Appendix A

CLU Reference Manual

For a nominal fee universities can obtain an implementation of CLU by writing to the authors. At present there are implementations for the DEC20 and DEC Vax and Motorola 68000-based Unix systems.

A.1 Syntax

We use an extended BNF grammar to define the syntax. The general form of a production is

nonterminal ::= alternative | alternative | ... | alternative

The following extensions are used:

a , ... a list of one or more *a*s separated by commas: "a" or "a, a" or "a, a, a", etc.

{ a } a sequence of zero or more *a*s: " " or "a" or "a a", etc.

[a] an optional *a*: " " or "a".

Nonterminal symbols appear in lightface. Reserved words appear in boldface. All other terminal symbols are nonalphabetic and appear in lightface:

module	::=	{ equate } procedure
	|	{ equate } iterator
	|	{ equate } cluster
procedure	::=	idn = **proc** [parms] args [returns] [signals] [where] routine_body **end** idn

```
iterator      ::= idn = iter [ parms ] args [ yields ] [ signals ] [ where ]
                  routine_body
                  end idn
cluster       ::= idn = cluster [ parms ] is idn , ... [ where ]
                  cluster_body
                  end idn
parms         ::= [ parm , ... ]
parm          ::= idn , ... : type | idn , ... : type_spec
args          ::= ( [ decl , ... ] )
decl          ::= idn , ... : type_spec
returns       ::= returns ( type_spec , ... )
yields        ::= yields ( type_spec , ... )
signals       ::= signals ( exception , ... )
exception     ::= name [ ( type_spec , ... ) ]
where         ::= where restriction , ...
restriction   ::= idn has oper_decl , ...
                | idn in type_set
type_set      ::= { idn | idn has oper_decl , ... { equate } } | idn
oper_decl     ::= op_name , ... : type_spec
op_name       ::= name [ [ constant , ... ] ]
constant      ::= expression | type_spec
routine_body  ::= { equate }
                  { own_var }
                  { statement }
cluster_body  ::= { equate } rep = type_spec { equate }
                  { own_var }
                  routine { routine }
routine       ::= procedure | iterator
equate        ::= idn = constant | idn = type_set
own_var       ::= own decl
                | own idn : type_spec := expression
                | own decl , ... := invocation
type_spec     ::= null | bool | int | real | char | string | any | rep | cvt
                | array [ type_spec ] | sequence [ type_spec ]
                | record [ field_spec , ... ] | struct [ field_spec , ... ]
```

	|	oneof [field_spec , . . .] | variant [field_spec , . . .]
	|	proctype ([type_spec , . . .]) [returns] [signals]
	|	itertype ([type_spec , . . .]) [yields] [signals]
	|	idn [constant , . . .] | idn
field_spec	::=	name , . . . : type_spec
statement	::=	decl
	|	idn : type_spec := expression
	|	decl , . . . := invocation
	|	idn , . . . := invocation
	|	idn , . . . := expression , . . .
	|	primary **.** name := expression
	|	primary [expression] := expression
	|	invocation
	|	**while** expression **do** body **end**
	|	**for** [decl , . . .] **in** invocation **do** body **end**
	|	**for** [idn , . . .] **in** invocation **do** body **end**
	|	**if** expression **then** body
		{ **elseif** expression **then** body }
		[**else** body]
		end
	|	**tagcase** expression
		tag_arm { tag_arm }
		[**others** : body]
		end
	|	**return** [(expression , . . .)]
	|	**yield** [(expression , . . .)]
	|	**signal** name [(expression , . . .)]
	|	**exit** name [(expression , . . .)]
	|	**break**
	|	**continue**
	|	**begin** body **end**
	|	statement **resignal** name , . . .
	|	statement **except** { when_handler }
		[others_handler]
		end
tag_arm	::=	**tag** name , . . . [(idn : type_spec)] : body
when_handler	::=	**when** name , . . . [(decl , . . .)] : body
	|	**when** name , . . . (*) : body
others_handler	::=	**others** [(idn : type_spec)] : body
body	::=	{ equate }
		{ statement }

expression	::=	primary	
	|	(expression)	
	|	~ expression	% 6 (precedence)
	|	− expression	% 6
	|	expression ** expression	% 5
	|	expression // expression	% 4
	|	expression / expression	% 4
	|	expression * expression	% 4
	|	expression || expression	% 3
	|	expression + expression	% 3
	|	expression − expression	% 3
	|	expression < expression	% 2
	|	expression <= expression	% 2
	|	expression = expression	% 2
	|	expression >= expression	% 2
	|	expression > expression	% 2
	|	expression ~< expression	% 2
	|	expression ~<= expression	% 2
	|	expression ~= expression	% 2
	|	expression ~>= expression	% 2
	|	expression ~> expression	% 2
	|	expression & expression	% 1
	|	expression **cand** expression	% 1
	|	expression | expression	% 0
	|	expression **cor** expression	% 0
primary	::=	**nil** | **true** | **false**	
	|	int_literal | real_literal | char_literal | string_literal	
	|	idn	
	|	idn [constant , ...]	
	|	primary . name	
	|	primary [expression]	
	|	invocation	
	|	type_spec${ field , ... }	
	|	type_spec$[[expression :] [expression , ...]]	
	|	type_spec$name [[constant , ...]]	
	|	**force** [type_spec]	
	|	**up** (expression)	
	|	**down** (expression)	
invocation	::=	primary ([expression , ...])	
field	::=	name , ... : expression	

A.2 Lexical Considerations

A module is written as a sequence of tokens and separators. A *token* is a sequence of "printing" ASCII characters (octal value 40 through 176) representing a reserved word, an identifier, a literal, an operator, or a punctuation symbol. A *separator* is a "blank" character (space, vertical tab, horizontal tab, carriage return, newline, form feed) or a comment. In general, any number of separators may appear between tokens. Tokens and separators are described in more detail in the following sections.

A.2.1 Reserved Words

The following character sequences are reserved words:

any	down	int	record	tagcase
array	else	is	rep	then
begin	elseif	iter	resignal	true
bool	end	itertype	return	type
break	except	nil	returns	up
cand	exit	null	sequence	variant
char	false	oneof	signal	when
cluster	for	others	signals	where
continue	force	own	string	while
cor	has	proc	struct	yield
cvt	if	proctype	tag	yields
do	in	real		

Upper- and lowercase letters are not distinguished in reserved words. For example, "end," "END," and "eNd" are all the same reserved word. Reserved words appear in boldface in this document, except for names of types.

A.2.2 Identifiers

An *identifier* is a sequence of letters, digits, and underscores that begins with a letter or underscore and that is not a reserved word. As in reserved words, upper- and lowercase letters are not distinguished in identifiers.

In the syntax there are two different nonterminals for identifiers. The nonterminal *idn* is used when the identifier has scope (see section A.4.1); idns are used for variables, parameters, and module names, and as abbreviations for constants. The nonterminal *name* is used when the identifier is not subject to scope rules; names are used for record and structure selectors, oneof and variant tags, operation names, and exceptional condition names.

A.2.3 Literals

There are literals for naming objects of the built-in types null, bool, int, real, char, and string. Their forms are discussed in section A.3.

A.2.4 Operators and Punctuation Symbols

The following character sequences are used as operators and punctuation symbols:

```
(    :    “    <     ~<     +   ||
)    :=   ’    <=    ~<=    -   **
{    ,    \    =     ~=     *   //
}    .         ¿=    ~¿=    /   &
[    $         ¿     ~¿         |
]                               ~
```

A.2.5 Comments and Other Separators

A *comment* is a sequence of characters that begins with a percent sign (%), ends with a newline character, and contains only printing ASCII characters and horizontal tabs in between.

A *separator* is a blank character (space, vertical tab, horizontal tab, carriage return, newline, form feed) or a comment. Zero or more separators may appear between any two tokens, except that at least one separator is required between any two adjacent non-self-terminating tokens: reserved words, identifiers, integer literals, and real literals.

A.3 Type Specifications

Within a program, a type is specified by a syntactic construct called a *type_spec*. The type specification for a type with no parameters is just the identifier (or reserved word) naming the type. For parameterized types, the type specification consists of the identifier (or reserved word) naming the parameterized type, together with the parameter values.

In addition to the built-in types (null, bool, int, real, char, string, and any) and the built-in type classes (array, sequence, record, struct, variant, oneof, proctype, and itertype), there are also type_specs for user-defined types. These have the form

idn **[** [constant , . . .] **]**

where *idn* names the user_defined type, each *constant* must be computable at compile time (see section A.4.3), and constants of the appropriate types and number must be supplied in the order specified by the type.

There are three special type specifications that are used for implementing new abstractions: **rep, cvt**, and **type**. These forms are discussed in sections A.8.3 and A.8.4. Within a module, formal parameters declared with **type** can be used as type specifications.

Finally, identifiers that have been equated to type specifications can also be used as type specifications. Equates are discussed in section A.4.3.

Specifications for the built-in types and type classes are given in section A.9.

A.4 Scopes, Declarations, and Equates

We now describe how to introduce and use constants and variables, and the scope of constant and variable names.

A.4.1 Scoping Units

Scoping units follow the nesting structure of statements. Generally, a scoping unit is a body and an associated "heading." The scoping units are

1. from the start of a module to its end,

2. from a **cluster**, **proc**, or **iter** to the matching **end**,

3. from a **for**, **do**, or **begin** to the matching **end**,

4. from a **then** or **else** in an **if** statement to the end of the corresponding body,

5. from a **tag** or **others** in a **tagcase** statement to the end of the corresponding body,

6. from a **when** or **others** in an **except** statement to the end of the corresponding body, and

7. from the start of a type_set to its end.

In this section we discuss only cases 1–6; the scope in a *type_set* is discussed in section A.8.4.

The structure of scoping units is such that if one scoping unit overlaps another scoping unit (textually), then one is fully contained in the other. The contained scope is called a *nested* scope, and the containing scope is called a *surrounding* scope.

New constant and variable names may be introduced in a scoping unit. Names for constants are introduced by equates, which are syntactically restricted to appear grouped together at or near the beginning of scoping units. For example, equates may appear at the beginning of a body, but not after any statements in the body. By contrast, declarations, which introduce new variables, are allowed wherever statements are allowed, and hence may appear throughout a scoping unit.

In the syntax there are two distinct nonterminals for identifiers: *idn* and *name*. Any identifier introduced by an equate or declaration is an idn, as is the name of the module being defined and any operations the module has. An idn refers to a specific type or object. A name generally refers to a piece of something and is always used in context; for example, names are used as record selectors. The scope rules apply only to idns.

The scope rules are very simple:

1. An idn may not be redefined in its scope.

2. Any idn that is used as an external reference in a module may not be used for any other purpose in that module.

Unlike other "block-structured" languages, CLU prohibits the redefinition of an identifier in a nested scope. An identifier used as an external reference must name a module or constant in its program.

A.4.2 Declarations

Declarations introduce new variables. The scope of a variable is from its declaration to the end of the smallest scoping unit containing its declaration; hence variables must be declared before use.

There are two sorts of declarations: those with initialization and those without. Simple declarations (those without initialization) take the form

decl ::= idn, ... : type_spec

A simple declaration introduces a list of variables, all having the type given by *type_spec*. This type determines the types of objects that can be assigned to the variable. The variables introduced in a simple declaration initially denote no objects; that is, they are uninitialized. Attempts to use uninitialized variables (if not detected at compile time) cause the runtime exception

failure("uninitialized variable")

A declaration with initialization combines declarations and assignments into a single statement. It is entirely equivalent to one or more simple declarations followed by an assignment statement. The two forms of declaration with initialization are

idn : type_spec := expression

and

$\text{decl}_1, \ \ldots, \ \text{decl}_n$:= invocation

These are equivalent to (respectively)

idn : type_spec
idn := expression

and

$\text{decl}_1 \ \ldots \ \text{decl}_n$ % declaring $\text{idn}_1 \ \ldots \ \text{idn}_m$
$\text{idn}_1, \ \ldots, \ \text{idn}_m$:= invocation

In the second form, the order of the idns in the assignment statement is the same as in the original declaration with initialization. (The invocation must return m objects.)

A.4.3 Equates and Constants

An equate allows a single identifier to be used as an abbreviation for a constant that may have a lengthy textual representation. An equate also permits a mnemonic identifier to be used in place of a commonly used constant, such as a numerical value. We use the term "constant" in a very narrow sense here: Constants, in addition to being immutable, must be computable at compile time. Constants are either types (built-in or user-defined), or objects that are the results of evaluating constant expressions. (Constant expressions are defined later.)

The syntax of equates is

equate ::= idn = constant | idn = type_set
constant ::= type_spec | expression

This section describes only the first form of equate; discussion of *type_sets* is deferred to section A.8.4.

An equated identifier may be used as an expression. The value of such an expression is the constant to which the identifier is equated. An equated identifier cannot be used on the left-hand side of an assignment statement.

The scope of an equated identifier is the smallest scoping unit surrounding the equate defining it; here we mean the entire scoping unit, not just the portion after the equate. All the equates in a scoping unit must appear grouped near the beginning of the scoping unit. The exact placement of equates depends on the containing syntactic construct; usually equates appear at the beginnings of bodies.

Equates may be in any order within the group. Forward references among equates in the same scoping unit are allowed, but cyclic dependencies are illegal. For example,

```
x = y
y = z
z = 3
```

is a legal sequence of equates, but

```
x = y
y = z
z = x
```

is not. Since equates introduce idns, the scoping restrictions on idns apply (that is, the idns may not be defined more than once).

Identifiers may be equated to type specifications, thus giving abbreviations for type names. Since equates cannot have cyclic dependencies, though, directly recursive type specifications cannot be written. This does not prevent the definition of recursive types, since we can use clusters to write them.

A *constant expression* is an expression that can be evaluated at compile time to produce an immutable object of a built-in type. Specifically this includes

1. literals,
2. identifiers equated to constants,
3. formal parameters (see section A.8.4),
4. procedure and iterator names, including **force**[*t*] for any type *t*, and
5. invocations of procedure operations of the built-in constant types, provided that all operands and all results are constant expressions; however, we explicitly forbid the use of formal parameters as operands to invocations in constant expressions, since the values of formal parameters are not known at compile time.

The built-in constant types are null, int, real, bool, char, string, sequence types, oneof types, structure types, procedure types, and iterator types. Any invocation in a constant expression must terminate normally; a program is illegal if evaluation of any constant expression would signal an exception. Illegal programs will not be executed.

A.5 Assignment and Invocation

Two fundamental actions of CLU are assignment of computed objects to variables and invocation of procedures (and iterators) to compute objects. Other actions are composed from these two by using various control flow mechanisms. Since the correctness of assignments and invocations depends on a type-checking rule, we describe that rule first, then assignment, and finally invocation.

A.5.1 Type Inclusion

CLU is designed to allow complete compile-time type checking. The type of each variable is known by the compiler. Furthermore, the type of object that could result from the evaluation of any expression (invocation) is known at compile time. Hence every assignment can be checked at compile time to make sure that the variable is only assigned objects of its declared type. The rule is that an assignment $v := E$ is legal only if the set of objects defined by the type of E (loosely, the set of all objects that could possibly result from evaluating the expression) is included in the set of all objects that could be denoted by v.

Instead of speaking of the set of objects defined by a type, we generally speak of the type and say that the type of the expression must be *included in* the type of the variable. If it were not for the type any, the inclusion rule would be an equality rule. This leads to a simple interpretation of the type inclusion rule: *The type of a variable being assigned an expression must be either the type of the expression or the type any.*

A.5.2 Assignment

Assignment is the means of causing a variable to denote an object. The simplest form of assignment is

idn := expression

In this case the expression is evaluated and the resulting object is assigned to the variable. The expression must return a single object (whose type must be included in that of the variable).

There are two forms of assignment that assign to more than one variable at the same time:

idn , . . . := expression , . . .

and

idn , . . . := invocation

The first form of multiple assignment is a generalization of simple assignment. The first variable is assigned the first expression, the second variable the second expression, and so on. The expressions are all evaluated (from left to right) before any assignments are performed. The number of variables in the list must equal the number of expressions, no variable may occur more than once, and the type of each variable must include the type of the corresponding expression. There is no form of this statement with declarations.

The second form of multiple assignment allows retention of the objects resulting from an invocation returning two or more objects. The first variable is assigned the first object, the second variable the second object, and so on. The order of the objects is the same as in the **return** statement of the invoked routine. The number of variables must equal the number of objects returned, no variable may occur more than once, and the type of each variable must include the corresponding return type of the invoked procedure.

The assignment symbol := is used in two other syntactic forms that are not true assignments, but rather abbreviations for certain invocations. These forms are used for updating collections such as records and arrays (see section A.7.2).

A.5.3 Invocation

Invocation is the other fundamental action of CLU. In this section we discuss procedure invocation; iterator invocation is discussed in section A.7.6. However, up to and including passing of arguments, the two are the same.

Invocations take the form

primary ([expression , ...])

A *primary* is a slightly restricted form of expression, which includes variables and routine names, among other things (see the next section).

The sequence of activities in performing an invocation are as follows:

1. The primary is evaluated. It must evaluate to a procedure or iterator.

2. The expressions are evaluated, from left to right.

3. New variables are introduced corresponding to the formal arguments of the routine being invoked (that is, a new environment is created for the invoked routine to execute in).

4. The objects resulting from evaluating the expressions (the actual arguments) are assigned to the corresponding new variables (the formal arguments). The first formal is assigned the first actual, the second formal the second actual, and so on. The type of each expression must be included in the type of the corresponding formal argument.

5. Control is transferred to the routine at the start of its body.

An invocation is considered legal in exactly those situations where all the (implicit) assignments involved in its execution are legal.

It is permissible for a routine to assign an object to a formal argument variable; the effect is just as if that object were assigned to any other variable. From the point of view of the invoked routine, the only difference between its formal argument variables and its other local variables is that the formals are initialized by its caller.

Procedures can terminate in two ways: *normally*, returning zero or more objects, or *exceptionally*, signaling an exception. When a procedure terminates normally, the result objects become available to the caller and will (usually) be assigned to variables or passed as arguments to other routines. When a procedure terminates exceptionally, the flow of control will not go to the point of return of the invocation, but rather will go to an exception handler, as described in section A.7.13.

A.6 Expressions

An expression evaluates to an object in the CLU universe. This object is said to be the *result* or *value* of the expression. Expressions are used to name the object to which they evaluate. The simplest expressions are literals, variables, parameters, and routine names. These forms directly name their result object. More complex expressions are generally built up out of nested procedure invocations. The result of such an expression is the value returned by the outermost invocation.

Like many other languages, CLU has prefix and infix operators for the common arithmetic and comparison operations and uses the familiar syntax for array indexing and record component selection (for example, $a[i]$ and $r.s$). However, in CLU these notations are considered to be abbreviations for procedure invocations. This allows built-in types and user-defined types to be treated as uniformly as possible, and also allows the programmer to use familiar notation when appropriate.

In addition to invocation, four other forms are used to build complex expressions out of simpler ones. These are the conditional operators **cand** and **cor** (see section A.6.10), and the type conversion operations **up** and **down** (see section A.6.12).

There is a syntactically restricted form of expression called a *primary*. A primary is any expression that does not have a prefix or infix operator, or parentheses, at the top level. In certain places the syntax requires a primary rather than a general expression. This has been done to increase the readability of the resulting programs.

As a general rule, procedures with side effects should not be used in expressions, and programs should not depend on the order in which expressions are evaluated. However, to avoid surprises, the subexpressions of any expression are evaluated from left to right.

A.6.1 Literals

Integer, real, character, string, boolean, and null literals are expressions. The syntax for literals is given in the sections describing these types. The type of a literal expression is the type of the object named by the literal.

A.6.2 Variables

Variables are identifiers that denote objects of a given type. The type of a variable is the type given in the declaration of that variable and determines which objects may be denoted by the variable.

A.6.3 Parameters

Parameters are identifiers that denote constants supplied when a parameterized module is instantiated (see section A.8.4). The type of a parameter is the type given in the declaration of that parameter. Parameters of type **type** cannot be used as expressions.

A.6.4 Equated Identifiers

Equated identifiers denote constants. The type of an equated identifier is the type of the constant that it denotes. Identifiers equated to types and type_sets cannot be used as expressions.

A.6.5 Procedure and Iterator Names

Procedures and iterators may be defined either as separate modules or within a cluster. Those defined as separate modules are named by expressions of the form

idn ⟦ [constant , . . .] ⟧

The optional constants are the parameters of the procedure or iterator abstractions. (Constants are discussed in section A.4.3.)

When a procedure or iterator is defined as an operation of a type, the type name must be part of the name of the routine. The form for naming an operation of a type is

type_spec$name ⟦ [constant, . . .] ⟧

The type of a procedure or iterator name is just the type of the named routine.

A.6.6 Procedure Invocations

Procedure invocations have the form

primary ([expression , ...])

The primary is evaluated to obtain a procedure object, and then the expressions are evaluated left to right to obtain the argument objects. The procedure is invoked with these arguments, and the object returned is the result of the entire expression. For a more detailed discussion see section A.5.3.

Any procedure invocation $P(E_1, \ldots, E_n)$ must satisfy two constraints: The type of P must be of the form

proctype ($\mathrm{T_1}, \ldots, \mathrm{T_n}$) **returns** (R) **signals** (...)

and the type of each expression E_i must be included in the corresponding type T_i. The type of the entire invocation expression is given by R.

Procedures can also be invoked as statements (see section A.7.1).

A.6.7 Selection Operations

Arrays, sequences, records, and structures are collections of objects. Selection operations provide access to the individual elements or components of the collection. Simple notations are provided for invoking the *fetch* and *store* operations of array types, the *fetch* operation of sequence types, the *get* and *set* operations of record types, and the *get* operations of structure types. In addition, these short forms may be used for user-defined types with the appropriate properties.

An element selection expression has the form

primary [expression]

This form is a short form for an invocation of a *fetch* operation and is completely equivalent to

T$fetch(primary, expression)

where T is the type of *primary*. For example, if *a* is an array of integers, then

a[27]

is completely equivalent to the invocation

array[int]$fetch(a, 27)

The expression is legal whenever the corresponding invocation is legal. In other words, T (the type of *primary*) must provide a procedure operation named *fetch*, which takes two arguments whose types include the types of *primary* and *expression*, and which returns a single result.

The use of *fetch* for user-defined types should be restricted to types with arraylike behavior. Objects of such types will contain (along with other information) a collection of objects, where the collection can be indexed in some way. For example, it might make sense for an associative memory type to provide a *fetch* operation to access the value associated with a key. *Fetch* operations are intended for use in expressions; thus they should never have side effects.

The component selection expression has the form

primary **.** name

This form is short for an invocation of a *get_name* operation and is completely equivalent to

T$get_*name*(primary)

where T is the type of *primary*. For example, if x has type RT = record[first: int, second: real], then

x**.**first

is completely equivalent to

RT$get_first(x)

The expression is legal whenever the corresponding invocation is legal. In other words, T (the type of *primary*) must provide a procedure operation named *get_name*, which takes one argument whose type includes the type of *primary* and returns a single result.

The use of *get* operations for user-defined types should be restricted to types with recordlike behavior. Objects of such types will contain (along with other information) one or more named objects. For example, it might make sense for a type that implements channels to files to provide a get_author operation, which returns the name of the file's creator. *Get* operations are intended for use in expressions; thus they should never have side effects.

A.6.8 Constructors

Constructors are expressions that enable users to create and initialize arrays, sequences, records, and structures. Constructors are not provided for user-defined types.

An array constructor has the form

type_spec $ [⟦ expression: ⟧ ⟦ expression , ... ⟧]

The type specification must name an array type: array[T]. This is the type of the constructed array. The expression preceding the colon must evaluate to an integer and becomes the low bound of the constructed array. If this expression is omitted, the low bound is 1. The expressions following the colon are evaluated to obtain the elements of the array. They correspond (from left to right) to the indexes *low_bound*, *low_bound*+1, *low_bound*+2, ... For an array of type array[T], the type of each element expression in the constructor must be included in T.

An array constructor is computationally equivalent to an array *create* operation, followed by a number of array *addh* operations. However, such a sequence of operations cannot be written as an expression.

A sequence constructor has the form

type_spec $ [⟦ expression , ... ⟧]

The type specification must name a sequence type: sequence[T]. This is the type of the constructed sequence. The expressions are evaluated to obtain the elements of the sequence. They correspond (from left to right) to the indexes 1, 2, 3, ... For a sequence of type sequence[T], the type of each element expression in the constructor must be included in T.

A sequence constructor is computationally equivalent to a sequence *new* operation, followed by a number of sequence *addh* operations.

A record constructor has the form

type_spec $ { field , ... }

where

field ::= name , ... : expression

Whenever a field has more than one name, it is equivalent to a sequence of fields, one for each name. Thus the following two constructors are equivalent:

R${a, b: 7, c: 9}
R${a: 7, b: 7, c: 9}

where

R = record[a: int, b: int, c: int]

In a record constructor, the type specification must name a record type: record[$S_1 : T_1, \ldots, S_n : T_n$]. This will be the type of the constructed record. The component names in the field list must be exactly the names $S_1, \ldots, S_n$, although these names may appear in any order. The expressions are evaluated from left to right, and there is one evaluation per component name even if several component names are grouped with the same expression. The type of the expression for component S_i must be included in T_i. The results of these evaluations form the components of a newly constructed record. This record is the value of the entire constructor expression.

A structure constructor has the form

type_spec${ field , ... }

where (as for records)

field ::= name , ... : expression

Whenever a field has more than one name, it is equivalent to a sequence of fields, one for each name.

In a structure constructor, the type specification must name a structure type: struct[$S_1 : T_1, \ldots, S_n : T_n$]. This will be the type of the constructed structure. The component names in the field list must be exactly the names $S_1, \ldots, S_n$, although these names may appear in any order. The expressions are evaluated from left to right, and there is one evaluation per component name even if several component names are grouped with the same expression. The type of the expression for component S_i must be included in T_i. The results of these evaluations form the components of a newly constructed structure. This structure is the value of the entire constructor expression.

A.6.9 Prefix and Infix Operators

CLU allows prefix and infix notation to be used as a shorthand for the following operations. Table A.1 shows the shorthand form and the equivalent expanded form for each operation. Here T is the type of the first operand.

Operator notation is used most heavily for the built-in types, but may be used for user-defined types as well. When these operations are provided for user-defined types, they should always be side-effect-free, and they should mean roughly the same thing as they do for the built-in types. For example, the comparison operations should only be used for types that have a natural partial or total order. Usually the comparison operations (*lt*, *le*, *equal*, *ge*, *gt*) will be of type

proctype (T, T) **returns** (**bool**)

The other binary operations (for example, *add* and *sub*) will be of type

proctype (T, T) **returns** (T) **signals** (...)

and the unary operations will be of type

proctype (T) **returns** (T) **signals** (...)

Table A.1 Shorthand forms for operations.

Shorthand Form	Expansion
$expr_1 ** expr_2$	$T\$power(expr_1, expr_2)$
$expr_1 // expr_2$	$T\$mod(expr_1, expr_2)$
$expr_1 / expr_2$	$T\$div(expr_1, expr_2)$
$expr_1 * expr_2$	$T\$mul(expr_1, expr_2)$
$expr_1 \| expr_2$	$T\$concat(expr_1, expr_2)$
$expr_1 + expr_2$	$T\$add(expr_1, expr_2)$
$expr_1 - expr_2$	$T\$sub(expr_1, expr_2$
$expr_1 < expr_2$	$T\$lt(expr_1, expr_2)$
$expr_1 <= expr_2$	$T\$le(expr_1, expr_2)$
$expr_1 = expr_2$	$T\$equal(expr_1, expr_2)$
$expr_1 >= expr_2$	$T\$ge(expr_1, expr_2)$
$expr_1 > expr_2$	$T\$gt(expr_1, expr_2)$
$expr_1 \sim< expr_2$	$\sim (expr_1 < expr_2)$
$expr_1 \sim<= expr_2$	$\sim (expr_1 <= expr_2)$
$expr_1 \sim= expr_2$	$\sim (expr_1 = expr_2)$
$expr_1 \sim>= expr_2$	$\sim (expr_1 >= expr_2)$
$expr_1 \sim> expr_2$	$\sim (expr_1 > expr_2)$
$expr_1 \& expr_2$	$T\$and(expr_1, expr_2)$
$expr_1 \mid expr_2$	$T\$or(expr_1, expr_2)$
$-expr$	$T\$minus(expr)$
$\sim expr$	$T\$not(expr)$

A.6.10 Cand and Cor

Two additional binary operators are provided. These are the conditional *and* operator, **cand**, and the conditional *or* operator, **cor**.

expression$_1$ **cand**expression$_2$

is the boolean *and* of $expression_1$ and $expression_2$. However, if $expression_1$ is false, $expression_2$ is never evaluated.

expression$_1$ **cor**expression$_2$

is the boolean *or* of $expression_1$ and $expression_2$, but $expression_2$ is not evaluated unless $expression_1$ is false. For both **cand** and **cor**, $expression_1$ and $expression_2$ must have type bool.

Because of the conditional expression evaluation involved, uses of **cand** and **cor** are not equivalent to any procedure invocation.

A.6.11 Precedence

When an expression is not fully parenthesized, the proper nesting of subexpressions may be ambiguous. The following precedence rules are used to resolve such ambiguity. The precedence of each infix operator is given below—note that higher precedence operations are performed first and that prefix operators always have precedence over infix operators:

Precedence	Operators
5	$**$
4	$* *$ $/$ $//$
3	$+$ $-$ $\Vert$
2	< <= = >= > $\sim$< $\sim$<= $\sim$= $\sim$>= $\sim$>
1	& **cand**
0	$\mid$ **cor**

The order of evaluation for operators of the same precedence is from left to right, except for $**$, which is from right to left. The following examples illustrate the precedence rules:

Expression	Equivalent Form
$a + b//c$	$a + (b//c)$
$a + b - c$	$(a + b) - c$
$a + b ** c ** d$	$a + (b ** (c ** d))$
$a = b \mid c = d$	$(a = b) \mid (c = d)$
$-a * b$	$(-a) * b$

A.6.12 Up and Down

There are no implicit type conversions in CLU. Two forms of expression exist for explicit conversions. These are

up (expression)
down (expression)

Up and **down** may be used only within the body of a cluster operation. **Up** changes the type of the expression from the representation type of the cluster to the abstract type. **Down** converts the type of the expression from the abstract type to the representation type. These conversions are explained further in section A.8.3.

A.6.13 Force

CLU has a single built-in procedure generator called **force**. **Force** takes one type parameter and is written

force [type_spec]

The procedure **force**[T] has type

proctype (any) **returns** (T) **signals** (wrong_type)

If **force**[T] is applied to an object that is included in type T, it returns that object. If **force**[T] is applied to an object that is not included in type T, it signals "wrong_type."

Force is a necessary companion to the type any. The type any allows programs to pass around objects of arbitrary type. However, to do anything substantive with an object, one must use the primitive operations of that object's type. This raises a conflict with compile-time type checking, since an operation can be applied only when the arguments are known to be of the correct types. This conflict is resolved by using **force**. **Force**[T] allows a program to check, at runtime, that a particular object is actually of type T. If this check succeeds, the object can be used in all the ways appropriate for objects of type T.

For example, the procedure **force**[T] allows us to write the following code:

```
x: any := 3
y: int := force[int](x)
```

while the following is illegal because the type of y (int) does not include the type of the expression x (any):

```
x: any := 3
y: int := x
```

A.7 Statements

CLU is a statement-oriented language; that is, statements are executed for their side effects and do not return any values. Most statements are *control* statements that permit the programmer to define how control flows through the program. The real work is done by the *simple* statements: assignment and invocation. Assignment has already been discussed in section A.5.2; the invocation statement is discussed in the following. Two special statements that look like assignments but are really invocations are discussed in section A.7.2.

The syntax of CLU is defined to permit a control statement to control a group of equates, declarations, and statements rather than just a single statement. Such a group is called a *body* and has the form

```
body  ::= { equate }
 { statement }     % statements include declarations
```

Scope rules for bodies are discussed in section A.4.1. No special terminator is needed to signify the end of a body; reserved words used in the various compound statements serve to delimit the bodies. Occasionally it is necessary to indicate explicitly that a group of statements should be treated like a single statement; this is done by the block statement, discussed in section A.7.3.

A.7.1 Invocation Statement

An invocation statement invokes a procedure. Its form is the same as an invocation expression:

primary ([expression , ...])

The primary must evaluate to a procedure object, and the type of each expression must be included in the type of the corresponding formal argument for that procedure. The procedure may or may not return results; if it does return results, they are discarded.

A.7.2 Update Statements

Two special statements are provided for updating components of records and arrays. They may also be used with user-defined types with the appropriate properties. These statements resemble assignments syntactically, but they are really invocations.

The *element update* statement has the form

primary [expression_1] := expression_2

This form is merely a shorthand for an invocation of a *store* operation and is completely equivalent to the invocation statement

T$store(primary, expression_1, expression_2)

where T is the type of *primary*.

The element update statement is not restricted to arrays. The statement is legal if the corresponding invocation statement is legal. In other words, T (the type of *primary*) must provide a procedure operation named *store*, which takes three arguments whose types include those of *primary*, $expression_1$, and $expression_2$, respectively.

We recommend that the use of *store* for user-defined types be restricted to types with arraylike behavior, that is, types whose objects contain mutable collections of indexable elements. For example, it might make sense for an associative memory type to provide a *store* operation for changing the value associated with a key.

The *component update* statement has the form

primary . name := expression

This form is merely a shorthand for an invocation of a *set_name* operation and is completely equivalent to the invocation statement

T$set_name(primary, expression)

where T is the type of *primary*. For example, if x has type RT = record[first: int, second: real], then

```
x.first := 6
```

is completely equivalent to

```
RT$set_first(x, 6)
```

The component update statement is not restricted to records. The statement is legal if the corresponding invocation statement is legal. In other words, T (the type of *primary*) must provide a procedure operation called *set_name*, which takes two arguments whose types include the types of *primary* and *expression*, respectively.

We recommend that *set_* operations be provided for user-defined types only if recordlike behavior is desired, that is, only if it is meaningful to permit selected parts of the abstract object to be modified. In general, *set_* operations should not perform any substantial computation, except possibly checking that the arguments satisfy certain constraints.

A.7.3 Begin Statement

The **begin** statement permits a sequence of statements to be grouped together into a single statement. Its form is

begin body **end**

Since the syntax already permits bodies inside control statements, the main use of the block statement is to group statements together for use with the **except** statement (see section A.7.13).

A.7.4 Conditional Statement

The form of the conditional statement is

if expression **then** body
  { **elseif** expression **then** body }
  [**else** body]
  end

The expressions must be of type bool. They are evaluated successively until one is found to be true. The body corresponding to the first true expression is executed, and the execution of the **if** statement then terminates. If none of the expressions is true, the body in the **else** clause is executed (if the **else** clause exists). The **elseif** form provides a convenient way to write a multipath branch.

A.7.5 While Statement

The **while** statement has the form

while expression **do** body **end**

Its effect is repeatedly to execute the body as long as the expression remains true. The expression must be of type bool. If the value of the expression is true, the body is executed, and then the entire **while** statement is executed again. When the expression evaluates to false, execution of the **while** statement terminates.

A.7.6 For Statement

The only way an iterator can be invoked is by use of a **for** statement. The iterator produces a sequence of *items* (where an item is a group of zero or more objects) one item at a time; the body of the **for** statement is executed for each item in the sequence.

The **for** statement has the form

for [idn , ...] **in** invocation **do** body **end**

or

for [decl , ...] **in** invocation **do** body **end**

The invocation must be an iterator invocation. The *idn* form uses previously declared variables to serve as the loop variables, while the *decl* form introduces new variables, local to the **for** statement, for this purpose. In either case, the type of each variable must include the corresponding yield type of the invoked iterator.

Execution of the **for** statement proceeds as follows. First the iterator is invoked, and it either yields an item or terminates. If the iterator yields an item, its execution is temporarily suspended, the objects in the item are assigned to the loop variables, and the body of the **for** statement is executed. The next cycle of the loop is begun by resuming execution of the iterator from its point of suspension. Whenever the iterator terminates, the entire **for** statement terminates. If the **for** statement terminates, this also terminates the iterator.

A.7.7 Continue Statement

The **continue** statement has the form

continue

Its effect is to terminate execution of the body of the smallest loop statement in which it appears and to start the next cycle of that loop (if any).

A.7.8 Break Statement

The **break** statement has the form

break

Its effect is to terminate execution of the smallest loop statement in which it appears.

A.7.9 Tagcase Statement

The **tagcase** statement is a special statement provided for decomposing oneof and variant objects; it permits the selection of a body to perform based on the tag of the object.

The form of the **tagcase** statement is

tagcase expression
    tag_arm { tag_arm }
    [**others** : body]
    end

where

tag_arm ::= **tag** name , ... [(idn: type_spec)] : body

The expression must evaluate to a oneof or variant object. The tag of this object is then matched against the names on the tag_arms. When a match is found, if a declaration (*idn*: *type_spec*) exists, the value component of the object is assigned to the local variable *idn*. The matching body is then executed; *idn* is defined only in that body. If no match is found, the body in the **others** arm is executed.

In a syntactically correct **tagcase** statement, the following constraints are satisfied. The type of the expression must be some oneof or variant type T. The tags named in the tag_arms must be a subset of the tags of T, and no tag may occur more than once. If all tags of T are present, there is no **others** arm; otherwise an **others** arm must be present. Finally, on any tag_arm containing a declaration (*idn*: *type_spec*), *type_spec* must equal (not include) the type specified as corresponding in T to the tag or tags named in the tag_arm.

A.7.10 Return Statement

The form of the **return** statement is

return [(expression , ...)]

The **return** statement terminates execution of the containing procedure or iterator. If the **return** statement is in a procedure, the type of each expression must be included in the corresponding return type of the procedure. The expressions (if any) are evaluated from left to right, and the objects obtained become the results of the procedure. If the **return** statement occurs in an iterator, no results can be returned.

A.7.11 Yield Statement

Yield statements may occur only in the body of an iterator. The form of a **yield** statement is

yield [(expression , . . .)]

It has the effect of suspending operation of the iterator and returning control to the invoking **for** statement. The values obtained by evaluating the expressions (from left to right) are passed to the **for** statement to be assigned to the corresponding list of identifiers. The type of each expression must be included in the corresponding yield type of the iterator. After the body of the **for** loop has been executed, execution of the iterator is resumed at the statement following the **yield** statement.

A.7.12 Signal Statement

An exception is signaled with a **signal** statement, which has the form

signal name [(expression , . . .)]

A **signal** statement may appear anywhere in the body of a routine. The execution of a **signal** statement begins with evaluation of the expressions (if any), from left to right, to produce a list of *exception results*. The activation of the routine is then terminated. Execution continues in the caller, as described in section A.7.13.

The exception name must be either one of the exception names listed in the routine heading or *failure*. If the corresponding exception specification in the heading has the form

name(T_1, . . . , T_n)

there must be exactly n expressions in the **signal** statement, and the type of the ith expression must be included in T_i. If the name is *failure*, there must be exactly one expression present, of type string.

A.7.13 Except Statement

When a routine activation terminates by signaling an exception, the corresponding invocation (the text of the call) is said to *raise* that exception. By attaching handlers to statements, the caller can specify the action to be taken when an exception is raised.

A statement with handlers attached is called an **except** statement and has the form

statement **except** { when_handler }
[others_handler]
end

where

when_handler ::= **when** name , ... [(decl , ...)] : body
| **when** name , ... (*) : body
others_handler ::= **others** [(idn : type_spec)] : body

Let S be the statement to which the handlers are attached, and let X be the entire **except** statement. Each when_handler specifies one or more exception names and a body. The body is executed if an exception with one of those names is raised by an invocation in S. All the names listed in the when_handlers must be distinct. The optional others_handler is used to handle all exceptions not explicitly named in the when_handlers. Here S can be any form of statement, even another **except** statement.

If, during the execution of S, some invocation in S raises an exception E, control immediately transfers to the closest applicable handler—that is, the closest handler for E that is attached to a statement containing the invocation. When execution of the handler is complete, control passes to the statement following the one to which the handler is attached. Thus if the closest handler is attached to S, the statement following X is executed next. If execution of S completes without raising an exception, the attached handlers are not executed.

An exception raised inside a handler is treated the same as any other exception: Control passes to the closest handler for that exception. Note that an exception raised in some handler attached to S cannot be handled by any handler attached to S; either the exception is handled within the handler or it is handled by some handler attached to a statement containing X.

We now consider the forms of handlers in more detail. The form

when name , ... [(decl , ...)] : body

is used to handle exceptions with the given names when the exception results are of interest. The optional declared variables, which are local to the handler, are assigned the exception results before the body is executed. Every exception potentially handled by this form must have the same number of results as there are declared variables, and the types of the results must equal (not include) the types of the variables. The form

when name , ... (∗) : body

handles all exceptions with the given names, regardless of the existence of exception results; any actual results are discarded. Hence exceptions with differing numbers and types of results can be handled together.

The form

others [(idn : type_spec)] : body

is optional and must appear last in a handler list. This form handles any exception not handled by other handlers in the list. If a variable is declared, it must be of type string. The variable, which is local to the handler, is assigned a lowercase string representing the actual exception name; any results are discarded.

Note that exception results are ignored when matching exceptions to handlers; only the names of exceptions are used. Thus the following is illegal, in that int$*div* signals *zero_divide* without any results, but the closest handler has a declared variable:

```
begin
    y: int := 0
    x: int := 3 / y except when zero_divide (z: int): return end
    end except when zero_divide: return end
```

An invocation need not be surrounded by **except** statements that handle all potential exceptions. This policy was adopted because in many cases the programmer can prove that a particular exception will not arise. For example, the invocation int$*div*(x, 7) will never signal *zero_divide*. However, this policy does lead to the possibility that some invocation may raise an exception for which there is no handler. To avoid this situation, every routine body is contained in an implicit **except** statement of the form

```
begin routine_body end
    except when failure (s: string): signal failure(s)
        others (s: string): signal failure("unhandled exception: " || s)
        end
```

Failure exceptions are propagated unchanged; an exception named *name* becomes

failure("unhandled exception: *name*")

A.7.14 Resignal Statement

A **resignal** statement is a syntactically abbreviated form of exception handling:

statement **resignal** name , . . .

Each name listed must be distinct, and each must be either one of the condition names listed in the routine heading or *failure*. The **resignal** statement acts like an **except** statement containing a handler for each condition named, where each handler simply signals that exception with exactly the same results. Thus if the **resignal** clause names an exception with a specification in the routine heading of the form

name(T_1, . . . , T_n)

then effectively there is a handler of the form

when name (x_1: T_1, . . . , x_n: T_n): **signal** name(x_1, . . . , x_n)

As for an explicit handler of this form, every exception potentially handled by this implicit handler must have the same number of results as are declared in the exception specification, and the types of the results must equal the types listed in the exception specification.

A.7.15 Exit Statement

A *local* transfer of control can be effected by using an **exit** statement, which has the form

exit name [(expression , . . .)]

An **exit** statement is similar to a **signal** statement, except that where the **signal** statement *signals* an exception to the *calling* routine, the **exit** statement *raises* the exception directly in the *current* routine. An exception raised by an **exit** statement must be handled explicitly by a containing **except** statement with a handler of the form

when name , . . . [(decl , . . .)] : body

As usual, the types of the expressions in the **exit** statement must equal the types of the variables declared in the handler. The handler must be an explicit one; that is, exits to the implicit handlers of **resignal** statements or to the implicit *failure* handler enclosing a routine body are illegal.

A.8 Modules

A CLU program consists of a group of modules. Three kinds of modules are provided, one for each kind of abstraction we have found to be useful in program construction:

```
module ::= { equate } procedure
         | { equate } iterator
         | { equate } cluster
```

Procedures support procedural abstraction, iterators support control abstraction, and clusters support data abstraction.

A module defines a new scope. The identifiers introduced in the equates (if any) and the identifier naming the abstraction (the *module name*) are local to that scope (and therefore may not be redefined in an inner scope). Abstractions implemented by other modules are referred to by using nonlocal identifiers.

The existence of an externally established meaning for an identifier does not preclude a local definition for that identifier. Within a module, any identifier may be used in a purely local fashion or in a purely nonlocal fashion, but no identifier may be used in both ways.

A.8.1 Procedures

A procedure performs an action on zero or more *arguments*, and terminates by returning zero or more *results*. It may terminate in one of a number of *conditions*; one of these is the *normal condition*, while the others are *exceptional conditions*. Differing numbers and types of results may be returned in the different conditions.

The form of a procedure is

```
idn = proc [ parms ] args [ returns ] [ signals ] [ where ]
        routine_body
        end idn
```

where

```
args          ::= ( [ decl , ... ] )
returns       ::= returns ( type_spec , ... )
signals       ::= signals ( exception , ... )
exception     ::= name [ ( type_spec , ... ) ]
routine_body  ::= { equate } { own_var } { statement }
```

The *idn* following the **end** of the procedure must be the same as the *idn* naming the procedure. In this section we discuss nonparameterized procedures, in which the *parms* and **where** clauses are missing. Parameterized modules are discussed in section A.8.4; own variables are discussed in section A.8.5.

The heading of a procedure describes the way in which the procedure communicates with its caller. The *args* clause specifies the number, order, and types of arguments required to invoke the procedure, while the **returns** clause specifies the number, order, and types of results returned when the procedure terminates normally (by executing a **return** statement or reaching the end of its body). A missing **returns** clause indicates that no results are returned.

The **signals** clause names the exceptional conditions in which the procedure can terminate and specifies the number, order, and types of result objects returned in each exception. In addition to the exceptions explicitly named in the **signals** clause, any procedure can terminate in the *failure* exception. The *failure* exception returns with one result, a string object. All names of exceptions in the **signals** clause must be distinct, and none can be *failure*.

A procedure is an object of some procedure type. For a nonparameterized procedure, this type is derived from the procedure heading by removing the procedure name, rewriting the formal argument declarations with one *idn* per *decl*, deleting the names of formal arguments, and, finally, replacing **proc** by **proctype**.

As was discussed in section A.5.3, the invocation of a procedure causes the introduction of the formal variables, and the actual arguments are assigned to these variables. Then the procedure body is executed. Execution terminates when a **return** statement or a **signal** statement is executed or when the textual end of the body is reached. If a procedure that should return results reaches the textual end of the body, the procedure terminates in the condition

failure("no return values")

At termination the result objects, if any, are passed back to the invoker of the procedure.

A.8.2 Iterators

An iterator computes a sequence of items, one at a time, where each item is a group of zero or more objects. It has the form

idn = **iter** [parms] args [yields] [signals] [where]
    routine_body
    end idn

where

yields ::= **yields** (type_spec , ...)

The *idn* following the **end** of the iterator must be the same as the *idn* naming the iterator. In this section we discuss nonparameterized iterators, in which the *parms* and **where** clauses are missing. Parameterized modules are discussed in section A.8.4; own variables are discussed in section A.8.5.

The form of an iterator is very similar to the form of a procedure. There are only two differences:

1. An iterator has a **yields** clause in its heading in place of the **returns** clause of a procedure. The **yields** clause specifies the number, order, and types of objects yielded each time the iterator produces the next item in the sequence. If zero objects are yielded, the **yields** clause is omitted.

2. Within the iterator body, the **yield** statement is used to present the caller with the next item in the sequence. An iterator terminates in the same manner as a procedure, but it may not return any results.

An iterator is an object of some iterator type. For a nonparameterized iterator, this type is derived from the iterator heading by removing the iterator name, rewriting the formal argument declarations with one *idn* per *decl*, deleting the formal argument names, and, finally, replacing **iter** by **itertype**.

An iterator can be invoked only by a **for** statement. The execution of iterators is described in section A.7.6.

A.8.3 Clusters

A cluster is used to implement a new data type, distinct from any other built-in or user-defined data type. The form is

```
idn = cluster [ parms ] is idn , ... [ where ]
      cluster_body
      end idn
```

where

```
cluster_body  ::= { equate } rep = type_spec { equate }
                  { own_var }
                  routine { routine }
routine  ::= procedure | iterator
```

The *idn* following the **end** of the cluster must be the same as the *idn* naming the cluster. In this section we discuss nonparameterized clusters, in which the *parms* and **where** clauses are missing. Parameterized modules are discussed in section A.8.4; own variables are discussed in section A.8.5.

The primitive operations are named by the list of *idn*s following the reserved word **is**. All of the *idn*s in this list must be distinct.

To define a new data type, it is necessary to choose a *concrete representation* for the objects of the type. The special equate

rep = type_spec

within the cluster body identifies *type_spec* as the concrete representation. Within the cluster, **rep** may be used as an abbreviation for *type_spec*.

The identifier naming the cluster is available for use in the cluster body. Use of this identifier within the cluster body permits the definition of recursive types.

In addition to specifying the representation of objects, the cluster must implement the primitive operations of the type. The operations may be either procedures or iterators. Most of the routines in the cluster body define the primitive operations (those whose names are listed in the cluster heading). Any additional routines are *hidden*; they are private to the cluster and may not be named directly by users of the abstract type. All the routines must be named by distinct identifiers; the scope of these identifiers is the entire cluster.

Outside the cluster, the type's objects may only be treated abstractly (that is, manipulated by using the primitive operations). To implement the operations, however, it is usually necessary to manipulate the objects in terms of their concrete representation. It is also convenient sometimes to manipulate the objects abstractly. Therefore, inside the cluster it is possible to view the type's objects either abstractly or in terms of their representation. The syntax is defined to specify unambiguously, for each variable that refers to one of the type's objects, which view is being taken. Thus inside a cluster named T, a declaration

v: T

indicates that the object referred to by *v* is to be treated abstractly, while a declaration

w: **rep**

indicates that the object referred to by *w* is to be treated concretely. Two primitives, **up** and **down**, are available for converting between these two points of view. The use of **up** permits a type **rep** object to be viewed abstractly, while **down** permits an abstract object to be viewed concretely. For example, the following two assignments are legal for the declarations just given:

v := **up**(w)
w := **down**(v)

Only routines inside a cluster may use **up** and **down**. Note that **up** and **down** are used merely to inform the compiler that the object is going to be viewed abstractly or concretely, respectively.

A common place where the view of an object changes is at the interface to one of the type's operations. The user, of course, views the object abstractly, while inside the operation the object is viewed concretely. To facilitate this use, a special type specification, **cvt**, is provided. The use of **cvt** is restricted to the *args*, **returns**, **yields**, and **signals** clauses of routines inside a cluster, and it may be used at the top level only (for example, **array**[**cvt**] is illegal). When used inside the *args* clause, it means that **down** is applied implicitly to the argument object when it is assigned to the formal argument variable. When **cvt** is used in the **returns**, **yields**, or **signals** clause, it means that **up** is applied implicitly to the result object as it is returned (or yielded) to the caller. Thus **cvt** means abstract outside, concrete inside; when constructing the type of a routine, **cvt** is equivalent to the abstract type, but when type checking the body of a routine, **cvt** is equivalent to the representation type.

The **cvt** form does not introduce any new ability over what is provided by **up** and **down**. It is merely a shorthand for a common case.

The type of each routine is derived from its heading in the usual manner, except that each occurrence of **cvt** is replaced by the abstract type.

Inside the cluster, it is not necessary to use the compound form (*type_spec$op_name*) for naming locally defined routines. Furthermore, the compound form cannot be used for invoking hidden routines.

A.8.4 Parameters

Procedures, iterators, and clusters can all be parameterized. Parameterization permits a set of related abstractions to be defined by a single module. Recall that in each module heading there is an optional *parms* clause and an optional **where** clause. The presence of the *parms* clause indicates that the module is parameterized; the **where** clause states certain constraints on permissible actual values for the parameters.

The form of the *parms* clause is

```
[ parm , ... ]
```

where

```
parm ::= idn , ... : type_spec
       | idn , ... : type
```

Each parameter is declared like an argument. However, only the following types of parameters are legal: int, real, bool, char, string, null, and **type**. Parameters are limited to these types because the actual values for parameters are required to be constants that can be computed at compile time. This requirement ensures that all types are known at compile time and permits complete compile-time type checking.

In a parameterized module, the scope rules permit the parameters to be used throughout the remainder of the module. Type parameters can be used freely as type specifications, and all other parameters can be used freely as expressions. For example, type parameters can be used in defining the types of arguments and results:

p = **proc** [t: **type**] (x: t) **returns** (t)

To use a parameterized module, we must first *instantiate* it, that is, provide actual, constant values for the parameters. (The exact forms of such constants are discussed in section A.4.3.) The result of instantiation is a procedure, iterator, or type (where the parameterized module was a procedure, iterator, or cluster, respectively) that may be used just like a nonparameterized module of the same kind. For each distinct instantiations (that is, for each distinct list of actual parameters), a distinct procedure, iterator, or type is produced.

The meaning of a parameterized module is most easily understood in terms of rewriting. When the module is instantiated, the actual parameter values are substituted for the formal parameters throughout the module, and the *parms* clause and **where** clause are deleted. The resulting module is a regular (nonparameterized) module. In the case of a cluster, some of the operations may have additional parameters; further rewriting will be performed when these operations are used.

In the case of a type parameter, constraints on permissible actual types can be given in the **where** clause. The **where** clause lists a set of operations that the actual type is required to have and also specifies the type of each required operation. The **where** clause constrains the parameterized module as well; the only primitive operations of the type parameter that can be used are those listed in the **where** clause.

The form of the **where** clause is

```
where      ::= where restriction , ...
```

where

```
restriction  ::= idn has oper_decl , ...
             |   idn in type_set
oper_decl    ::= op_name , ... : type_spec
op_name      ::= name [ [ constant , ... ] ]
type_set     ::= { idn | idn has oper_decl , ... { equate } }
             |   idn
```

There are two forms of restrictions. In both forms, the initial *idn* must be a type parameter. The **has** form lists the set of required operations directly, by means of *oper_decls*. The *type_spec* in each *oper_decl* must name a routine type. Note that if some of the type's operations are parameterized, particular instantiations of those operations must be given. The **in** form requires that the actual type be a member of a *type_set*, a set of types having the required operations. The two identifiers in the type_set must match, and the notation is read like set notation; an example is

{t | t **has** f: . . . }

which means "the set of all types *t* such that *t* **has** *f* . . . ." The scope of the identifier is the type_set.

The **in** form is useful because an abbreviation can be given for a type_set via an equate. If it is helpful to introduce some abbreviations in defining the type_set, these are given in the optional equates within the type_set. The scope of these equates is the type_set.

A routine in a parameterized cluster may have a **where** clause in its heading and can place further constraints on the cluster parameters. For example, any type is permissible for the array element type, but the array *similar* operation requires that the element type have a *similar* operation. This means that array[T] exists for any type T, but array[T]$*similar* exists only when T$*similar* exists. Note that a routine need not include in its **where** clause any of the restrictions included in the cluster **where** clause.

A.8.5 Own Variables

Occasionally it is desirable to have a module that retains information internally between invocations. Without such an ability, the information would have to be either reconstructed at every invocation, which can be expensive (and may even be impossible if the information depends on previous invocations), or passed in through arguments, which is undesirable because the information is then subject to uncontrolled modification in other modules.

Procedures, iterators, and clusters can all retain information through the use of own variables. An own variable is similar to a normal variable, except that it exists for the life of the program, rather than being bound to the life of any particular routine activation. Syntactically, own variable declarations must appear immediately after the equates in a routine or cluster body; they cannot appear in bodies nested within statements. Own variable declarations have the form

```
own_var  ::= own decl
           |  own idn : type_spec := expression
           |  own decl ,  . . . := invocation
```

Note that initialization is optional.

Own variables are created when a program begins execution, and they always start out uninitialized. The own variables of a routine (including cluster operations) are initialized in textual order as part of the first invocation of that routine, before any statements in the body of the routine

```
C = cluster [t: type] is ...
   ...
   own x: int := init(...)
   P = proc (...)
      own y:  ...
       ...
      end P
   Q = proc [i: int] (...)
      own z:  ...
       ...
      end Q
   end C
```

Figure 1.1 Own variables.

are executed. Cluster own variables are initialized in textual order as part of the first invocation of the first cluster operation to be invoked (even if the operation does not use the own variables). Cluster own variables are initialized before any operation own variables are initialized.

Aside from the placement of their declarations, the time of their initialization, and their lifetime, own variables act just like normal variables and can be used in all the same places. As for normal variables, attempts to use uninitialized own variables (if not detected at compile time) cause the runtime exception

failure("uninitialized variable")

Own variable declarations in different modules always refer to distinct own variables, and distinct executions of programs never share own variables (even if the same module is used in several programs). Furthermore, own variable declarations within a parameterized module produce distinct own variables for each instantiation of the module. For a given instantiation of a parameterized cluster, all instantiations of the type's operations share the same set of cluster own variables, but distinct instantiations of parameterized operations have distinct routine own variables. For example, in the cluster in figure 1.1 there is a distinct x and y for every type t, and a distinct z for every type-integer pair (t, i).

Own variable declarations cannot be enclosed by an **except** statement, so care must be exercised when writing initialization expressions. If an exception is raised by an initialization expression, it will be treated as an exception raised, but not handled, in the body of the routine whose invocation caused the initialization to be attempted. This routine will then signal *failure* to its caller. In the example cluster just given, if procedure P were the first operation of C[string] to be invoked, causing initialization of

x to be attempted, then an *overflow* exception raised in the initialization of x would result in P signaling

failure("unhandled exception: overflow")

to its caller.

With the introduction of own variables, procedures and iterators become potentially mutable objects. If the abstract behavior of a routine depends on its history, this should be stated in its specification. In general, own variables should not be used to modify the abstract behavior of a module.

A.9 Built-in Types

In this section we describe the built-in types and type classes. Familiarity with chapter 2 is assumed in this discussion. In defining the built-in type classes, we do not depend on users satisfying any constraints beyond those that can be checked at compile time (that is, that a type parameter has an operation with a particular name and signature). This decision leads to more complicated specifications. For example, for the *elements* iterator for arrays, we must define the behavior when the loop modifies the array.

A.9.1 Null

null = **data type is** copy, equal, similar

Overview

The type null has exactly one, immutable object, represented by the literal nil. Nil is generally used as a place holder in type definitions using oneofs or variants.

Operations

copy = **proc** (n: null) **returns** (null)
 effects Returns nil.

equal = **proc** (n1, n2: null) **returns** (bool)
 effects Returns true.

similar = **proc** (n1, n2: null) **returns** (bool)
 effects Returns true.

end null

A.9.2 Booleans

bool = **data type is** and, or, not, equal, similar, copy

Overview

The two immutable objects of type bool, with literals true and false, represent logical truth values.

Operations

and = **proc** (b1, b2: bool) **returns** (bool)
  effects Returns the logical and of *b1* and *b2*.

or = **proc** (b1, b2: bool) **returns** (bool)
  effects Returns the logical or of *b1* and *b2*.

not = **proc** (b: bool) **returns** (bool)
  effects Returns the logical negation of *b*.

equal = **proc** (b1, b2: bool) **returns** (bool)
similar = **proc** (b1, b2: bool) **returns** (bool)
  effects Returns true if *b1* and *b2* are both true or both false; otherwise returns false.

copy = **proc** (b: bool) **returns** (bool)
  effects If *b* is true returns true; otherwise returns false.

end bool

A.9.3 Integers

int = **data type is** add, sub, mul, minus, div, mod, power, abs, max, min, lt, le, ge, gt, equal, similar, copy, from_to_by, from_to, parse, unparse

Overview

Objects of type int are immutable and are intended to model a subrange of the mathematical integers. The exact range is not part of the language definition and can vary somewhat from implementation to implementation. Each implementation is constrained to provide a closed interval [*int_min*, *int_max*], with $int\_min < 0$ and $int\_max \geq char\_top$ (the number of characters—see section A.9.5). An *overflow* exception is signaled by an operation if the result would lie outside this interval.

Integer literals are written as a sequence of one or more decimal digits.

Operations

add = **proc** (x, y: int) **returns** (int) **signals** (overflow)
sub = **proc** (x, y: int) **returns** (int) **signals** (overflow)
mul = **proc** (x, y: int) **returns** (int) **signals** (overflow)

effects These are the standard integer addition, subtraction, and multiplication operations. They signal *overflow* if the result would lie outside the represented interval.

minus = **proc** (x: int) **returns** (int) **signals** (overflow)
effects Returns the negative of x; signals *overflow* if the result would lie outside the represented interval.

div = **proc** (x, y: int) **returns** (int) **signals** (zero_divide, overflow)
effects Signals *zero_divide* if $y = 0$. Otherwise returns the integer quotient of dividing x by y; signals *overflow* if the result would lie outside the represented interval.

mod = **proc** (x, y: int) **returns** (int) **signals** (zero_divide, overflow)
effects Signals *zero_divide* if $y = 0$. Otherwise returns the integer remainder of dividing x by y; signals *overflow* if the result would lie outside the represented interval.

power = **proc** (x, y: int) **returns** (int) **signals** (negative_exponent, overflow)
effects Signals *negative_exponent* if $y < 0$. Otherwise returns x^y; signals *overflow* if the result would lie outside the represented interval.

abs = **proc** (x: int) **returns** (int) **signals** (overflow)
effects Returns the absolute value of x; signals *overflow* if the result would lie outside the represented interval.

max = **proc** (x, y: int) **returns** (int)
effects Returns the larger of x and y.

min = **proc** (x, y: int) **returns** (int)
effects Returns the smaller of x and y.

lt = **proc** (x, y: int) **returns** (bool)
gt = **proc** (x, y: int) **returns** (bool)
le = **proc** (x, y: int) **returns** (bool)
ge = **proc** (x, y: int) **returns** (bool)
equal = **proc** (x, y: int) **returns** (bool)
effects These are the standard ordering relations.

similar = **proc** (x, y: int) **returns** (bool)
effects Returns $(x = y)$.

copy = **proc** (x: int) **returns** (y: int)
effects Returns y such that $x = y$.

from_to_by = **iter** (from, to, by: int) **yields** (int)

effects Yields the integers from *from* to *to*, incrementing by *by* each time, that is, yields *from*, *from*+*by*, ... , $from + n * by$, where n is the largest positive integer such that $from + n * by \leq to$. If $by = 0$, then yields *from* indefinitely. Yields nothing if $from > to$ and $by > 0$, or if $from < to$ and $by < 0$.

from_to = **iter** (from, to: int) **yields** (int)
effects Identical to *from_to_by*(*from*, *to*, 1).

parse = **proc** (s: string) **returns** (int) **signals** (bad_format, overflow)
effects *s* must be an integer literal, with an optional leading plus or minus sign; if *s* is not of this form, signals *bad_format*. Otherwise returns the integer corresponding to *s*; signals *overflow* if the result would be outside of the represented interval.

unparse = **proc** (x: int) **returns** (string)
effects Produces the string corresponding to the integer value of x, preceded by a minus sign if $x < 0$. Leading zeros are suppressed, and there is no leading plus sign for positive integers.

end int

A.9.4 Reals

real = **data type is** add, sub, mul, minus, div, power, abs, max, min, exponent, mantissa, i2r, r2i, trunc, parse, unparse, lt, le, ge, gt, equal, similar, copy

Overview

The type real models a subset of the mathematical real numbers. Reals are immutable and are written as a *mantissa* with an optional *exponent*. A mantissa is either a sequence of one or more decimal digits or two sequences (one of which may be empty) joined by a period. The mantissa must contain at least one digit. An exponent is E or e, optionally followed by + or −, followed by one or more decimal digits. An exponent is required if the mantissa does not contain a period. As is usual, $m\mathrm{E}x = m * 10^x$. Examples of real literals are:
ch3.14 ch3.14E0 ch314e−2 ch.0314E+2
ch3. ch.14
Each implementation represents numbers in
chD $= \{-\mathrm{real\_max}, -\mathrm{real\_min}\} \cup \{0\} \cup \{\mathrm{real\_min}, \mathrm{real\_max}\}$
where
ch0 $< \mathrm{real\_min} < 1 < \mathrm{real\_max}$
Numbers in D are approximated by the implementation with precision p such that

$\forall \mathrm{r} \in \mathrm{D}$ $\quad \mathrm{Approx(r)} \in \mathrm{Real}$
$\forall \mathrm{r} \in \mathrm{Real}$ $\quad \mathrm{Approx(r)} = \mathrm{r}$
$\forall \mathrm{r} \in \mathrm{D} - \{0\}$ $\quad |(\mathrm{Approx(r)} - \mathrm{r})/\mathrm{r}| < 10^{1-\mathrm{p}}$ We define
$\forall \mathrm{r}, \mathrm{s} \in \mathrm{D}$ $\quad \mathrm{r} \le \mathrm{s} \Rightarrow \mathrm{Approx(r)} \le \mathrm{Approx(s)}$
$\forall \mathrm{r} \in \mathrm{D}$ $\quad \mathrm{Approx(-r)} = -\mathrm{Approx(r)}$

Max_width and *Exp_width* to be the smallest integers such that every nonzero element of real can be represented in "standard" form (exactly one digit, not zero, before the decimal point) with no more than *Max_width* digits of mantissa and no more than *Exp_width* digits of exponent.
Real operations signal an exception if the result of a computation lies outside of D; *overflow* occurs if the magnitude exceeds *real_max*, and *underflow* occurs if the magnitude is less than *real_min*.

Operations

add = **proc** (x, y: real) **returns** (real) **signals** (overflow, underflow)
effects Computes the sum z of x and y; signals *overflow* or *underflow* if z is outside of D, as explained earlier. Otherwise returns *approx*(z) such that
ch $(\mathrm{x}, \mathrm{y} \ge 0 \vee \mathrm{x}, \mathrm{y} \le 0) \Rightarrow \mathrm{add(x, y)} = \mathrm{Approx(x + y)}$
ch $\mathrm{add(x, y)} = (1 + \epsilon)(\mathrm{x} + \mathrm{y})$ ch$|\epsilon| < 10^{1-\mathrm{p}}$
ch $\mathrm{add(x, 0)} = \mathrm{x}$
ch $\mathrm{add(x, y)} = \mathrm{add(y, x)}$
ch $\mathrm{x} \le \mathrm{x}' \Rightarrow \mathrm{add(x, y)} \le \mathrm{add(x', y)}$

sub = **proc** (x, y: real) **returns** (real) **signals** (overflow, underflow)

effects Computes $x - y$; the result is identical to $add(x, -y)$.

minus = **proc** (x: real) **returns** (real)
effects Returns $-x$.

mul = **proc** (x, y: real) **returns** (real) **signals** (overflow, underflow)
effects Returns $approx(x * y)$; signals *overflow* or *underflow* if $x * y$ is outside of D.

div = **proc** (x, y: real) **returns** (real)
signals (zero_divide, overflow, underflow)
effects If $y = 0$, signals *zero_divide.* Otherwise returns $approx(x/y)$; signals *overflow* or *underflow* if x/y is outside of D.

power = **proc** (x, y: real) **returns** (real)
signals (zero_divide, complex_result, overflow, underflow)
effects If $x = 0$ and $y < 0$, signals *zero_divide.* If $x < 0$ and y is nonintegral, signals *complex_result.* Otherwise returns an approximation to x^y; signals *overflow* or *underflow* if x^y is outside of D.

abs = **proc** (x: real) **returns** (real)
effects Returns the absolute value of x.

max = **proc** (x, y: real) **returns** (real)
effects Returns the larger of x and y.

min = **proc** (x, y: real) **returns** (real)
effects Returns the smaller of x and y.

exponent = **proc** (x: real) **returns** (int) **signals** (undefined)
effects If $x = 0$, signals *undefined.* Otherwise returns the exponent that would be used in representing x as a literal in standard form, that is, returns
$\mathrm{chmax}(\{\mathrm{i}|\mathrm{abs}(\mathrm{x}) \geq 10^{\mathrm{i}}\})$

mantissa = **proc** (x: real) **returns** (real)
effects Returns the mantissa of x when represented in standard form, that is, returns $approx(x/10^e)$, where $e = exponent(x)$. If $x = 0.0$, returns 0.0.

i2r = **proc** (i: int) **returns** (real) **signals** (overflow)
effects Returns $approx(i)$; signals overflow if i is not in D.

r2i = **proc** (x: real) **returns** (int) **signals** (overflow)
effects Rounds x to the nearest integer and toward zero in case of a tie. Signals *overflow* if the result lies outside the represented range of integers.

trunc = **proc** (x: real) **returns** (int) **signals** (overflow)
effects Truncates x toward zero; signals *overflow* if the result would be outside the represented range of integers.

parse = **proc** (s: string) **returns** (real) **signals** (bad_format, overflow, underflow)
effects Computes the exact value z corresponding to a real or integer literal and returns $approx(z)$. s must be a real or integer literal with an optional leading plus or minus sign; otherwise signals *bad_format*. Signals *underflow* or *overflow* if z is not in D.

unparse = **proc** (x: real) **returns** (string)
effects Returns a real literal such that $parse(unparse(x)) = x$. The general form of the literal is
ch [$-$] *i_field.f_field* [$e \pm$ *x_field*]
Leading zeros in *i_field* and trailing zeros in *f_field* are suppressed. If x is integral and within the range of CLU integers, then *f_field* and the exponent are not present. If x can be represented by a mantissa of no more than *Max_width* digits and no exponent (that is, if $-1 \leq exponent(arg1) < Max\_width$), then the exponent is not present. Otherwise the literal is in standard form, with *Exp_width* digits of exponent.

lt = **proc** (x, y: real) **returns** (bool)
le = **proc** (x, y: real) **returns** (bool)
ge = **proc** (x, y: real) **returns** (bool)
gt = **proc** (x, y: real) **returns** (bool)
equal = **proc** (x, y: real) **returns** (bool)
effects The standard ordering relations.

similar = **proc** (x, y: real) **returns** (bool)
effects Returns $(x = y)$.

copy = **proc** (x: real) **returns** (real)
effects Returns y such that $x = y$.

end real

A.9.5 Characters

char = **data type is** i2c, c2i, lt, le, ge, gt, equal, similar, copy

Overview

Type char provides the alphabet for text manipulation. Characters are immutable and form an ordered set. Every implementation must provide at least 128, but no more than 512, characters; the first 128 characters are the ASCII characters in their standard order.

Operations *i2c* and *c2i* convert between ints and chars. The smallest character corresponds to zero, and characters are numbered sequentially up to *char_top*, the integer corresponding to the largest character. This numbering determines the ordering of the characters.

Printing ASCII characters (octal 40 through octal 176), other than single quote or backslash, can be written as that character enclosed in single quotes. Any character can be written by enclosing one of the following escape sequences in single quotes:

escape sequence	**character**
\ ’	’ (single quote)
\ “	’ (double quote)
\\	\ (backslash)
\ n	NL (newline)
\ t	HT (horizontal tab)
\ p	FF (form feed, newpage)
\ b	BS (backspace)
\ r	CR (carriage return)
\ v	VT (vertical tab)
\ ***	specified by octal value (exactly three octal digits)

The escape sequences may also be written using upper case letters. Examples of character literals are

ch‘7’ ch‘a’ ch“ ” ch‘\”’

ch‘\’’ ch‘\B’ ch‘\177’

Operations

i2c = **proc** (x: int) **returns** (char) **signals** (illegal_char)

effects Returns the character corresponding the x; signals *illegal_argument* if x is not in the range [0, *char_top*].

c2i = **proc** (c: char) **returns** (int)

effects Returns the integer corresponding to c.

lt = **proc** (c1, c2: char) **returns** (bool)
le = **proc** (c1, c2: char) **returns** (bool)
ge = **proc** (c1, c2: char) **returns** (bool)
gt = **proc** (c1, c2: char) **returns** (bool)
equal = **proc** (c1, c2: char) **returns** (bool)

effects These are the standard ordering relations, where the order is consistent with the numbering of characters.

similar = **proc** (c1, c2: char) **returns** (bool)
effects Returns (*c1* = *c2*).

copy = **proc** (c1: char) **returns** (c2: char)
effects Returns *c2* such that *c1* = *c2*.

end char

A.9.6 Strings

string = **data type is** size, empty, concat, append, fetch, rest, indexs, indexc, substr, lt, le, ge, gt, equal, similar, copy, c2s, s2ac, ac2s, s2sc, sc2s, chars

Overview

Type string is used for representing text. A string is an immutable sequence of zero or more characters. The characters of a string are indexed sequentially starting from one. Strings are lexicographically ordered based on the ordering for characters. A string is written as a sequence of zero or more character representations enclosed in double quotes. Within a string literal, a printing ASCII character other than double quote or backslash is represented by itself. Any character can be represented by using the escape sequences listed for characters. Examples of string literals are

"Item\tCost"   "altmode (\033) = \\033"   " "   " "

If the result of a string operation would be a string containing more than *int_max* characters, the operation signals *failure*.

Operations

size = **proc** (s: string) **returns** (int)
effects Returns the number of characters in *s*.

empty = **proc** (s: string) **returns** (bool)
effects Returns true if *s* is empty (contains no characters); otherwise returns false.

concat = **proc** (s1, s2: string) **returns** (string)
effects Returns a new string containing the characters of *s1* followed by the characters of *s2*. Signals *failure* if the new string would contain more than *int_max* characters.

append = **proc** (s: string, c: char) **returns** (string)
effects Returns a new string containing the characters of *s* followed by *c*. Signals *failure* if the new string would contain more than *int_max* characters.

fetch = **proc** (s: string, i: int) **returns** (char) **signals** (bounds)

effects Signals *bounds* if $i < 0$ or $i > size(s)$; otherwise returns the ith character of s.

rest = **proc** (s: string, i: int) **returns** (string) **signals** (bounds)
effects Signals *bounds* if $i < 0$ or $i > size(s) + 1$; otherwise returns a new string containing the characters $s[i], s[i+1], \ldots, s[size(s)]$. Note that if $i = size(s)+1$, *rest* returns the empty string.

indexs = **proc** (s1, s2: string) **returns** (int)
effects If *s1* occurs as a substring in *s2*, returns the least index at which *s1* occurs. Returns 0 if *s1* does not occur in *s2*, and 1 if *s1* is the empty string. For example,
chindexs("bc", "abcbc") = 2
chindexs("", "abcde") = 1

indexc = **proc** (c: char, s: string) **returns** (int)
effects If c occurs in s, returns the least index at which c occurs; returns 0 if c does not occur in s.

substr = **proc** (s: string, at, cnt: int) **returns** (string)
signals (bounds, negative_size)
effects If $cnt < 0$, signals *negative_size*. If $at < 1$ or $at > size(s) + 1$, signals *bounds*. Otherwise returns a new string containing the characters $s[at], s[at+1], \ldots$; the new string contains $min(cnt, size - at + 1)$ characters. For example,
chsubstr ("abcdef", 2, 3) = "bcd"
chsubstr ("abcdef", 2, 7) = "bcdef"
chsubstr ("abcdef", 7, 1) = ""
Note that if $min(cnt, size - at + 1) = 0$, *substr* returns the empty string.

lt = **proc** (s1, s2: string) **returns** (bool)
le = **proc** (s1, s2: string) **returns** (bool)
ge = **proc** (s1, s2: string) **returns** (bool)
gt = **proc** (s1, s2: string) **returns** (bool)
equal = **proc** (s1, s2: string) **returns** (bool)
effects These are the usual lexicographic ordering relations on strings, based on the ordering of characters. For example,
ch"abc" < "aca"
ch"abc" < "abca"

similar = **proc** (s1, s2: string) **returns** (bool)
effects Returns true if $s1 = s2$; otherwise returns false.

copy = **proc** (s1: string) **returns** (s2: string)
effects Returns *s2* such that $s1 = s2$.

c2s = **proc** (c: char) **returns** (string)
effects Returns a string containing *c* as its only character.

s2ac = **proc** (s: string) **returns** (array[char])
effects Stores the characters of *s* as elements of a new array of characters, *a*. The low bound of the array is 1, the size is *size*(*s*), and the *i*th element of the array is the *i*th character of *s*, for $1 \leq i \leq size(s)$.

ac2s = **proc** (a: array[char]) **returns** (string)
effects Does the inverse of *s2ac*. The result is a string with characters in the same order as in *a*. That is, the *i*th character of the string is the $(i + low(a) - 1)$th element of *a*.

s2sc = **proc** (s: string) **returns** (sequence[char])
effects Transforms a string into a sequence of characters. The size of the sequence is *size*(*s*). The *i*th element of the sequence is the *i*th character of *s*.

sc2s = **proc** (s: sequence[char]) **returns** (string)
effects Does the inverse of *s2sc*. The result is a string with characters in the same order as in *s*. That is, the *i*th character of the string is the *i*th element of *s*.

chars = **iter** (s: string) yields (char)
effects Yields, in order, each character of *s*.

end string

A.9.7 Any

A type specification is used to restrict the class of objects that a variable can denote, a procedure or iterator can take as arguments, a procedure can return, and so forth. There are times when no restrictions are desired, when any object is acceptable. At such times the type specification *any* is used. For example, one might wish to implement a table mapping strings to arbitrary objects, with the intention that different strings should map to objects of different types. The lookup operation, used to get the object corresponding to a string, would have its result declared to be of type any.

The type any is the *union* of all possible types. Every object is of type any, as well as being of some base type. The type any has no operations; however, the base type of an object can be tested at runtime (see section A.6.13).

A.9.8 Arrays

array = **data type** [t: type] **is** create, new, predict, fill, low, high, size, empty, set_low, trim, fetch, bottom, top, store, addh, addl, remh, reml, elements, indexes, equal, similar, similar1, copy, copy1, fill_copy

Overview

Arrays are one-dimensional mutable objects that can grow and shrink dynamically. Each array has a low bound and a sequence of elements that are indexed sequentially, starting from the low bound. All elements of an array are of the same type. Arrays can be created by calling array operations *create*, *new*, *fill*, *fill_copy*, and *predict*. They can also be created by means of a special constructor form that specifies the array low bound, and an arbitrary number of initial elements. For example,

charray[int]$[5: 1, 2, 3, 4]

creates an integer array with low bound 5 and four elements. The low bound can be omitted if it is 1; for example,

charray[string]$["a", "b", "c"]

creates a three-element array with low bound 1. Array constructors are discussed further in section A.6.8.

Operations *low*, *high*, and *size* return the current low and high bounds and size of the array. For array a, $size(a)$ is the number of elements in a, and $high(a) = low(a) + size(a) - 1$.

For any index i between the low and high bound of an array, there is a defined element $a[i]$. Any operation call that receives as an index an integer outside the defined range terminates with a *bounds* exception. Any call that would lead to an array whose low or high bound or size is outside the defined range of integers terminates with the *failure* exception.

Operations *similar*, *similar1*, *copy*, and *fill_copy* require that the element type t provide certain operations. No constraints are assumed on the behavior of these t operations; the behavior of the array operations is defined even if the t operation behaves strangely (for example, modifies the array being copied).

Operations

create = **proc** (lb: int) **returns** (array[t])
  effects Returns a new, empty array with low bound *lb*.

new = **proc** ( ) **returns** (array[t])
  effects Returns a new, empty array with low bound 1.

predict = **proc** (lb, cnt: int) **returns** (array[t])

effects Returns a new, empty array with low bound *lb*. Argument *cnt* is a prediction of how many *addh*s or *addl*s are likely to be performed on this new array. If $cnt > 0$, *addh*s are expected; otherwise *addl*s are expected. These operations may execute faster than if the array had been produced by calling *create*.

fill = **proc** (lb, cnt: int, elem: t) **returns** (array[t]) **signals** (negative_size)
effects If $cnt < 0$, signals *negative_size*. Returns a new array with low bound *lb* and size *cnt*, and with *elem* as each element; if this new array would have high bound $< int\_min$ or $> int\_max$, signals failure.

low = **proc** (a: array[t]) **returns** (int)
effects Returns the low bound of *a*.

high = **proc** (a: array[t]) **returns** (int)
effects Returns the high bound of *a*.

size = **proc** (a: array[t]) **returns** (int)
effects Returns a count of the number of elements of *a*.

empty = **proc** (a: array[t]) **returns** (bool)
effects Returns true if *a* contains no elements; otherwise returns false.

set_low = **proc** (a: array[t], lb: int)
modifies *a*
effects Modifies the low and high bounds of *a*; the new low bound of *a* is *lb* and the new high bound is $high(a_{\text{pre}} - lb + low(a_{\text{pre}}))$. If the new high bound is outside the represented range of integers, signals *failure* and does not modify *a*.

trim = **proc** (a: array[t], lb, cnt: int) **signals** (bounds, negative_size)
modifies *a*
effects If $lb < low(a)$ or $lb > high(a) + 1$, signals *bounds*. If $cnt < 0$, signals *negative_size*. Otherwise modifies *a* by removing all elements with index $< lb$ or greater than or equal to $lb + cnt$; the new low bound is *lb*. For example, if *a* = array[int]$[1,2,3,4,5], then
trim(a, 2, 2) results in *a* having value [2: 2, 3]
trim(a, 4, 3) results in *a* having value [4: 4, 5]

fetch = **proc** (a: array[t], i: int) **returns** (t) **signals** (bounds)
effects If $i < low(a)$ or $i > high(a)$, signals *bounds*; otherwise returns the element of *a* with index *i*.

store = **proc** (a: array[t], i: int, elem: t) **signals** (bounds)
modifies *a*.
effects If $i < low(a)$ or $i > high(a)$, signals *bounds*; otherwise makes *elem* the element of *a* with index *i*.

bottom = **proc** (a: array[t]) **returns** (t) **signals** (bounds)
effects If *a* is empty, signals *bounds*; otherwise returns $a[low(a)]$.

top = **proc** (a: array[t]) **returns** (t) **signals** (bounds)
effects If *a* is empty, signals *bounds*; otherwise returns $a[high(a)]$.

addh = **proc** (a: array[t], elem: t)
modifies *a*.
effects If extending *a* on the high end causes the high bound or size of *a* to be outside the defined range of integers, signals *failure*. Otherwise extends *a* by 1 in the high direction and stores *elem* as the new element.

addl = **proc** (a: array[t], elem: t) **signals** (overflow)
modifies *a*.
effects If extending *a* on the low end causes the low bound or size of *a* to be outside the defined range of integers, signals *failure*. Otherwise extends *a* by 1 in the low direction and stores *elem* as the new element.

remh = **proc** (a: array[t]) **returns** (t) **signals** (bounds)
modifies *a*.
effects If *a* is empty, signals *bounds*. Otherwise shrinks *a* by removing its high element and returning the removed element.

reml = **proc** (a: array[t]) **returns** (t) **signals** (bounds)
modifies *a*.
effects If *a* is empty, signals *bounds*. Otherwise shrinks *a* by removing its low element and returning the removed element.

elements = **iter** (a: array[t]) **yields** (t)
effects The effect is equivalent to the following body:

```
    for i: int in int$from_to(array[t]$low(a), array[t]$high(a))
  do
        yield (a[i])
        end
```

Note that if *a* is not modified by the loop body, the effect is to yield the elements of *a*, each exactly once, from the low bound to the high bound.

indexes = **iter** (a: array[t]) **yields** (int)
effects Yields the indexes of *a* from the low bound of a_{pre} to the high bound of a_{pre}. Note that *indexes* is unaffected by any modifications done by the loop body.

equal = **proc** (a1, a2: array[t]) **returns** (bool)

effects Returns true if *a1* and *a2* refer to the same array object; otherwise returns false.

similar = **proc** (a1, a2: array[t]) **returns** (bool)
requires *t* has operation
similar: **proctype** (t, t) **returns** (bool)
effects Returns true if *a1* and *a2* have the same low and high bounds and if their elements are pairwise similar as determined by *t$similar*. This operation is equivalent to the following body:

```
at = array[t]
if at$low(a) ~= at$low(a2) cor at$size(a1) ~= at$size(a2)
    then return (false)
    end
for i: int in at$indexes(a1) do
    if ~t$similar(a1[i], a2[i]) then return (false) end
    end
return (true)
```

similar1 = **proc** (a1, a2: array[t]) **returns** (bool)
requires *t* has operation
equal: **proctype** (t, t) **returns** (bool)
effects Returns true if *a1* and *a2* have the same low and high bounds and if their elements are pairwise equal as determined by *t$equal*. This operation works the same way as *similar*, except that *t$equal* is used instead of *t$similar*.

copy = **proc** (a: array[t]) **returns** (b: array[t])
requires *t* has operation
copy: **proctype** (t) **returns** (t)
effects Returns a new array *b* with the same low and high bounds as *a* and such that each element *b[i]* contains *t$copy(a[i])*. This operation is equivalent to the following body:

```
b := array[t]$copy1(a)
for i: int in array[t]$indexes(a) do
    b[i] := t$copy(a[i])
    end

return (b)
```

copy1 = **proc** (a: array[t]) **returns** (b: array[t])
effects Returns a new array *b* with the same low and high bounds as *a* and such that each element *b[i]* contains the same element as *a[i]*.

fill_copy = **proc** (lb, cnt: int, elem: t) **returns** (array[t])
signals (negative_size)

requires *t* has operation
copy: **proctype** (t) **returns** (t)
effects The effect is like *fill* except that *elem* is copied. If $cnt < 0$, signals *negative_size*. Otherwise returns a new array with low bound *lb* and size *cnt* and with each element a distinct copy of *elem*, as produced by *t$copy*; if the new array has high bound $<$ *int_min* or $>$ *int_max*, signals *failure*. This operation is equivalent to the following body:

```
if cnt < 0 then signal negative_size end
x: array[t] := array[t]$predict(lb, cnt)
for j: int in int$from_to(1, cnt) do
    at$addh(x, t$copy(elem))
    end
return (x)
```

end array

A.9.9 Sequences

sequence = **data type** [t: type] **is** new, fill, size, empty, fetch, bottom, top, replace, addh, addl, remh, reml, concat, subseq, e2s, a2s, s2a, elements, indexes, equal, similar, copy, fill_copy

Overview

Sequences are immutable sequences of elements; they always have low bound 1. The elements of the sequence can be indexed sequentially from 1 up to the size of the sequence.

Sequences can be created by calling sequence operations and by means of a special constructor; arguments of the constructor are simply the elements of the new sequence; for example,

sequence[int]$[3, 7]

creates a two-element sequence containing 3 at index 1 and 7 at index 2. The special constructor is discussed further in section A.6.8.

Any operation call that attempts to access a sequence with an index that is not within the defined range terminates with the *bounds* exception. Any operation call that would give rise to a sequence whose size is greater than *int_max* terminates with the *failure* exception.

Operations

new = **proc** ( ) **returns** (sequence[t])
effects Returns the empty sequence.

fill = **proc** (cnt: int, elem: t) **returns** (sequence[t]) **signals** (negative_size)

effects If $cnt < 0$, signals *negative_size*. Otherwise returns a sequence containing *cnt* elements each of which is *elem*.

size = **proc** (s: sequence[t]) **returns** (int)
effects Returns a count of the number of elements in *s*.

empty = **proc** (s: sequence[t]) **returns** (bool)
effects Returns true if *s* contains no elements; otherwise returns false.

fetch = **proc** (s: sequence[t], i: int) **returns** (t) **signals** (bounds)
effects If $i < 1$ or $i > size(s)$, signals *bounds*. Otherwise returns the *i*th element of *s*.

bottom = **proc** (s: sequence[t]) **returns** (t) **signals** (bounds)
effects If *s* is empty, signals *bounds*. Otherwise returns $s[1]$.

top = **proc** (s: sequence[t]) **returns** (t) **signals** (bounds)
effects If *s* is empty, signals *bounds*. Otherwise returns $s[size(s)]$.

replace = **proc** (s: sequence[t], i: int, elem: t) **returns** (sequence[t]) **signals** (bounds)
effects If $i < 1$ or $i > high(s)$, signals *bounds*. Otherwise returns a new sequence containing the same elements as *s*, except that *elem* is in the *i*th position. For example,
replace(sequence[int]$[2,5], 1, 6]) = [6, 5]

addh = **proc** (s: sequence[t], elem: t) **returns** (sequence[t])
effects Returns a new sequence containing the same elements as *s* followed by one additional element, *elem*. If the resulting sequence has size $> int\_max$, signals *failure*.

addl = **proc** (s: sequence[t], elem: t) **returns** (sequence[t])
effects Returns a new sequence containing *elem* as the first element followed by the elements of *s*. If the resulting sequence has $size > int\_max$, signals failure.

remh = **proc** (s: sequence[t]) **returns** (sequence[t]) **signals** (bounds)
effects If *s* is empty, signals *bounds*. Otherwise returns a new sequence containing all elements of *s* except the last one.

reml = **proc** (s: sequence[t]) **returns** (sequence[t]) **signals** (bounds)
effects If *s* is empty, signals *bounds*. Otherwise returns a new sequence containing all elements of *s* except the first one.

concat = **proc** (s1, s2: sequence[t]) **returns** (sequence[t])
effects Returns a new sequence containing the elements of *s1* followed by the elements of *s2*. Signals *failure* if the size of the new sequence is $> int\_max$.

e2s = **proc** (elem: t) **returns** (sequence[t])
effects Returns a one-element sequence containing *elem* as its only element.

a2s = **proc** (a: array[t]) **returns** (sequence[t])

effects Returns a sequence containing the elements of *a* in the same order as in *a*.

s2a = **proc** (s: sequence[t]) **returns** (array[t])
effects Returns a new array with low bound 1 and containing the elements of *s* in the same order as in *s*.

elements = **iter** (s: sequence[t]) **yields** (t)
effects Yields the elements of *s* in order.

indexes = **iter** (s: sequence[t]) **yields** (int)
effects Yields the indexes of *s* from 1 to *size*(*s*).

equal = **proc** (s1, s2: sequence[t]) **returns** (bool)
requires *t* has operation
equal: **proctype** (t, t) **returns** (bool)
effects Returns true if *s1* and *s2* have equal values as determined by *t$equal*. This operation is equivalent to the following body:

```
qt = sequence [t]
if qt$size(s1) ~= qt$size(s2) then return (false) end
for i: int in qt$indexes(s1) do
    if s1[i] ~= s2[i] then return(false) end
end
return (true)
```

similar = **proc** (s1, s2: sequence[t]) **returns** (bool)
requires *t* has operation
similar: **proctype** (t, t) **returns** (bool)
effects Returns true if *s1* and *s2* have similar values as determined by *t$similar*. *Similar* works in the same way as *equal*, except that *t$similar* is used instead of *t$equal*.

copy = **proc** (s: sequence[t]) **returns** (sequence[t])
requires *t* has operation
copy: **proctype** (t) **returns** (t)
effects Returns a new sequence containing as elements copies of the elements of *s*. This operation is equivalent to the following body:

```
qt = sequence[t]
y: qt := qt$new()
for e: t in qt$elements(s) do
    y := qt$addh(y, t$copy(e))
    end
return (y)
```

fill_copy = **proc** (cnt: int, elem: t) **returns** (sequence[t])
signals (negative_size)
requires t has operation
copy: **proctype** (t) **returns** (t)

effects If *cnt* < 0, signals *negative_size*.
Otherwise returns a new sequence containing *cnt* elements each of
which is a copy of *elem*. This operation is equivalent to the following body:

```
qt = sequence[t]
if cnt < 0 then signal negative_size end
x: qt := qt$new()
for i: int in int$from_to(1, cnt) do
    x := qt$addh(x, T$copy(elem))
    end
return (x)
```

end sequence

A.9.10 Records

record = **data type** $[n_1 : t_1, \ldots, n_k : t_k]$ **is** get_ , set_ , r_gets_r,
r_gets_s, equal,similar, similar1, copy, copy1

Overview

A record is a mutable collection of one or more named objects. The names are called *selectors*, and the objects are called *components*. Different components may have different types. A record type specification has the form

record [field_spec , ...]

where

field_spec ::= name, ...: type_spec

Selectors must be unique within a specification, but the ordering and grouping of selectors is unimportant. For example, the following name the same type:

record[last, first, middle: string, age: int]
record[last: string, age: int, first, middle: string]

A record is created using a record constructor, such as

info $ {last: "Jones", first: "John", age: 32, middle: "J."}

(assuming that "info" has been equated to one of the above type specifications; see section A.4.3). An expression must be given for each selector, but the order and grouping of selectors need not resemble the corresponding type specification. Record constructors are discussed in section A.6.8.
In the following definitions of record operations, let

rt = record$[n_1 : t_1, \ldots, n_k : t_k]$

Operations

get_n_i = **proc** (r: rt) **returns** (t_i)

effects Returns the component of r whose selector is n_i. There is a *get_* operation for each selector.

set_n = **proc** (r: rt, e: t_i)
modifies *r*.
effects Modifies r by making the component whose selector is n_i be e. There is a *set_* operation for each selector.

r_gets_r = **proc** (r1, r2: rt)
modifies *r1*.
effects Sets each component of *r1* to be the corresponding component of *r2*.

r_gets_s = **proc** (r: rt, s: st)
modifies *r*.
effects Here *st* is a struct type whose components have the same selectors and types as *rt*. Sets each component of r to be the corresponding component of s.

equal = **proc** (r1, r2: rt) **returns** (bool)
effects Returns true if *r1* and *r2* are the very same record object; otherwise returns false.

similar = **proc** (r1, r2: rt) **returns** (bool)
requires Each t_i has operation
similar: **proctype** (t_i, t_i) **returns** (bool)
effects Returns true if *r1* and *r2* contain similar objects for each component as determined by the $t_i\$similar$ operations. The comparison is done in lexicographic order; if any comparison returns false, false is returned immediately.

similar1 = **proc** (r1, r2: rt) **returns** (bool)
requires Each t_i has operation
equal: **proctype** (t_i, t_i) **returns** (bool)
effects Returns true if *r1* and *r2* contain equal objects for each component as determined by the $t_i\$equal$ operations. The comparison is done in lexicographic order; if any comparison returns false, false is returned immediately.

copy1 = **proc** (r: rt) **returns** (rt)
effects Returns a new record containing the components of r as its components.

copy = **proc** (r: rt) **returns** (rt)
requires Each t_i has operation
copy: **proctype** (t_i) **returns** (t_i)

effects Returns a new record obtained by performing *copy1*(r) and then replacing each component with a copy of the corresponding component of r. Copies are obtained by calling the $t_i\$copy$ operations. Copying is done in lexicographic order.

end record

A.9.11 Structs

struct = **data type** $[n_1 : t_1, \ldots, n_k : t_k]$ **is** get_ , replace_ , s2r, r2s, equal, similar, copy

Overview

A struct is an immutable record. A struct type specification has the form

struct [field_spec, ...]

where (as for records)

field_spec **::=** name, ...: type_spec

A struct is created using a struct constructor, which syntactically is identical to a record constructor. Struct constructors are discussed in section A.6.8.

In the following operation descriptions,

st = struct$[n_1 : t_1, \ldots, n_k : t_k]$

Operations

get_n_i = **proc** (s: st) **returns** (t_i)

effects Returns the component of s whose selector is n_i. There is a *get_* operation for each selector.

replace_n_i = **proc** (s: st, e: t_i) **returns** (st)

effects Returns a new struct object whose components are those of s except that component n_i is e. There is a *replace_* operation for each selector.

s2r = **proc** (s: st) **returns** (rt)

effects Here *rt* is a record type whose components have the same selectors and types as *st*. Returns a new record object whose components are those of s.

r2s = **proc** (r: rt) **returns** (st)

effects Here *rt* is a record type whose components have the same selectors and types as *st*. Returns a struct object whose components are those of r.

equal = **proc** (s1, s2: st) **returns** (bool)

requires Each t_i has operation

equal: **proctype** (t_i, t_i) **returns** (bool)

effects Returns true if *s1* and *s2* contain equal objects for each component as determined by the $t_i\$equal$ operations. The comparison is done in lexicographic order; if any comparison returns false, false is returned immediately.

similar = **proc** (s1, s2: st) **returns** (bool)

requires Each t_i has operation

similar: **proctype** $(\mathrm{t_i}, \mathrm{t_i})$ **returns** (bool)

effects Returns true if *s1* and *s2* contain similar objects for each component as determined by the $t_i\$similar$ operations. The comparison is done in lexicographic order; if any comparison returns false, false is returned immediately.

copy = **proc** (s: st) **returns** (st)

requires Each t_i has operation

copy: **proctype** $(\mathrm{t_i})$ **returns** $(\mathrm{t_i})$

effects Returns a struct containing a copy of each component of *s*; copies are obtained by calling the $t_i\$copy$ operations. Copying is done in lexicographic order.

end struct

A.9.12 Oneofs

oneof = **data type** $[\mathrm{n_1} : \mathrm{t_1}, \ldots, \mathrm{n_k} : \mathrm{t_k}]$ **is** make_ , is_ , value_ , o2v, v2o, equal, similar, copy

Overview

A oneof type is a *tagged, discriminated union.* A oneof is a labeled object, to be thought of as "one of" a set of alternatives. The label is called the *tag*, and the object is called the *value*.
A oneof type specification has the form

oneof [field_spec , . . .]

where (as for records

field_spec **: : =** name, ... : type_spec

Tags must be unique within a specification, but the ordering and grouping of tags is unimportant.
Although there are oneof operations for decomposing oneof objects, they are usually decomposed via the **tagcase** statement, which is discussed in section A.7.9.
In the following descriptions of oneof operations,

ot = oneof$[\mathrm{n_1} : \mathrm{t_1}, \ldots, \mathrm{n_k} : \mathrm{t_k}]$

Operations

make_$\mathrm{n_i}$ = **proc** (e: $\mathrm{t_i}$) **returns** (ot)

effects Returns a oneof object with tag n_i and value e. There is a *make_* operation for each selector.

is_n_i = **proc** (o: ot) **returns** (bool)

effects Returns true if the tag of o is n_i, else returns false. There is an *is_* operation for each selector.

value_n_i = **proc** (o: ot) **returns** (t_i) **signals** (wrong_tag)

effects If the tag of o is n_i, returns the value of o; otherwise signals *wrong_tag*. There is a *value_* operation for each selector.

o2v = **proc** (o: ot) **returns** (vt)

effects Here *vt* is a variant type with the same selectors and types as *ot*. Returns a new variant object with the same tag and value as *o*.

v2o = **proc** (v: vt) **returns** (ot)

effects Here *vt* is a variant type with the same selectors and types as *ot*. Returns a oneof object with the same tag and value as *v*.

equal = **proc** (o1, o2: ot) **returns** (bool)

requires Each t_i has operation

equal: **proctype** (t_i, t_i) **returns** (bool)

effects Returns true if *o1* and *o2* have the same tag and equal values as determined by the *equal* operation of their type.

similar = **proc** (o1, o2: ot) **returns** (bool)

requires Each t_i has operation

similar: **proctype** (t_i, t_i) **returns** (bool)

effects Returns true if *o1* and *o2* have the same tag and similar values as determined by the *similar* operation of their type.

copy = **proc** (o: ot) **returns** (ot)

requires Each t_i must have operation

copy: **proctype** (t_i) **returns** (t_i)

effects Returns a oneof object with the same tag as *o* and containing as a value a copy of *o*'s value; the copy is made using the *copy* operation of the value's type.

end oneof

A.9.13 Variants

variant = **data type** [$n_1 : t_1, \ldots, n_k : t_k$] **is** make_, change_, is_, value_, v_gets_v, v_gets_0, equal, similar, similar1, copy, copy1

Overview

A variant is a mutable oneof. A variant type specification has the form

variant [field_spec , ...]

where (as for records)

field_spec ::= name, ... : type_spec

Although there are variant operations for decomposing variant objects, they are usually decomposed via the **tagcase** statement, which is discussed in section A.7.9.

In the following descriptions of variant operations,

vt = variant[$n_1 : t_1, \ldots, n_k : t_k$]

Operations

make_n_i = **proc** (e: t_i) **returns** (vt)
effects Returns a new variant object with tag n_i and value *e*. There is a *make_* operation for each selector.

change_n_i = **proc** (v: vt, e: t_i)
modifies *v*.
effects Modifies *v* to have tag n_i and value *e*. There is a *change_* operation for each selector.

is_n_i = **proc** (v: vt) **returns** (bool)
effects Returns true if the tag of *v* is n_i; otherwise returns false. There is an *is_* operation for each selector.

value_n_i = **proc** (v: vt) **returns** (t_i) **signals** (wrong_tag)
effects If the tag of *v* is n_i, returns the value of *v*; otherwise signals *wrong_tag*. There is a *value_* operation for each selector.

v_gets_v = **proc** (v1, v2: vt)
modifies *v1*.
effects Modifies *v1* to contain the same tag and value as *v2*.

v_gets_o = **proc** (v: vt, o: ot)
modifies *v*.
effects Here *ot* is the oneof type with the same selectors and types as *vt*. Modifies *v* to contain the same tag and value as *o*.

equal = **proc** (v1, v2: vt) **returns** (bool)
effects Returns true if *v1* and *v2* are the same variant object.

similar = **proc** (v1, v2: vt) **returns** (bool)
requires Each t_i must have operation
similar: **proctype** (t_i, t_i) **returns** (bool)
effects Returns true if *v1* and *v2* have the same tag and similar values as determined by the *similar* operation of their type.

similar1 = **proc** (v1, v2: vt) **returns** (bool)

requires Each t_i must have operation
equal: **proctype** (t_i, t_i) **returns** (bool)
effects Returns true if *v1* and *v2* have the same tag and equal values as determined by the *equal* operation of their type.

copy = **proc** (v: vt) **returns** (vt)
requires Each t_i must have operation
copy: **proctype** (t_i) **returns** (t_i)
effects Returns a variant object with the same tag as *v* and containing as a value a copy of *v*'s value; the copy is made using the *copy* operation of the value's type.

copy1 = **proc** (v: vt) **returns** (vt)
effects Returns a new variant object with the same tag as *v* and containing *v*'s value as its value.

end variant

A.9.14 Procedure and Iterator Types

Procedures and iterators are objects created by the CLU system. The type specification for a procedure or iterator contains most of the information stated in a procedure or iterator heading; a procedure type specification has the form

proctype ([type_spec , ...]) [returns] [signals]

and an iterator type specification has the form

itertype ([type_spec , ...]) [yields] bnflbrack signals]

where

returns	::=	**returns** (type_spec , ...)
yields	::=	**yields** (type_spec , ...)
signals	::=	**signals** (exception , ...)
exception	::=	name [(type_spec , ...)]

The first list of type specifications describes the number, types, and order of arguments. The **returns** or **yields** clause gives the number, types, and order of the objects to be returned or yielded. The **signals** clause lists the exceptions raised by the procedure or iterator; for each exception name, the number, types, and order of the objects to be returned are also given. All names used in a **signals** clause must be unique, and none can be *failure*, which has a standard meaning in CLU. The ordering of exceptions is not important. For example, both of the following type specifications name the procedure type for string$substr:

proctype (string, int, int) **returns** (string)
signals (bounds, negative_size)
proctype (string, int, int) **returns** (string)
signals (negative_size, bounds)

In the following operation descriptions, t stands for a proctype or itertype.

Operations

equal = **proc** (x, y: t) **returns** (bool)
similar = **proc** (x, y: t) **returns** (bool)
    effects These operations return true if and only if x and y are the same module with the same parameters.

copy = **proc** (x: t) **returns** (y: t)
    effects Returns y such that $x = y$.

A.10 Input/Output

This section describes a set of standard "library" data types and procedures for CLU, provided primarily to support I/O. We do not consider this facility to be part of the language proper, but we feel the need for a set of commonly used functions that have some meaning on most systems. This facility is minimal because we wish it to be general, that is, to be implementable, at least in large part, under almost any operating system. The facility also provides a framework in which some other operations that are not always available can be expressed.

Some thought has been given to portability of programs, and possibly even data, but we expect that programs dealing with all but the simplest I/O will have to be written very carefully to be portable, and might not be portable no matter how careful one is.

We shall describe types for naming files, for providing access to text and image files, and for attaching calendar date and time to files. No type "file" exists, as will be explained in section A.10.3.

A.10.1 Files

Our notion of file is a general one that includes not only storage files (disk files) but also terminals and other devices (for example, tape drives). Each file will in general support only a subset of the operations described here.

There are two basic kinds of files, *text files* and *image files*. The two kinds of files may be incompatible. However, on any particular system, it may not be possible to determine what kind a given file is.

A text file consists of a sequence of characters and is divided into lines terminated by newline (\n) characters. A nonempty last line might not be terminated. By convention, the start of a new page is indicated by placing a newpage (\p) character at the beginning of the first line of that page.

A text file will be stored in the (most appropriate) standard text file format of the local operating system. As a result, certain control characters (such as NUL, CR, FF, *uparrow*C, ↑Z) may be ignored when written. In addition, a system may limit the maximum length of lines and may add (remove) trailing spaces to (from) lines.

Image files are provided to allow more efficient storage of information than is provided by text files. Unlike text files, there is no need for image files to be compatible with any local file format; thus image files can be defined more precisely than text files.

An image file consists of a sequence of encoded objects. Objects are written and read using *encode* and *decode* operations of their types. (These in turn will call *encode* and *decode* on their components until built-in types are reached.) The objects stored in an image file are not tagged by the system according to their types. Thus if a file is written by performing a specific sequence of *encode* operations, it must be read back using the corresponding sequence of *decode* operations to be meaningful.

A.10.2 File Names

file_name = **data type is** create, get_dir, get_name, get_suffix,
get_other, parse, unparse, make_output, make_temp, equal, similar,
copy

Overview

File names are immutable objects used to name files. The system file name format is viewed as consisting of four string components:

1. directory—specifies a file directory or device;
2. name—the primary name of the file (for example, "thesis");
3. suffix—a name normally indicating the type of file (for example, "clu" for a CLU source file);
4. other—all other components of the system file name form.

The *directory* and *other* components may have internal syntax. The *name* and *suffix* should be short identifiers. (For example, in the TOPS-20 file name "ps:⟨cluser⟩ref.lpt.3";, the *directory* is "ps:⟨cluser⟩", the *name* is "ref;', the *suffix* is "lpt", and the *other* is "3". In the UNIX path name "/usr/snyder/doc/refman.r", the *directory* is "/usr/snyder/doc", the *name* is 'refman', the *suffix* is "r", and there is no *other*.)
A null component has the following interpretation:

1. directory—denotes the current "working" directory (for example, the "connected directory" under TOPS-20 and the "current directory" under UNIX—see also section A.10.6);
2. name—may be illegal, have a unique interpretation, or be ignored (for example, under TOPS-20, a null name is illegal for most directories, but for some devices the name is ignored);
3. suffix—may be illegal, have a unique interpretation, or be ignored (for example, under TOPS-20, a null suffix is legal, as in "⟨rws⟩foo");
4. other—should imply a reasonable default.

Operations

create = **proc** (dir, name, suffix, other: string) **returns** (file_name)
  signals (bad_format)

effects Creates a new file name from its components; if any component is not in the form required by the underlying system, signals *bad_format*. In the process of creating a file name, the string arguments may be transformed—for example, by truncation or case-conversion.

get_dir = **proc** (fn: file_name) **returns** (string)
get_name = **proc** (fn: file_name) **returns** (string)
get_suffix = **proc** (fn: file_name) **returns** (string)
get_other = **proc** (fn: file_name) **returns** (string)

effects These operations return string forms of the components of a file name. If the file name was created using the *create* operation, the strings returned may be different than those given as arguments to *create*—for example, they may be truncated or case-converted.

parse = **proc** (s: string) **returns** (file_name) **signals** (bad_format)

effects This operation creates a file name given a string in the system standard file name syntax.

unparse = **proc** (fn: file_name) **returns** (string)

effects This operation transforms a file name into the system standard file name syntax. We require that

parse(unparse(fn)) = fn
create(fn.dir, fn.name, fn.suffix, fn.other) = fn

for all file names *fn*. One implication of this rule is that there can be no file name that can be created by *create* but not by *parse*; if a system does have file names that have no string representation in the system standard file name syntax, *create* must reject those file names as having a bad format. Alternatively, the file name syntax can be extended so that it can express all possible file names.

make_output =**proc** (fn: file_name, suffix: string)
returns (file_name) **signals** (bad_format)

effects This operation is used by programs that take input from a file and write new files whose names are based on the input file name. The operation transforms the file name into one that is suitable for an output file. The transformation is done as follows: (1) the suffix is set to *suffix*; (2) if the old directory is not suitable for writing, it is set to null; (3) the name, if null and meaningless, is set to "output". (Examples of directories that may not be suitable for writing are directories that involve transferring files over a slow network.)

make_temp = **proc** (dir, prog, file_id: string) **returns** (file_name)
signals (bad_format)

effects This operation creates a file name appropriate for a temporary file, using the given preferred directory name (*dir*), program name (*prog*), and file identifier (*file_id*). To be useful, both *prog* and *file_id* should be short and alphabetic. The returned file name, when used as an argument to *stream$open* or *istream$open* to open a new file for writing, is guaranteed to create a new file and will not overwrite an existing file. Further file name references to the created file should be made using the name returned by the stream or istream *get_name* operation.

equal = **proc** (fn1, fn2: file_name) **returns** (**bool**)
effects Returns true if and only if the two file_names will *unparse* to equal strings.

similar = **proc** (fn1, fn2: file_name) **returns** (**bool**)
effects The same as the *equal* operation.

copy = **proc** (fn: file_name) **returns** (file_name)
effects Returns a file_name that is equal to *fn*.

end file_name

A.10.3 A File Type

Although files are the basic information-containing objects in this package, we do not introduce a file type. Our reason is that few systems provide an adequate representation for files. On many systems, the most reliable representation of a file (accessible to the user) is a channel (stream) to that file. However, this representation is inappropriate for a CLU file type, since possession of a channel to a file often implies locking that file. Another possible representation is a file name. However, file names are one level removed from files, via the file directory. As a result, the relationship of a file name to a file object is time-varying. Using file names as representations for files would imply that all file operations could signal *non_existent_file*.

For these reasons, operations related to file objects are performed by two clusters, *stream* and *istream*, and operations related to the directory system are performed by procedures.

Note that two opens for read with the same file name might return streams to two different files. We cannot guarantee anything about what happens to a file after a program obtains a stream to it.

A.10.4 Streams

stream = **data type is** open, primary_input, primary_output,error_output, can_read, can_write, reset, flush, close, abort, is_closed, is_terminal,

getc, peekc, empty, getl, gets,
putc, puts, putzero, putleft, putright, putspace,
getc_image, putc_image, puts_image, gets_image,
get_lineno, set_lineno, get_line_length,
get_page_length,get_date, set_date,
get_name, set_output_buffered, get_output_buffered,
equal, similar, copy,
create_input, create_output, get_contents, % string I/O
get_buf, get_prompt, set_prompt, % terminal I/O
get_input_buffered, set_input_buffered, % terminal I/O
add_script, rem_script, unscript % scripting

Overview

Streams provide the means to read and write text files and to perform some other operations on file objects. The operations allowed on any particular stream depend upon the access mode. In addition, certain operations may have no effect in some implementations.

When an operation cannot be performed because of an incorrect access mode, implementation limitations, or properties of an individual file or device, the operation will signal *not_possible* (unless the description of the operation explicitly says that the invocation will be ignored). *End_of_file* is signaled by reading operations when there are no more characters to read.

Streams provide operations to connect streams to strings, to interact with a user at a terminal, and to record input/output in one stream on another. These operations are described in subsequent sections.

Operations

open = **proc** (fn: file_name, mode: string) **returns** (stream)
  signals (not_possible(string))

effects Opens a stream to *fn* in the given *mode*. The possible access modes are "read", "write", and "append". If *mode* is not one of these strings, *not_possible*("bad access mode") is signaled. In those cases where the system is able to detect that the specified preexisting file is not a text file, *not_possible*("wrong file type") is signaled. If *mode* is "read", the named file must exist and a stream is returned upon which input operations can be performed. If *mode* is "write", a new file is created or an old file is rewritten. A stream is returned upon which output operations can be performed. Write mode to storage files should guarantee exclusive access to the file, if possible. If *mode* is "append" and if the named file does not exist, one is created. A stream is returned, positioned at the end of the file, upon which output operations can be performed. Append mode to storage files should guarantee exclusive access to the file, if possible.

primary_input = **proc**( ) **returns** (stream)
effects Returns the "primary" input stream, suitable for reading. This is usually a stream to the user's terminal, but may be set by the operating system.

primary_output = **proc**( ) **returns** (stream)
effects Returns the "primary" output stream, suitable for writing. This is usually a stream to the user's terminal, but may be set by the operating system.

error_output = **proc**( ) **returns** (stream)
effects Returns the "primary" output stream for error messages, suitable for writing. This is usually a stream to the user's terminal, but may be set by the operating system.

can_read = **proc** (s: stream) **returns** (bool)
effects Returns true if input operations appear possible on *s*.

can_write = **proc** (s: stream) **returns** (bool)
effects Returns true if output operations appear possible on *s*.

getc = **proc** (s: stream) **returns** (char)
signals (end_of_file, not_possible(string))
modifies *s*.
effects Removes the next character from *s* and returns it. Signals *end_of_file* if there are no more characters.

peekc = **proc** (s: stream) **returns** (char)
signals (end_of_file, not_possible(string))
effects This input operation is like *getc*, except that the character is not removed from *s*.

empty = **proc** (s: stream) **returns** (bool)
signals (not_possible(string))
effects Returns true if and only if there are no more characters in the stream. It is equivalent to an invocation of *peekc*, where true is returned if *peekc* returns a character and false is returned if *peekc* signals *end_of_file*. Thus in the case of terminals, for example, this operation may wait until additional characters have been typed by the user.

getl = **proc** (s: stream) **returns** (string)
signals (end_of_file, not_possible(string))
modifies *s*.
effects Reads and returns (the remainder of) the current input line and reads but does not return the terminating newline (if any). This operation signals *end_of_file* only if there are no characters and end-of-file is detected.

gets = **proc** (s: stream, term: string) **returns** (string)
signals (end_of_file, not_possible(string))
modifies *s*.
effects Reads characters until a terminating character (one in *term*) or end-of-file is seen. The characters up to the terminator are returned; the terminator (if any) is left in the stream. Signals *end_of_file* only if there are no characters and end-of-file is detected.

putc = **proc** (s: stream, c: (char) **signals** (not_possible(string))
modifies *s*.
effects Appends *c* to *s*. Writing a newline indicates the end of the current line.

putl = **proc** (s: stream, str: string) **signals** (not_possible(string))
modifies *s*.
effects Writes the characters of *str* onto *s*, followed by a newline.

puts = **proc** (s: stream, str: string) **signals** (not_possible(string))
modifies *s*.
effects Writes the characters in *str* using *putc*. Naturally it may be somewhat more efficient than doing a series of individual *putc*s.

putzero = **proc** (s: stream, str: string, cnt: int)
signals (negative_field_width, not_possible(string))
modifies *s*.

effects Outputs *str*. However, if the length of *str* is less than *cnt*, outputs $cnt - length(str)$ zeros before the first digit or period in the string (or at the end, if no such characters).

putleft = **proc** (s: stream, str: string, cnt: int)
signals (negative_field_width, not_possible(string))
modifies *s*.
effects Outputs *str*. However, if the length of *str* is less than *cnt*, outputs $cnt - length(str)$ spaces after the string.

putright = **proc** (s: stream, str: string, cnt: int)
signals (negative_field_width, not_possible(string))
modifies *s*.
effects Outputs *str*. However, if the length of *str* is less than *cnt*, outputs $cnt - length(str)$ spaces before the string.

putspace = **proc** (s: stream, cnt: int)
signals (negative_field_width, not_possible(string))
modifies *s*.
effects Outputs *cnt* spaces on *s*.

putc_image = **proc** (s: stream, c: char) **signals** (not_possible(string))
modifies *s*.
effects Like *putc*, except that an arbitrary character may be written and the character is not interpreted by the CLU I/O system. (For example, the ITS XGP program expects a text file containing certain escape sequences. An escape sequence consists of a special character followed by a fixed number of arbitrary characters. These characters could be the same as an end-of-line mark, but they are recognized as data by their context. On a record-oriented system, such characters would be part of the data. In either case, writing a newline in image mode would not be interpreted by the CLU system as indicating an end-of-line.) Characters written to a terminal stream with this operation can be used to cause terminal-dependent control functions.

getc_image = **proc** (s: stream) **returns** (char)
signals (end_of_file, not_possible(string))
modifies *s*.
effects Provided to read escape sequences in text files, as might be written using *putc_image*. Using this operation inhibits the recognition of end-of-line marks, where used. When reading from a terminal stream, the character is not echoed and is not subject to interpretation as an editing command.

puts_image = **proc** (s: stream, str: string) **signals** (not_possible(string))

modifies *s*.
effects Writes the characters in *str* using *putc_image*. Naturally it may be somewhat more efficient than doing a series of individual *putc_image*s.

gets_image = **proc** (s: stream, str: string) **returns** (string)
signals (end_of_file, not_possible(string))
modifies *s*.
effects Reads characters until a terminating character (one in *str*) or end-of-file is seen. Using this operation inhibits the recognition of end-of-line marks, where used. When reading from a terminal stream, the characters read are not echoed and are not subject to interpretation as editing commands. The characters up to the terminator are returned; the terminator (if any) is left in the stream. This operation signals *end_of_file* only if there are no characters and end-of-file is detected.

close = **proc** (s: stream) **signals** (not_possible(string))
modifies *s*.
effects Attempts to terminate I/O and remove the association between the stream and the file. If successful, further use of operations that signal *not_possible* will signal *not_possible*. This operation will fail if buffered output cannot be written.

abort = **proc** (s: stream)
modifies *s*.
effects Terminates I/O and removes the association between the stream and the file. If buffered output cannot be written, it will be lost, and if a new file is being written, it may or may not exist.

is_closed = **proc** (s: stream) **returns** (bool)
effects Returns true if and only if the stream is closed.

is_terminal = **proc** (s: stream) **returns** (bool)
effects Returns true if and only if the stream is attached to an interactive terminal (see below).

get_lineno = **proc** (s: stream) **returns** (int)
signals (end_of_file, not_possible(string))
effects Returns the line number of the current (being or about to be read) line. If the system maintains explicit line numbers in the file, said line numbers are returned. Otherwise lines are implicitly numbered, starting with 1.

set_lineno = **proc** (s: stream, num: int)
signals (not_possible(string))
modifies *s*.

effects If the system maintains explicit line numbers in the file, sets the line number of the next (not yet started) line to *num*. Otherwise it is ignored.

get_line_length = **proc** (s: stream) **returns** (int)
signals (no_limit)
effects If the file or device to which the stream is attached has a natural maximum line length, that length is returned. Otherwise *no_limit* is signaled. The line length does not include newline characters.

get_page_length = **proc** (s: stream) **returns** (int)
signals (no_limit)
effects If the device to which the stream is attached has a natural maximum page length, that length is returned. Otherwise *no_limit* is signaled. Storage files will generally not have page lengths.

get_date = **proc** (s: stream) **returns** (date)
signals (not_possible(string))
effects Returns the date of the last modification of the corresponding storage file.

set_date = **proc** (s: stream, d: date) **signals** (not_possible(string))
modifies *s*.
effects Sets the modification date of the corresponding storage file. The modification date is set automatically when a file is opened in "write" or "append" mode.

get_name = **proc** (s: stream) **returns** (file_name)
signals (not_possible(string))
effects Returns the name of the corresponding file. It may be different than the name used to open the file, in that defaults have been resolved and link indirections have been followed.

set_output_buffered = **proc** (s: stream, b: bool)
signals (not_possible(string))
modifies *s*.
effects Sets the output buffering mode. Normally output may be arbitrarily buffered before it is actually written out. Unbuffered output can be used on some systems to decrease the amount of information lost if the program terminates prematurely. For terminal streams, unbuffered output is useful in programs that output incomplete lines as they are working to allow the user to watch the progress of the program.

get_output_buffered = **proc** (s: stream) **returns** (bool)

effects Returns true if and only if output to the stream is being buffered.

equal = **proc** (s1, s2: stream) **returns** (bool)

similar = **proc** (s1, s2: stream) **returns** (bool)
effects Returns true if and only both arguments are the same stream.

copy = **proc** (s: stream) **returns** (stream)
effects Returns a stream equal to *s*.

String Input/Output

It is occasionally useful to be able to construct a stream that is not connected to a file, but instead simply collects the output text into a string. Conversely, it is occasionally useful to be able to take a string and convert it into an input stream so that it can be given to a procedure that expects a stream. String streams allow these functions to be performed. A string stream does not have a file name, a creation date, a maximum line or page length, or explicit line numbers. The following stream operations deal with string streams:

create_input = **proc** (s: string) **returns** (stream)
effects An input stream is created that will return the characters in the given string. If the string is nonempty and does not end with a newline, an extra terminating newline will be appended to the string.

create_output = **proc**( ) **returns** (stream)
effects An output stream is created that will collect output text in an internal buffer. The text may be extracted using the *get_contents* operation.

get_contents = **proc** (s: stream) **returns** (string) **signals** (not_possible(string))
effects Returns the text that has so far been output to *s*. Signals *not_possible* if the stream was not created by *create_output*.

Terminal I/O

Terminal I/O is performed via streams attached to interactive terminals. Such a stream is normally obtained via the *primary_input* and *primary_output* operations. A terminal stream is capable of performing both input and output operations. A number of additional operations are possible on terminal streams, and a number of standard operations have special interpretations.

Terminal input will normally be buffered so that the user can perform editing functions, such as deleting the last character on the current line, deleting the current line, redisplaying the current line, and redisplaying the current line after clearing the screen. Specific characters for causing these functions are not suggested. In addition, some means must be provided for the user to indicate end-of-file, so that a terminal stream can be given to a program that expects an arbitrary stream and reads it until end-of-file. The end-of-file status of a stream is cleared by the *reset* operation.

Input buffering is normally provided on a line basis. When a program first asks for input (using *getc*, for example), an entire line of input is read from the terminal and stored in an internal buffer. Further input is not taken from the terminal until the existing buffered input is read.

New input caused to be read by the *getbuf* operation will be buffered as a unit. Thus one can read in a large amount of text and allow editing of that entire text. In addition, when the internal buffer is empty, the *getc_image* operation will read a character directly from the terminal, without interpreting it or echoing it.

The user may specify a prompt string to be printed whenever a new buffer of input is requested from the terminal; the prompt string will be reprinted when redisplay of the current line is requested by the user. However, if new input is requested just when an unfinished line has been output to the terminal, that unfinished line is used instead as a prompt.

The routine *putc_image* can be used to cause control functions, such as "\007" (bell) and "\p" (new-page or clear-screen). We cannot guarantee the effect caused by any particular control character, but we recommend that the standard ASCII interpretation of control characters be supported wherever possible.

Terminal output may be buffered by the system up to one line at a time. However, the buffer must be flushed when new input is requested from the terminal. Terminal streams do not have modification dates, but they should have file names and implicit line numbers. The following additional operations are provided:

getbuf = **proc** (s: stream, str: string) **returns** (string)
  signals (end_of_file, not_possible(string))
 modifies *s*.
 effects This operation is the same as *gets*, but for terminals with input buffering, the new input read by *getbuf* is buffered as a unit, rather than a line at a time, allowing input editing of the entire text.

get_prompt = **proc** (s: stream) **returns** (string)
 effects Returns the current prompt string. The prompt string is initially empty (""). The empty string is returned for nonterminal streams.

set_prompt = **proc** (s: stream. str: string)
 modifies *s*.

effects Sets the string to be used for prompting to *str*. For non-terminal streams there is no effect.

get_input_buffered = **proc** (s: stream) **returns** (bool)
effects Returns true if and only if *s* is attached to a terminal and input is being buffered.

set_input_buffered = **proc** (s: stream, b: bool) **signals** (not_possible(string))
modifies *s*.
effects Sets the input buffering mode. Only buffered terminal input is subject to editing.

Scripting

Streams provide a mechanism for recording the input and/or output from one stream onto any number of other streams. This can be particularly useful in recording terminal sessions. The following additional operations are provided:

add_script = **proc** (s1, s2: stream) **signals** (script_failed)
modifies *s1*.
effects Adds *s2* as a script stream of *s1*. All subsequent input from and output to *s1* will also be output to *s2*. *Not_possible* exceptions that arise in actually outputting to *s2* will be ignored. This operation will fail if *s2* cannot be written to or if either stream is a direct or indirect script stream of the other.

rem_script = **proc** (s1, s2: stream)
modifies *s2*.
effects Removes, but does not close, *s2* as a direct script stream of *s1*.

unscript = **proc** (s: stream)
effects Removes, but does not close, all direct script streams of *s*.

end stream

A.10.5 Istreams

istream = **data type is** open, can_read, can_write, empty, reset, flush, get_date, set_date, get_name, close, abort, is_closed, equal, similar, copy

Overview

Istreams provide the means to read and write image files and to perform some other operations on file objects. The operations allowed on any particular istream depend upon the access mode. In addition, certain operations may be null in some implementations.

When an operation cannot be performed, because of an incorrect access mode, implementation limitations, or properties of an individual file or device, the operation will signal *not_possible* (unless the description of the operation explicitly says that the invocation will be ignored).

Actual reading and writing of objects is performed by *encode* and *decode* operations of the types involved. All of the built-in CLU types and type generators (except the routine type generators), and the file_name and date types, provide these operations. Designers of abstract types are encouraged to provide them also. The type specifications of the *encode* and *decode* operations for a type T are

encode = **proc** (c: T, s: istream)
  signals (not_possible(**string**))
decode = **proc** (s: istream) **returns** (T)
  signals (end_of_file not_possible(**string**))

For parameterized types, *encode* will have a **where** clause requiring *encode* operations for all components, and *decode* will have a **where** clause requiring *decode* operations for all components.

The *encode* operations are output operations. They write an encoding of the given object onto the istream. The *decode* operations are input operations. They decode the information written by *encode* operations and return an object "similar" to the one encoded. If the sequence of *decode* operations used to read a file does not match the sequence of *encode* operations used to write it, meaningless objects may be returned. The system may in some cases be able to detect this condition, in which case the *decode* operation will signal *not_possible*("bad format"). The system is not guaranteed to detect all such errors.

Operations

open = **proc** (fn: file_name, mode: string)
  signals (not_possible(string))

effects The possible access modes are "read", "write", and "append". If *mode* is not one of these strings, not_possible("bad access mode") is signaled. In those cases where the system is able to detect that the specified preexisting file is not an image file, *not_possible*("wrong file type") is signaled. If *mode* is "read", the named file must exist. If the file exists, an image stream is returned upon which *decode* operations can be performed. If *mode* is "write", a new file is created or an old file is rewritten. An image stream is returned upon which *encode* operations can be performed. Write mode to storage files should guarantee exclusive access to the file, if possible. If *mode* is "append" and if the named file does not exist, one is created. An image stream is returned, positioned at the end of the file, upon which *encode* operations can be performed. Append mode to storage files should guarantee exclusive access to the file, if possible.

can_read = **proc** (s: istream) **returns** (bool)
effects Returns true if *decode* operations appear possible on *s*.

can_write = **proc** (s: istream) **returns** (bool)
effects Returns true if *encode* operations appear possible on *s*.

empty = **proc** (istream) **returns** (bool)
effects Returns true if and only if there are no more objects in the associated file.

reset = **proc** (s: istream) **signals** (not_possible(string))
effects Resets *s* so that the next input or output operation will read or write the first item in the file.

flush = **proc** (s: istream) **signals** (not_possible(string))
effects Writes any buffered output to the associated file, if possible.

get_date = **proc** (s: istream) **returns** (date) **signals** (not_possible(string))
effects Returns the date of the last modification of the corresponding storage file.

set_date = **proc** (s: istream, d: date) **signals** (not_possible(string))
modifies *s*.
effects Sets the modification date of the corresponding storage file. The modification date is set automatically when a file is opened in "write" or "append" mode.

get_name = **proc** (s: istream) **returns** (file_name)
effects Returns the name of the corresponding file. It may be different than the name used to open the file, in that defaults have been resolved and link indirections have been followed.

close = **proc** (s: istream) **signals** (not_possible(string))
  modifies *s*.
  effects Attempts to terminate I/O and remove the association between *s* and the file. If successful, further use of operations that signal *not_possible* will signal *not_possible*. This operation will fail if buffered output cannot be written.

abort = **proc** (s: istream)
  modifies *s*.
  effects Terminates I/O and removes the association between the istream and the file. If buffered output cannot be written, it will be lost, and if a new file is being written, it may or may not exist.

is_closed = **proc** (s: istream) **returns** (bool)
  effects Returns true if and only if *s* is closed.

equal = **proc** (s1, s2: istream) **returns** (bool)
  effects Returns true if and only both arguments are the same istream.

similar = **proc** (s1, s2: istream) **returns** (bool)
  effects Returns true if and only both arguments are the same istream.

copy = **proc** (s: istream) **returns** (istream)
  effects Returns a stream that is equal to *s*.

end istream

A.10.6 Miscellaneous Procedures

There are a number of miscellaneous procedures that are useful for input/output:

working_dir = **proc** ( ) **returns** (string)
  effects Returns the name of the current working directory. The working directory is used by the I/O system to fill in a null directory in a file name.

set_working_dir = **proc** (s: string) **signals** (bad_format)
  effects Used to change the working directory. No checking of directory access privileges is performed.

delete_file = **proc** (fn: file_name) **signals** (not_possible(string))
  effects Deletes the specified storage file. An exception may be signaled even if the specified file does not exist, but an exception will not be signaled solely because the file does not exist. For example, an exception may be signaled if the specified directory does not exist or if the user does not have access to the directory.

rename_file = **proc** (fn1, fn2: file_name) **signals** (not_possible(string))
effects Renames the file specified by *fn1* to have the name specified by *fn2*. Renaming across directories and devices may or may not be allowed.

user_name = **proc** ( ) **returns** (string)
effects Returns some identification of the user who is associated with the executing process.

now = **proc** ( ) **returns** (date)
effects Returns the current date and time (see the next section).

e_form = **proc** (r: real, x,y: int) **returns** (string) **signals** (illegal_field_width)
effects Returns a real literal of the form
[–] *i_field* [*.f_field*] e ± x_field)]
where *i_field* is x digits, *f_field* is y digits, and *x_field* is *Exp_width* digits (see section A.9.4). If $y = 0$, the decimal point and *f_field* are not present. If $r \neq 0.0$, the leftmost digit of *i_field* is not zero. If $r = 0.0$, *x_field* is all zeros. *Illegal_field_width* occurs if $x < 0$ or $y < 0$ or $x + y < 1$. If necessary, x may be rounded off to fit the specified form.

f_form = **proc** (r: real, x,y: int) **returns** (string)
signals (illegal_field_width, insufficient_field_width)
effects Returns a real literal of the form
[–] *i_field.f_field*
where *f_field* is y digits. If $x > 0$, *i_field* is at least one digit, with leading zeros suppressed. If $x = 0$, *i_field* is not present. *Illegal_field_width* occurs if $x < 0$ or $y < 0$ or $x + y < 1$. If necessary, r may be rounded off to fit the specified form. *Insufficient_field_width* occurs if real\$$exponent(r) \geq x$ after any rounding.

g_form = **proc** (r: real, x,y: int) **returns** (string)
signals (illegal_field_width, insufficient_field_width)
effects If $r = 0.0$ or $-1 \leq$ real\$$exponent(r) < x$, the result returned is *f_form*(r, x, y). Otherwise the result is *e_form*$(r, 1, x + y - Exp\_width - 3)$. *Illegal_field_width* occurs if $x < 0$ or $y < 0$ or $x + y < 1$. If necessary, r may be rounded off to fit the specified form. *Insufficient_field_width* occurs if $r \neq 0.0$ and $\sim (-1 \leq$ real\$$exponent(x) < y)$ and $(x + y < Exp\_width + 3)$ after any rounding.

A.10.7 Dates

date = **data type is** create, get_all, get_day, get_month, get_year, get_hour, get_minute, get_second

Overview

Dates are immutable objects that represent calendar dates and times.

Operations

create = **proc** (day, month, year, hour, minute, second: int)
    returns (date) **signals** (bad_format)
  effects Creates the specified date. The ranges for the arguments are (1 .. 31), (1 .. 12), (1 ..), (0 .. 23), (0 .. 59), (0 .. 59), respectively; *bad_format* is signaled if an argument is out of range.

get_all = **proc** (d: date) **returns** (int, int, int, int, int, int)
  effects Returns the components in the same order as given to *create*.

get_day = **proc** (d: date) **returns** (int)
get_month = **proc** (d: date) **returns** (int)
get_year = **proc** (d: date) **returns** (int)
get_hour = **proc** (d: date) **returns** (int)
get_minute = **proc** (d: date) **returns** (int)
get_second = **proc** (d: date) **returns** (int)
  effects Returns the specified component of *d*.

unparse = **proc** (d: date) **returns** (string)
  effects Returns a string representation of *d*—for example, "12 January 1978 01:36:59".

unparse_date = **proc** (d: date) **returns** (string)
  effects Returns a string representation of the date part of *d*—for example, "12 January 1978".

unparse_time = **proc** (d: date) **returns** (string)
  effects Returns a string representation of the time part of *d*—for example, "01:36:59".

lt = **proc** (d1, d2: date) **returns** (bool))
le = **proc** (d1, d2: date) **returns** (bool)
ge = **proc** (d1, d2: date) **returns** (bool)
gt = **proc** (d1, d2: date) **returns** (bool)
equal = **proc** (d1, d2: date) **returns** (bool)
  effects The obvious relational operations;
    if $d1 < d2$, then *d1* occurs earlier than *d2*.

similar = **proc** (d1, d2: date) **returns** (bool)
  effects The same as the *equal* operation.

copy = **proc** (d: date) **returns** (date)
  effects Returns a date equal to *d*.

end date