ISSN 0316-6295

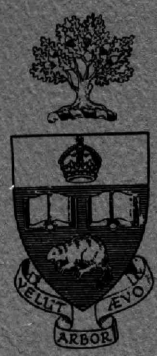
SPECIFICATION OF CONCURRENT EUCLID

(Preliminary Version)

James R. Cordy
Richard C. Holt

Technical Report CSRG-115
July 1980

COMPUTER SYSTEMS RESEARCH GROUP
UNIVERSITY OF TORONTO



SPECIFICATION OF CONCURRENT EUCLID

(Preliminary Version)

James R. Cordy
Richard C. Holt

Technical Report CSRG-115
July 1980

Computer Systems Research Group
University of Toronto
Toronto, Canada
M5S 1A1

The Computer Systems Research Group (CSRG) is an interdisciplinary group formed to conduct research and development relevant to computer systems and their application. It is jointly administered by the Department of Electrical Engineering and the Department of Computer Science of the University of Toronto, and is supported in part by the Natural Sciences and Engineering Research Council of Canada.

ABSTRACT

Concurrent Euclid (Euclid-C) is a programming language designed for implementing system software. It consists of a subset of the Euclid programming language [1] called Euclid-S plus concurrency features based on monitors [2].

This paper defines Concurrent Euclid independently of Euclid. It begins with a definition of Euclid-S and then describes the concurrency features to give Euclid-C. An understanding of the basic concepts of the Pascal family of programming languages is assumed.

CONTENTS

INTRODUCTION

This paper defines a subset of the Euclid programming language [1] called Euclid-S and a set of concurrency extensions to Euclid based on monitors [2]. Euclid-S plus the concurrency features combine to form a new language called Concurrent Euclid or Euclid-C. Euclid-C has been designed specifically for implementing system software, and in particular for implementing operating systems and compilers.

The first part of this document defines of the Euclid-S subset independently of Euclid. The second section describes the concurrency features added to form Euclid-C. The last section describes extensions to Euclid-C designed to allow separate compilation of procedures, functions, modules and monitors.

I. THE EUCLID-S SUBSET

This section describes the Euclid-S subset of Euclid. Euclid-S is defined independently of Euclid and no previous knowledge of the Euclid programming language is required. An understanding of the basic concepts of the Pascal family of programming languages is assumed.

IDENTIFIERS AND LITERALS

An identifier consists of any string of at most 50 letters, digits and underscores (_) beginning with a letter. Upper and lower case letters are considered identical in Euclid-S, hence aa, aA, Aa and AA all represent the same identifier. Keywords and predefined identifiers of Euclid must not be redeclared in a Euclid-S program. A list of these is given in Appendix 1.

A string literal is any sequence of characters not including a single quote (') surrounded by quotes. Within strings, the characters quote, dollar sign, new line and end of file are represented as $', $$, $N and $E respectively. As well, $T, $S and $F may be used for tab, space, and form feed respectively.

A character literal is a dollar sign ($) followed by any single character. The Euclid-S character literals corresponding to quote, dollar sign, space, tab, form feed, new line and end of file are $$', $$$, $$S, $$T, $$F, $$N and $$E respectively.

A integer literal is a decimal number, an octal number or a hexadecimal number. A decimal number is any sequence of decimal digits. An octal number is any sequence of octal digits followed by #8. A hexadecimal number is any sequence of hexadecimal digits (represented as the decimal digits plus the capital letters A through F) beginning with a decimal digit and followed by #16.

SOURCE PROGRAM FORMAT

A comment is any sequence of characters not including comment brackets surrounded by the comment brackets { and }. Comments may cross line boundaries.

A separator is a comment, blank, tab, form feed or source line boundary. Euclid-S programs are free-format; that is, the identifiers, keywords, literals, operators and special characters which make up a program may have any number of separators between them. Separators cannot be embedded in identifiers, keywords, literals or operators, except that blanks may appear as part of the value of a string literal. Identifiers, keywords and literals must not cross line boundaries. At least one separator must appear between adjacent identifiers, keywords and literals.

NOTATION

The following sections define the syntax of Euclid-S.

The following notation is used:
{item} means zero or more of the item
[item] means the item is optional

Keywords are given in lower case. Special symbols are enclosed in double quotes (").

The following abbreviations are used:
id for identifier
expn for expression
typeDefn for typeDefinition

In both Euclid-S and Euclid-C, all specified semicolons are optional. Whenever ";" appears, it can be omitted.

PROGRAMS

A main program consists of a module declaration.

A program is:

```
moduleDeclaration ";"
```

Execution of a program consists of initializing the main module, see "Modules".

Modules, procedures and functions can be compiled separately; see "Separate Compilation".

MODULES

A moduleDeclaration is:

```
var id ":"
   module
      [imports "(" [var] id {"," [var] id} ")" ";"]
      [exports "(" id {"," id} ")" ";"]
      [[not] checked ";"]
      {declarationInModule ";"}
      [initially
         procedureBody ";"]
   end module
```

Execution of a module declaration consists of executing the declarations in the module and then calling the "initially" procedure of the module. Execution of a Euclid-S program consists of executing the main module declaration in this way.

Module declarations may be nested inside other modules but must not be nested inside procedures and functions.

A module defines a package of variables, constants, types, procedures and functions. The interface of the module to the rest of the program is defined by its imports and exports clauses.

The imports clause lists the global identifiers which are to be visible inside the module. Variable identifiers may be imported "var" (or not). Only those variable identifiers which are imported "var" may be assigned to or passed to var parameters within the module. Imported module and collection identifiers must be imported using "var". Imported identifiers must not be redeclared inside the module.

The exports clause lists those identifiers defined inside the module which may be accessed outside the module using the "." operator. Exported variables cannot be assigned to or passed as var parameters. Unexported identifiers cannot be referenced outside the module.

Named types declared inside a module are opaque outside the module, that is, they are not considered equivalent to any other type. Exported variables and constants whose type is opaque cannot be subscripted, field selected or compared for other than equality.

Modules may be "checked"; this causes all assert statements, subscripts and case statements in the module to be checked for validity at run-time. In addition, a particular implementation may check other conditions such as ranges in assignments. Modules not already nested inside an unchecked module are checked by default and must be explicitly declared using "not checked" to turn off run-time checking.

Even though declared like variables, modules are not variables and cannot be assigned, compared, passed as parameters or exported. Module identifiers must be imported using "var".

Modules can be separately compiled if desired; see "Separate Compilation".

DECLARATIONS

A declarationInModule is one of the following:

a. constantDeclaration
b. variableDeclaration
c. typeDeclaration
d. variableBinding
e. moduleDeclaration
f. collectionDeclaration
g. procedureDeclaration
h. functionDeclaration
i. converterDeclaration
j. assert ["(" expn ")"]

Forms (a) through (i) are declarations for new identifiers as explained in the following sections. Form (j) is an assert statement; see "Statements". An identifier must be declared textually preceding any references to it.

CONSTANT DECLARATIONS

A constantDeclaration is one of:

```
a.  [pervasive] const id ":=" expn
b.  [pervasive] const id ":" typeDefn ":="
        "(" manifestExpn {"," manifestExpn} ")"
```

A constantDeclaration gives a name to a value which is constant throughout the scope of the declaration. The value of a scalar constant can be manifest or nonmanifest. A manifest constant or expression is one whose value is known at compile-time. A nonmanifest constant or expression must be evaluated at run-time.

Form (a) names a scalar constant or string literal. The type of the constant is the type of the value expression. The value of the scalar expression may be manifest or nonmanifest.

Form (b) declares an array constant. The typeDefn must be an array type or named array type whose component type is scalar. The list of expressions gives the values of the elements of the array constant. The element values must be manifest expressions assignable to the element type of the array. The number of element values specified must be exactly the number of elements in the array.

Constants declared using "pervasive" are automatically imported into all subscopes of the scope in which they are declared. Such constants need not be explicitly imported.

VARIABLE DECLARATIONS

A variableDeclaration is:

```
[register] var id ["(" at manifestExpn ")"] ":" typeDefn
        [":=" expn]
```

A variableDeclaration declares a variable of the specified type. The "at" clause declares a variable at an absolute machine location. Variables may optionally be declared with an initial value which is assigned to the variable when the declaration is executed. Fields of records cannot have initial values or "at" clauses.

Local variables in procedures and functions may optionally be declared "register". This is a hint to the compiler that it should attempt to allocate the variable to a register. Register variables cannot be bound to nor passed to a reference parameter.

A typeDeclaration is:

[pervasive] type id "=" typeBody

The typeBody is one of:

a. typeDefn
b. forward

A typeDeclaration gives a name to a type. The type name can subsequently be used in place of the full type definition. A named type is equivalent to the type that it names.

Named types may optionally be declared "pervasive". Type names declared using "pervasive" are automatically imported into all subscopes of the scope in which they are declared. Such types need not be explicitly imported.

Form (b) declares a forward type. A forward type declares a type name whose definition will be given in a later type declaration in the scope. A forward type can be used only as the element type of a collection until its real type definition is given. This allows the declaration of collections whose elements contain pointers to other elements in the collection.

A typeDefn is one of the following:

a. standardType
b. manifestConstant ".." manifestExpn
c. [packed] array indexType of typeDefn
d. set of baseType
e. [packed] recordType
f. pointerType
g. namedType

The following are standardTypes of Euclid-S:

SignedInt	- signed integer, implementation defined range
UnsignedInt	- unsigned integer, implementation defined range
LongInt	- signed integer, implementation defined range
ShortInt	- unsigned integer, implementation defined range
Boolean	- values are "true" and "false"
Char	- single character
StorageUnit	- no operations or literals, smallest addressable memory unit (typically a byte)
AddressType	- implementation defined integer range

The standard types and the constants true and false are implicitly declared pervasive in the global scope and need not be

imported.

Form (b) is a subrange type. The leading constant must be a literal or manifest named constant and gives the lower bound of the range of values for variables declared using the type. The expression, which must be manifest, gives the upper bound of the range. The bounds must be both integer values or both character values. The lower bound must be less than or equal to the upper bound.

A scalar type is a subrange, pointer or one of the standard types.

Form (c) is an array type. The indexType must be a subrange type, Char or a named type which is an indexType. The indexType gives the range of subscripts. The typeDefn gives the type of the elements of the array.

Elements of an array variable are referenced using subscripts (see "Variables and Constants") and themselves used as variables. Array variables and constants may also be assigned (but not compared) as a whole.

Arrays can be "packed", which allows the compiler to pack the elements more efficiently if possible. The type of string literals in Euclid-S is "packed array 1..n of Char" where n is the length of the string.

Form (d) is a set type. The baseType of the set must be a subrange of integer with lower bound 0 or a namedType which is a baseType. An implementation may limit the upper bound of a set type to insure efficient code; a typical limit could be 15.

A recordType is:

```
record
   {var id ":" typeDefn ";"}
end record
```

Variables declared using a record type have the fields given by the variable declarations in the recordType. Fields of a record variable may be referenced using the "." operator (see "Variables and Constants") and themselves used as variables. Record variables may be assigned (but not compared) as a whole.

The variableDeclarations in a record type must not have initial values and cannot be declared to be at absolute locations.

A pointerType is:

```
"^" collectionId
```

Variables declared using a pointerType are pointers to dynamically allocated and freed elements of the specified collection (see "Collections"). Pointer variables are used as subscripts of the specified collection to select the element to which they

point. The selected element can be used as a variable. Pointer variables may be assigned, compared for equality and passed as parameters.

A namedType is:

```
[moduleId "."] typeId
```

The typeId must be a previously declared type name. Type names exported from a module are referenced outside the module using the "." operator.

TYPE EQUIVALENCE AND ASSIGNABILITY

Two types are defined to be equivalent if they are

(a) subranges with equal first and last values

(b) arrays (both packed or both unpacked) with equivalent index types and equivalent component types

(c) sets with equivalent base types

(d) pointers to the same collection

A declared type identifier is equivalent to the type it names, with the following exception. When an exported type identifier is used outside its module, as "moduleId.typeId", it is a unique type, equivalent to no other type.

Each type definition for a record type creates a new type that is not equivalent to any other record type definition.

An array value can be assigned to an array variable, a record value assigned to a record variable, a set value assigned to a set variable and a pointer value assigned to a pointer variable only if the source and target of the assignment have equivalent types.

An expression can be assigned to a scalar variable only if (i) the "root" type of the expression and the "root" type of the variable are equivalent, and (ii) the value of the expression is in the range of the variable's type. The "root" type of Char and character subrange types is Char. The root type of SignedInt, UnsignedInt, AddressType and integer subranges is integer. The root type of any other type is the type itself.

A variable can be passed to a reference parameter only if its type is equivalent to the parameter type. An expression can be passed to a value parameter only if it is assignable to the parameter type.

VARIABLE BINDINGS

A variableBinding is one of:

```
a.  bind [var] id to variable
b.  bind "(" [var] id to variable
          {"," [var] id to variable} ")"
```

A variableBinding declares a new identifier for an arbitrary variable reference which may contain subscripts and "." operators. The new identifier is subsequently used in place of the variable reference within the scope in which the binding appears. If the bound variable is to be assigned to or passed to a var parameter, the binding must be declared using "var". Euclid-S does not allow "aliasing" of variables (i.e., having two names for the same variable in a scope). Hence the "root" variable (the first identifier in the variable reference) becomes inaccessible for the scope of the binding.

Form (b) allows bindings to different elements or fields of the same variable or module. Since Euclid-S does not allow aliasing of variables, bindings to the same field, element or variable are not allowed.

COLLECTIONS

A collectionDeclaration is:

```
var id ":" collection of typeDefn
```

A collection is essentially an array whose elements are dynamically allocated and freed at run-time. Elements of a collection are referenced by subscripting the collection name with a variable of the collection's pointer type. This subscripting selects the particular element of the collection located by the pointer variable.

Elements of a collection are allocated and freed dynamically by calls to the built-in operations New and Free. "C.New(p)" allocates a new element in the collection C and sets p to point at it. "C.Free(p)" frees the element of C pointed at by p. In each case p must be a variable of the pointer type of C. These operations are invoked as statements in procedures, see "Statements". They cannot be used in functions.

The built-in constant "C.nil" is the null pointer value for the collection.

Collections themselves cannot be assigned, compared, passed as parameters or exported. A collection must be imported using "var".

PROCEDURES AND FUNCTIONS

A procedureDeclaration is:

```
procedure id ["(" [var] id ":" parameterType
        {"," [var] id ":" parameterType} ")"] "="
   procedureBody
```

A functionDeclaration is:

```
function id ["(" id ":" parameterType
        {"," id ":" parameterType} ")"]
      returns id ":" resultType "="
   procedureBody
```

A procedure is invoked by a procedure call statement, with actual parameters if required. A function is invoked by using its name, with actual parameters if required, in an expression.

A procedure may return explicitly by executing a return statement or implicitly by reaching the end of the procedure body. A function must return via "return(expn)".

Procedures and functions may optionally take parameters, the types of which are defined in the header. The parameters can be referred to inside the procedure or function using the names declared in the header. Parameters to a procedure may be declared using "var", which means the parameter may be assigned to or further passed as a var parameter inside the procedure. Parameters declared without using "var" are constants and cannot be assigned to or passed as var parameters. Functions are not allowed to have any side-effects and cannot have var parameters. Only variable references can be passed to var parameters.

A parameter is a reference parameter if it is declared using "var" or if its type is an array or record. Other parameters are value parameters. Hence, a value parameter is a non-var parameter whose type is a scalar or set.

A parameterType is one of:

a. typeDefn
b. [packed] array manifestConstant ".." parameter of
   typeDefn
c. universal

The type of a variable, record or array passed to a reference parameter must be equivalent to the parameter's type with the following exceptions. (1) The upper bound of the index type of an array parameter can be declared using the keyword "parameter" in which case any array value whose element type and index type lower bound are equivalent to the parameter's can be passed to the parameter. (2) The type of a parameter can be specified as "universal", in which case a value of any type can be passed to the parameter. Inside the procedure, a universal parameter is equivalent to a parameter of type "packed array 1..parameter of

StorageUnit", where the upper bound is the size of the actual parameter in StorageUnits. Parameters declared using "parameter" or "universal" cannot be assigned or compared as a whole. (Note: Full Euclid does not allow forms (b) and (c).) Elements of packed arrays and fields of packed records cannot be passed to a reference parameter.

The type of an expression passed to a value parameter must be assignable to the parameter's type.

Euclid-S does not allow "aliasing" of variables (i.e., having two names for a given variable or part of a given variable in the same scope). Hence a variable or part of a variable which is imported directly or indirectly into a procedure cannot be passed to a reference parameter of the procedure. (A variable is directly imported if it appears in the procedure's import list. It is indirectly imported if an imported module or procedure directly or indirectly imports it.)

The returns clause defines the result type of a function. The return identifier is required for compatibility with full Euclid but cannot be used in Euclid-S.

The resultType of a function must be a scalar type or set. The expression in a function's return statement must be assignable to the result type.

A procedureBody is:

```
[imports "(" [var] id {"," [var] id} ")" ";"]
begin
   [[not] checked ";"]
   {declarationInRoutine ";"}
   {statement ";"}
end [id]
```

The identifier following the "end" must be the procedure or function identifier. If the procedure is the initially procedure of a module then the end identifier must not be present.

The imports clause of a procedure or function specifies those global identifiers which are to be visible inside the procedure or function. Only those variables imported into a procedure using "var" may be assigned to or passed to a var parameter inside the procedure. Functions are not allowed to have side-effects and cannot import modules, collections and "var" variables. This restriction is transitive; hence a function cannot import a procedure which imports anything "var".

Procedures and functions may be "checked"; this causes assert statements, subscripts and case statements to be checked for validity at run-time. In addition, a particular implementation may check other conditions, such as ranges in assignments. Procedures and functions not nested inside an unchecked module are checked by default and must be explicitly declared using "not checked" to turn off run-time checking.

A procedure returns when it executes a return statement or reaches the end of the procedure. A function is executed similarly but must return via "return(expn)".

Procedures and functions can be separately compiled; see "Separate Compilation".

A declarationInRoutine is one of:

a. constantDeclaration
b. variableDeclaration
c. typeDeclaration
d. variableBinding
e. collectionDeclaration
f. converterDeclaration
g. assert ["("expn")"]

Modules, procedures and functions cannot be nested inside a procedure or function. Form (g) allows assert statements to appear in declaration lists.

TYPE CONVERTERS

A converterDeclaration is:

```
converter id "(" typeId ")" returns typeId
```

A converterDeclaration declares a type converter. A type converter is a special function which converts an expression to a type other than its declared type. Both the parameter and result type of a type converter must be named types. An implementation is not expected to generate any code for a type conversion.

A type converter may be used to convert a variable that is passed to a reference parameter.

If the size of the target type is larger than the size of the source type, the conversion may be meaningless.

STATEMENTS

A statement is one of:

```
a. variable ":=" expn
b. [moduleId"."] procedureId ["(" expn {"," expn} ")"]
c. assert ["("expn")"]
d. return ["("expn")"]
e. if expn then
      {statement ";"}
   {elseif expn then
      {statement ";"}}
   [else
      {statement ";"}]
   end if
```

```
f.  loop
       {statement ";"}
    end loop
g.  exit [when expn]
h.  case expn of
       {manifestExpn {"," manifestExpn} "=>"
          {statement ";"}
          end manifestExpn ";"}
       [otherwise "=>"
          {statement ";"}]
    end case
i.  begin
       {declarationInRoutine ";"}
       {statement ";"}
    end
j.  collectionId "." New "(" variable ")"
k.  collectionId "." Free "(" variable ")"
```

Form (a) is an assignment statement. The expression is evaluated and the value assigned to the variable. The expression must be assignable to the variable type (see "Type Equivalence and Assignability").

Form (b) is a procedure call. An exported procedure is called outside the module in which it was declared using the "." operator.

The type of an expression passed to a value parameter must be assignable to the parameter's type. The type of a variable or value passed to a reference parameter must be equivalent to the parameter's type. If the upper bound of the type of an array parameter is declared using "parameter", any array whose element type and index type lower bound are the same as the parameter's can be passed to the parameter.

An actual parameter passed to a var parameter must be a variable, a bound variable or a var formal parameter. If it is an imported (or exported) variable, it must have been imported (or exported) using "var".

Form (c) is an assert statement. The parenthesized expression is optional; if it is omitted, it can be replaced by a comment. If present, it must be of type Boolean. The expression is evaluated and checked at run time if it appears in a checked scope. Assert statements may appear in both statement lists and declaration lists. They cannot appear inside records.

Form (d) is a return statement. The return statement causes an immediate return from the procedure or function when executed. The optional parenthesized expression gives the value to be returned from a function. The return expression is required for function returns. It is forbidden for procedure returns. A function must return via a return statement and not implicitly by reaching the end of the function body. A procedure may return either via a return statement or implicitly by reaching the end of the procedure body.

Form (e) is an if statement. The conditional expression following "if" and each "elseif" is evaluated until one of them is found to be true, in which case the statements following the corresponding "then" are executed. If none of the expressions evaluates to true then the statements following "else" are executed; if no "else" is present then execution continues following the if statement. The conditional expressions must be of type Boolean.

Form (f) is the looping construct of Euclid-S. The statements within the loop are repeated until one of its "exit" statements or a "return" statement is executed.

Form (g) is a loop exit. When executed, it causes an immediate exit from the nearest enclosing loop. The optional "when" expression makes the exit conditional. If the expression, which must be Boolean, evaluates to true then the exit is executed, otherwise execution of the loop continues. An exit statement cannot appear outside a loop.

Form (h) is a case statement. The case expression is evaluated and compared with each of the label values. The statements which follow the matching label value are executed. If the case expression value does not match any of the label values then the statements following "otherwise" are executed. If no "otherwise" is present, the case expression must match one of the label values. When execution of the statements following the selected label is completed, execution continues following the case statement.

The type of the case expression must be SignedInt or Char. All of the label expressions must be of the same type as the case expression. Label expressions must be manifest, i.e., their values must be known at compile time. The values of all label expressions in a given case statement must be distinct. The value of the manifest expression following the end of an alternative must be equal to the first label expression of the alternative.

Form (i) is a begin block. Begin blocks can be used to group local declarations within a procedure or function. In particular, a they can be used to make local binds.

Forms (j) and (k) are the built-in collection operations New and Free (see "Collections").

VARIABLES AND CONSTANTS

A variable is:

```
[moduleId "."] id {componentSelector}
```

The syntax of variable includes variable and constant references. An exported variable or constant is referenced outside the module in which it is declared using the "." operator.

A componentSelector is one of:

a. "(" expn ")"
b. "." id

Form (a) allows subscripting of variable and constant arrays. The type of the subscript expression must be assignable to the index type of the array. The value of the subscript expression must be in the declared range of the index type of the array. Subscripts which appear in checked scopes are checked for validity at run-time.

Form (a) also allows references to elements of a collection. In this case, the subscript expression must be a pointer to an element of the collection.

Form (b) allows record field selection. Fields of a record variable are referenced using the "." operator.

EXPRESSIONS

An expn is one of the following:

a. variable
b. literalConstant
c. setTypeId "(" elementList ")"
d. collectionId "." nil
e. [moduleId "."] functionId ["(" expn {"," expn} ")"]
f. [moduleId "."] converterId "(" expn ")"
g. "(" expn ")"
h. "-" expn
i. expn arithmeticOperator expn
j. expn comparisonOperator expn
k. not expn
l. expn booleanOperator expn
m. expn setOperator expn

The arithmeticOperators are +, -, * (multiply), div (integer divide) and mod (integer remainder). Operands of the arithmetic operators and unary minus must be integers or expressions having root type integer. The arithmetic operators yield an integer result. (Note: + and - are also set operators; see below.)

The comparisonOperators are <, >, =, <=, >= and "not =". Operands of comparison operators must either be the same type or have the same root type; see "Type Equivalence and Assignability". The comparison operators yield a Boolean result. Arrays and records cannot be compared. Sets and Boolean expressions can be compared for equality only. (Note: <= and >= are also set operators; see below.)

The booleanOperators are "and" (intersection), "or" (union) and -> (implication). The Boolean operators and the "not" operator take Boolean operands and yield a Boolean result.

The setOperators are * (set intersection), + (set union), - (set difference), <= and >= (set inclusion), and "in" and "not in" (element containment). The set operators + and - take operands of equivalent set types and yield a set result. The set operators <= and >= take operands of equivalent set types and yield a Boolean result. The operators "in" and "not in" take a set as right operand and an integer expression as left operand. They yield a Boolean result.

The order of precedence is among the following classes of operators (most binding first):

1. unary -
2. *, div, mod
3. +, -
4. <, >, =, <=, >=, not =, in, not in
5. not
6. and
7. or
8. ->

Expression form (a) includes references to constants and variables including elements of arrays and collections, fields of records, and constants and variables exported from a module.

Form (b) includes integer, character and string literal constants.

Form (c) is a set constructor. The setTypeId must be the name of a set type. The set constructor returns a set containing the specified elements.

A setElementList is one of:

a. [expn {"," expn}]
b. all

The element list can be a (possibly empty) list of expressions of the base type of the set, or "all". If "all" is specified, the constructor returns the complete set. If no elements are specified, the constructor returns the empty set.

Expression form (d) is the null pointer value of the specified collection.

Form (e) is a function call. Functions exported from a module are referenced outside the module using the "." operator. An actual parameter to a function must be an expression assignable to the parameter type.

Form (f) is a type conversion. The type of the actual parameter is changed to the result type of the type converter. The actual parameter must be an expression assignable to the source type of the converter. Type converters exported from a module are referenced outside the module using the "." operator.

BUILT-IN FUNCTIONS

Euclid-S has two built-in functions, Chr and Ord. "Chr(i)" returns the character whose machine representation is the positive integer value i. "Char.Ord(c)" returns the positive integer machine representation of the character c. Chr and Ord are defined such that for all characters "c" in the machine character set, Chr(Char.Ord(c)) = c.

STANDARD COMPONENTS

Euclid-S defines two standard components, size and address. "T.size" returns the length in StorageUnits (typically bytes) of the machine representation of the variable or type T. "V.address" returns the unsigned integer machine address of the variable V. (Note: In full Euclid, "V.address" is legal only for variables of type StorageUnit.)

SOURCE INCLUSION FACILITY

Other Euclid-S source files may be included as part of a program using the "include" statement.

An includeStatement is:

```
include stringLiteral ";"
```

The stringLiteral gives the name of a Euclid-S source file to be included in the compilation. The include statement is replaced in the program source by the contents of the specified file.

Include statements can appear anywhere in a program and can contain any valid source fragment. Included source files can themselves contain include statements.

II. CONCURRENCY FEATURES OF EUCLID-C

The Euclid-C language is an extension to Euclid-S designed to allow concurrent programming with monitors. Euclid-S is a subset of Euclid but Euclid-C is not, because concurrency and monitors are not features of Euclid.

The concurrency features of Euclid-C will be presented in the following order:

(1) processes, reentrant procedures and modules;
(2) monitors, entry procedures and functions;
(3) conditions, signalling and waiting;
(4) simulation, the busy statement.

PROCESSES

Each Euclid-C module (including the main module) can have any number of concurrently run processes associated with it.

A moduleDeclaration is:

```
var id ":"
   module
      [imports "(" [var] id {"," [var] id} ")" ";"]
      [exports "(" id {"," id} ")" ";"]
      [[not] checked ";"]
      {declarationInModule ";"}
      [initially
         procedureBody ";"]
      {process id ["(" memoryRequirement ")"]
         procedureBody ";"}
   end module
```

Each process is like a parameterless procedure. Concurrent execution of the processes of the module begins following execution of the initially procedure of the module. A process terminates by executing its last statement or by executing a return statement. The process identifier is for documentation only since processes cannot be called.

Processes can communicate with each other by changing and inspecting variables declared in the module or imported into it. Generally, however, processes communicate by means of monitors.

Each process requires a certain amount of memory space for its variables. When the process calls a procedure or function, the requirement increases to provide space for the new local variables. When the procedure or function returns, the requirement decreases to its former amount. The programmer can provide his own estimate of the process's required space as a parenthesized manifest integer expression following the keyword "process". This estimate is in StorageUnits (normally bytes) and can be based on previous program executions. If this estimate is omitted, the implementation provides a default space allocation.

All procedures and functions declared in a Euclid-C program are reentrant, meaning that they can be executed simultaneously by more than one process.

Modules, monitors, procedures and functions cannot be nested inside a process.

MONITORS

A monitor is essentially a special kind of module which implements inter-process communication with synchronization.

A declarationInModule is one of the following:

a. constantDeclaration
b. variableDeclaration
c. typeDeclaration
d. variableBinding
e. moduleDeclaration
f. monitorDeclaration
g. collectionDeclaration
h. procedureDeclaration
i. functionDeclaration
j. converterDeclaration
k. assert ["(" expn ")"]

Monitors may only be declared inside modules. Monitors cannot be nested inside procedures, functions or other monitors.

A monitorDeclaration is:

```
var id ":"
   monitor
      [imports "(" [var] id {"," [var] id} ")" ";"]
      [exports "(" id {"," id} ")" ";"]
      [[not] checked ";"]
      {declarationInMonitor ";"}
      [initially
         procedureBody ";"]
   end monitor
```

The imports list of a monitor specifies the global identifiers which are accessible inside the monitor, exactly like the imports list in a module.

The exports list of a monitor specifies those identifiers defined inside the monitor which may be accessed outside the monitor using the "." operator. Unlike modules, monitors cannot export variables.

Procedures and functions which are exported from a monitor are called monitor entries. Entry procedures and functions of a monitor cannot be invoked inside the monitor. Outside the monitor, entry procedures and functions can be invoked exactly like the procedures and functions of a module, using the "." operator.

Procedures and functions which are entries of a monitor cannot be separately compiled except as part of the entire monitor.

It is guaranteed that only one process at a time will be executing inside a monitor. As a result, mutually exclusive access to a monitor's variables is implicitly provided, since a monitor cannot export any variables. If a process calls an entry of a monitor while another process is executing in the monitor, the calling process will be blocked and not allowed in the monitor until no other process is executing in the monitor.

A declarationInMonitor is one of the following:

a. constantDeclaration
b. variableDeclaration
c. typeDeclaration
d. variableBinding
e. conditionDeclaration
f. collectionDeclaration
g. procedureDeclaration
h. functionDeclaration
i. converterDeclaration
j. assert ["(" expn ")"]

Modules and monitors cannot be declared inside a monitor. A monitor cannot contain a nested process.

Monitors can be separately compiled; see "Separate Compilation".

CONDITIONS

Euclid-C introduces conditions.

A conditionDeclaration is one of:

a. var id ":" [priority] condition
b. var id ":" array indexType of [priority] condition

The only place a condition can be declared is as a field of a monitor. The only allowed use of conditions is in the "wait" and "signal" statements and in the "empty" built-in function. Conditions cannot be assigned, compared or passed as parameters. Arrays of conditions are allowed. Conditions must be imported using "var".

Two new statements are introduced:

```
wait "(" conditionVar [","  priorityValue] ")"
signal "(" conditionVar ")"
```

Where a conditionVar is:

```
conditionId ["(" expn ")"]
```

The wait and signal statements each specify a conditionVar. Each of these must be a conditionId or a subscripted condition array. These statements can appear only in monitors, but not in a monitor's initially procedure.

When a process executes a wait statement for condition C it is blocked and is removed from the monitor. When a process executes a signal statement for condition C, one of the processes (if there are any) waiting for condition C is unblocked and allowed immediately to continue executing the monitor. The signalling process is temporarily removed from the monitor and is not allowed to continue execution until no processes are in the monitor. If no processes were waiting for condition C, the only effect of the signal statement is that the signalling process may be removed from the monitor. The signalling process cannot in general know whether other processes have entered the monitor before the signaller continues in the monitor.

If the condition variable is declared with the "priority" option, the wait statement must specify a priority value; otherwise the priority value is not allowed in wait. The priority value is a SignedInt expression that must evaluate to a nonnegative integer value. The processes waiting for a priority condition are ranked in order of their specified priority values, and the process with the smallest priority value is the first to be unblocked by a signal statement.

In the case of processes waiting for non-priority conditions, or waiting with identical priorities for a priority condition, the scheduling is "fair", meaning that a particular waiting process will eventually be unblocked given enough signals on the condition.

A predefined function named "empty" accepts a condition as a parameter. It returns the Boolean value "true" if no processes are waiting for the condition, otherwise "false". Like wait and signal, "empty" can appear only inside a monitor, but not in the initially procedure of a monitor.

The variables in a monitor represent its state. For example, if a monitor allocates a single resource, only one variable inside the monitor is needed and it can be declared as Boolean. When this variable is true, it represents the state in which the resource is available, when false it represents the state of being allocated. When a process enters the monitor and finds that it does not have the desired state, the process leaves the monitor and becomes blocked by executing a wait statement on a condition. The condition corresponds to the state that the process is waiting for. Suppose a process enters a monitor and changes its state to a state that may be waited for by other processes. The process should execute a signal statement for the condition corresponding to the new state. If there are processes waiting for this state transition, then they will be blocked on the condition, and one of them will immediately resume execution in the monitor. Because of this immediate resumption, the signalled process knows the monitor is in the desired state, without

testing monitor variables. The signalling process is allowed to continue executing only when no other processes are in the monitor. If no processes were waiting on the condition, the only effect of the signal statement is to temporarily remove the signaller from the monitor.

As specified by Hoare, monitors and conditions are intended to be used in the following manner. The programmer should associate with the monitor's variables a consistency criterion. The consistency criterion is a Boolean expression that should be true between monitor activations, or whenever a process enters or leaves a monitor. Hence, the programmer should see that it is made true before each signal or wait statement in the monitor and before each return from an entry of the monitor. The programmer should also associate a Boolean expression, call it Ei, with each condition Ci. The expression Ei should be true whenever a signal is executed for condition Ci. A process that is unblocked after waiting for a condition knows that Ei is true because the signalled process (not the signalling process) executes first. (The consistency criterion and each Ei for a condition do not necessarily appear as executable code in the monitor.) In general, when a process changes the monitor's state so that one of the awaited relations Ei becomes true, the corresponding condition Ci should be signalled.

THE BUSY STATEMENT

A statement is introduced to allow simulation using timing delays:

```
busy "(" time ")"
```

The "time" must be a nonnegative SignedInt expression. The busy statement can be understood in terms of simulated time recorded by a system clock. This clock is set to zero at the beginning of execution of a program. With the exception of the busy statement (or wait statements causing an indirect delay for a busy statement), statements take negligible simulated time to execute. When the programmer wants to specify that a certain action takes time to complete, the busy statement is used. The process that executes the busy statement is delayed until the system clock ticks (counts off) the specified number of time units.

III. SEPARATE COMPILATION

This section describes the extensions made to Euclid-C to allow separate compilation of procedures, functions, modules and monitors.

EXTERNAL DECLARATIONS

Procedures, functions, modules and monitors may be declared "external", which means that they are to be separately compiled and joined with the program at link time.

An externalProcedureDeclaration is:

```
procedure id ["(" [var] id ":" parameterType
        {"," [var] id ":" parameterType} ")"] "="
   external
```

An externalFunctionDeclaration is:

```
function id ["(" id ":" parameterType
        {"," id ":" parameterType} ")"]
      returns id ":" resultType "="
   external
```

An externalModuleDeclaration is:

```
var id ":"
   external module
      [imports "(" [var] id {"," [var] id} ")" ";"]
      [exports "(" id {"," id} ")" ";"]
      {declarationInExternalModule ";"}
   end module
```

A declarationInExternalModule is one of:

a. constantDeclaration
b. typeDeclaration
c. collectionDeclaration
d. externalProcedureDeclaration
e. externalFunctionDeclaration

An externalMonitorDeclaration is:

```
var id ":"
   external monitor
      [imports "(" [var] id {"," [var] id} ")" ";"]
      [exports "(" id {"," id} ")" ";"]
      {declarationInExternalMonitor ";"}
   end monitor
```

A declarationInExternalMonitor is one of:

a. constantDeclaration
b. typeDeclaration
c. collectionDeclaration
d. externalProcedureDeclaration
e. externalFunctionDeclaration

An external declaration can appear in place of the real declaration and specifies that the corresponding procedure, function, module or monitor is to be compiled separately.

Processes and initially procedures of modules cannot be declared external. Procedures and functions which are entries of a monitor cannot be declared external except as part of an external monitor declaration. Constants declared within an external module or monitor must have manifest values.

COMPILATIONS

A compilation can consist of a main program (see "Programs") or a separate compilation.

A separateCompilation is:

{separateDeclaration ";"}

Each separateDeclaration is one of the following:

a. constantDeclaration
b. typeDeclaration
c. collectionDeclaration
d. procedureDeclaration
e. functionDeclaration
f. moduleDeclaration
g. monitorDeclaration

Each separateDeclaration can be a manifest constant declaration, a type declaration, a collection declaration, a procedure or function declared as "external" in another compilation, or a module or monitor declared as "external" in another compilation.

Separately compiled procedures, functions, modules and monitors can be linked to form a complete program. Constants, types, collections and variables are not linked across compilations; it is the programmer's responsibility to insure that the number and type of formal parameters agree across compilations.

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 12,1 (February 1977).

2. Hoare, C.A.R., Monitors: An Operating System Structuring Concept. Comm. ACM 17,10 (October 1974), 549-557.

APPENDIX 1.
COLLECTED SYNTAX OF EUCLID-S AND EUCLID-C

The collected syntax of Euclid-S is given first. Throughout the following, {item} means zero or more of the item, and [item] means the item is optional.

The following abbreviations are used:
id for identifier
expn for expression
typeDefn for typeDefinition

A program is:

```
moduleDeclaration ";"
```

A moduleDeclaration is:

```
var id ":"
   module
      [imports "(" [var] id {"," [var] id} ")" ";"]
      [exports "(" id {"," id} ")" ";"]
      [[not] checked ";"]
      {declarationInModule ";"}
      [initially
          procedureBody ";"]
   end module
```

A declarationInModule is one of the following:

a. constantDeclaration
b. variableDeclaration
c. typeDeclaration
d. variableBinding
e. moduleDeclaration
f. collectionDeclaration
g. procedureDeclaration
h. functionDeclaration
i. converterDeclaration
j. assert ["(" expn ")"]

A constantDeclaration is one of:

a. [pervasive] const id ":=" expn
b. [pervasive] const id ":" typeDefn ":="
   "(" manifestExpn {"," manifestExpn} ")"

A variableDeclaration is:

```
[register] var id ["(" at manifestExpn ")"] ":" typeDefn
        [":=" expn]
```

A typeDeclaration is:

```
[pervasive] type id "=" typeBody
```

The typeBody is one of:

a. typeDefn
b. forward

A typeDefn is one of the following:

a. standardType
b. manifestConstant ".." manifestExpn
c. [packed] array indexType of typeDefn
d. set of baseType
e. [packed] recordType
f. pointerType
g. namedType

Note: The following are standardTypes of Euclid-S: SignedInt, UnsignedInt, LongInt, ShortInt, Boolean, Char, StorageUnit, AddressType.

A recordType is:

```
record
   {var id ":" typeDefn ";"}
end record
```

A pointerType is:

```
"^" collectionId
```

A namedType is:

```
[moduleId "."] typeId
```

A variableBinding is one of:

a. bind [var] id to variable
b. bind "(" [var] id to variable
        {"," [var] id to variable} ")"

A collectionDeclaration is:

```
var id ":" collection of typeDefn
```

A procedureDeclaration is:

```
procedure id ["(" [var] id ":" parameterType
        {"," [var] id ":" parameterType} ")"] "="
   procedureBody
```

A functionDeclaration is:

```
function id ["(" id ":" parameterType
        {"," id ":" parameterType} ")"]
      returns id ":" resultType "="
   procedureBody
```

A parameterType is one of:

a. typeDefn
b. [packed] array manifestConstant ".." parameter of
   typeDefn
c. universal

A procedureBody is:

```
[imports "(" [var] id {"," [var] id} ")" ";"]
begin
   [[not] checked ";"]
   {declarationInRoutine ";"}
   {statement ";"}
end [id]
```

A declarationInRoutine is one of:

a. constantDeclaration
b. variableDeclaration
c. typeDeclaration
d. variableBinding
e. collectionDeclaration
f. converterDeclaration
g. assert ["("expn")"]

A converterDeclaration is:

```
converter id "(" typeId ")" returns typeId
```

A statement is one of:

a. variable ":=" expn
b. [moduleId"."] procedureId ["(" expn {"," expn} ")"]
c. assert ["("expn")"]
d. return ["("expn")"]
e. if expn then

```
            {statement ";"}
         {elseif expn then
            {statement ";"}}
         [else
            {statement ";"}]
         end if
   f.    loop
            {statement ";"}
         end loop
   g.    exit [when expn]
   h.    case expn of
            {manifestExpn {"," manifestExpn} "=>"
               {statement ";"}
               end manifestExpn ";"}
            [otherwise "=>"
               {statement ";"}]
         end case
   i.    begin
            {declarationInRoutine ";"}
            {statement ";"}
         end
   j.    collectionId "." New "(" variable ")"
   k.    collectionId "." Free "(" variable ")"
```

A variable is:

```
   a.    moduleId "." id
   b.    id {componentSelector}
```

A componentSelector is one of:

```
   a.    "(" expn ")"
   b.    "." id
```

An expn is one of the following:

```
   a.    variable
   b.    literalConstant
   c.    setTypeId "(" elementList ")"
   d.    collectionId "." nil
   e.    [moduleId "."] functionId ["(" expn {"," expn} ")"]
   f.    [moduleId "."] converterId "(" expn ")"
   g.    "(" expn ")"
   h.    "-" expn
   i.    expn arithmeticOperator expn
   j.    expn comparisonOperator expn
   k.    not expn
   l.    expn booleanOperator expn
   m.    expn setOperator expn
```

Note: The arithmeticOperators are +, -, * (multiply), div (integer divide) and mod (integer remainder). The

comparisonOperators are <, >, =, <=, >= and "not =". The booleanOperators are "and" (intersection), "or" (union) and -> (implication). The setOperators are * (set intersection), + (set union), - (set difference), <= and >= (set inclusion), and "in" and "not in" (element containment).

The order of precedence is among the following classes of operators (most binding first):

1. unary -
2. *, div, mod
3. +, -
4. <, >, =, <=, >=, not =, in, not in
5. not
6. and
7. or
8. ->

A setElementList is one of:

a. [expn {"," expn}]
b. all

An includeStatement is:

include stringLiteral ";"

Note: Include statements can appear anywhere in a program.

The following changes and additions are made to form Euclid-C:

A moduleDeclaration is:

```
var id ":"
   module
      [imports "(" [var] id {"," [var] id} ")" ";"]
      [exports "(" id {"," id} ")" ";"]
      [[not] checked ";"]
      {declarationInModule ";"}
      [initially
         procedureBody ";"]
      {process id ["(" memoryRequirement ")"]
         procedureBody ";"}
   end module
```

A declarationInModule is one of the following:

a. constantDeclaration
b. variableDeclaration
c. typeDeclaration
d. variableBinding
e. moduleDeclaration
f. monitorDeclaration
g. collectionDeclaration
h. procedureDeclaration
i. functionDeclaration
j. converterDeclaration
k. assert ["(" expn ")"]

A monitorDeclaration is:

```
var id ":"
   monitor
      [imports "(" [var] id {"," [var] id} ")" ";"]
      [exports "(" id {"," id} ")" ";"]
      [[not] checked ";"]
      {declarationInMonitor ";"}
      [initially
         procedureBody ";"]
   end monitor
```

A declarationInMonitor is one of the following:

a. constantDeclaration
b. variableDeclaration
c. typeDeclaration
d. variableBinding
e. conditionDeclaration
f. collectionDeclaration
g. procedureDeclaration
h. functionDeclaration

```
i.  converterDeclaration
j.  assert ["(" expn ")"]
```

A conditionDeclaration is one of:

```
a.  var id ":" [priority] condition
b.  var id ":" array indexType of [priority] condition
```

A statement is one of:

```
a.  variable ":=" expn
b.  [moduleId"."] procedureId ["(" expn {"," expn}")"]
c.  assert ["("expn")"]
d.  return ["("expn")"]
e.  if expn then
        {statement ";"}
    {elseif expn then
        {statement ";"}}
    [else
        {statement ";"}]
    end if
f.  loop
        {statement ";"}
    end loop
g.  exit [when expn]
h.  case expn of
        {manifestExpn {"," manifestExpn} "=>"
            {statement ";"}}
        [otherwise "=>"
            {statement ";"}]
    end case
i.  begin
        {declarationInRoutine ";"}
        {statement ";"}
    end
j.  collectionId "." New "(" variable ")"
k.  collectionId "." Free "(" variable ")"
l.  wait "(" conditionVar ["," priorityValue] ")"
m.  signal "(" conditionVar ")"
n.  busy "(" time ")"
```

A conditionVar is:

```
conditionId ["(" expn ")"]
```

The following extensions allow separate compilation of procedures, functions, modules and monitors:

An externalProcedureDeclaration is:

```
procedure id ["(" [var] id ":" parameterType
        {"," [var] id ":" parameterType} ")"] "="
   external
```

An externalFunctionDeclaration is:

```
function id ["(" id ":" parameterType
        {"," id ":" parameterType} ")"]
      returns id ":" resultType "="
   external
```

An externalModuleDeclaration is:

```
var id ":"
   external module
      [imports "(" [var] id {"," [var] id} ")" ";"]
      [exports "(" id {"," id} ")" ";"]
      {declarationInExternalModule ";"}
   end module
```

A declarationInExternalModule is one of:

a. constantDeclaration
b. typeDeclaration
c. collectionDeclaration
d. externalProcedureDeclaration
e. externalFunctionDeclaration

An externalMonitorDeclaration is:

```
var id ":"
   external monitor
      [imports "(" [var] id {"," [var] id} ")" ";"]
      [exports "(" id {"," id} ")" ";"]
      {declarationInExternalMonitor ";"}
   end monitor
```

A declarationInExternalMonitor is one of:

a. constantDeclaration
b. typeDeclaration
c. collectionDeclaration
d. externalProcedureDeclaration
e. externalFunctionDeclaration

Note: An external declaration can appear in place of the real declaration anywhere in a program.

A separateCompilation is:

```
{separateDeclaration ";"}
```

Each separateDeclaration is one of the following:

a. constantDeclaration
b. typeDeclaration
c. collectionDeclaration
d. procedureDeclaration
e. functionDeclaration
f. moduleDeclaration
g. monitorDeclaration

APPENDIX 2.
KEYWORDS AND PREDEFINED IDENTIFIERS OF EUCLID-S AND EUCLID-C

The following are reserved words of Euclid. These must not be used as identifiers in Euclid-S and Euclid-C programs. Those which are not in the Euclid-S subset are marked with an *.

*abstraction	*aligned	all	and
*any	array	assert	at
begin	bind	*bits	*bound
case	*checkable	checked	*code
collection	const	converter	*counted
*decreasing	*default	*dependent	div
else	elseif	end	exit
exports	*finally	*for	forward
*from	function	if	imports
in	include	initially	*inline
*invariant	loop	machine	mod
not	of	or	otherwise
packed	parameter	pervasive	*post
*pre	procedure	*readonly	record
return	returns	set	then
*thus	to	type	*unknown
var	when	*with	*xor

The following are additional reserved words of Euclid-S and Euclid-C. These also must not be used as identifiers in Euclid-S and Euclid-C programs.

busy	condition	empty	monitor
priority	process	register	signal
wait			

The following are predefined identifiers of Euclid. In general, these are pervasive and must not be redeclared in Euclid programs. Those which are not in the Euclid-S subset are marked with an *.

*Abs	address	AddressType	*alignment
*BaseType	Boolean	Char	Chr
*ComponentType	false	*first	Free
*Index	*IndexType	*Integer	*itsTag
*ItsType	*last	*Max	*Min
New	nil	*ObjectType	*Odd
Ord	*Pred	*refCount	SignedInt
size	*sizeInBits	StorageUnit	*String
*StringIndex	*stringMaxLength		*Succ
*SystemZone	true	UnsignedInt	

The following are additional predefined identifiers of Euclid-S and Euclid-C. These also must not be redeclared in Euclid-S and Euclid-C programs.

LongInt	ShortInt

APPENDIX 3.
INPUT/OUTPUT IN CONCURRENT EUCLID

This paper presents the standard input/output package for Euclid-S and Euclid-C. The user can access the I/O facility by including in his program the stub input/output module which corresponds to the level of I/O which his program requires. In this way, the user's compiled and linked program will include code only for the I/O facilities required.

The package provides four levels of sophistication, which are called "IO/1" through "IO/4". Each level includes all the facilities of the previous levels plus certain new features. The levels are as follows:

IO/1: Terminal (standard) input and output; Formatted text input/output of integers, characters and strings (Get and Put).

IO/2: Program argument sequential files; Open and close on argument files; Formatted text input/output of integers, characters and strings to files (FGet and FPut); Internal representation input/output of integers, characters and strings to files (Read and Write); End of file detection (EndFile).

IO/3: Temporary and non-argument sequential files (Assign, Deassign, Delete); Program arguments (FetchArg).

IO/4: Record, array and storage input/output (Read and Write); Random access files (Stat and Seek); Error detection (Error).

The procedures and functions of the input/output system are all part of the module "IO" and must be referenced using "IO.". The types and constants which form the interface to the module are global. The user can access the level n facilities of the input/output module by beginning his program with the statement

```
include 'IOn'
```

We now describe the input/output facilities in detail.

IO/1: Terminal Formatted Text I/O

```
pervasive const newLine := $$N
pervasive const endOfFile := $$E
pervasive const maxStringLength :=
                { Implementation defined; >= 128 }
```

Strings read and written by the input/output routines may be up to maxStringLength characters in length.

```
IO.PutChar (c: Char)
```

Prints the character c on the terminal.

IO.PutInt (i: SignedInt, w: SignedInt)
Prints the integer i on the terminal, right justified in a field of w characters. Leading blanks are supplied to fill the field. If w is an insufficient width, the value is printed in the minimum possible width with no leading blanks. In particular, if w is 1 then the exact number of characters needed is used. The specified width must be greater than zero and less than maxStringLength.

IO.PutString (s: packed array 1..parameter of Char)
Prints the string s on the terminal. The string must be terminated by an endOfFile character ('$E'), which is not output. It can contain embedded newLines ('$N') if desired. (Note: An endOfFile character ($$E) can be output using PutChar.)

IO.GetChar (var c:Char)
Gets a the next input character from the terminal. End of file is indicated by a return of endOfFile ($$E).

IO.GetInt (var i: SignedInt)
Gets an integer from the terminal. The input must consist of any number of optional blanks, tabs and newlines, followed by an optional minus sign, followed by any number of decimal digits.

IO.GetString (var s: packed array 1..parameter of Char)
Gets a line of character input from the terminal. The string returned is ended with the newLine character ('$N') followed by an endOfFile character ('$E'). The returned string may be up to maxStringLength characters in length. End of file is indicated by returning a string containing endOfFile ('$E') as the first character.

IO/2: Sequential Argument File I/O

```
pervasive const stdInput := -2
pervasive const stdOutput := -1
pervasive const stdError := 0
pervasive const maxArgs := { Implementation defined; >= 9 }
pervasive const maxFiles :=
                { Implementation defined; >= maxArgs+5 }
type File = stdError..maxFiles
```

Concurrent Euclid input/output refers to files using a file number. Certain file numbers are preassigned as follows: -2 refers to the terminal input; -1 is the terminal output; 0 is the standard diagnostic output. The file numbers 1..maxArgs refer to the program arguments. The remaining file numbers (maxArgs+1..maxFiles) can be dynamically assigned to files using the "IO.Assign" operation; see "IO/3".

```
pervasive const inFile := 0
pervasive const outFile := 1
pervasive const inOutFile := 2
type FileMode = inFile..inOutFile
```

Files can be opened for input, output, or input/output using modes inFile, outFile and inOutFile respectively. (Note: The input/output mode is not available under Unix V6.)

```
IO.Open (f: File, m: FileMode)
IO.Close (f: File)
```
With the exception of terminal input/output and the standard diagnostic output, files must be opened before they are used and closed before the program returns. Open opens an existing file for the operations specified by the mode. If the opened file does not exist, it is created. The file number specified must be a preassigned file number or a file number returned from a call to "IO.Assign"; see "IO/3".

```
IO.FPutChar (f: File, c: Char)
IO.FPutInt (f: File, i: SignedInt, w: SignedInt)
IO.FPutString (f: File, s: packed array 1..parameter of Char)
IO.FGetChar (f: File, var c: Char)
IO.FGetInt (f: File, var i: SignedInt)
IO.FGetString (f: File, var s: packed array 1..parameter of Char)
```
These operations are identical to the terminal input/output operations of IO/1 except that the put or get is done on the specified file.

```
IO.WriteChar (f: File, c: Char)
```
Identical to FPutChar.

```
IO.WriteInt (f: File, i: SignedInt)
```
Writes the internal representation of integer i to the specified file.

```
IO.WriteString (f:File, s: packed array 1..parameter of Char)
```
Identical to FPutString.

```
IO.ReadChar (f: File, var c: Char)
```
Identical to FGetChar.

```
IO.ReadInt (f: File, var i: SignedInt)
```
Reads an integer in internal representation from the specified file into i.

```
IO.ReadString (f: File, var s: packed array 1..parameter of Char)
```
Identical to FPutString.

```
IO.EndFile (f: File)
```
A function which returns true if the last operation on the specified input file encountered end of file and false otherwise.

IO/3: Temporary and Non-argument Files

```
pervasive const maxArgLength :=
                { Implementation defined; >= 32 }
```

File names and arguments to a program may be up to maxArgLength characters in length.

IO.Assign (var f: File, s: packed array 1..parameter of Char)
A file number is assigned to the file name supplied in s. The file name is given as a string terminated by the endOfFile character ('$E'), which is not part of the name. Before the file can be used it must be opened using "IO.Open".

IO.Deassign (f: File)
The specified file number is freed for assignment to another file name. An open file cannot be deassigned.

IO.Delete (f: File)
The specified file is destroyed. An open file cannot be deleted. Note that a program can have temporary files using "IO.Assign" and "IO.Delete".

IO.FetchArg (n: 1..maxArgs, var s: packed array 1..parameter of Char)
The program argument specified by "n" is returned in string s. The returned string is terminated by the endOfFile character ('$E') and may be up to maxArgLength characters in length.

IO/4: Structure Input/Output and Random Access Files

IO.Write (f: File, u: universal, n: SignedInt)
The number of StorageUnits specified by "n" are written to the file from u. Write can be used to write out whole arrays and records using a call of the form "IO.Write (f, v, v.size)". The value of n must be positive or zero.

IO.Read (f: File, var u: universal, n: SignedInt)
The number of StorageUnits specified by "n" are read from the file into u. Read can be used to read in whole arrays and records using a call of the form "IO.Read (f, v, v.size)". The value of n must be positive or zero.

```
type FileIndex =
     record
        var b: SignedInt
        var c: SignedInt
     end record
IO.Stat (f: File, var x: FileIndex)
IO.Seek (f: File, x: FileIndex)
```

These operations provide random access input/output by allowing the program to sense a file position, represented as two integers, and reset the file to a remembered position. Stat returns the current position of the specified file in x.b and x.c (conceptually, the "block number" and "character

number within block"). Seek sets the current position of the specified file to the position specified by the values of x.b and x.c. In both cases, the values of b and c are integers whose meaning is implementation-dependent. The programmer should not assume that the values of b and c are restricted to any particular range. (Note: "IO.Stat" and "IO.Seek" are not supported under Unix V6.)

IO.Error (f: File)
A function which returns true if the last operation on the specified file encountered an error and false otherwise.

Interfacing to Unix*

The input/output package is based on standard Unix input/output and is designed to be interfaced to Unix with a minimum of overhead. The Unix implementation is written in C and uses only facilities of the C "stdio" package. This implementation can be compiled unchanged under both V6 and V7 Unix.

Using Euclid-S Standard I/O with Toronto Euclid

Euclid-S programs may be compiled using Toronto Euclid by wrapping the entire Euclid-S program (including all "include" statements) in a module type, thus:

```
type EuclidS = module

include 'IOn'
{ The Euclid-S program goes here }

end module {EuclidS};
```

The program can then be compiled using the command "euc -E prog.e". The "-E" option specifies that the program is to be linked with the Euclid-S standard I/O library.

The Euclid-S input/output facility has been installed in the Toronto Euclid library and can be referenced directly using

```
include '/lib/euclid/IOn'
```

if desired. In this way, the user need not have his own copies or links to the input/output package.

* "Unix" is a trademark of Bell Telephone Laboratories.

University of Toronto
Computer Systems Research Group

BIBLIOGRAPHY OF CSRG TECHNICAL REPORTS+

* CSRG-1 EMPIRICAL COMPARISON OF LR(k) AND PRECEDENCE PARSERS
J.J. Horning and W.R. Lalonde, September 1970
[ACM SIGPLAN Notices, November 1970]

* CSRG-2 AN EFFICIENT LALR PARSER GENERATOR
W.R. Lalonde, February 1971
[M.A.Sc. Thesis, EE 1971]

* CSRG-3 A PROCESSOR GENERATOR SYSTEM
J.D. Gorrie, February 1971
[M.A.Sc. Thesis, EE 1971]

* CSRG-4 DYLAN USER'S MANUAL
P.E. Bonzon, March 1971

CSRG-5 DIAL - A PROGRAMMING SYSTEM FOR INTERACTIVE ALGEBRAIC MANIPULATION
Alan C.M. Brown and J.J. Horning, March 1971

CSRG-6 ON DEADLOCK IN COMPUTER SYSTEMS
Richard C. Holt, April 1971
[Ph.D. Thesis, Dept. of Computer Science,
Cornell University, 1971]

CSRG-7 THE STAR-RING SYSTEM OF LOOSELY COUPLED DIGITAL DEVICES
John Neill Thomas Potvin, August 1971
[M.A.Sc. Thesis, EE 1971]

* CSRG-8 FILE ORGANIZATION AND STRUCTURE
G.M. Stacey, August 1971

CSRG-9 DESIGN STUDY FOR A TWO-DIMENSIONAL COMPUTER-ASSISTED ANIMATION SYSTEM
Kenneth B. Evans, January 1972
[M.Sc. Thesis, DCS, 1972]

* CSRG-10 HOW A PROGRAMMING LANGUAGE IS USED
William Gregg Alexander, February 1972
[M.Sc. Thesis, DCS 1971; Computer, v.8, n.11, November 1975]

* CSRG-11 PROJECT SUE STATUS REPORT
J.W. Atwood (ed.), April 1972

+ Abbreviations:
DCS - Department of Computer Science, University of Toronto
EE - Department of Electrical Engineering, University of Toronto

* - Out of print

* CSRG-12 THREE DIMENSIONAL DATA DISPLAY WITH HIDDEN LINE REMOVAL
Rupert Bramall, April 1972
[M.Sc. Thesis, DCS, 1971]

* CSRG-13 A SYNTAX DIRECTED ERROR RECOVERY METHOD
Lewis R. James, May 1972
[M.Sc. Thesis, DCS, 1972]

CSRG-14 THE USE OF SERVICE TIME DISTRIBUTIONS IN SCHEDULING
Kenneth C. Sevcik, May 1972
[Ph.D. Thesis, Committee on Information Sciences, University of Chicago, 1971; JACM, January 1974]

CSRG-15 PROCESS STRUCTURING
J.J. Horning and B. Randell, June 1972
[ACM Computing Surveys, March 1972]

CSRG-16 OPTIMAL PROCESSOR SCHEDULING WHEN SERVICE TIMES ARE HYPEREXPONENTIALLY DISTRIBUTED AND PREEMPTION OVERHEAD IS NOT NEGLIGIBLE
Kenneth C. Sevcik, June 1972
[Proceedings of the Symposium on Computer-Communication, Networks and Teletraffic, Polytechnic Institute of Brooklyn, 1972]

* CSRG-17 PROGRAMMING LANGUAGE TRANSLATION TECHNIQUES
W.M. McKeeman, July 1972

CSRG-18 A COMPARATIVE ANALYSIS OF SEVERAL DISK SCHEDULING ALGORITHMS
C.J.M. Turnbull, September 1972

CSRG-19 PROJECT SUE AS A LEARNING EXPERIENCE
K.C. Sevcik *et al*, September 1972
[Proceedings AFIPS Fall Joint Computer Conference, v. 41, December 1972]

* CSRG-20 A STUDY OF LANGUAGE DIRECTED COMPUTER DESIGN
David B. Wortman, December 1972
[Ph.D. Thesis, Computer Science Department, Stanford University, 1972]

CSRG-21 AN APL TERMINAL APPROACH TO COMPUTER MAPPING
R. Kvaternik, December 1972
[M.Sc. Thesis, DCS, 1972]

* CSRG-22 AN IMPLEMENTATION LANGUAGE FOR MINICOMPUTERS
G.G. Kalmar, January 1973
[M.Sc. Thesis, DCS, 1972]

CSRG-23 COMPILER STRUCTURE
W.M. McKeeman, January 1973
[Proceedings of the USA-Japan Computer Conference, 1972]

* CSRG-24 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING
J.D. Gannon (ed.), March 1973

CSRG-25 THE INVESTIGATION OF SERVICE TIME DISTRIBUTIONS
Eleanor A. Lester, April 1973
[M.Sc. Thesis, DCS, 1973]

* CSRG-26 PSYCHOLOGICAL COMPLEXITY OF COMPUTER PROGRAMS: AN INITIAL EXPERIMENT
Larry Weissman, August 1973

* CSRG-27 STRUCTURED SUBSETS OF THE PL/I LANGUAGE
Richard C. Holt and David B. Wortman, October 1973

* CSRG-28 ON REDUCED MATRIX REPRESENTATION OF LR(k) PARSER TABLES
Marc Louis Joliat, October 1973
[Ph.D. Thesis, EE 1973]

* CSRG-29 A STUDENT PROJECT FOR AN OPERATING SYSTEMS COURSE
B. Czarnik and D. Tsichritzis (eds.), November 1973

* CSRG-30 A PSEUDO-MACHINE FOR CODE GENERATION
Henry John Pasko, December 1973
[M.Sc. Thesis, DCS 1973]

* CSRG-31 AN ANNOTAED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING
J.D. Gannon (ed.), Second Edition, March 1974

* CSRG-32 SCHEDULING MULTIPLE RESOURCE COMPUTER SYSTEMS
E.D. Lazowska, May 1974
[M.Sc. Thesis, DCS, 1974]

* CSRG-33 AN EDUCATIONAL DATA BASE MANAGEMENT SYSTEM
F. Lochovsky and D. Tsichritzis, May 1974
[INFOR, 14 (3), pp.270-278, 1976]

* CSRG-34 ALLOCATING STORAGE IN HIERARCHICAL DATA BASES
P. Bernstein and D. Tsichritzis, May 1974
[Information Systems Journal, v.1, pp.133-140]

* CSRG-35 ON IMPLEMENTATION OF RELATIONS
D. Tsichritzis, May 1974

* CSRG-36 SIX PL/I COMPILERS
D.B. Wortman, P.J. Khaiat, and D.M. Lasker, August 1974
[Software Practice and Experience, v.6, n.3, July-Sept. 1976]

* CSRG-37 A METHODOLOGY FOR STUDYING THE PSYCHOLOGICAL COMPLEXITY OF COMPUTER PROGRAMS
Laurence M. Weissman, August 1974
[Ph.D. Thesis, DCS, 1974]

* CSRG-38 AN INVESTIGATION OF A NEW METHOD OF CONSTRUCTING SOFTWARE
David M. Lasker, September 1974
[M.Sc. Thesis, DCS, 1974]

CSRG-39 AN ALGEBRAIC MODEL FOR STRING PATTERNS
Glenn F. Stewart, September 1974
[M.Sc. Thesis, DCS, 1974]

* CSRG-40 EDUCATIONAL DATA BASE SYSTEM USER'S MANUAL
J. Klebanoff, F. Lochovsky, A. Rozitis, and
D. Tsichritzis, September 1974

* CSRG-41 NOTES FROM A WORKSHOP ON THE ATTAINMENT OF
RELIABLE SOFTWARE
David B. Wortman (ed.), September 1974

* CSRG-42 THE PROJECT SUE SYSTEM LANGUAGE REFERENCE MANUAL
B.L. Clark and F.J.B. Ham, September 1974

* CSRG-43 A DATA BASE PROCESSOR
E.A. Ozkarahan, S.A. Schuster and K.C. Smith,
November 1974 [Proceedings National Computer
Conference 1975, v.44, pp.379-388]

* CSRG-44 MATCHING PROGRAM AND DATA REPRESENTATION TO A
COMPUTING ENVIRONMENT
Eric C.R. Hehner, Novemver 1974
[Ph.D. Thesis, DCS, 1974]
See Computer, Vol.9, No.9, August 1976, pp.65-70.

* CSRG-45 THREE APPROACHES TO RELIABLE SOFTWARE; LANGUAGE DESIGN,
DYADIC SPECIFICATIONS, COMPLEMENTARY SEMANTICS
J.E. Donahue, J.D. Gannon, J.V. Guttag and
J.J. Horning, December 1974

CSRG-46 THE SYNTHESIS OF OPTIMAL DECISION TREES FROM
DECISION TABLES
Helmut Schumacher, December 1974
[M.Sc. Thesis, DCS, 1974; CACM, v.19, n.6, June 1976]

* CSRG-47 LANGUAGE DESIGN TO ENHANCE PROGRAMMING RELIABILITY
John D. Gannon, January 1975
[Ph.D. Thesis, DCS, 1975]

* CSRG-48 DETERMINISTIC LEFT TO RIGHT PARSING
Christopher J.M. Turnbull, January 1975
[Ph.D. Thesis, EE, 1974]

* CSRG-49 A NETWORK FRAMEWORK FOR RELATIONAL IMPLEMENTATION
D. Tsichritzis, February 1975 [in Data Base Description,
Dongue and Nijssen (eds.), North Holland Publishing Co.]

* CSRG-50 A UNIFIED APPROACH TO FUNCTIONAL DEPENDENCIES AND RELATIONS
P.A. Bernstein, J.R. Swenson and D.C. Tsichritzis
February 1975 [Proceedings of the ACM SIGMOD Conference, 1975]

* CSRG-51 ZETA: A PROTOTYPE RELATIONAL DATA BASE MANAGEMENT SYSTEM
M. Brodie (ed). February 1975 [Proceedings Pacific ACM Conference, 1975]

* CSRG-52 AUTOMATIC GENERATION OF SYNTAX-REPAIRING AND PARAGRAPHING PARSERS
David T. Barnard, March 1975
[M.Sc. Thesis, DCS, 1975]

* CSRG-53 QUERY EXECUTION AND INDEX SELECTION FOR RELATIONAL DATA BASES
J.H. Gilles Farley and Stewart A. Schuster, March 1975

CSRG-54 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING
J.V. Guttag (ed.), Third Edition, April 1975

CSRG-55 STRUCTURED SUBSETS OF THE PL/1 LANGUAGE
Richard C. Holt and David B. Wortman, May 1975

* CSRG-56 FEATURES OF A CONCEPTUAL SCHEMA
D. Tsichritzis, June 1975 [Proceedings Very Large Data Base Conference, 1975]

* CSRG-57 MERLIN: TOWARDS AN IDEAL PROGRAMMING LANGUAGE
Eric C.R. Hehner, July 1975
see Acta Informatica Col.10, No.3, pp.229-243, 1978

CSRG-58 ON THE SEMANTICS OF THE RELATIONAL DATA MODEL
Hans Albrecht Schmid and J. Richard Swenson,
July 1975 [Proceedings of the ACM SIGMOD Conference, 1975]

* CSRG-59 THE SPECIFICATION AND APPLICATION TO PROGRAMMING OF ABSTRACT DATA TYPES
John V. Guttag, September 1975
[Ph.D. Thesis, DCS, 1975]

* CSRG-60 NORMALIZATION AND FUNCTIONAL DEPENDENCIES IN THE RELATIONAL DATA BASE MODEL
Phillip Alan Bernstein, October 1975
[Ph.D. Thesis, DCS, 1975]

* CSRG-61 LSL: A LINK AND SELECTION LANGUAGE
D. Tsichritzis, November 1975 [Proceedings ACM SIGMOD Conference, 1976]

* CSRG-62 COMPLEMENTARY DEFINITIONS OF PROGRAMMING LANGUAGE SEMANTICS
James E. Donahue, November 1975
[Ph.D. Thesis, DCS, 1975]

CSRG-63 AN EXPERIMENTAL EVALUATION OF CHESS PLAYING HEURISTICS
Lazlo Sugar, December 1975
[M.Sc. Thesis, DCS, 1975]

CSRG-64 A VIRTUAL MEMORY SYSTEM FOR A RELATIONAL ASSOCIATIVE PROCESSOR
S.A. Schuster, E.A. Ozkarahan, and K.C. Smith,
February 1976 [Proceedings National Computer Conference 1976, v.45, pp.855-862]

CSRG-65 PERFORMANCE EVALUATION OF A RELATIONAL ASSOCIATIVE PROCESSOR
E.A. Ozkarahan, S.A. Schuster, and K.C. Sevcik,
February 1976 [ACM Transactions on Database Systems, v.1, n:4, December 1976]

CSRG-66 EDITING COMPUTER ANIMATED FILM
Michael D. Tilson, February 1976
[M.Sc. Thesis, DCS, 1975]

CSRG-67 A DIAGRAMMATIC APPROACH TO PROGRAMMING LANGUAGE SEMANTICS
James R. Cordy, March 1976
[M.Sc. Thesis, DCS, 1976]

* CSRG-68 A SYNTHETIC ENGLISH QUERY LANGUAGE FOR A RELATIONAL ASSOCIATIVE PROCESSOR
L. Kerschberg, E.A. Ozkarahan, and J.E.S. Pacheco,
April 1976

CSRG-69 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING
D. Barnard and D. Thompson (eds.), Fourth Edition,
May 1976

* CSRG-70 A TAXONOMY OF DATA MODELS
L. Kerschberg, A. Klug, and D.Tsichritzis, May 1976
[Proceedings Very Large Data Base Conference, 1976]

* CSRG-71 OPTIMIZATION FEATURES FOR THE ARCHITECTURE OF A DATA BASE MACHINE
E.A. Ozkarahan and K.C. Sevcik, May 1976
[ACM Transactions of Database Systems, v.2, n.4, December 1977]

* CSRG-72 THE RELATIONAL DATA BASE SYSTEM OMEGA - PROGRESS REPORT
H.A. Schmid (ed.), P.A. Bernstein (ed.), B. Arlow,
R. Baker and S. Pozgaj, July 1976

CSRG-73 AN ALGORITHMIC APPROACH TO NORMALIZATION OF RELATIONAL DATA BASE SCHEMAS
P.A. Bernstein and C. Beeri, September 1976

* CSRG-74 A HIGH-LEVEL MACHINE-ORIENTED ASSEMBLER LANGUAGE FOR A DATA BASE MACHINE
E.A. Ozkarahan and S.A. Schuster, October 1976

CSRG-75 DO CONSIDERED OD: A CONTRIBUTION TO THE PROGRAMMING CALCULUS
Eric C.R. Hehner, November 1976
Acta Informatica to appear 1979

CSRG-76 SOFTWARE HUT: A COMPUTER PROGRAM ENGINEERING PROJECT IN THE FORM OF A GAME
J.J. Horning and D.B. Wortman, November 1976
[IEEE Transactions on Software Engineering, v.SE-3, n.4, July 1977]

CSRG-77 A SHORT STUDY OF PROGRAM AND MEMORY POLICY BEHAVIOUR
G. Scott Graham, January 1977

CSRG-78 A PANACHE OF DBMS IDEAS
D. Tsichritzis (ed.), February 1977

CSRG-79 THE DESIGN AND IMPLEMENTATION OF AN ADVANCED LALR PARSE TABLE CONSTRUCTOR
David H. Thompson, April 1977
[M.Sc. Thesis, DCS, 1976]

CSRG-80 AN ANNOTATED BIBLIOGRAPHY ON COMPUTER PROGRAM ENGINEERING
D. Barnard (ed.), Fifth Edition, May 1977

CSRG-81 PROGRAMMING METHODOLOGY: AN ANNOTATED BIBLIOGRAPHY FOR IFIP WORKING GROUP 2.3
Sol J. Greenspan and J.J. Horning (eds.), First Edition, May 1977

CSRG-82 NOTES ON EUCLID
edited by W. David Elliot and David T. Barnard, August 1977

CSRG-83 TOPICS IN QUEUEING NETWORK MODELING
edited by G. Scott Graham, July 1977

CSRG-84 TOWARD PROGRAM ILLUSTRATION
Edward Yarwood, September 1977
[M.Sc. Thesis, DCS, 1974]

CSRG-85 CHARACTERIZING SERVICE TIME AND RESPONSE TIME DISTRIBUTIONS IN QUEUEING NETWORK MODELS OF COMPUTER SYSTEMS
Edward D. Lazowska, September 1977
[Ph.D. Thesis, DCS, 1977]

CSRG-86 MEASUREMENTS OF COMPUTER SYSTEMS FOR QUEUEING NETWORK MODELS
Martin G. Kienzle, October 1977
[M.Sc. Thesis, DCS, 1977; Proc. Int. Symp. on Modelling and Performance Evaluation of Computer Systems, Vienna, 1979]

CSRG-87 'OLGA' LANGUAGE REFERENCE MANUAL
B. Abourbih, H. Trickey, D.M. Lewis, E.S. Lee, P.I.P. Boulton, November 1977

CSRG-88 USING A GRAMMATICAL FORMALISM AS A PROGRAMMING LANGUAGE
Brad A. Silverberg, January 1978
[M.Sc. Thesis, DCS, 1978]

CSRG-89 ON THE IMPLEMENTATION OF RELATIONS: A KEY TO EFFICIENCY
Joachim W. Schmidt, January 1978

CSRG-90 DATA BASE MANAGEMENT SYSTEM USER PERFORMANCE
Frederick H. Lochovsky, April 1978
[Ph.D. Thesis, DCS, 1978]

CSRG-91 SPECIFICATION AND VERIFICATION OF DATA BASE SEMANTIC INTEGRITY
Michael Lawrence Brodie, April 1978
[Ph.D. Thesis, DCS, 1978]

CSRG-92 STRUCTURED SOUND SYNTHESIS PROJECT (SSSP): AN INTRODUCTION
by William Buxton, Guy Fedorkow, with Ronald Baecker, Gustav Ciamaga, Leslie Mezei and K.C. Smith, June 1978

CSRG-93 A DEVICE-INDEPENDENT, GENERAL-PURPOSE GRAPHICS SYSTEM IN A MINICOMPUTER TIME-SHARING ENVIRONMENT
William T. Reeves, August 1978
[M.Sc. Thesis, DCS, 1978]

CSRG-94 ON THE AXIOMATIC VERIFICATION OF CONCURRENT ALGORITHMS
Christian Lengauer, August 1978
[M.Sc. Thesis, DCS, 1978]

CSRG-95 PISA: A PROGRAMMING SYSTEM FOR INTERACTIVE PRODUCTION OF APPLICATION SOFTWARE
Rudolf Marty, August 1978

CSRG-96 ADAPTIVE MICROPROGRAMMING AND PROCESSOR MODELING
Walter G. Rosocha
[Ph.D. Thesis, EE, August 1978]

CSRG-97 DESIGN ISSUES IN THE FOUNDATION OF A COMPUTER-BASED TOOL FOR MUSIC COMPOSITION
William Buxton
[M.Sc. Thesis, CSRG, October 1978]

CSRG-98 THEORY OF DATABASE MAPPINGS
Anthony C. Klug
[Ph.D. Thesis, DCS, December 1978]

CSRG-99 HIERARCHICAL COROUTINES: A MECHANISM FOR IMPROVED PROGRAM STRUCTURE
Leonard I. Vanek, February 1979

CSRG-100 TOPICS IN PERFORMANCE EVALUATION
G. Scott Graham (ed.), July 1979

CSRG-101 A PANACHE OF DBMS IDEAS II
F.H. Lochovsky (ed.), May 1979

CSRG-102 A SIMPLE SET THEORY FOR COMPUTING SCIENCE
Eric C.R. Hehner, May 1979

CSRG-103 THE CENTRALIZED ALGORITHM IN DISTRIBUTED SYSTEMS
Ernest J.H. Chang
[Ph.D. Thesis, DCS, July 1979]

CSRG-104 ELIMINATING THE VARIABLE FROM DIJKSTRA'S MINI-LANGUAGE
D. Hugh Redelmeier, July 1979

CSRG-105 A LANGUAGE FACILITY FOR DESIGNING INTERACTIVE DATABASE-INTENSIVE APPLICATIONS
John Mylopoulos, Philip A. Bernstein, Harry K.T. Wong, July 1979

CSRG-106 ON APPROXIMATE SOLUTION TECHNIQUES FOR QUEUEING NETWORK MODELS OF COMPUTER SYSTEMS
Satish Kumar Tripathi, July 1979

CSRG-107 A FRAMEWORK FOR VISUAL MOTION UNDERSTANDING
John K. Tsotsos, John Mylopoulos, H. Dominic Covvey
Steven W. Zucker, DCS, June 1979

CSRG-108 DIALOGUE ORGANIZATION AND STRUCTURE FOR INTERACTIVE INFORMATION SYSTEMS
John Leonard Barron
[M.Sc. Thesis, DCS, 1980]

CSRG-109 A UNIFYING MODEL OF PHYSICAL DATABASES
D.S. Batory, C.C. Gotlieb, April 1980

CSRG-110 OPTIMAL FILE DESIGNS AND REORGANIZATION POINTS
D.S. Batory, April 1980

CSRG-111 A PANACHE OF DBMS IDEAS III
D. Tsichritzis (ed.), April 1980

CSRG-112 TOPICS IN PSN - II: EXCEPTIONAL CONDITION HANDLING IN PSN; REPRESENTING PROGRAMS IN PSN; CONTENTS IN PSN
Yves Lesperance, Byran M. Kramer, Peter F. Schneider
April, 1980

CSRG-113 SYSTEM-ORIENTED MACRO-SCHEDULING
C.C. Gotlieb and A. Schonbach
May 1980

CSRG-114 A FRAMEWORK FOR VISUAL MOTION UNDERSTANDING
John Konstantine Tsotsos
[Ph.D. Thesis, DCS, June 1980]

CSRG-115 SPECIFICATION OF CONCURRENT EUCLID
James R. Cordy and Richard C. Holt
July 1980