDE_COMMENT := TRUE
1273            ; OUT_LITERALLY ( '(* &' )
1274            ; OUT_AND_GET (* NAME *)
1275            ; OUT_AND_GET (* LPARSY *)
1276            ; OUT_LITERALLY ( ' *)&' )
1277            ; INSIDE_COMMENT := FALSE
1278            ; EXPN
1279            ; INSIDE_COMMENT := TRUE
1280            ; OUT_LITERALLY ( '(* &' )
1281            ; OUT_AND_GET (* RPARSY *)
1282            ; OUT_LITERALLY ( ' *)&' )
1283            ; INSIDE_COMMENT := FALSE
1284           END
1285
1286          (* THIS PROCEDURE TRANSLATES A VARIABLE. IT ASSUMES
1287          THAT THE SYNTAX FOR A VARIABLE USED IN EUCLID IS:
1288
1289              VAR--> { "." <ID> ! "(" <EXPN> { "," <EXPN>} }
1290             *)
1291        ; PROCEDURE PROC_AND_OUT_VAR
1292
1293          ; VAR KIND_OF_ID : KINDOFID
1294            ; PROCESS_VAR : BOOLEAN
1295
1296          ; BEGIN
1297              IF TOK = INLINESY
1298              THEN OUT_TOK_AS_COMM
1299            ; OUT_FIXED_ID ( KIND_OF_ID )
1300            ; IF TOK = LPARSY
1301              THEN
1302            (* IS EITHER A PROCEDURE OR ARRAY *)
1303              BEGIN
1304                IF KIND_OF_ID = PROCID
1305                THEN OUT_LITERALLY ( '(&' )
1306                ELSE OUT_LITERALLY ( '[&' )
1307              ; NEXT
1308              ; EXPN
1309              ; WHILE TOK = COMMASY DO BEGIN OUT_AND_GET ; EXPN END
1310              ; IF KIND_OF_ID = PROCID
1311                THEN OUT_LITERALLY ( ') &' )
1312                ELSE OUT_LITERALLY ( '] &' )
1313              ; NEXT
1314            (* AT THIS STAGE OF THE GAME, IF THERE ARE ANY       *)
1315            (* "." OR "(" LEFT, WE ARE INSIDE ARRAYS AND RECORDS,*)
1316            (* NOT PROCEDURES OR FUNCTIONS.                      *)
1317              ; PROCESS_VAR := TRUE
1318              ; WHILE PROCESS_VAR
1319                DO BEGIN
1320                  PROCESS_VAR := FALSE
1321                ; WHILE TOK = DOTSY
1322                  DO BEGIN
1323                    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
```

```
1382
1383        ; BEGIN
1384            OUT_LITERALLY ( '(&' )
1385          ; TERM
1386          ; WHILE TOK = ADDOPSY DO BEGIN OUT_AND_GET ; TERM END
1387          ; OUT_LITERALLY ( ')&' )
1388          END
1389
1390      ; PROCEDURE RELATION
1391
1392        ; BEGIN
1393            OUT_LITERALLY ( '(&' )
1394          ; SUM
1395          ; IF ( TOK = RELOPSY ) OR ( TOK = EQUALSSY )
1396            THEN BEGIN OUT_AND_GET ; SUM END
1397            ELSE IF TOK = NOTSY
1398              THEN BEGIN OUT_LITERALLY ( '<> &' )
1399                         ; NEXT ; NEXT ; SUM END
1400          ; OUT_LITERALLY ( ')&' )
1401          END
1402
1403      ; PROCEDURE NEGATION
1404
1405        ; BEGIN
1406            OUT_LITERALLY ( '(&' )
1407          ; IF TOK = NOTSY
1408            THEN OUT_AND_GET
1409          ; RELATION
1410          ; OUT_LITERALLY ( ')&' )
1411          END
1412
1413      ; PROCEDURE CONJUNCTION
1414
1415        ; BEGIN
1416            OUT_LITERALLY ( '(&' )
1417          ; NEGATION
1418          ; WHILE TOK = ANDSY DO BEGIN OUT_AND_GET ; NEGATION END
1419          ; OUT_LITERALLY ( ')&' )
1420          END
1421
1422      ; PROCEDURE DISJUNCTION
1423
1424        ; BEGIN
1425            OUT_LITERALLY ( '(&' )
1426          ; CONJUNCTION
1427          ; WHILE TOK = ORSY DO BEGIN OUT_AND_GET
1428                                    ; CONJUNCTION END
1429          ; OUT_LITERALLY ( ')&' )
1430          END
1431
1432      ; PROCEDURE EXPN
1433
1434        ; BEGIN
1435            OUT_LITERALLY ( '( &' )
1436          ; DISJUNCTION
1437          ; IF TOK = IMPLYSY
1438            THEN BEGIN
1439              OUT_ERROR ( ' (* &' )
```

```
1440            ; OUT_LITERALLY ( ' THIS CAN NOT BE TRANSLATED &' )
1441            ; OUT_LITERALLY ( ' *) &' )
1442            ; NUMB_ERR := SUCC ( NUMB_ERR )
1443            ; OUT_LITERALLY ( '(* &' )
1444            ; INSIDE_COMMENT := TRUE
1445            ; OUT_AND_GET
1446            ; DISJUNCTION
1447            ; INSIDE_COMMENT := FALSE
1448            ; OUT_LITERALLY ( '*) &' )
1449            END
1450          ; OUT_LITERALLY ( ')&' )
1451          END
1452
1453      (* THIS PARSES A COMPILE TIME EXPRESSION, ASSIGNS IT
1454         TO A GENERATED CONSTANT ID, AND PASSES THAT ID
1455         TO THE CALLING PROC.
1456         THIS PROC IS THE FIRST ONE IN THIS LISTING
1457         THAT USES THE TECHNIQUES OF DECLARING
1458         INPUT CONSTRUCTS AS GENERATED SYSTEM IDS, AND PASSING
1459         THOSE DECLARED GENERATED IDS TO THE CALLING PROC. *)
1460      ; PROCEDURE COMP_EXPN ( VAR SYS_ID : BUFFER )
1461
1462        ; BEGIN
1463            GET_SYS_ID ( SYS_ID )
1464          ; OUT_LITERALLY ( 'CONST &' )
1465          ; OUT_AFTER_BL ( SYS_ID )
1466          ; OUT_LITERALLY ( ' = &' )
1467          ; SUM
1468          ; OUT_LITERALLY ( '; &' )
1469          END
1470
1471      ; PROCEDURE RANGE ( VAR SYS_TYPE : BUFFER )
1472
1473        ; VAR SYS_CONST_1 , SYS_CONST_2 : BUFFER
1474          ; GARBAGE : KINDOFID
1475          ; PTR : PTR_TO_SYM_TAB
1476
1477        ; BEGIN
1478            GET_SYS_ID ( SYS_TYPE )
1479          ; SEARCH_FOR_ID ( TEXT , PTR )
1480          ; IF ( PTR = NIL ) OR ( PTR ^ . TYPE_OF_ID = OTHERID )
1481            THEN
1482       (* RANGE HAS ELIPSIS *)
1483            BEGIN
1484              COMP_EXPN ( SYS_CONST_1 )
1485            ; NEXT
1486            ; COMP_EXPN ( SYS_CONST_2 )
1487            ; OUT_LITERALLY ( 'TYPE &' )
1488            ; OUT_AFTER_BL ( SYS_TYPE )
1489            ; OUT_LITERALLY ( ' = &' )
1490            ; OUT_AFTER_BL ( SYS_CONST_1 )
1491            ; OUT_LITERALLY ( '.. &' )
1492            ; OUT_AFTER_BL ( SYS_CONST_2 )
1493            ; OUT_LITERALLY ( '; &' )
1494            END
1495       (* RANGE IS TYPE *)
1496            ELSE BEGIN
1497              OUT_LITERALLY ( 'TYPE &' )
```

```
1498            ; OUT_AFTER_BL ( SYS_TYPE )
1499            ; OUT_LITERALLY ( ' = &' )
1500            ; OUT_FIXED_ID ( GARBAGE )
1501            ; OUT_LITERALLY ( ';&' )
1502            END
1503          END
1504
1505      ; PROCEDURE VAR_TYPE ( VAR SYS_ID : BUFFER )
1506
1507        ; VAR WHERE : PTR_TO_SYM_TAB
1508          ; SYS_TYPE : BUFFER
1509          ; GARBAGE_KIND : KINDOFID
1510
1511        ; BEGIN
1512            GET_SYS_ID ( SYS_ID )
1513          ; CASE TOK
1514            OF SIGINTSY , BOOLSY , CHARSY , LPARSY
1515              : BEGIN
1516                OUT_LITERALLY ( 'TYPE &' )
1517              ; OUT_AFTER_BL ( SYS_ID )
1518              ; OUT_LITERALLY ( ' = &' )
1519              ; CASE TOK
1520                OF SIGINTSY : OUT_LITERALLY ( 'INTEGER &' )
1521                ; BOOLSY , CHARSY : OUT_AFTER_BL ( TEXT )
1522                ; LPARSY
1523                  : BEGIN
1524                    OUT_AFTER_BL ( TEXT )
1525                  ; REPEAT NEXT
1526                    ; IF TOK = IDENTSY
1527                      THEN BEGIN
1528                        ADD_TO_SYM_TAB ( TEXT , OTHERID , WHERE )
1529                      ; APPEND ( WHERE )
1530                      END
1531                      ELSE OUT_AFTER_BL ( TEXT )
1532                    UNTIL TOK = RPARSY
1533                  END
1534              END
1535              ; OUT_LITERALLY ( '; &' )
1536              ; NEXT
1537              END
1538            ; IDENTSY
1539              : BEGIN
1540                SEARCH_FOR_ID ( TEXT , WHERE )
1541              ; IF WHERE ^ . TYPE_OF_ID <> OTHERID
1542                THEN BEGIN
1543                  OUT_LITERALLY ( 'TYPE &' )
1544                ; OUT_AFTER_BL ( SYS_ID )
1545                ; OUT_LITERALLY ( ' = &' )
1546                ; OUT_FIXED_ID ( GARBAGE_KIND )
1547                ; OUT_LITERALLY ( ' ;&' )
1548                END
1549                ELSE BEGIN
1550                  RANGE ( SYS_TYPE )
1551                   (* RANGE CAN BEGIN WITH AN ID *)
1552                ; OUT_LITERALLY ( 'TYPE &' )
1553                ; OUT_AFTER_BL ( SYS_ID )
1554                ; OUT_LITERALLY ( ' = &' )
1555                ; OUT_AFTER_BL ( SYS_TYPE )
```

```
1556              ; OUT_LITERALLY ( '; &' )
1557              END
1558            END
1559          ; CODESY
1560        (* THIS IS AN ADDITION BY DR. BATES.
1561           THIS WAY IT IS POSSIBLE TO DECLARE
1562           VARIANT RECORDS. *)
1563            : BEGIN
1564              OUT_LITERALLY ( 'TYPE &' )
1565            ; OUT_AFTER_BL ( SYS_ID )
1566            ; OUT_LITERALLY ( ' = &' )
1567            ; COPY_VERBATUM
1568            ; OUT_LITERALLY ( ';&' )
1569            ; NEXT
1570            END
1571          ; ELSE
1572              (* ANY OTHER POSSIBILITY MEANS THE ID IS
1573                 THE FIRST OF A RANGE *)
1574            : BEGIN
1575              RANGE ( SYS_TYPE )
1576            ; OUT_LITERALLY ( 'TYPE &' )
1577            ; OUT_AFTER_BL ( SYS_ID )
1578            ; OUT_LITERALLY ( ' = &' )
1579            ; OUT_AFTER_BL ( SYS_TYPE )
1580            ; OUT_LITERALLY ( '; &' )
1581            END
1582          END
1583        END
1584
1585    (* THIS PROCEDURE PROCESSES ONE FIELD INSIDE OF A
1586       RECORD DECLARATION. IT DOES NOT EXACTLY PARSE THE
1587       FIELD, BUT IT TAKES CORRECT ACTION. I.E., THE
1588       SYNTAX IS NOT CHECKED, BUT ASSUMING CORRECT SYNTAX,
1589       THE TRANSLATOR TAKES CORRECT ACTION. *)
1590     ; PROCEDURE REC_VAR_DCL
1591
1592       ; VAR GARBAGE : KINDOFID
1593         ; PTR : PTR_TO_SYM_TAB
1594
1595       ; BEGIN
1596           IF TOK = VARSY
1597           THEN BEGIN
1598             OUT_TOK_AS_COMM
1599           ; OUT_AND_GET
1600           ; OUT_AND_GET
1601           ; WHILE NOT ( ( TOK = VARSY ) OR ( TOK = ENDSY ) )
1602             DO CASE TOK
1603               OF SIGINTSY
1604                 : BEGIN
1605                   OUT_LITERALLY ( 'INTEGER &' )
1606                 ; NEXT
1607                 END
1608               ; IDENTSY : OUT_FIXED_ID ( GARBAGE )
1609               ; LPARSY
1610                 : BEGIN
1611                   REPEAT OUT_AND_GET
1612                   ; ADD_TO_SYM_TAB ( TEXT , OTHERID , PTR )
1613                   ; APPEND ( PTR )
```

```
1614                    ; NEXT (* ID *)
1615                    UNTIL TOK = RPARSY
1616                  ; OUT_AND_GET
1617                  END
1618                ; ARRAYSY
1619                  : BEGIN
1620                    OUT_AND_GET
1621                  ; OUT_LITERALLY ( '[ &' )
1622                  END
1623                ; OFSY : BEGIN OUT_LITERALLY ( '] &' )
1624                                ; OUT_AND_GET END
1625                ; BECOMESSY
1626                  : BEGIN
1627                    OUT_AND_GET
1628                  ; OUT_ERROR ( '(* ERROR - INIT IN RECORD *) &' )
1629                  ; NUMB_ERR := SUCC ( NUMB_ERR )
1630                  END
1631                ; PACKEDSY : OUT_TOK_AS_COMM
1632                ; ELSE
1633                  : BEGIN
1634                    IF NOT ( TOK IN QGOODRECORD )
1635                    THEN OUT_ERROR ( '(* RANGE HAS EXPN *)&' )
1636                  ; NUMB_ERR := SUCC ( NUMB_ERR )
1637                  ; OUT_AND_GET
1638                  END
1639                END
1640            END
1641          END
1642
1643      (* THIS IS THE FIRST OF SEVERAL STRAIGHT FORWARD PARSING
1644        PROCEDURES. THEY FOLLOW THE SYNTAX DIAGRAMS PRETTY
1645        DIRECTLY, SO THEY ARE NOT COMMENTED. *)
1646       ; PROCEDURE TYPE_DEF ( VAR SYS_TYPE_OUT : BUFFER )
1647         ; FORWARD
1648
1649       ; PROCEDURE TYPE_DEF
1650
1651         ; VAR SYS_TYPE , SYS_TYPE_2 : BUFFER
1652
1653         ; PROCEDURE PUT_OUT_TYPE_PRE
1654
1655           ; BEGIN
1656               OUT_LITERALLY ( 'TYPE &' )
1657             ; OUT_AFTER_BL ( SYS_TYPE_OUT )
1658             ; OUT_LITERALLY ( ' = &' )
1659             END
1660
1661         ; BEGIN
1662             GET_SYS_ID ( SYS_TYPE_OUT )
1663           ; IF TOK = PACKEDSY
1664             THEN OUT_TOK_AS_COMM
1665           ; IF TOK = RECORDSY
1666             THEN BEGIN
1667               PUT_OUT_TYPE_PRE
1668             ; OUT_AND_GET
1669             ; REC_VAR_DCL
```

```
1672              OUT_LITERALLY ( '; &' )
1673            ; REC_VAR_DCL
1674            END
1675          ; OUT_AND_GET
1676          ; OUT_LITERALLY ( ' ; &' )
1677          ; OUT_TOK_AS_COMM
1678          END
1679          ELSE IF TOK = ARRAYSY
1680            THEN BEGIN
1681              NEXT
1682            ; RANGE ( SYS_TYPE )
1683            ; NEXT
1684            ; IF TOK = ARRAYSY
1685              THEN TYPE_DEF ( SYS_TYPE_2 )
1686              ELSE VAR_TYPE ( SYS_TYPE_2 )
1687            ; PUT_OUT_TYPE_PRE
1688            ; OUT_LITERALLY ( 'ARRAY [&' )
1689            ; OUT_AFTER_BL ( SYS_TYPE )
1690            ; OUT_LITERALLY ( '] OF &' )
1691            ; OUT_AFTER_BL ( SYS_TYPE_2 )
1692            ; OUT_LITERALLY ( '; &' )
1693            END
1694            ELSE BEGIN
1695              VAR_TYPE ( SYS_TYPE_2 )
1696            ; PUT_OUT_TYPE_PRE
1697            ; OUT_AFTER_BL ( SYS_TYPE_2 )
1698            ; OUT_LITERALLY ( '; &' )
1699            END
1700        END
1701
1702    ; PROCEDURE CONST_DCL
1703
1704      ; VAR SYS_ID : BUFFER
1705        ; WHERE : PTR_TO_SYM_TAB
1706
1707      ; BEGIN
1708          NEXT
1709        ; ADD_TO_SYM_TAB ( TEXT , OTHERID , WHERE )
1710        ; NEXT
1711        ; IF TOK = BECOMESSY
1712          THEN
1713        (* IS SIMPLE CONST *)
1714          BEGIN
1715            OUT_LITERALLY ( 'CONST &' )
1716          ; APPEND ( WHERE )
1717          ; OUT_LITERALLY ( ' = &' )
1718          ; NEXT
1719          ; EXPN
1720          END
1721          ELSE
1722        (* IS STRUCTURED CONST *)
1723          BEGIN
1724            NEXT
1725          ; TYPE_DEF ( SYS_ID )
1726          ; NEXT
1727          ; OUT_LITERALLY ( 'CONST &' )
1728          ; APPEND ( WHERE )
1729          ; OUT_LITERALLY ( ' = STRUCTCON &' )
```

```
1730            ; 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
1740      ; PROCEDURE FORMAL_SECT
1741
1742        ; VAR WHERE : PTR_TO_SYM_TAB
1743          ; GARBAGE : KINDOFID
1744
1745        ; BEGIN
1746            IF TOK = PERVASIVESY
1747            THEN OUT_TOK_AS_COMM
1748          ; CASE TOK
1749            OF CONSTSY , READONLYSY : 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 (
1760              '(* NOT LEGAL PARM DCL-CHECK RANGE,TOO *) &' )
1761            ; NUMB_ERR := SUCC ( 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 = OFSY
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 := SUCC ( 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
```

```
1788              END
1789            ; IDENTSY
1790              : BEGIN
1791                OUT_FIXED_ID ( GARBAGE )
1792              ; IF NOT ( ( TOK = COMMASY ) OR ( TOK = RPARSY ) )
1793                THEN BEGIN
1794                  OUT_ERROR ( '(* ILLEGAL IN PARM TYPE *)&' )
1795                ; NUMB_ERR := SUCC ( NUMB_ERR )
1796                ; WHILE NOT ( ( TOK = COMMASY )
1797                              OR ( TOK = RPARSY ) )
1798                  DO IF TOK = IDENTSY
1799                    THEN OUT_FIXED_ID ( GARBAGE )
1800                    ELSE OUT_AND_GET
1801                END
1802              END
1803            ; ELSE
1804              : BEGIN
1805                OUT_ERROR ( '(* NOT LEGAL IN PARM TYPE *)&' )
1806              ; NUMB_ERR := SUCC ( NUMB_ERR )
1807              ; WHILE NOT ( ( TOK = COMMASY )
1808                            OR ( TOK = RPARSY ) )
1809                DO IF TOK = IDENTSY
1810                  THEN OUT_FIXED_ID ( GARBAGE )
1811                  ELSE OUT_AND_GET
1812              END
1813            END
1814          END
1815
1816      ; PROCEDURE PARM_LIST
1817
1818        ; BEGIN
1819            IF TOK = LPARSY
1820            THEN BEGIN
1821              OUT_AND_GET
1822            ; FORMAL_SECT
1823            ; WHILE TOK = COMMASY
1824              DO BEGIN
1825                NEXT
1826              ; OUT_LITERALLY ( ';&' )
1827              ; FORMAL_SECT
1828              END
1829            ; OUT_AND_GET
1830            END
1831          END
1832
1833      ; PROCEDURE TYPE_DCL
1834        ; FORWARD
1835
1836      ; PROCEDURE MOD_OR_VAR_DCL
1837        ; FORWARD
1838
1839      (* THIS IS THE PROCEDURE THAT OUTS ALL THE "COMPILE-
1840        TIME" INITIALIZATIONS. IT POPS THE STACK TWICE
1841        (TWICE FOR SYNCRONIZATION) AND USES THE INFO
1842        TO GENERATE ASSIGNMENT STATEMENTS FOR THE INITS. *)
1843      ; PROCEDURE OUT_VAR_INIT
1844
1845        ; VAR INIT_VAR : PTR_TO_SYM_TAB
```

```
1846          ; INIT_CONST : BUFFER
1847
1848        ; BEGIN
1849            FOR I := 1 TO 2
1850            DO BEGIN
1851              POP ( INIT_VAR , INIT_CONST )
1852            ; WHILE INIT_VAR <> STACK_MARKER
1853              DO BEGIN
1854                APPEND ( INIT_VAR )
1855              ; OUT_LITERALLY ( ':= &' )
1856              ; OUT_AFTER_BL ( INIT_CONST )
1857              ; OUT_LITERALLY ( ' ; &' )
1858              ; POP ( INIT_VAR , INIT_CONST )
1859              END
1860            END
1861          END
1862
1863      (* THIS OUTPUTS THE TOKENS "CHECKED" AND "NOT CHECKED"
1864        AS TOKENS IF THEY OCCUR IN THE SOURCE. *)
1865       ; PROCEDURE CHECK_CHECKED
1866
1867         ; BEGIN
1868             IF TOK = NOTSY
1869             THEN OUT_TOK_AS_COMM
1870           ; IF TOK = CHECKEDSY
1871             THEN OUT_TOK_AS_COMM
1872           END
1873
1874       ; PROCEDURE CONVERTER_DCL
1875         ; FORWARD
1876
1877       ; PROCEDURE ROUT_DEF ( WHERE2 : PTR_TO_SYM_TAB )
1878
1879         ; VAR THIS_LOOP_LABEL : CHAR
1880
1881         ; PROCEDURE STMT ( WHICH_LAB : CHAR )
1882           ; FORWARD
1883
1884         ; PROCEDURE STMT_LIST ( WHICH_LAB : CHAR )
1885
1886           ; BEGIN
1887               OUT_LITERALLY ( 'BEGIN &' )
1888             ; STMT ( WHICH_LAB )
1889             ; WHILE TOK IN QFIRSTSTMT
1890               DO BEGIN
1891                 OUT_LITERALLY ( '; &' )
1892               ; STMT ( WHICH_LAB )
1893               END
1894             ; OUT_LITERALLY ( 'END &' )
1895             END
1896
1897       (* PLEASE REFER TO THE ACCOMPANYING DOCUMENTATION FOR
1898         THE EXPLANATION OF THE TRANSLATION OF THE RETURN,
1899         LOOP, ETC. STATEMENTS. *)
1900          ; PROCEDURE STMT "WHICH_LAB: CHAR"
1901
1902           ; VAR GARBAGE_KIND : KINDOFID
1903             ; SAVE_LABEL : CHAR
```

```
1904
1905          ; BEGIN
1906              CASE TOK
1907              OF RETURNSY
1908                : BEGIN
1909                  OUT_LITERALLY ( 'BEGIN &' )
1910                ; NEXT
1911                ; IF TOK = LPARSY
1912                  THEN BEGIN
1913                    APPEND ( WHERE2 )
1914                  ; OUT_LITERALLY ( ':= &' )
1915                  ; NEXT
1916                  ; EXPN
1917                  ; OUT_LITERALLY ( '; &' )
1918                  ; NEXT
1919                  END
1920                ; OUT_LITERALLY ( 'GOTO 1    END &' )
1921                END
1922              ; IDENTSY , INLINESY
1923                : BEGIN
1924                  PROC_AND_OUT_VAR
1925                ; IF TOK = BECOMESSY
1926                  THEN BEGIN OUT_AND_GET ; EXPN END
1927                END
1928              ; ASSERTSY
1929                : BEGIN
1930                  OUT_LITERALLY ( '(* &' )
1931                ; INSIDE_COMMENT := TRUE
1932                ; OUT_AND_GET
1933                ; IF TOK = LPARSY
1934                  THEN BEGIN OUT_AND_GET ; EXPN ; OUT_AND_GET END
1935                ; INSIDE_COMMENT := FALSE
1936                ; OUT_LITERALLY ( ' *) &' )
1937                END
1938              ; BEGINSY
1939                : BEGIN
1940                  NEXT
1941                ; CHECK_CHECKED
1942                ; STMT_LIST ( WHICH_LAB )
1943                ; NEXT
1944                END
1945              ; LOOPSY
1946                : BEGIN
1947                  THIS_LOOP_LABEL := SUCC ( THIS_LOOP_LABEL )
1948                ; SAVE_LABEL := THIS_LOOP_LABEL
1949                ; OUT_LITERALLY ( 'WHILE TRUE DO &' )
1950                ; NEXT
1951                ; STMT_LIST ( THIS_LOOP_LABEL )
1952                ; OUT_LITERALLY ( ';&' )
1953                ; WRITE ( SAVE_LABEL )
1954                ; NUMB_OUTPUT := SUCC ( NUMB_OUTPUT )
1955                ; OUT_LITERALLY ( ': &' )
1956                ; OUT_TOK_AS_COMM
1957                ; OUT_TOK_AS_COMM
1958                END
1959              ; CASESY
1960                : BEGIN
1961                  OUT_AND_GET
```

```
1962                ; 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              ; IFSY
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
```

```
2020                ; OUT_LITERALLY ( 'GOTO &' )
2021                ; WRITE ( WHICH_LAB )
2022                ; NUMB_OUTPUT := SUCC ( NUMB_OUTPUT )
2023                END
2024              ; ELSE :
2025              END
2026            END
2027
2028        ; BEGIN
2029            OUT_LITERALLY ( '(* &' )
2030          ; INSIDE_COMMENT := TRUE
2031          ; WHILE NOT ( ( TOK = BEGINSY ) OR ( TOK = CODESY ) )
2032            DO OUT_AND_GET
2033          ; INSIDE_COMMENT := FALSE
2034          ; OUT_LITERALLY ( ' *) &' )
2035          ; MARK_STACK
2036          ; IF TOK = CODESY
2037            THEN BEGIN
2038              OUT_LITERALLY ( 'BEGIN &' )
2039            ; OUT_VAR_INIT
2040            ; COPY_VERBATUM
2041            ; NEXT
2042            END
2043            ELSE BEGIN
2044              NEXT
2045            ; CHECK_CHECKED
2046            ; WHILE TOK IN QDCL
2047              DO CASE TOK
2048                OF TYPESY : TYPE_DCL
2049                ; CONSTSY : CONST_DCL
2050                ; VARSY : MOD_OR_VAR_DCL
2051                ; CONVERTERSY : CONVERTER_DCL
2052                ; ELSE
2053                  : BEGIN
2054                    OUT_ERROR (
2055                    '(*LOGIC ERROR - DCL-IN-ROUT *) &' )
2056                  ; NUMB_ERR := SUCC ( NUMB_ERR )
2057                  END
2058                END
2059            ; OUT_LITERALLY ( 'BEGIN &' )
2060            ; OUT_VAR_INIT
2061            ; THIS_LOOP_LABEL := '1'
2062            ; STMT_LIST ( '1' )
2063            END
2064          END
2065
2066      ; PROCEDURE PROC_DCL
2067
2068        ; VAR WHERE3 , WHERE , WHERE2 : PTR_TO_SYM_TAB
2069          ; SAVE : BUFFER
2070          ; GARBAGE : KINDOFID
2071          ; FUNCT_IND : BOOLEAN
2072          ; LAB : CHAR
2073
2074        ; BEGIN
2075            IF TOK = FUNCSY
2076            THEN FUNCT_IND := TRUE
2077            ELSE FUNCT_IND := FALSE
```

```
2078          ; 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_IND
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
2096                  :=
2097
'INTEGER                                                                '
2098                ; NEXT
2099                END
2100            ; BOOLSY , CHARSY : BEGIN SAVE := TEXT ; OUT_AND_GET
END
2101            ; LPARSY
2102              : BEGIN
2103                OUT_ERROR
2104                  (
2105
'(* ILLEGAL FUNCTION DCL-WILL CAUSE MULTIPL ERR-CHANGE SOURCE *)&'
2106                  )
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
```

```
2133              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        END
2180
2181    (* THIS PROC ADDS A CONVERTER ID TO THE SYMBOL TABLE,
2182      AND DELETES ITS DECLARATION. *)
2183     ; PROCEDURE CONVERTER_DCL
2184
2185       ; VAR WHERE : PTR_TO_SYM_TAB
2186
2187       ; BEGIN
2188           INSIDE_COMMENT := TRUE
2189         ; OUT_LITERALLY ( '(* &' )
2190         ; OUT_AND_GET (* CONVERTERSY *)
```

```
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 )
```

```
2249              ; OUT_WOUT_BL_LIT ( '_INITIAL ;&' )
2250              ; QUEUE_INIT_PROC ( TOP_OF_DISP ^ . APPEND_ID )
2251              ; OUT_LITERALLY ( 'BEGIN &' )
2252              ; OUT_VAR_INIT
2253              ; OUT_LITERALLY ( 'END; &' )
2254              END
2255            ; GET_RID_OF_TAB ( FALSE )
2256            END
2257            ELSE
2258          (* IT WAS A VAR DCL *)
2259            BEGIN
2260              WHERE_ADDED ^ . TYPE_OF_ID := OTHERID
2261            ; TYPE_DEF ( SYS_TYPE )
2262            ; OUT_LITERALLY ( 'VAR &' )
2263            ; APPEND ( WHERE_ADDED )
2264            ; OUT_LITERALLY ( ' : &' )
2265            ; OUT_AFTER_BL ( SYS_TYPE )
2266            ; OUT_LITERALLY ( '; &' )
2267            ; IF TOK = BECOMESSY
2268              THEN BEGIN
2269                NEXT
2270              ; GET_SYS_ID ( SYS_CONST )
2271              ; PUSH ( WHERE_ADDED , SYS_CONST )
2272              ; OUT_LITERALLY ( 'CONST &' )
2273              ; OUT_AFTER_BL ( SYS_CONST )
2274              ; OUT_LITERALLY ( ' = &' )
2275              ; EXPN
2276              ; OUT_LITERALLY ( '; &' )
2277              END
2278            END
2279          END
2280
2281      ; PROCEDURE MODULE_TYPE "VAR INITIALLY_PART: BOOLEAN"
2282
2283        ; VAR GARBAGE : PTR_TO_SYM_TAB
2284          ; LAB : CHAR
2285
2286        ; BEGIN
2287            INITIALLY_PART := FALSE
2288          ; OUT_LITERALLY ( '(* &' )
2289          ; INSIDE_COMMENT := TRUE
2290          ; WHILE NOT ( TOK IN QFIRSTMODDCL ) DO OUT_AND_GET
2291          ; OUT_LITERALLY ( ' *) &' )
2292          ; INSIDE_COMMENT := FALSE
2293          ; MARK_STACK
2294          ; WHILE TOK <> ENDSY
2295            DO CASE TOK
2296              OF PERVASIVESY : OUT_TOK_AS_COMM
2297              ; IMPORTSSY , EXPORTSSY
2298                : BEGIN
2299                  OUT_LITERALLY ( ' (* &' )
2300                ; INSIDE_COMMENT := TRUE
2301                ; OUT_AND_GET
2302                ; WHILE TOK <> RPARSY DO OUT_AND_GET
2303                ; OUT_LITERALLY ( ' *) &' )
2304                ; INSIDE_COMMENT := FALSE
2305                ; OUT_TOK_AS_COMM
2306                END
```

```
2307              ; TYPESY : TYPE_DCL
2308              ; CONSTSY : CONST_DCL
2309              ; FUNCSY , PROCSY : PROC_DCL
2310              ; CONVERTERSY : CONVERTER_DCL
2311              ; VARSY : MOD_OR_VAR_DCL
2312              ; INLINESY : OUT_TOK_AS_COMM
2313              ; INITSY
2314                : BEGIN
2315                  INITIALLY_PART := TRUE
2316                ; OUT_LITERALLY ( 'PROCEDURE &' )
2317                ; OUT_AFTER_BL ( TOP_OF_DISP ^ . APPEND_ID )
2318                ; OUT_WOUT_BL_LIT ( '_INITIAL; &' )
2319                ; QUEUE_INIT_PROC ( TOP_OF_DISP ^ . APPEND_ID )
2320                ; START_NEW_TAB ( BLANKS )
2321                ; LAB := '1'
2322                ; OUT_LITERALLY ( 'LABEL &' )
2323                ; NUMB_OUTPUT := SUCC ( NUMB_OUTPUT )
2324                ; WRITE ( LAB )
2325                ; REPEAT OUT_LITERALLY ( ', &' )
2326                  ; LAB := SUCC ( LAB )
2327                  ; NUMB_OUTPUT := SUCC ( NUMB_OUTPUT )
2328                  ; WRITE ( LAB )
2329                  UNTIL LAB = '9'
2330                ; OUT_LITERALLY ( '; &' )
2331                ; ROUT_DEF ( GARBAGE )
2332                ; OUT_AND_GET
2333                ; OUT_LITERALLY ( ';&' )
2334                ; GET_RID_OF_TAB ( TRUE )
2335                END
2336              ; ELSE
2337                : BEGIN
2338                  OUT_ERROR ( ' (* LOGIC ERROR IN MOD DCL *) &' )
2339                ; NUMB_ERR := SUCC ( NUMB_ERR )
2340                ; OUT_TOK_AS_COMM
2341                END
2342              END
2343          ; OUT_TOK_AS_COMM
2344          ; OUT_TOK_AS_COMM
2345          END
2346
2347      (* THIS GENERATES THE MAIN BODY OF THE PASCAL OBJECT.
2348           THIS GENERATES PROCEDURE CALLS ON ALL THE
2349           MODULE INITIALIZATION PROCEDURES. *)
2350       ; PROCEDURE INITIALIZE_ALL_ROUT
2351
2352         ; VAR PTR_Q : INIT_PTR
2353
2354         ; BEGIN
2355             PTR_Q := FRONT_OF_QUEUE
2356           ; REPEAT PTR_Q := PTR_Q ^ . NEXT_QUEUE
2357             ; OUT_AFTER_BL ( PTR_Q ^ . INIT_ID )
2358             ; OUT_WOUT_BL_LIT ( '_INITIAL&' )
2359             ; OUT_LITERALLY ( '; &' )
2360             UNTIL PTR_Q ^ . NEXT_QUEUE = NIL
2361           END
2362
2363          (* THIS PROC READS IN AND OUTPUTS THE PROPER PREFIX *)
2364       ; PROCEDURE OUT_PREFIX
```

```
2365
2366        ; CONST PREFIXFILE = 1
2367
2368        ; VAR BLOCK : PAGE
2369          ; NEXT_CHAR_NO , NEXT_BLOCK_NO : INTEGER
2370
2371        ; BEGIN
2372            GET ( PREFIXFILE , 1 , BLOCK )
2373          ; NEXT_BLOCK_NO := 2
2374          ; NEXT_CHAR_NO := 1
2375          ; WHILE BLOCK [ NEXT_CHAR_NO ] <> EM
2376            DO BEGIN
2377              WRITE ( BLOCK [ NEXT_CHAR_NO ] )
2378            ; IF NEXT_CHAR_NO = PAGELENGTH
2379              THEN BEGIN
2380                GET ( PREFIXFILE , NEXT_BLOCK_NO , BLOCK )
2381              ; NEXT_BLOCK_NO := SUCC ( NEXT_BLOCK_NO )
2382              ; NEXT_CHAR_NO := 1
2383              END
2384              ELSE NEXT_CHAR_NO := SUCC ( NEXT_CHAR_NO )
2385            END
2386          END
2387
2388      ; PROCEDURE INITIALIZE
2389
2390        ; VAR I : INTEGER
2391          ; BIG_TEMP : BUFFER
2392          ; TEMP : ARRAY [ 1 .. 18 ] OF CHAR
2393
2394       (* THIS PROCEDURE LOADS THE SYMBOL TABLE WITH WHATEVER
2395          ID AND KIND_OF_ID IT IS PASSED. IT'S PURPOSE
2396          IS TO SAVE WORK, BECAUSE THERE ARE MANY SUCH
2397          IDENTIFIERS. *)
2398        ; PROCEDURE INIT_RSVD_WDS
2399            ( WHAT_RSV_WORD : BUFFER ; WHAT_KIND : KINDOFID )
2400
2401          ; VAR PASS_TO_TAB : BUFFER
2402            ; I : INTEGER
2403            ; PTR : PTR_TO_SYM_TAB
2404
2405          ; BEGIN
2406              PASS_TO_TAB := BLANKS
2407            ; I := 1
2408            ; REPEAT PASS_TO_TAB [ I ] := WHAT_RSV_WORD [ I ]
2409              ; I := SUCC ( I )
2410              UNTIL WHAT_RSV_WORD [ I ] = '@'
2411            ; ADD_TO_SYM_TAB ( PASS_TO_TAB , WHAT_KIND , PTR )
2412            END
2413
2414        ; BEGIN
2415            DIGITS
2416            := [ '0' , '1' , '2' , '3' , '4' , '5' , '6' , '7' ,
2417                 '8' , '9'
2418               ]
2419          ; LETTER_DIG
2420            := DIGITS
2421               + [ 'A' , 'B' , 'C' , 'D' , 'E' , 'F' , 'G' , 'H' ,
2422                   'I'
```

```
2423                , 'J' , 'K' , 'L' , 'M' , 'N' , 'O' , 'P' , 'Q'
2424                , 'R'
2425                , 'S' , 'T' , 'U' , 'V' , 'W' , 'X' , 'Y' , 'Z'
2426                , '_'
2427                , 'a' , 'b' , 'c' , 'd' , 'e' , 'f' , 'g' , 'h'
2428                , 'i'
2429                , 'j' , 'k' , 'l' , 'm' , 'n' , 'o' , 'p' , 'q'
2430                , 'r'
2431                , 's' , 't' , 'u' , 'v' , 'w' , 'x' , 'y' , 'z' ]
2432          ; QFIRSTSTMT
2433            := [ RETURNSY , IDENTSY , ASSERTSY , BEGINSY , LOOPSY
2434                 , CASES , IFSY , EXITSY ]
2435          ; QCASE := [ OTHERWISESY , ENDSY ]
2436          ; QFIRSTMODDCL
2437            := [ VARSY , PROCSY , FUNCSY , CONSTSY , TYPESY
2438               , CONVERTERSY
2439               , NOTSY , CHECKEDSY , IMPORTSSY , EXPORTSSY ]
2440          ; QDCL
2441            := [ TYPESY , CONSTSY , VARSY , CONVERTERSY , NOTSY
2442               , CHECKEDSY ]
2443          ; QGOODRECORD := [ CONSTANTSY , ELIPSISSY , CHARSY
2444                           , BOOLSY ]
2445          ; INSIDE_COMMENT := FALSE
2446          ; INITRWTABLE
2447          ; FREE_STACK := NIL
2448          ; FREE_DISP := NIL
2449          ; FREE_TABLE := NIL
2450          ; READ ( INPUT_CHAR )
2451          ; NEXT
2452          ; NUMB_ERR := 0
2453          ; NUMB_OUTPUT := 0
2454        (* INITIALIZATION OF PROCEDURES QUEUE *)
2455          ; NEW ( FRONT_OF_QUEUE )
2456          ; ADD_ON_QUEUE := FRONT_OF_QUEUE
2457          ; ADD_ON_QUEUE ^ . NEXT_QUEUE := NIL
2458        (* STACK INIT *)
2459          ; STACK_TOP := NIL
2460        (* SYMBOL TABLE INIT *)
2461          ; NEW ( BOTTOM_OF_DISP )
2462          ; TOP_OF_DISP := BOTTOM_OF_DISP
2463          ; NEW ( PTR_T )
2464          ; WITH TOP_OF_DISP ^
2465            DO BEGIN
2466              EARLIER_PTR := NIL
2467            ; LATER_PTR := NIL
2468            ; APPEND_ID := BLANKS
2469            ; SYM_TAB_PTR := PTR_T
2470            END
2471          ; WITH PTR_T ^
2472            DO BEGIN
2473              LEFT_SON := NIL
2474            ; RT_SON := NIL
2475            ; ID := BLANKS
2476            ; BACK_TO_DISP := TOP_OF_DISP
2477            END
2478          ; OUT_LITERALLY ( 'TYPE StorageUnit = INTEGER;&' )
2479          ; INIT_RSVD_WDS ( 'StorageUnit@' , TYPEID )
2480          ; 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 ( 'FF@' , OTHERID )
2487          ; INIT_RSVD_WDS ( 'CR@' , OTHERID )
2488          ; INIT_RSVD_WDS ( 'EM@' , 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 ( 'IOOPERATION@' , 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 ( 'WRITEEOF@' , OTHERID )
2516          ; INIT_RSVD_WDS ( 'REWIND@' , OTHERID )
2517          ; INIT_RSVD_WDS ( 'UPSPACE@' , OTHERID )
2518          ; INIT_RSVD_WDS ( 'BACKSPACE@' , OTHERID )
2519          ; INIT_RSVD_WDS ( 'IORESULT@' , 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 ( 'STARMEDIUM@' , 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 ( 'ARGTAG@' , 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 )
```

```
2539          ; INIT_RSVD_WDS ( 'PASSPTR@' , TYPEID )
2540          ; INIT_RSVD_WDS ( 'PASSLINK@' , TYPEID )
2541          ; INIT_RSVD_WDS ( 'ARGTYPE@' , TYPEID )
2542          ; INIT_RSVD_WDS ( 'MAXARG@' , OTHERID )
2543          ; INIT_RSVD_WDS ( 'ARGLIST@' , OTHERID )
2544          ; INIT_RSVD_WDS ( 'ARGS@' , TYPEID )
2545          ; INIT_RSVD_WDS ( 'INP@' , OTHERID )
2546          ; INIT_RSVD_WDS ( 'OUT@' , OTHERID )
2547          ; INIT_RSVD_WDS ( 'PROGRESULT@' , TYPEID )
2548          ; INIT_RSVD_WDS ( 'TERMINATED@' , OTHERID )
2549          ; INIT_RSVD_WDS ( 'OVERFLOW@' , OTHERID )
2550          ; INIT_RSVD_WDS ( 'POINTERERROR@' , OTHERID )
2551          ; INIT_RSVD_WDS ( 'RANGERROR@' , OTHERID )
2552          ; INIT_RSVD_WDS ( 'VARIANTERROR@' , OTHERID )
2553          ; INIT_RSVD_WDS ( 'HEAPLIMIT@' , OTHERID )
2554          ; INIT_RSVD_WDS ( 'STACKLIMIT@' , OTHERID )
2555          ; INIT_RSVD_WDS ( 'CODELIMIT@' , OTHERID )
2556          ; INIT_RSVD_WDS ( 'TIMELIMIT@' , OTHERID )
2557          ; INIT_RSVD_WDS ( 'CALLERR@' , OTHERID )
2558          ; INIT_RSVD_WDS ( 'READ@' , PROCID )
2559          ; INIT_RSVD_WDS ( 'WRITE@' , PROCID )
2560          ; INIT_RSVD_WDS ( 'PfxOpen@' , PROCID )
2561          ; INIT_RSVD_WDS ( 'PfxClose@' , PROCID )
2562          ; INIT_RSVD_WDS ( 'PfxGet@' , PROCID )
2563          ; INIT_RSVD_WDS ( 'PfxPut@' , PROCID )
2564          ; INIT_RSVD_WDS ( 'PfxLength@' , PROCID )
2565          ; INIT_RSVD_WDS ( 'MARK@' , PROCID )
2566          ; INIT_RSVD_WDS ( 'RELEASE@' , PROCID )
2567          ; INIT_RSVD_WDS ( 'IDENTIFY@' , PROCID )
2568          ; INIT_RSVD_WDS ( 'PfxAccept@' , PROCID )
2569          ; INIT_RSVD_WDS ( 'PfxDisplay@' , PROCID )
2570          ; INIT_RSVD_WDS ( 'RUN@' , PROCID )
2571          ; INIT_RSVD_WDS ( 'PfxAbort@' , PROCID )
2572          ; INIT_RSVD_WDS ( 'P@' , PROCID )
2573          ; INIT_RSVD_WDS ( 'IOMode@' , TYPEID )
2574          ; INIT_RSVD_WDS ( 'inFile@' , OTHERID )
2575          ; INIT_RSVD_WDS ( 'out@' , OTHERID )
2576          ; INIT_RSVD_WDS ( 'inOutFile@' , OTHERID )
2577          ; INIT_RSVD_WDS ( 'DIRECTION@' , OTHERID )
2578          ; INIT_RSVD_WDS ( 'size@' , PROCID )
2579       (* SYSTEM ID INIT *)
2580          ; SYSTEM_ID := BLANKS
2581          ; TEMP := 'SYSID00000        '
2582          ; FOR I := 1 TO 18 DO SYSTEM_ID [ I ] := TEMP [ I ]
2583          END
2584
2585       ; BEGIN
2586          BREAKPNT
2587          ; OUT_PREFIX
2588          ; INITIALIZE
2589          ; OUT_LITERALLY ( '(* &' )
2590          ; INSIDE_COMMENT := TRUE
2591          ; OUT_AFTER_BL ( TEXT )
2592          ; WHILE TOK <> EQUALSSY
2593               DO BEGIN NEXT ; OUT_AFTER_BL ( TEXT ) END
2594          ; OUT_LITERALLY ( ' *) &' )
2595          ; 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.