

PHOTOGRAPH THIS SHEET

AD-A188 619

DTIC ACCESSION NUMBER

LEVEL

INVENTORY

AFWA1-TR-87-1169

DOCUMENT IDENTIFICATION

Dec

This document has been approved for public release and sale; its distribution is unlimited.

DISTRIBUTION STATEMENT

ACCESSION FOR

NTIS GRA&I

DTIC TAB

UNANNOUNCED

JUSTIFICATION

BY

DISTRIBUTION

AVAILABILITY CODES

DIST

AVAIL AND/OR SPECIAL

A-1

DISTRIBUTION STAMP

S DTIC ELECTE **D**
FEB 09 1988
E

DATE ACCESSIONED

DATE RETURNED

88 2 05 100

DATE RECEIVED IN DTIC

REGISTERED OR CERTIFIED NO.

PHOTOGRAPH THIS SHEET AND RETURN TO DTIC-DDAC

AD-A188 619

AFWAL-TR-87-1169

ARCTIC - PROGRAMMERS'S MANUAL AND TUTORIAL

Dean Rubine and Roger Dannenberg

Carnegie-Mellon University
Computer Science Department
Pittsburgh, PA 15213-3890

December 1987

Interim



Approved for Public Release; Distribution is Unlimited

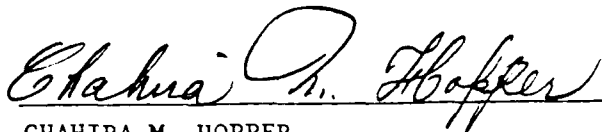
AVIONICS LABORATORY
AIR FORCE WRIGHT AERONAUTICAL LABORATORIES
AIR FORCE SYSTEMS COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433-6543

NOTICE

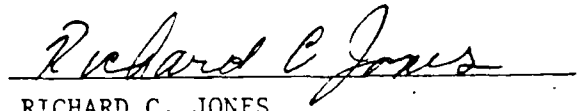
When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the Government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report has been reviewed by the Office of Public Affairs (ASD/PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.



CHAHIRA M. HOPPER
Project Engineer



RICHARD C. JONES
Ch, Advanced Systems Research Gp
Information Processing Technology Br

FOR THE COMMANDER



EDWARD L. GLIATTI
Ch, Information Processing Technology Br
Systems Avionics Div

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify AFWAL/AAAT, Wright-Patterson AFB, OH 45433-6543 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified		1b. RESTRICTIVE MARKINGS	
2a. SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.	
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE			
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU-CS-87-110		5. MONITORING ORGANIZATION REPORT NUMBER(S) AFWAL-TR-87-1169	
6a. NAME OF PERFORMING ORGANIZATION Carnegie-Mellon University	6b. OFFICE SYMBOL (if applicable)	7a. NAME OF MONITORING ORGANIZATION Air Force Wright Aeronautical Laboratories AFWAL/AAAT-3	
6c. ADDRESS (City, State, and ZIP Code) Computer Science Dept Pittsburgh PA 15213-3890		7b. ADDRESS (City, State, and ZIP Code) Wright-Patterson AFB OH 45433-6543	
8a. NAME OF FUNDING / SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F33615-84-K-1520	
8c. ADDRESS (City, State, and ZIP Code)		10. SOURCE OF FUNDING NUMBERS	
		PROGRAM ELEMENT NO 61101E	PROJECT NO 4976
		TASK NO 00	WORK UNIT ACCESSION NO 01
11. TITLE (Include Security Classification) Arctic Programmer's Manual and Tutorial			
12. PERSONAL AUTHOR(S) Dean Rubine, Roger Dannenberg			
13a. TYPE OF REPORT Interim	13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day) 1987 December	15. PAGE COUNT 42
16. SUPPLEMENTARY NOTATION			
17. COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)	
FIELD	GROUP	SUB-GROUP	
19. ABSTRACT (Continue on reverse if necessary and identify by block number)			
<p>7 ARCTIC is a programming language for describing real-time systems with many concurrent activities. Unlike conventional languages that model concurrency as multiple sequential threads of control, ARCTIC models concurrency as multiple functions of time, whose domains may overlap. This radical departure from convention has many advantages, including a declarative programming style, implicit synchronization, convenient specification of timing relationships, and an integrated approach to event-driven and data-driven real-time computation.</p> <p>This document is a specification of the ARCTIC language. Examples have been included in the hope that this specification may be also used as a tutorial. Details of a preliminary implementation of ARCTIC are given in the appendix.</p>			
20. DISTRIBUTION / AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT <input type="checkbox"/> DTIC USERS		21. ABSTRACT SECURITY CLASSIFICATION Unclassified	
22a. NAME OF RESPONSIBLE INDIVIDUAL Chahira M. Hopper		22b. TELEPHONE (Include Area Code) (513) 255-7865	22c. OFFICE SYMBOL AFWAL/AAAT-3

Table of Contents

1. Introduction	1
2. Lexical Conventions	2
3. Identifiers	3
4. Keywords	3
5. Constants	3
6. Syntax Notation	3
7. Scope and Visibility	3
8. ARCTIC Types	5
9. Expressions	5
9.1. Variable Names	6
9.2. Attribute Selectors	7
9.3. Builtin Operations	7
9.3.1. Arithmetic Operations: + - * /	8
9.3.2. Relational Operations: = <> <= >= < >	8
9.3.3. Logical Operations: and or not	9
9.3.4. Time Operations: ~ ~~ @ @@	10
9.4. Assignment Operators	10
9.5. Applications	13
9.6. Collections	15
9.7. Alternative Expressions	15
9.8. Until Expressions	17
9.9. Event Expressions	17
10. Declarations	19
11. Programs	20
12. Implementation Considerations	21
12.1. Evaluation Order	22
12.2. Function Representation	22
12.3. Data Dependencies	23
13. Conclusions	24
I. Builtin Prototypes	25
II. Interpreter Implementation Details and Limitations	28
II.1. Using the Interpreter	28
II.2. Restrictions and Deviations	29
II.3. Input/Output File Formats	29
II.4. Porting Considerations	30
II.4.1. machdep.h	30
II.4.2. Graphics	30
III. Some Examples	32

List of Figures

Figure 9-1: Example arithmetic expressions	8
Figure 9-2: Example relational expressions	9
Figure 9-3: Example time operations	11
Figure 9-4: Example sum assignment operation	13
Figure 9-5: Example collections (x is plotted)	16
Figure 9-6: Example of until	18

1. Introduction

The Tralfamadorians can look at all the different moments just the way we can look at a stretch of the Rocky Mountains, for instance. They can see how permanent all the moments are, and they can look at any moment that interests them. It is just an illusion we have here on Earth that one moment follows another one, like beads on a string, and that once a moment is gone it is gone forever.

Kurt Vonnegut, Jr., *Slaughterhouse Five* (1971)

ARCTIC is a language based on a new way of thinking about real-time programs. The fundamental idea in ARCTIC is that changing values of program variables can be described as functions of time. The same concept is at work when a logic designer draws a timing diagram, illustrating logic states as functions of time, or when a biologist plots cell populations as functions of time. In each case, the time dimension is transformed from a linear sequence of states to one or more functions, making the system's behavior much easier to specify and reason about.

To illustrate this, suppose we wish for the value of a variable to increase smoothly from 0 to 1 in 2 seconds and then remain at 1 for 5 seconds. (This variable might determine the position of a servo-controlled actuator in a mechanical system, for example.) In a sequential language, we might write something like the following:

```
while current_time < 2 do
  x := current_time / 2;
x := 1;
while current_time < 7 do { nothing };
```

Notice how much effort is involved in sequencing. The programmer must make sure that in addition to computing the right sequence of values, his program computes them at the right time.

In contrast, ARCTIC programs are not sequential and values are functions of time. The following expression is functionally equivalent to the program above:

```
x := (ramp ~ 2) + (unit ~ 5 @ 2);
```

This expression says that x is the sum of a ramp of length 2 and a unit function whose value is 1 for an extent of 5 time units, starting at time 2.

Notice how easy it is to specify rather complex functions by combining primitive functions and by using explicit operators to shift and stretch functions. The notation is declarative in the sense that we describe values by writing expressions, but we do not need to provide a sequential, imperative program that tells how to compute the values or in what order to compute them.

The idea that we should explicitly indicate timing rather than allow timing to exist as a consequence of sequential execution can be applied to discrete events as well as to continuously changing variables. For example, suppose that we want to perform two actions at specified time delays after an event occurs. In a sequential language, we might write the following program:

```
start := currenttime;
WaitUntil (start + delay1);
action1;
WaitUntil (start + delay2);
action2;
```

However, this program will only work if delay₂ is greater than delay₁ and if action₁ completes before it is time to perform action₂. Of course, it is possible to write a more robust version of this program, but our point is to illustrate that the semantics of sequential languages can make programming difficult.

Sometimes, it is more convenient to say *when* actions should happen without explicitly ordering the actions. In ARCTIC, timing is nearly always explicit and order is implicit. The program given above can be rewritten in ARCTIC as follows:

```
[ action1 @ delay1; action2 @ delay2 ]
```

In this ARCTIC expression, the actions occur at the indicated delays relative to the *instantiation time* of the overall expression. The semicolon in ARCTIC implies parallelism, so the order of instantiation is determined by the values of delay₁ and delay₂, not by the lexical order of action₁ and action₂.

What follows is an ARCTIC specification and manual; a bottom up description of ARCTIC concepts and constructs. The text contains many examples in the hope that these will serve as a tutorial aid.

2. Lexical Conventions

There are five classes of tokens: identifiers, keywords, constants, operators, and other separators. White space (blanks, tabs, newlines, and comments) is ignored except as it serves to separate otherwise adjacent identifiers, keywords, and constants. The characters "--" introduce a comment, which terminates with a newline character.

3. Identifiers

An identifier is a sequence of letters, digits, and underscores; the first character must be a letter. Upper and lower case letters are distinct. All characters are significant.

4. Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

and	if	start
causes	in	stop
do	is	sum
else	length	then
elseif	not	true
end	or	until
event	out	value
false	prod	

5. Constants

There are two kinds of constants in ARCTIC, real and boolean scalar constants.

Real constants consist of an integer part, a decimal point, and a fraction part. The integer and fraction parts both consist of a sequence of digits. Either part, but not both, may be missing.

There are two boolean constants: **true** and **false**.

6. Syntax Notation

Syntax is described in this document using a variant of BNF. Non-terminals are indicated in *italic* type, and literal words and characters in **bold** type. Alternative categories are listed on separate lines. An optional symbol has the subscript "opt."

7. Scope and Visibility

For each declaration there is a certain portion of program text called the *scope* of the declaration. By definition, the scope of any entity (variable or prototype) declared by a declaration is the scope of the declaration. An entity may be *visible* over a portion of its scope; there and only there the identifier that names the entity may be used to refer to the entity. Scoping in ARCTIC is static in the sense that given a use of an identifier it is possible by examining the text of the program to determine which declaration is referred to by that use.

There is also a dynamic aspect to variables and their scopes. Variables and scopes are contained

within *prototy, es*, which may be instantiated any number of times during the execution of an ARCTIC program. It is even possible for there to be multiple instantiations of a prototype executing simultaneously. Each instantiation of a prototype will have its own *instances* of variables (including parameters) declared in the prototype, and each instance of a variable is unrelated to other instances of the same variable. In other words, instantiating a prototype is equivalent to textually copying the prototype, renaming it and all of its variables, and instantiating the derived prototype. When not ambiguous, the phrase "an instance of the variable" will be replaced by "the variable." In particular, it is understood that when discussing the value of a variable what really is meant is the value of a particular instance of the variable.

There are four places where a variable declaration may occur: at the top level (an *item*, see section 11), as a parameter declaration (sections 9.5, 11), as a collection declaration (section 9.6), and in an event expression (section 9.9). In the latter case, the declaration of the variable is implicit; i.e. the syntax of Section 10 is not used. A variable declared at the top level is known as a *global* variable – the scope of a global begins at the point of its declaration and ends at the lexical end of the program. A variable declared as a prototype argument is a *parameter* – the scope of a parameter is the body of the prototype in which it is declared. Each instantiation gets a new set of parameter instances unrelated to the parameters of different instantiations of the same prototype. A variable declared in a collection is known as a *local* variable – the scope of a local begins at the point of its declaration and ends at the end of the collection in which it is declared. Locals in multiple evaluations of the same collection are unrelated. Finally, the scope of a variable declared by an event expression is the innermost *until* clause in which the event expression is lexically enclosed. Event variables are also considered to be local variables.

ARCTIC prototypes are analogous to functions or procedures in other languages. There are two classes of prototypes: *builtin* prototypes and *user-defined* prototypes. Builtin prototype are implicitly declared at the beginning of every ARCTIC program. The scope of each builtin prototype and user-defined prototype declaration is the entire program. Thus, user-defined prototypes may be mutually recursive without the need for forward declarations.

An identifier is potentially visible throughout its scope – it may however be hidden by a variable of the same name declared in an inner scope. Globals and prototypes are declared in the outermost scope. A parameter will hide a global or prototype of the same name over the scope of the parameter. Similarly, a local may hide a parameter, global, or prototype within the local's scope, and a local may be hidden by another local declared in a subexpression in the scope of the first local. This is the standard form of hiding seen in lexically scoped languages – each use of an identifier corresponds to the innermost declaration of the identifier whose scope includes the use.

It is an error for a name to be declared more than once as an *item* (e.g. as both a builtin prototype and a global variable). Similarly, it is an error for more than one parameter of a single prototype to have the same name, for more than one local declared in a single collection to have the same name, and for more than one event variable declared in the same `until` clause to have the same name.

8. ARCTIC Types

In ARCTIC, values are typed. A variable may contain a value of any type, and a prototype may return a value of any type. It is not possible to declare the type of a variable or prototype in ARCTIC.

ARCTIC supports five different types:

Type	Example Expressions
boolean	<code>true, 4 < 10</code>
function of boolean	<code>ramp < 0.5</code>
real	<code>10, 105.7-6</code>
function of real	<code>unit, ramp*sin(1)</code>
null	<code>[]</code>

The two ground types are boolean and real. A non-null ARCTIC value will either be a scalar of one of the ground types or a function whose range is one of the ground types. The domain of the functional types is the reals; the functions are normally considered to be functions of time, measured in seconds. The origin, `time = 0`, is the time that the ARCTIC program begins to run.

Functional values have three attributes: **start**, **stop**, and **length**. The **start** of a function is the ordinate of the first point in the function. Similarly, **stop** is the ending time of the function. Functions may be non-zero (**true**) between their **start** and **stop** times, but are by definition zero (**false**) outside the range **start** to **stop**. The **length** attribute of a function is its **start** attribute subtracted from its **stop** attribute. The interval between **start** and **stop** is open on the left but closed on the right.

Some constructs and builtin prototypes return the **null** value, which is the only member of the null type. It is an error to use the **null** value as an operand of a builtin-operation or as an argument of a builtin prototype.

9. Expressions

Most ARCTIC constructs are expressions:

expression:

variable-name
attribute-selector
builtin-operation

assignment
application
collection
alternative-expression
until-expression
event-expression
(expression)

All expressions are evaluated in an environment where three implicit variables are defined: **time**, **dur**, and **quit**. Intuitively, **time** and **dur** may be thought of as the desired start time and duration that the result of an expression should have, if it is a function. **Quit** arises from *until-expressions*, and may be considered the abort time of an expression. An expression that normally evaluates to a function of a given duration will evaluate to a truncated version of the function if the start time plus the normal duration exceeds **quit**. In this case the stop time of the function will be **quit**.

The implicit variables **time**, **dur**, and **quit** are of type real, and they can affect the result of an expression in one of three ways. An implicit variable can be named, and its value used directly. A builtin prototype may make its result dependent on these three variables, as well as its parameters. Finally, the variable **quit** may affect the value bound to a variable in an assignment statement. The three variables are dynamically inherited from expression to sub-expression; in addition, certain ARCTIC constructs alter these variables to modify the environment in which their sub-expressions are evaluated.

9.1. Variable Names

variable-name:
identifier

A variable name, used in an expression context, evaluates to an ARCTIC value. Each variable has a value. In ARCTIC *the value of a variable does not change over the course of a program.*^{1,2} Values are normally bound to variables by assignment statements, and only one assignment per variable is allowed. If the variable has a value which is a function, the value of the function at a particular time may be used before the value of the function at a later time has been calculated.

Thus, the result of evaluating a variable name is the value the variable denotes. This value is the value bound to the variable name with the same declaration.

¹The word 'variable' is perhaps a misnomer, but it does have many of the correct connotations so we adopt it here.

²As mentioned in Section 7, we are using the term "a variable" to mean "an instance of a variable."

9.2. Attribute Selectors

attribute-selector:

variable-name . **start**
variable-name . **stop**
variable-name . **length**

The variable name must denote a value which is a function. The result is a real number, the requested attribute as described in Section 8. It may be possible to determine an attribute of a variable by examining the assignments to the variable, without necessarily determining the value of the variable.

9.3. Builtin Operations

builtin-operation:

expression + *expression*
expression - *expression*
expression * *expression*
expression / *expression*
 - *expression*
expression = *expression*
expression ◇ *expression*
expression <= *expression*
expression >= *expression*
expression < *expression*
expression > *expression*
expression or *expression*
expression and *expression*
not *expression*
expression ~ *expression*
expression ~ ~ *expression*
expression @ *expression*
expression @@ *expression*

Builtin operations include the arithmetic operations, comparisons, logical operations, and time operations. The precedence of the operators is given from lowest to highest by the following list, with operators on the same line having the same precedence.

~ ~ ~ @ @@
 or
 and
 not
 = ◇ <= >= < >
 + -
 * /
 - (*unary minus*)

9.3.1. Arithmetic Operations: + - * /

Addition, subtraction, and multiplication are defined over real scalars and real functions. If both operands are scalars the result is a scalar; otherwise the result is a function. If one operand is a scalar and the other is a function, the scalar is converted into a constant function whose **start** and **stop** attribute are those of the other operand. The **start** attribute of the result is the minimum of the **start** attributes of the operands, and the **stop** attribute of the result is the maximum of the **stop** attributes of the operands. Division is only defined on scalar operands. The arithmetic negation operation $- \text{expression}$ is equivalent to $0 - \text{expression}$. The result of an arithmetic operation approximates the corresponding operation on real numbers or real functions. See figure 9-1 for some examples of arithmetic operations on real functions.

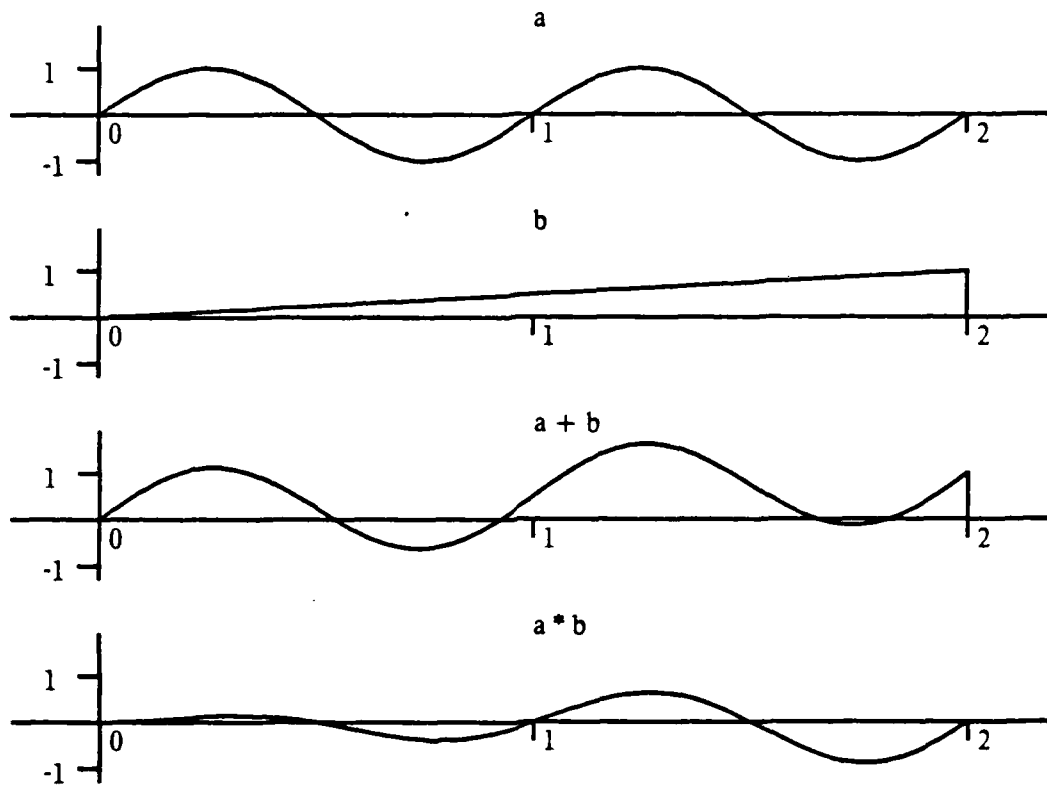


Figure 9-1: Example arithmetic expressions

9.3.2. Relational Operations: = <> <= >= < >

The relational operators all take real scalars or real functions as operands. In addition, equality and inequality ($=$ and $<>$) are defined over boolean scalars and functions. The same conversion rule as in the arithmetic operations apply when one operand is a scalar and the other is a function. The **start** and **stop** of the result are also determined as in arithmetic operations. If both operands are scalars, the result is a boolean scalar; otherwise the result is a boolean function. Conceptually, the result of comparing two functions will be at every point the result of comparing the values of the two functions at the point. An implementation must take into account the fact that equality may hold only at one

point, for example in the expression $\text{ramp} = 0.5$ (ramp is a builtin function that increases linearly from 0 to 1 on the interval $(0, 1]$ as shown in Figure 9-3). In such a case, the implementations *may* approximate the resulting boolean function with one that is true (or false) on a very small interval beginning very near the point in question. Section 9.8 describes and illustrates another related constraint on implementations. Figure 9-2 shows some examples of relational operations on real functions.

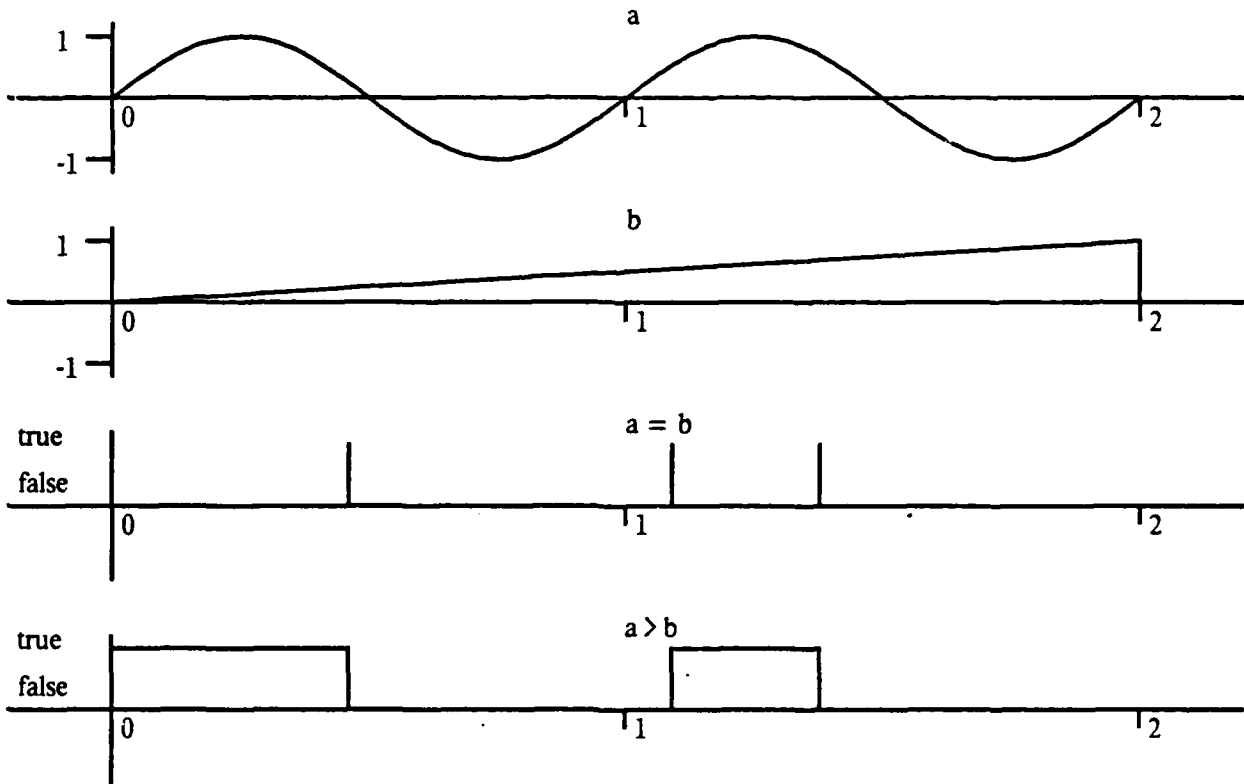


Figure 9-2: Example relational expressions

9.3.3. Logical Operations: and or not

The logical operators **and** and **or** take boolean scalars or functions as operands and result in boolean scalars or functions. Analogous to arithmetic operations, if both operands are scalars, the result is a scalar; otherwise the scalar argument is converted to a function, and the result is a function. At each point, the result of **and** (**or**) of functional operands is the **and** (**or**) of the values of the functions at the point. The **start** and **stop** of the result are determined as in arithmetic operations.

The logical operator **not** applied to a boolean scalar results in the logical complement of the scalar. If the operand is a boolean function, then the result has the same **start** and **stop** attributes as the operand. Between **start** and **stop**, the value at each point of the result is the complement of the operand at the same point.

9.3.4. Time Operations: ~ ~ @ @@

The time operations modify the implicit variables **time** and **dur**. There are four time operations: **@** (read "shift") produces a time shift relative to the current **time**. **@@** (read "at") produces an absolute time shift, *i.e.* a time shift relative to time zero, the starting time of the program. **~** (read "stretch") produces an increase in duration relative to the current duration. Finally, **~~** (read "stretch stretch") produces an absolute duration, independent of the current duration. These are now explained in detail.

The operator **@** affects the time variable. Assume **time** = t_0 and **dur** = d_0 . Then evaluating the expression

$$expr @ t_1$$

is the same as evaluating *expr* in an environment in which **time** = $t_0 + t_1 d_0$ and **dur** = d_0 . The effect is to evaluate *expr* t_1 units in the future, where the time unit is the current duration.

Similarly, evaluating

$$expr @@ t_1$$

is the same as evaluating *expr* where **time** = t_1 and **dur** = d_0 . The effect is to evaluate *expr* at absolute time t_1 seconds.

Evaluating

$$expr ~ d_1$$

is the same as evaluating *expr* where **time** = t_0 and **dur** = $d_0 d_1$. Lastly, evaluating

$$expr ~~ d_1$$

is the same as evaluating *expr* at **time** = t_0 and **dur** = d_1 .

Figure 9-3 gives some examples of time operations. It is important to notice that time operations do not operate on the result of an expression by applying some transformation. Instead, time operations serve to modify the environment in which expressions are evaluated.

9.4. Assignment Operators

assignment:

```
variable-name := expression
variable-name += expression
variable-name -= expression
variable-name *= expression
```

There are a number of assignment operators, all of which group right to left. The result of evaluating an assignment is the right operand, even for the cumulative operators **+=**, **-=**, and ***=**.

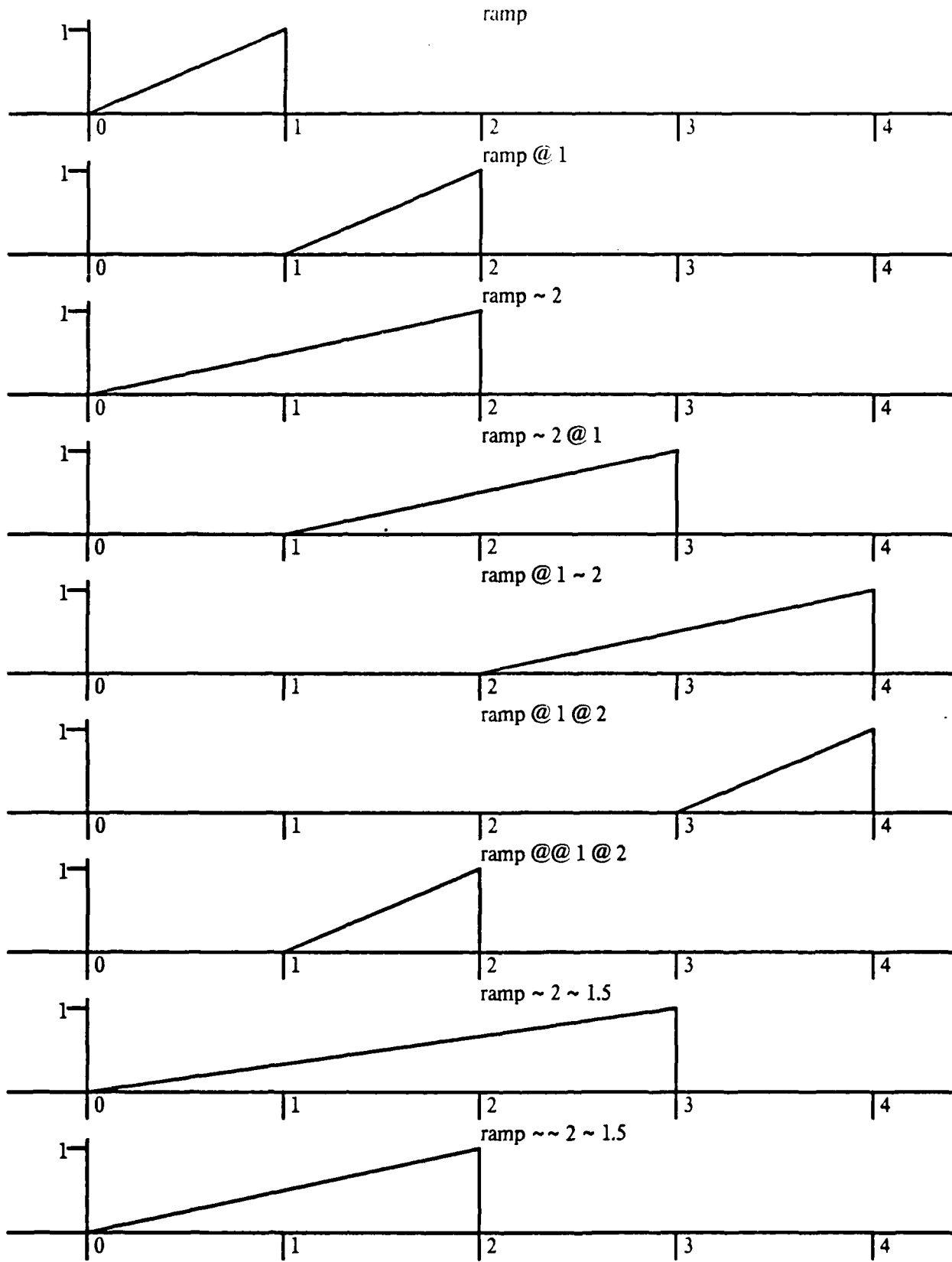


Figure 9-3: Example time operations

A variable on the left side of a simple assignment ($:=$) must have been declared **value** or **out**. The value of the variable is the result of evaluating the right side expression. (There is an exception to this rule, see the paragraph regarding **quit**, below). Assigning a value to an **out** parameter has the effect of binding that value to the corresponding actual parameter (which must be declared **value** or **out**).

A variable bound using $+=$ and/or $-=$ must have been declared **sum** or **out sum**. The value of the variable is the accumulation of the right sides of all the assignments to this variable that have been evaluated (over the course of the entire program). (But see the discussion of **quit**, below). Values assigned using $+=$ are accumulated by addition; those using $-=$ are accumulated by subtraction, the initial value of the accumulator is zero.

Similarly, a variable assigned using $*=$ must have been declared **prod** or **out prod** and is accumulated in the same manner as **sum** variables except that multiplication is used, and the initial value is unity between its **start** and **stop** attributes. Let us call all instances of expressions appearing on the right hand side of $*=$ assignments to a particular variable its *factors*. Then the **start** (**stop**) attribute of the variable must be the minimum (maximum) of the **start** (**stop**) attribute of each factor.

The implicit variable **quit** is normally positive infinity, and does not affect assignments. However, when **quit** has a finite value, its effect is this: If a function value to be assigned (using any assignment operator) has a **stop** attribute greater than **quit**, the value assigned will be the portion of the function value from its **start** to **quit** (i.e. the assigned value will have the value of **quit** as its **stop** attribute). On its interval, the assigned value has the same value as the function value before the truncation.

If assignments to a variable and uses of a variable conceptually occur in parallel, ARCTIC arranges that all assignments occur before any uses are evaluated. It is an error if there is no ordering of evaluation that allows assignments to occur before uses.

Figure 9-4 shows the value of the variables in the following example.

```
[
    sum x;
    value a, b;
    a := ramp;
    x += a;
    b := unit @ 1;
    x += b;
];
```

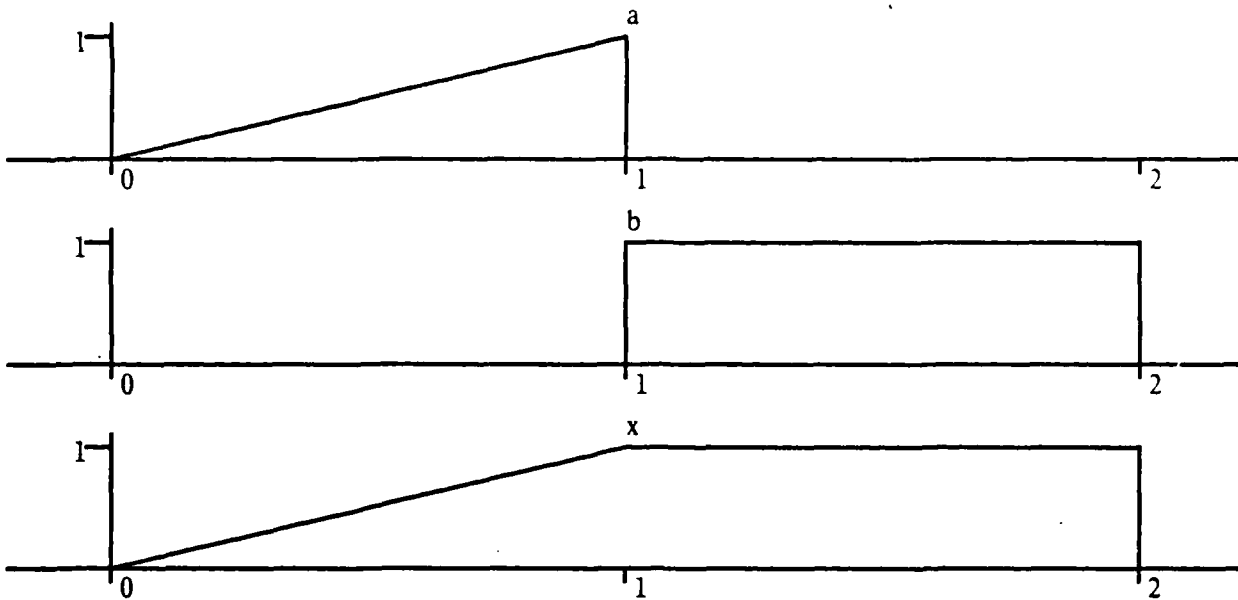


Figure 9-4: Example sum assignment operation

9.5. Applications

Application is the collective term in ARCTIC that refers to the instantiation of a prototype or the evaluation of a function value at a point.

application:

identifier
identifier (expressions_{opt})
identifier (expressions_{opt}) (expressions_{opt})

expressions:

expression
expression , expressions

An application with no parameter list is syntactically equivalent to a variable name, just an identifier. The identifier is an instantiation of a prototype with no parameters if the identifier is visibly declared to be a prototype. An identifier followed by a parenthesized list of expressions may either be an instantiation or the evaluation of a function at a point. These are distinguished by the (innermost) declaration for the identifier. If and only if the (innermost) declaration is a prototype declaration (or there is no declaration but the identifier names a builtin prototype) then the application is an instantiation. The form of application with the two parenthesized expression lists must be an instantiation of a prototype. The result of this instantiation must be a function, which is evaluated at a point.

Instantiations are evaluated as follows: The actual parameters in the call are matched positionally with the formal parameters of the prototype declaration. For user-defined prototypes, it is an error for

there to be a different number of actual and formal parameters. However, some builtin prototypes may take a variable number of arguments. After the matching, all actual parameters that correspond to **in** parameters are evaluated in the environment of the caller. These are bound to their respective formal parameters and the body of the prototype is evaluated with the same **time**, **dur**, and **quit** as the caller. The result of the instantiation is the result of the evaluation of the body. The values of all **out** parameters are bound to their corresponding actual parameters. It is an error if the actual parameter corresponding to a formal parameter is not a variable declared **value** or **out**. Notice that if an actual parameter is paired with a formal **out** parameter, then there can be no other assignments to the actual.

An **in** formal parameter is declared using the **in** declaration form. The corresponding actual parameter may be any expression. As stated, that expression is evaluated and bound to the formal (as if with **=**) when the prototype is instantiated.

An **out** formal parameter may be declared **out**, **out sum**, or **out prod**. (Out parameters are not allowed in certain prototype declarations, see Section 10). The corresponding actual parameter must be a variable that is capable of being bound using **=**, **+ =**, or *** =** respectively.³ Upon completion of the instantiation, the value of each formal is bound to its corresponding actual using the appropriate assignment operator.

An application may also be the evaluation of a function at a point. The first argument is an expression which gives the ordinate at which the value of the function is desired. If the ordinate is outside the interval of the function, the result of the evaluation is zero. If the function is continuous at the ordinate, the value of the application is the value of the function at the ordinate. If the function has a jump discontinuity at the ordinate, the result of the evaluating is the value of the function, approaching the discontinuity from the left. This is consistent with the view that functions are defined on intervals open to the left and closed to the right.

The result of evaluating a function at a discontinuity may be modified by a second argument to the evaluation. If the value of the second argument is negative, the application results in the value of the function approaching from the left. If positive, the ordinate is approached from the right. If zero, it is an error if the ordinate refers to a discontinuity.

³At the time of this writing, other parameter binding rules are under consideration. For example, it might be useful to allow **out sum** formals to be matched with **value** actuals, or to allow **out** formals to be matched with **sum** actuals. While evaluating the instantiation, the formal parameter would be bound using its appropriate assignment operator, and upon completion of the instantiation the value of the formal would be bound to the actual using the appropriate assignment operator for the actual.

9.6. Collections

There are two forms of collections: parallel and sequential.

collection:

[*parallel*]
[*sequential*]

parallel:

*expression-or-declaration*_{opt}
*expression-or-declaration*_{opt} ; *parallel*

sequential:

*expression-or-declaration*_{opt}
*expression-or-declaration*_{opt} | *sequential*

expression-or-declaration:

expression
variable-declaration

A collection is a sequence of expressions and variable declarations, separated by vertical bars or semicolons. In the degenerate case where there are no expressions in the collection, the result of the collection is **null**. If there is only one expression in the collection, the result of the collection is the result of evaluating the one expression.

Each expression in a parallel collection is evaluated with the same **time**, **dur**, and **quit** as the collection itself. The result of the collection is the result of the first expression in the collection.

For a sequential collection, the first expression is evaluated with the same **time**, **dur**, and **quit** as the collection itself. Each subsequent expression is evaluated with the same **dur** and **quit** as the collection, but with **time** equal to the **stop** attribute of the previous expression. All expressions in a sequential collection must evaluate to functions. The result of a sequential collection is the sum of the results of the constituent expressions. See figure 9-5 for some examples of collections.

Any variables declared in a collection are local to that collection. The only allowable declarations in a collection are **value**, **sum**, and **prod**.

9.7. Alternative Expressions

alternative-expression:

*if expression then expression elseif-part*_{opt} *else-part*_{opt}

elseif-part:

elseif-clause
elseif-clause elseif-part

elseif-clause:

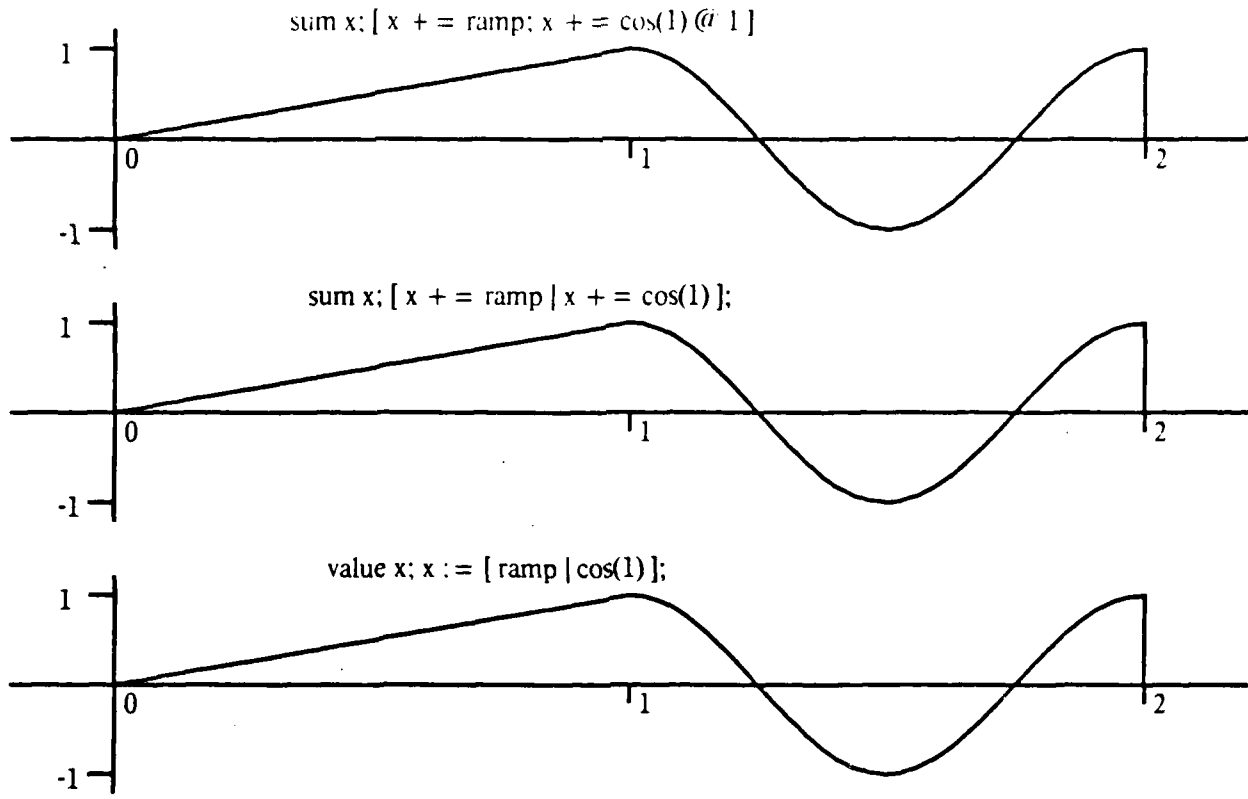


Figure 9-5: Example collections (x is plotted)

elseif *expression* **then** *expression*

else-part:

else *expression*

An *alternative-expression* is evaluated as follows: The expression specified after the **if**, and any expressions specified after **elseif**, are evaluated up to and including the first expression whose value is **true** (treating a final **else** as **elseif true then**) and the corresponding **then** expression is evaluated. The result of this evaluation becomes the result of the *alternative-expression*. If none of the expressions after **if** or **elseif** evaluate to **true** and there is no **else** part, the result of the *alternative-expression* is **null**. It is an error if one of the expressions after **if** or **elseif** evaluates to something other than a boolean scalar.

Examples

```
if vibrato then freq += sin(6) * 0.3;
```

```
if AlarmEnabled then RingBell else BlinkLight;
```


9.8. Until Expressions

until-expression:
do *expression* *until-clauses*

until-clauses:
until-clause
until-clause *until-clauses*

until-clause:
until *expression* **then** *expression*

Intuitively, the intent of an *until-expression* is simple. The expression after the **do** is evaluated until one of the expressions after **until** becomes **true**, then the expressions after the corresponding **then** clause is evaluated. The result is the sum of the evaluated **do** and **then** expressions. Also, any assignments that happen while evaluating the **do** expressions are abruptly truncated when the first **until** expression becomes **true**.

Abstractly, an *until-expression* is evaluated in a rather roundabout manner. The first expression after **until** in each until clause is evaluated, and each expression must be a boolean function. If none of the functions are **true** after **time**, the result of the *until-expression* is the result of evaluating the expression after the **do**, using the environment of the **until**.

Otherwise, there must be a minimum time t_{min} at which some function becomes **true** after **time**. The expression after **do** is evaluated in the same environment (**time** and **dur**) as the **until** clause except **quit** is t_{min} . This result is truncated, so that its **stop** attribute is at most the value of **quit**. This truncated value is added to the result of evaluating the **then** part of the until clause that gave rise to the minimum time. When evaluating the expression after this **then**, **time** is set to t_{min} , while **dur** and **quit** are that of the until expression.

Figure 9-6 shows the value of the following expression.

```
do x := [ ramp + sin(2) ] ~ 3
until x > 1.5
then (unit * x(time)) ~ (3 - time);
```

This expression computes a function of length 3 in which the first part terminates as soon as its value reaches 1.5. The function is then given a constant value of 1.5 until time 3.

9.9. Event Expressions

event-expression:
event *application*

Informally, an **event** expression denotes a demon that waits for the instantiation of a prototype.

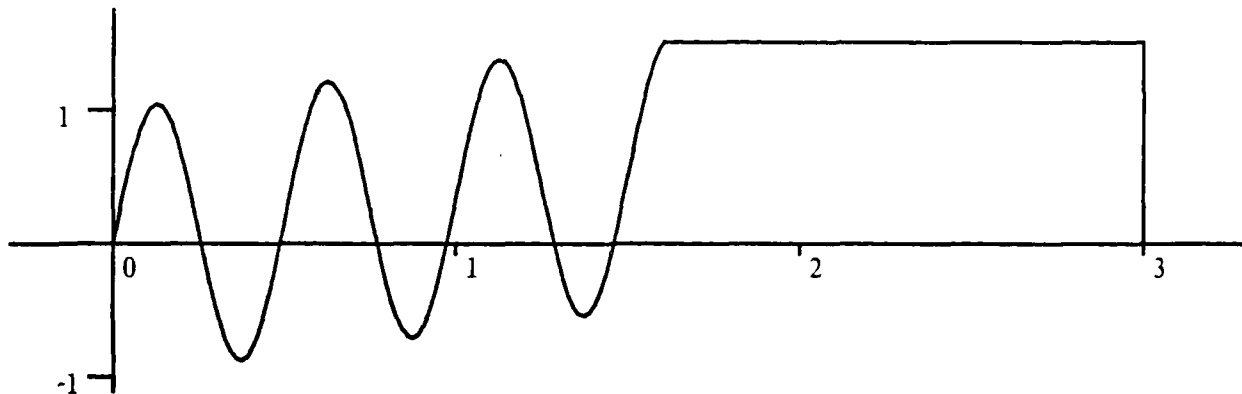


Figure 9-6: Example of **until**

When the prototype is instantiated, the demon becomes **true** and *event variables* take on values of parameters passed to the instances of the prototype. This is something like an Ada rendezvous entry or an exception handler.

An *event-expression* can only appear in the conditional part of an **until** clause. In an event expression, the identifier of the application must be the name of a prototype. The expressions of the application must all be legal variable names, as if the prototype required all **out** parameters. Although the syntax is that of an application, the parameters are all formal arguments called *event variables*, which are declared by the event-expression. The scope of these variables are the innermost **until** clause in which the event expression is lexically enclosed. The actual parameters of every instantiation of the prototype must be scalars. An event-expression results in a boolean function that is **true** only at the instantaneous times that the named prototype has been instantiated. At these times, the event variables will be bound to values of the corresponding actual parameters.

A semantic problem arises if two instantiations of a prototype occur simultaneously with different actual parameters and there is a corresponding event expression. What values do the event variables take on? We would like to avoid this problem by assuming no two events happen at the same time, but this raises more problems than it solves. Instead, the problem is solved by saying that a conditional containing an event expression must be evaluated once for each instance of the corresponding prototype, even if two or more instances are concurrent. In implementations on a sequential processor, instantiations will naturally be ordered even when they are specified to occur simultaneously, so the problem is likely to exist only in the formal model.

Example

This example illustrates the use of an **until** expression as an exceptional condition handler. The prototype `Operate` is instantiated with a `quit` time corresponding to the time of the first instantiation of `StopRequest(Emergency)` or `StopRequest(Normal)`. Depending on the value of the argument to

StopRequest, either EmergencyShutdown or NormalShutdown is instantiated.

```
do Operate
  until event StopRequest(reason) and reason = Emergency
    then EmergencyShutdown;
  until event StopRequest(reason) and reason = Normal
    then NormalShutdown;
```

10. Declarations

declaration:

```
prototype-declaration
variable-declaration
```

There are two kinds of declarations: prototype declarations and variable declarations. Prototype declarations define new prototypes. Variable declarations define new variables.

prototype-declaration:

```
identifier formalsopt is expression
identifier formalsopt causes expression
identifier formalsopt in causes expression
identifier formalsopt causes out
```

formals:

```
( formal-list )
```

formal-list:

```
variable-declaration
variable-declaration ; formal-list
```

variable-declaration:

```
in identifier-list
out identifier-list
out sum identifier-list
out prod identifier-list
sum identifier-list
prod identifier-list
value identifier-list
```

identifier-list:

```
identifier
identifier , identifier-list
```

There are four forms of prototype declarations. The first two (*is* and *causes*) are semantically identical in all but one respect. An instantiation of the *is* form is equivalent to the result of evaluating its body expressions with each actual parameter bound to its corresponding formal. The *cause* form also causes the evaluation of its body expressions as in the *is* form, but if the result is a scalar, *null* is

returned and if the result is a function the result has the same **start** and **stop** attributes but the function value is undefined. Thus, the **is** form is used when a value is to be returned, and the **cause** form is used when no meaningful value is to be returned. The **start** and **stop** attributes are returned for use in constructs like the *sequence*.

Parameter passing is accomplished as defined in Section 9.5. Because of the single-assignment rule, it is an error for two **out** formal parameters to be aliases for the same actual. **In**, **out**, **out sum**, and **out prod** declarations are allowed for the formal parameters of these first two forms of prototype declarations.

The third form (**in causes**) is equivalent to the **cause** form except that it can be instantiated by an external event. It is only allowed to have **in** parameters. The mapping of an external event to a particular input prototype name is undefined in the language. When such a prototype is instantiated from an external event, **dur** is 1, **quit** is positive infinity, and **start** is the time at which the event occurred.

The fourth form (**causes out**) defines an external event. It is only allowed to have **in** parameters. Instantiating a prototype of this form causes the external event to occur at time **time**. The external event has access to any actual parameters as well as the values of the implicit parameters **time**, **dur**, and **quit**.

There are seven forms of variable declarations. A variable declaration may occur in one of three contexts: as an *item* (Section 11) in which case the variable is global, in a prototype declarations (Section 10) in which case the variable is a parameter, and in a collection (Section 9.6) in which case the variable is local. Not all five forms are allowed in all three contexts – see the section noted for the restrictions of each context. The scope and visibility of variables in each context is discussed in Section 7. The various kinds of assignment to each variable declaration for is discussed in Sections 9.4.

11. Programs

The constructs group into two main classes, expressions and declarations.

```
arctic-program:
    items
```

```
items:
    item
    item items
```

```
item:
```

```
expression ;
declaration ;
```

A program is a list of declarations and expressions. An ARCTIC program may have at most one expression at the top level. This expression, if given, is evaluated at `time = 0` and `dur = 1`. In any case, prototypes instantiated by external events are evaluated. The result of an ARCTIC program consists of the values of all the global `out` variables together with any external events instantiated by the ARCTIC program.

All other *items* are declarations. Prototype declarations have been discussed in Section 10. *Items* that are variable declarations declare global variables. All seven forms of variable declarations are allowed for global variables. Globals declared `in` are inputs to the ARCTIC program; they correspond to external (physical) inputs. Globals declared `out`, `out sum`, and `out prod` are the outputs of the ARCTIC program; they correspond to external outputs. Globals declared `value`, `sum`, and `prod` do not refer to inputs or outputs of the ARCTIC program. The method by which global inputs and outputs are assigned to actual physical inputs and outputs is undefined in the language.

Example

This program specifies a simple music synthesizer. The input consists of instantiations of the note prototype, whose parameter is the desired frequency. The output consists of two functions, frequency and amplitude, that control the instantaneous frequency and amplitude of an oscillator.

```
sum amp, freq;
out amplitude, frequency;

note(in pitch) input causes [
    amp += [ ramp ~ 0.1 | (1 - ramp) ~ 0.9 ];
    freq += unit * pitch;
];

[ amplitude := amp; frequency := freq ];
```

12. Implementation Considerations

Because Arctic programs ordinarily describe parallel activities and manipulate data structures (functions) that change over time, there is no simple translation from an Arctic program into a conventional uniprocessor instruction set. Implementation of Arctic is still an area of research, but a discussion of implementation strategies is warranted nevertheless. An implementation can be categorized with respect to several orthogonal characteristics or dimensions. The first of these is whether or not the implementation computes entire functions at a time or computes functions

incrementally in time order. The second characteristic is the representation of functions, either with splines or in sampled form. The third characteristic is the manner in which data dependencies are handled. Each of these characteristics is discussed in greater detail below.

12.1. Evaluation Order

Let us start with the first characteristic. Our interpreter has the property that it computes one entire function as the value of an Arctic expression or subexpression before going on to the next. This implies that the implementation cannot operate in real time because it does not use on-line methods. For example, to compute the result of the following expression:

$$(ramp + a) * b$$

the interpreter first constructs a representation for *ramp*. Then it evaluates *a* and adds the result to *ramp*. This intermediate result is then added to the value of *b* to form the result. Note that the entire futures of *a* and *b* must be known before any of the result can be computed. Therefore, the interpreter requires that inputs are presented in advance. Output functions are computed one at a time and plotted or written to a file.

The second approach is essential to achieve real-time execution. In this approach, all functions are computed in a quasi-parallel fashion. That is, the values of all functions at time *t* are computed before any values are computed at times greater than *t*. For example, to compute the value of the expression given above at time *t*, we could compute

$$(ramp(t) + a(t)) * b(t).$$

This would only compute one point of the resulting function, so the implementation must iterate the computation, interleaving it with the computation of other functions.

12.2. Function Representation

The second characteristic of an implementation is its representation of functions. Our interpreter uses piece-wise linear approximations in all of its computations. The functions are stored as linked lists of x-y coordinate pairs, where each pair represents a point of inflection. This provides an efficient representation where functions are often simple ones like ramp and unit, and computation time is linear in the number of inflection points computed. Another advantage of this representation is that very simple hardware (basically just an adder and two registers) can be used to interpolate between the inflection points to produce a smoothly changing function in real time.

A real-time implementation of Arctic using this representation would be difficult because in order to

compute the value of a function at time t , one must have the coordinates of the first points of inflection before and after t . For functions produced by prototypes like ramp, the future is known, but for real-time input functions, the future is not known. One can, however, sample inputs sufficiently often to obtain a good approximation in piece-wise linear form. The implementation then becomes event-driven, where events are the computation of points of inflection for various functions.

Another approach is to represent functions as a sequence of samples. This representation is almost universal in the field of digital signal processing and has the desirable property that the representation uses a constant amount of storage per unit of time. Furthermore, most operations like function addition or multiplication take a constant amount of computation time per unit of function time, facilitating a real-time implementation. If only the current value of a function is needed at any given time, then functions can be represented by using a single scalar storage cell whose content is updated every sample period.

12.3. Data Dependencies

The final important characteristic of an implementation is the manner in which data dependencies are handled. Our interpreter assumes that the order of execution can be statically determined. Furthermore, it assumes that a prototype can be evaluated in full before a prototype instantiated in parallel is evaluated. The following example illustrates how this implementation can fail:

```

value x, y;
A is [ y; x := ramp];
B is [ y := x ];
[ A; B ];

```

The interpreter must either evaluate A first, in which case y will be undefined, or it must evaluate B first, in which case x will be undefined.

It is also possible to deal with dependencies dynamically. Think of each assignment statement as a process that produces values that are in turn used by other processes. These processes must be executed in an order such that values are defined before they are used. The order can be determined by a topological sort or by synchronization primitives, but neither of these methods is ideal. The problem with topological sorts is that the sort must be performed each time a new process is created, corresponding to the instantiation of a new assignment expression. On the other hand, synchronization mechanisms add considerable overhead when performed at such a fine granularity of computation. Special hardware, as in data-flow processors, could reduce this synchronization

overhead considerably.

13. Conclusions

The design and ongoing implementation of ARCTIC has been a useful and illuminating task. Beyond the routine applications of ARCTIC as a tool to compute functions of time, the language has given us a way to formalize the notion of behavior and response. It has also provided a new way to think about real-time computation.

As a practical language, ARCTIC has found many applications in the production of computer music and other digital audio sounds. ARCTIC is used to generate control functions for parameters such as amplitude and frequency. These functions are then used to control software or hardware digital oscillators which produce sound. ARCTIC has been used to generate sounds ranging from the simulation of a music box in a musical composition to the generation of the international standard for nuclear power plant alarm signals proposed by Westinghouse.

ARCTIC has also been useful as a conceptual tool. Behaviors that are difficult to specify verbally or in sequential languages are often easy to program in ARCTIC. The explicit specification of timing in ARCTIC tends to make programs easy to read, write, and modify. ARCTIC also leads to modular program construction. In particular, complex computations that must take place simultaneously in real-time can be specified and tested separately. They can then be combined using arithmetic operators or a collection construct. This sort of modular program construction is often difficult in sequential languages. Another attraction of ARCTIC has been that computed behaviors can be plotted and studied visually. This can be of enormous help in debugging and understanding programs, but it is not a usual feature of sequential programming languages.

Finally, ARCTIC has led us to think about alternative paradigms for implementing real-time programs. While we do not have a real-time implementation at this point, some of the techniques developed for ARCTIC have already been used to advantage in complex real-time program for controlling a signal processor. We believe that we will be able to run ARCTIC programs in real-time using similar techniques, and that ARCTIC will then become a practical language for a large class of real-time systems.

I. Builtin Prototypes

The following is an alphabetized list of prototypes built into an interpretive, non-real-time implementation of ARCTIC. Each prototype name is followed by a short description of its operation. Most prototypes are 'normal' in the sense that they evaluate their arguments as described in section 9.5, execute, and return a value and possibly have side effects. There are a few 'special' prototypes that either do not evaluate their arguments or evaluate them more than once. If a prototype is special, it will be noted in its description. Some prototypes affect the operations of the interpreter itself; these too will be noted.

- cos(frequency, initialphase)** returns a cosine function on the interval (*time*, *time + dur*]. *frequency* gives the frequency of the cosine in Hertz, and *initialphase* gives the initial phase of the cosine, in degrees. The *initialphase* argument is optional, and defaults to zero.
- error(arg1, ...)** enters the interpreter debugger. In the debugger the values of the arguments to *error* and other variables may be examined; type '?' for help.
- exit** causes the interpreter to exit. The currently running ARCTIC program is terminated abruptly, and control is returned to the shell.
- exp(f)** returns e^f . *f* may be a real scalar or a real function.
- extract(f, btime, etime)** returns the function *f* on the interval (*btime*, *etime*]. The resultant function equals *f* on the intersection of (*f.start*, *f.stop*] and (*btime*, *etime*] and is zero elsewhere.
- fnorm(f)** returns $(f - fmin) / (fmax - fmin)$. *f* must be a real function, and *fmax* and *fmin* are the maximum and minimum of the function, respectively. If *fmax* equals *fmin*, then the zero valued function of the interval (*f.start*, *f.stop*) is returned.
- fullnorm(f)** returns *fnorm*(*tnorm*(*f*)).
- int(s)** returns the integer part of *s*, truncating toward zero.
- interp(f1, f2, m)** returns the result of interpolating *f1* and *f2* with interpolation factor *m*. *f1* and *f2* must be real functions; furthermore, they must be piecewise linear functions (see *pwl*) with the same number of inflection points. For each (*x1*, *y1*) and (*x2*, *y2*) that are corresponding points in *f1* and *f2*, the result will have a point ($x1 * (1 - m) + x2 * m$, $y1 * (1 - m) + y2 * m$). Note that extrapolation may be achieved with *m* less than zero or greater than one.
- irnd(s)** returns an equally distributed integral random number between zero and *s* - 1 inclusive.
- ln(f)** returns the natural log of *f*. *f* may be a real number or a real function.
- max(f)** returns the maximum abscissa of the real function *f*.

- min(f)** returns the minimum abscissa of the real function *f*.
- norm(f)** returns *f* shifted and stretched so that the result is defined on the interval (*time*, *time* + *dur* * *f.length*).
- osc(amp, freq, wave, voice, nvoices)**
is the oscillator allocation function. First, a search is made for an integer *N* with the following properties: 1) *N* has remainder *voice* when divided by *nvoices*, and 2) the time interval of the values of the global variable *ampN* does not intersect the interval (*amp.start*, *amp.stop*]. Given *N*, the expression [*ampN* + = *amp*; *freqN* + = *freq*; *waveN* + = *wave*] is performed. As many *ampN*, *freqN*, and *waveN* function must be declared as are necessary. It is an error if no *N* can be found.
- play** causes the output file to be written, and the shell command "play.sh file.out" to be executed.
- plot(f1, f2, ...)** does not evaluate its arguments. During interactive mode, the variables *f1*, *f2*, ... are plotted after each interactive expression is evaluated. Each *fi* is plotted only if it is a function of time.
- print(a1, a2, ...)** returns *a1*, with the side effect of the interpreter printing *a1*, *a2*...
- proto(f)** returns *f* shifted and stretched so as to be defined on the interval (*f.start* + *time*, *f.start* + *time* + *f.length* * *dur*].
- pwl(x1,y1, x2,y2, ..., xn,yn)**
returns a piecewise linear function constructed by connecting the points (0,0), (x1,y1), (x2,y2), ... (xn,yn), (xn,0) and then stretching and shifting the function to lie on the interval (*time*, *time* + *dur*]. A missing *yn* is taken to be zero.
- ramp** returns *pwl(1,1)*.
- randomize** causes the random number generator to be given a random seed, and returns **null**.
- repeat(expr, n)** returns [*expr* | *expr* | ... | *expr*] where there are *n* *expr*'s in the above collection. This is a special form; *expr* is evaluated *n* times.
- rnd** returns a random number between zero and one.
- sin(frequency, initialphase)**
returns *cos(frequency, initialphase - 90)*. The *initialphase* argument is optional, and defaults to zero.
- tnorm(f)** returns *f* shifted and stretched so that the result is defined on the interval (*time*, *time* + *dur*].
- undef** causes the values of all global variables to be forgotten.
- unit** returns *pwl(0,1, 1,1)*.

`unplot(f1, f2, ...)` does not evaluate its arguments. During interactive mode, the variables `f1`, `f2`, ... are no longer plotted after each interactive expression is evaluated.

`zero` returns `pwl(1,0)`.

II. Interpreter Implementation Details and Limitations

There is a non-real-time interpreter of ARCTIC available on UNIX. This implementation currently runs on both a DEC VAX (running a CMU-modified 4.2bsd) and a SUN workstation (running 4.2bsd with modifications for Andrew). The interpreter is written in C and it is expected that it will be portable to most any computer system with a C compiler. The interpreter has the ability to graphically display functions of time if there is hardware to support this.

II.1. Using the Interpreter

The interpreter is invoked with the command:

```
arctic [ options ] [ file.a ]
```

If a file (whose name must end in '.a') is specified, that file is read for the ARCTIC program (see Section 11). If it exists, the file whose name is formed by changing to '.a' suffix to '.in' is read to get the values of global in variables as well as input events (see Section II.3).

If the file contains an *item* that is an expression, that expression is evaluated (as well as any prototypes instantiated by extern events in the '.in' file). The result of the expression is printed and, if possible, plotted. If there are any global **out** declarations or prototypes defined using **causes out**, a '.out' file is written. The interpreter then exits.

If no input file is specified, or if file.a does not contain an *item* that is an expression, the interpreter reads the '.in' file and then prompts for *items*. Each *item* has its effect as soon as it is entered. Variable and prototype declarations hide previous global declarations of the same identifier, and expressions are evaluated immediately.

The following command line options effect the operation of the interpreter.

- v turns on verbose mode. The most noticeable effect of this option is that when printing functions of time all inflection points are listed, rather than just the start and stop attributes.
- p turns off plotting mode. This options tells the interpreter not to bother to plot the results of top-level expressions. This mode is useful for the common 'batch mode' case where, given the '.a' and '.in' files, one is not interested in seeing the result of the expression *item*, but only the '.out' file.

When the interpreter encounters an error during the execution of an ARCTIC program, it enters a debugging mode. In this mode the user can, among other things, examine the values of variables and parameters. Type '?' for help in this mode.

II.2. Restrictions and Deviations

The interpreter does not implement exactly the language specified in this manual. This section describes the differences between the implementation and specifications. The differences occur in due to the treatment of two issues: function representation and evaluation ordering.

The interpreter represents functions of time as *piecewise-linear functions*. A real function is approximated by a sequence of points ordered by non-decreasing ordinate. Jump discontinuities are represented by two successive points with the same ordinate and different abscissa. The value of a function at an ordinate between (the ordinates of) two successive points is the result of linear interpolation between the two enclosing points. The value of a function at a jump discontinuity is generally the value obtain from approaching the discontinuity from the left, though there are ways (see Section 9.5) to get the value approaching from the right. The value of a function outside the endpoints of the function (typically the **start** and **stop** attributes of the function) is defined to be zero.

This implementation of ARCTIC does *not per se* restrict the language. What does restrict the language is the interpreter's strategy of evaluating a function of time all at once. This prevents the value of function at some time from depending on its value at an earlier time. For example the interpreter is unable to evaluate the expression in Figure 9-6 since the value of x at later times is dependent of the values of x at an earlier time. (In this case, due to the semantics of **until**, the **stop** attribute of x is exactly the point where x first has value greater than 1.5. The interpreter needs x to compute x , and fails.) The fragment that actually generated the picture is:

```
[
    do d until d > 1.5 then (unit * d(time)) ~~ (3-time);
    d := ( ramp + sin(2) ) ~ 3;
];
```

The limitations of the interpreter's function representation and evaluation order have already been discussed in Section 12.

II.3. Input/Output File Formats

The '.in' and '.out' files are ascii files with a rigid format. These files may contain two kinds of data: (*piecewise linear*) functions of time, and discrete events. Each kind, function or event, has the following form:

```

function-or-prototype-name
datum-1
datum-2
...
datum-n

```

a blank line

In the case of functions, each datum is a pair of numbers, representing the ordinate and abscissa of the startpoint and/or endpoint of each segment in a piecewise-linear function. The ordinates of the points are in non-decreasing order, and by convention the abscissa of the first and last points should be zero. The ordinate of the first and last points are the **start** and **stop** attributes of the function, respectively.

In the case of discrete events, each datum represents one event. Each datum contains at least one number. The first number in each datum is the instantiation time of the event that the datum represents. Remaining numbers are the parameters to the instantiation of the event. These actual parameters are mapped to the formal parameters of the named parameters by position. In this implementation all input and output events are required to have only real scalar in parameters.

II.4. Porting Considerations

Porting ARCTIC to a new environment is a straightforward task. First, the file 'machdep.h' should be modified to reflect the characteristics of the new environment. Next, a set of low-level plotting routines must be written if the graphical display of the interpreter is to function.

II.4.1. machdep.h

The only non-obvious issue in porting this file is the definitions of the two quantities PLUSINF and NEGINF. PLUSINF and NEGINF are double precision constants that must be larger (in absolute value) than any quantity likely to be encountered in the interpreter. In particular, they must be larger than the maximum time ever to be encountered. Also, it must be possible to add and subtract any plausible quantity to PLUSINF and NEGINF without overflow occurring. This is because additions involving these arise in the denominator of interpolation calculations whose numerator is zero. In these case the result must be zero, and no overflow should occur.

II.4.2. Graphics

Porting the ARCTIC graphics is also straightforward. The interpreter routine `b_plot_term()` is called to plot the desired functions on the screen. It in turn calls the following functions, all of which use integers as coordinates:

```
PlotGetCorners(lly, lly, urx, ury) int *llx, *lly, *urx, *ury;
```

The parameters are pointers to integers which are set to the x and y coordinates of the lower left and upper right of the screen respectively. The interpreter

depends on the fact that $*lly < *ury$ and $*lly < *ury$. This is true for the normal Cartesian plane, but for many plotting devices the upper left corner is the origin so that, at least at first glance, $*lly > *ury$. When this happens it is easiest to set $*ury$ to zero and $*lly$ to the negation of the size of the y dimension. Then all the routines `PlotLine`, `PlotPoint`, and `PlotText` need do is negate the y coordinates before they are used.

`PlotGetMiscParameters(interfunctiondistance, charx, chary) int *interfunctiondistance, *charx, *chary;`

Sets $*charx$ to the average width of a character, $*chary$ to the average height, and $*interfunctiondistance$ to a value that makes a pleasing break between functions, usually between $1 * *chary$ and $3 * *chary$.

`PlotStartDraw()` is called just before a new plot is to be made. It should clear the screen in preparation for the new plot.

`PlotLine(fromX, fromY, toX, toY) int fromX, fromY, toX, toY;`

This function should draw a line from $(fromX, fromY)$ to (toX, toY) . Remember to negate $fromY$ and toY if you played the negation game in `PlotGetCorners()`.

`PlotPoint(X, Y) int X, Y;`

plots a point at the passed coordinate. Again, remember to negate Y if necessary.

`PlotText(X, Y, text) int X, Y; char *text;`

prints the passed string so that the lower left corner of the first character is at (X, Y) . Again, remember to negate Y if need be.

`PlotEndDraw()` is called after the new plot is complete. This routine should do things like cause the screen to be updated if, for example, the previous plot routines have been buffering up commands without redrawing.

`PlotTerminate()` is called just before the ARCTIC interpreter exits. It should do any major cleanup required (like the removal of temporary files).

The function `b_plot_term()` may be invoked when the screen needs to be refreshed. If, for example, the size of the plotting window is changed, all that needs to be done is to call `b_plot_term()`, which will call `PlotGetCorners()` to get the size of the new window and redisplay.

III. Some Examples

This appendix shows an example session using the ARCTIC interpreter on a SUN workstation. Typed input is shown in boldface, output is shown in italics or in graphical form, and comments on the session are shown in the normal typeface.

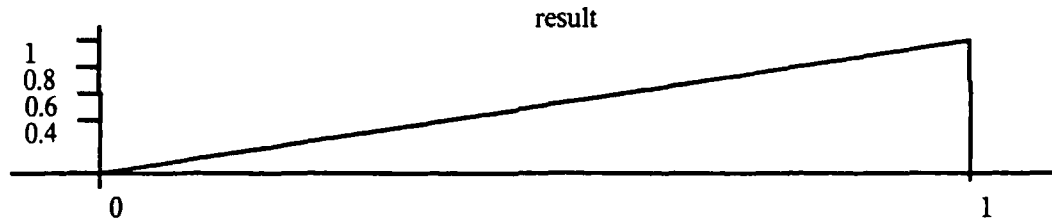
In the first example, we will use ARCTIC to control a single sine-wave oscillator. The oscillator is controlled by two functions of time: *amp0* controls the amplitude of the oscillator and should be between zero and one hundred. *freq0* controls the frequency of the oscillator. The value is the number of semitones above four octaves below middle C (i.e. 48 is middle C, and 57 is A 440Hz).

The file *ampfreq.a* contains the declarations of *amp0* and *freq0*.

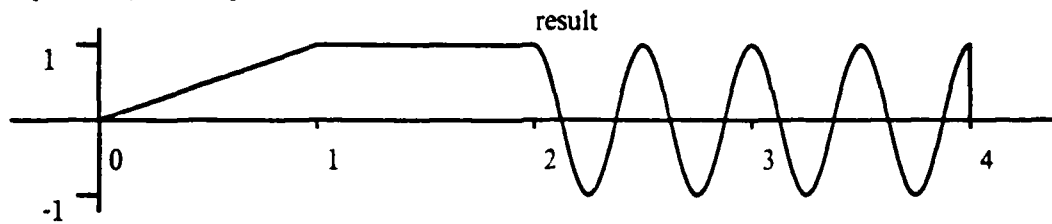
```
% cat ampfreq.a
out amp0, freq0;
% arctic ampfreq.a
```

Now that the interpreter is invoked we execute some example expressions.

```
arctic> ramp;
result := [ 0.0000, 1.0000 ]
```



```
arctic> [ ramp | unit | cos(2) ~ 2 ];
result := [ 0.0000, 4.0000 ]
```

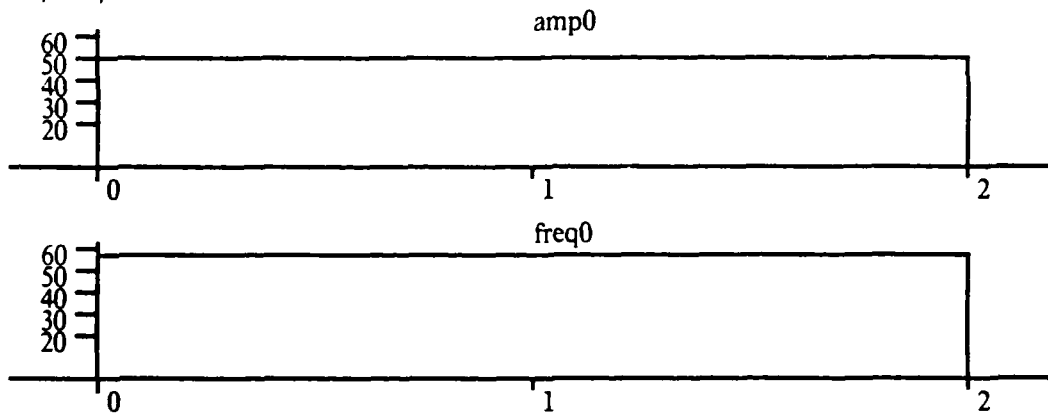


We can use ARCTIC to generate a two second 440Hz sine wave and then listen to it.


```

arctic> amp0 := 50 * unit ~ 2;
Reinitializing amp0
result := [ 0.0000, 2.0000 ]
arctic> freq0 := 57 * unit ~ 2;
Reinitializing freq0
result := [ 0.0000, 2.0000 ]
arctic> plot(amp0, freq0);
result := NULL
arctic> play;
result := NULL
writing ampfreq.out

```

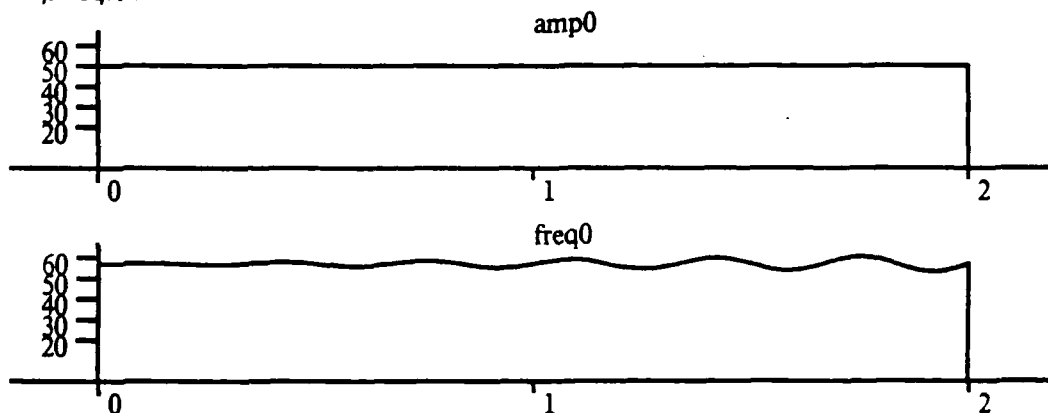


We can change the frequency to achieve a vibrato of increasing depth.

```

arctic> freq0 := (57 * unit + 4 * ramp * sin(3)) ~ 2;
Reinitializing freq0
result := [ 0.0000, 2.0000 ]
arctic> play;
result := NULL
writing ampfreq.out

```



We now define a monophonic synthesizer. First, we make *amp0* and *freq0* sum variables (the interpreter will now treat them as out sum variables). Then, *undef* is called to reset their values to zero. This is necessary, as sum variables would not by default be reset when assigned to during interactive mode.

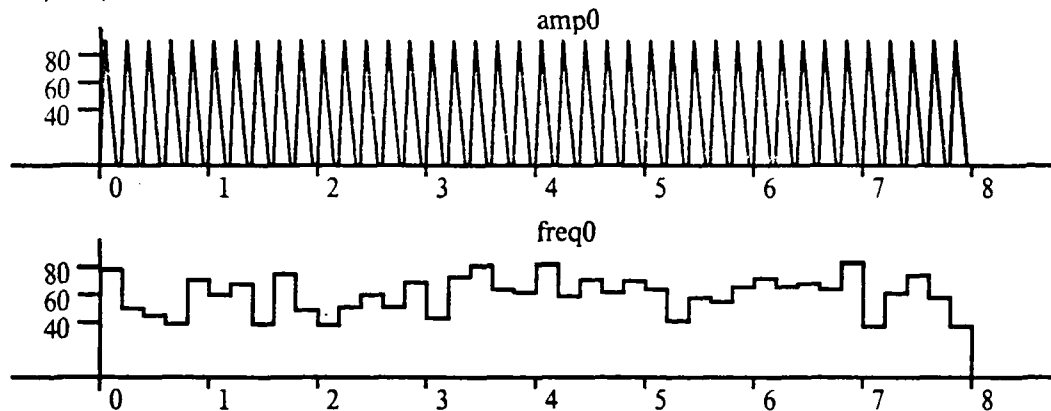
```
arctic> sum amp0, freq0;
arctic> undef;
Reinitializing amp0
Reinitializing freq0
result := NULL
```

Now, two prototypes are defined to implement the synthesizer.

```
arctic> Env is 90 * [ ramp ~ .2 | (1 - ramp) ~ .6 | zero ~ .2 ];
arctic> Note(in pitch) causes [ amp0 += Env; freq0 += pitch * unit ];
```

Given the Note prototype we can now generate some random notes and hear them.

```
arctic> repeat(Note(36 + irnd(48)), 40) ~ .2;
Not reinitializing amp0
Not reinitializing freq0
result := [ 0.0000, 8.0000 ]
arctic> play;
result := NULL
writing ampfreq.out
```



This example is finished, so we terminate the interpreter.

```
arctic> exit;
%
```

In this next example we illustrate a prototype that performs a time warping. The prototype *warp(original, modulator)* takes two functions of time, *original* and *modulator*, as arguments. *Warp* returns a new function constructed as follows: The *original* function is chopped into *N* equal length segments (*N* is set to 20 in this implementation). The *modulator* function is sampled at *N* equally spaced points along its length. Each segment of *original* is then stretched to a length proportional to the value of its corresponding *modulator* sample. The stretched segments are rejoined, and the result of this rejoining normalized to be the same length as *original*. See the figures that follow for examples.

Here is the code for *warp*, from the file *warp.a*:

```

warp_help(in orig, mod, i, n) is
  if i < n then [
    value segment, stretchfactor;
    [ tnorm(segment) ~~ stretchfactor |
      warp_help(orig, mod, i+1, n) ]];
    segment := extract(orig, orig.start + i/n*orig.length,
      orig.start + (i+1)/n*orig.length);
    stretchfactor := mod(mod.start + (i+.5)/n*mod.length);
  ]
  else
    zero ~ 0;

warp(in original, modulator) is
  tnorm(warp_help(original, modulator, 0, 20)) ~~ original.length;

```

Warp calls a recursive prototype *warp_help* to do the real work of extracting, stretching, and connecting the segments. The arguments to *warp_help* are the original function, the modulator function, the segment number currently being extracted (zero initially) and the total number of segments to be extracted (20 in this implementation). The function returned from *warp_help* is normalized using *tnorm* to be the same length as *original*.

Warp_help has a standard recursive form. The recursion terminates when the current segment, *i*, is greater than or equal to *n*, the number of segments to be processed. If so, *warp_help* returns a zero length function. Otherwise, it returns the extracted segment stretched to a length gotten from sampling the modulator, followed by the result of *warp_help* of the next segment.

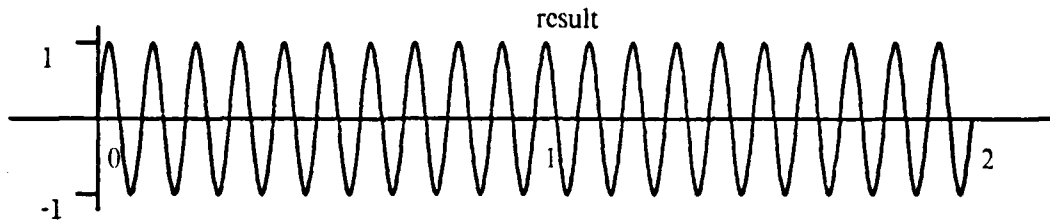
Note how *warp_help* depends on the fact that a parallel collection has a value that is the first expression in the collection, and the fact that the assignment to a variable happens before the value of the variable is used, even though lexically the assignment is written after the use. Note also that it would in general be impossible to evaluate *warp* in real-time (since it would be necessary to predict the future), but it is easily evaluated by the non-real-time ARCTIC interpreter.

Now we test out *warp*.

```

% arctic warp.a
arctic> value o, m;
arctic> o := sin(10)~2;
Reinitializing o
result := [ 0.0000, 2.0000 ]

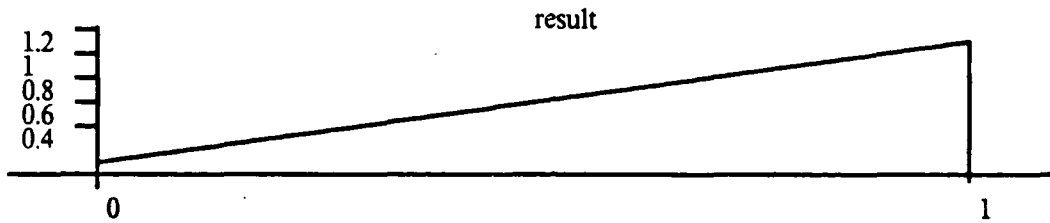
```



```

arctic> m := ramp + .1;
Reinitializing m
result := [ 0.0000, 1.0000 ]

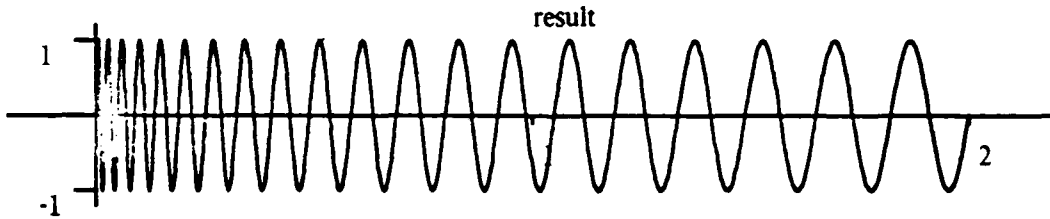
```



```

arctic> vary(o, m);
result := [ 0.0000, 2.0000 ]

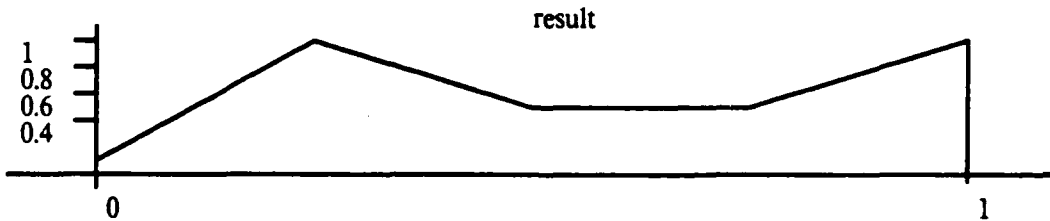
```



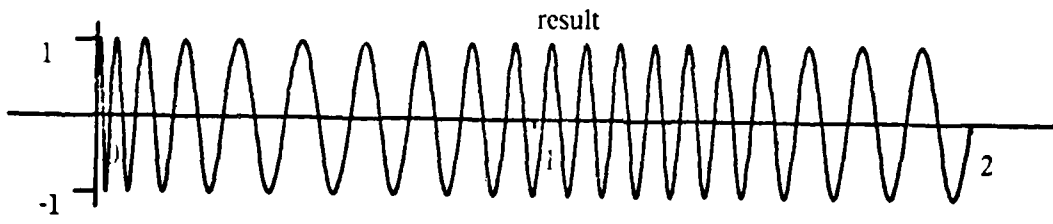
```

arctic> m := pwl(0,.1, 1,1, 2,.5, 3,.5, 4,1);
Reinitializing m
result := [ 0.0000, 1.0000 ]

```



```
arctic> vary(o, m);  
result := [0.0000, 2.0000]
```



```
arctic> exit;  
%
```

END

FILMED

MARCH, 19 88

DTIC