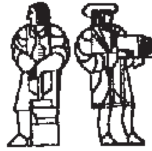
LABORATORY FOR COMPUTER SCIENCE



MASSACHUSETTS INSTITUTE OF TECHNOLOGY

CLU Information Package

Computation Structures Group Memo 154
November 1977

Robert W. Scheifler
Alan Snyder

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of aval Research under contract N00014-75-C-0661, and in part by the National Science Foundation under grant DCR74-21892.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

This memo is intended as a temporary source of information about CLU, in conjunction with [1] and [2], while the new reference manual is in preparation. This document is in four parts. Part A describes the current syntax of CLU, using an extended BNF grammar. Part B defines the objects and operations of the basic types. Although no changes in the syntax are anticipated, new operations related to input/output eventually will be added to the basic types, but their design has not been completed. Part C gives a simple example of a cluster, an implementation of priority queues. Part D presents a more complex program, a text formatter.

Table of Contents

Part A – CLU Syntax 2
Part B – Basic Types and Type Generators 12
Part C – Priority Queue Cluster 27
Part D – Text Formatter 30

References

1. Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C. Abstraction Mechanisms in CLU. *Comm. ACM 20, 8* (Aug 1977), 564–576.

2. Liskov, B., and Snyder, A. Structured Exception Handling. Computation Structures Group Memo, Laboratory for Computer Science, M.I.T., Cambridge, Mass., forthcoming.

Part A - CLU Syntax

1. Lexical Considerations

A CLU module is written as a sequence of tokens and separators. A *token* is a sequence of ASCII "printing" characters (octal 37 < value < octal 177) representing a reserved word, a name, an identifier, a literal, an operator, or a punctuation symbol. A *separator* is a "blank" character (space,[1] VT, HT, CR, NL, FF) or a comment. In general, any number of separators may appear between tokens. Tokens and separators are described in more detail in the sections below.

1.1 Reserved Words

The following character sequences are reserved words:

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

Upper and lower case letters are not distinguished in reserved words. For example, 'end', 'END', and 'eNd' are all the same reserved word.

1.2 Idns and Names

An *idn* (identifier) is a sequence of letters, digits, and underscores that begins with a letter or underscore, and is not a reserved word. As in reserved words, upper and lower case letters are not distinguished in idns. Idns have scope, and are used primarily for variables, parameters, module names, and as abbreviations for constants.

1. Spaces usually serve as separators, but can be used within character and string literals.

A *name* is textually the same as an idn, and upper and lower case letters are not distinguished. Names have no scope, and are used primarily in naming operations of types and selectors of records and oneofs.

1.3 Literals

There is one object of type null; the reserved word **nil** is used as a literal for this object.

The reserved words **true** and **false** are used as literals for the two objects of type **bool**.

An *integer literal* is a sequence of one or more decimal digits.

A *real literal* is 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 null, joined by a period. A mantissa must contain at least one digit. An *exponent* is 'E', 'E+', or 'E-' (or 'e', 'e+', 'e-') followed by one or more decimal digits. An exponent is required if the mantissa does not contain a period. Examples of real literals are:

3.14   3.14E0   314e-2   .0314E+2   3.   .14

A *character literal* for a "printing" ASCII character, other than single quote or backslash, is that character enclosed in single quotes. Literals for other characters are formed 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 (newpage)
\b	BS (backspace)
\r	CR (carriage return)
\v	VT (vertical tab)
***	specified by octal value (* is an octal digit)

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

'7'   'a'   '\''   '\"'   '\B'   '\177'

A *string literal* is 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. Other characters can be represented by using the escape sequences listed above. Examples of string literals are:

```
"Item\tCost"     ""     " "
```

1.4 Operators and Punctuation Symbols

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

```
(     :     <     ~<     +     ||
)     ;     <=    ~<=    -     **
{     ,     =     ~=     *     //
}     .     >=    ~>=    /     &
[     $     >     ~>           |
]     :=                       ~
```

1.5 Comments and Other Separators

A *comment* is a sequence of characters that begins with '%', ends with a newline character (NL), and contains only "printing" ASCII characters and horizontal tabs (HT) in between. For example:

```
z := a[i] + % a comment in an expression
        b[i];
```

A *separator* is a "blank" character (space, VT, HT, CR, NL, FF) 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, idns, names, integer literals, and real literals. This rule is necessary to avoid lexical ambiguities.

2. CLU 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}    stands for  (ε | a | a a | a a a | ...)
[a]    stands for  (ε | a)
%      begins a meta-comment that continues to the end of the line
```

All semicolons are optional in CLU; for simplicity, they appear in the syntax without meta-brackets. Nonterminal symbols appear in normal face. Reserved words appear in bold face. All other terminal symbols are non-alphabetic, and appear in normal face.

```
full_module:  { equate ; } [ module ; ]

module:         procedure
              | iterator
              | cluster

procedure:      idn = proc [ parms ] args [ returns ] [ signals ] [ where ]
                      body
                      end idn

              % The two idns must match.

iterator:       idn = iter [ parms ] args [ yields ] [ signals ] [ where ]
                      body
                      end idn

              % The two idns must match.

cluster:        idn = cluster [ parms ] is idn { , idn } [ where ]
                      c_equates
                      operation ; { operation ; }
                      end idn

              % The first and last idn must match.

parms:          [ parm { , parm } ]

parm:           idn { , idn } : type
              | idn { , idn } : type_spec

args:           ( [ decl { , decl } ] )

decl:           idn { , idn } : type_spec

returns:        returns ( type_spec { , type_spec } )
```

```
yields:         yields ( type_spec { , type_spec } )
signals:        signals ( exception { , exception } )
exception:      name [ ( type_spec { , type_spec } ) ]
where:          where restriction { , restriction }
restriction:    idn has oper_decl { , oper_decl }
              | idn in type_set
              % The idn must be a type parameter.
type_set:       { idn | idn has oper_decl { , oper_decl } ; { equate ; } }
              % The two idns must match.
              | idn
              % The idn must be equated to a type_set.
oper_decl:      op_name { , op_name } : type_spec
op_name:        name [ [ constant { , constant } ] ]
constant:       expression
              % It must be possible to evaluate the expression at compile-time.
              | type_spec
body:           ; { equate ; } { statement ; }
c_equates:      ; { equate ; }  rep = type_spec ;  { equate ; }
operation:      procedure
              | iterator
equate:         idn = constant
              | idn = type_set
```

```
type_spec:     null
             | bool
             | int
             | real
             | char
             | string
             | any

             % any is the union of all types.

             | rep

             % rep may be used only within clusters.

             | cvt

             % cvt may be used only at the top level of the args, returns, yields, and signals
             % clauses of cluster operation headings.

             | array [ type_spec ]
             | record [ field_spec { , field_spec } ]
             | oneof [ field_spec { , field_spec } ]
             | proctype ( [ type_spec { , type_spec } ] ) [ returns ] [ signals ]
             | itertype ( [ type_spec { , type_spec } ] ) [ yields ] [ signals ]
             | idn [ constant { , constant } ]
             | idn

field_spec:    name { , name } : type_spec
```

```
statement:     decl
             | idn : type_spec := expression
             | decl { , decl } := invocation
             | idn { , idn } := invocation
             | idn { , idn } := expression { , expression }
             % The number of idns must equal the number of expressions.
             | primary . name := expression
             | primary [ expression ] := expression
             | invocation
             | while expression do body end
             | for [ decl { , decl } ] in invocation do body end
             | for [ idn { , idn } ] in invocation do body end
             % The invocations in the above two statements must be of an iterator.
             | if expression then body
                  { elseif expression then body }
                  [ else body ]
                  end
             | tagcase expression
                  tag_arm { tag_arm }
                  [ others : body ]
                  end
             % The others arm is present if and only if selectors are missing on the tag_arms.
             | return [ ( expression { , expression } ) ]
             | yield [ ( expression { , expression } ) ]
             | signal name [ ( expression { , expression } ) ]
             | exit name [ ( expression { , expression } ) ]
             | break
             | continue
             % break and continue must occur within a while or for statement.
             | begin body end
```

```
              | statement ; except { when_arm }
                                  [ others [ ( idn : type_spec ) ] : body ]
                                  end

                % The named exceptions arising from statement are caught by the arms.
                % The type_spec must resolve to string.  At least one arm must be present.

tag_arm:          tag name { , name } [ ( idn : type_spec ) ] : body

when_arm:         when name { , name } [ ( decl { , decl } ) ] : body
                | when name { , name } ( * ) : body

expression:       expression ** expression        % 5    (precedence)
                | 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
                | prim

                % The higher the precedence the tighter the binding.
                % All operators are left associative except **, which is right associative.
```

```
prim:           ~ prim
              | - prim
              | ( expression )
              | primary

primary:        nil
              | true
              | false
              | int_literal
              | real_literal
              | char_literal
              | string_literal
              | idn
              | idn [ constant { , constant } ]
              | type_spec $ name [ [ constant { , constant } ] ]
              | primary . name
              | primary [ expression ]
              | invocation
              | type_spec $ { field { , field } }
              % The above is a record constructor.
              | type_spec $ [ [ expression : ] [ expression { , expression } ] ]
              % The above is an array constructor.
              | force [ type_spec ]
              % force[T] is a procedure which takes any object and checks that its type is T.
              | up ( expression )
              | down ( expression )
              % up and down are used within cluster operations to convert between the abstract
              % and representation types.

invocation:     primary ( [ expression { , expression } ] )

field:          name { , name } : expression
```

3. Sugarings

Below is a complete list of operation "sugarings" and their corresponding expansions. These are equivalent in semantics and type-correctness. In the following, x, y, and z are expressions, T_x is the syntactic type of x, and n is a name.

```
Sugar            Expansion

x.n              T_x$get_n(x)
x.n := z         T_x$put_n(x, z)
x[y]             T_x$fetch(x, y)
x[y] := z        T_x$store(x, y, z)
x ** y           T_x$power(x, y)
x // y           T_x$mod(x, y)
x / y            T_x$div(x, y)
x * y            T_x$mul(x, y)
x || y           T_x$concat(x, y)
x + y            T_x$add(x, y)
x - y            T_x$sub(x, y)
x < y            T_x$lt(x, y)
x <= y           T_x$le(x, y)
x = y            T_x$equal(x, y)
x >= y           T_x$ge(x, y)
x > y            T_x$gt(x, y)
x ~< y           ~(x < y)
x ~<= y          ~(x <= y)
x ~= y           ~(x = y)
x ~>= y          ~(x >= y)
x ~> y           ~(x > y)
x & y            T_x$and(x, y)
x | y            T_x$or(x, y)
~x               T_x$not(x)
-x               T_x$minus(x)
```

Part B - Basic Types and Type Generators

1. Introduction

The following sections describe the basic types and the types produced by the basic type generators. For each type, the objects of the type are characterized, and all operations of the type are defined.

In defining an operation, *arg1*, *arg2*, etc., refer to the arguments (the objects, not the syntactic expressions), and *res* refers to the result of the operation. If execution of an operation terminates in an exception, we say the exception "occurs". The order in which exceptions are listed in the operation type is the order in which the various conditions are checked.

The definition of an operation consists of an "interface specification" and an explanation of the relation between arguments and results. An interface specification has the form

name: *type_spec* *side_effects*
restrictions

If *side_effects* is null, no side-effects can occur. "PSE" (primary side-effect) indicates that the state of *arg1* may change. "SSE" (secondary side-effect) indicates that a state change may occur in some object that is, or is contained in, an argument.[1] *Restrictions*, if present, is either a standard **where** clause, or a clause of the form

where each T_i **has** oper_decl$_i$

which is an abbreviation for

where T_1 **has** oper_decl$_1$, ..., T_n **has** oper_decl$_n$

Arithmetic expressions and comparisons used in defining operations are to be computed over the domain of mathematical integers or the domain of mathematical reals; the particular domain will be clear from context.

Definitions of several of the types will involve tuples. A tuple is written <e_1, ..., e_n>; e_i is called the ith element. A tuple with n elements is called an n-tuple. We define the following functions on

1. Secondary side-effects occur when a subsidiary abstraction performs unwanted side-effects. For example, side-effects are not expected when **array[T]$similar** calls **T$similar**, but their absence cannot be guaranteed.

tuples:

$\mathrm{Size}(\langle e_1, ..., e_n \rangle) = n$

$A = B$ if and only if $(\mathrm{Size}(A) = \mathrm{Size}(B)) \wedge (\forall i \mid 1 \le i \le \mathrm{Size}(A))[a_i = b_i]$

$\langle a, ..., b \rangle \,\|\, \langle c, ..., d \rangle = \langle a, ..., b, c, ..., d \rangle$

$\mathrm{Front}(\langle a, ..., b, c \rangle) = \langle a, ..., b \rangle$.

$\mathrm{Tail}(\langle a, b, ..., c \rangle) = \langle b, ..., c \rangle$

$\mathrm{Tail}^0(A) = A$ and $\mathrm{Tail}^{n+1}(A) = \mathrm{Tail}(\mathrm{Tail}^n(A))$

$\mathrm{Occurs}(A, B, i) = (\exists C,D)[(B = C \,\|\, A \,\|\, D) \wedge (\mathrm{Size}(C) = i - 1)]$

If Occurs(A, B, i) holds, we say that A occurs in B at index i.

2. Null

There is one immutable object of type **null**, denoted **nil**.

equal: **proctype (null, null) returns (bool)**
similar: **proctype (null, null) returns (bool)**

Both operations always return **true**.

copy: **proctype (null) returns (null)**

Copy always returns **nil**.

3. Bool

There are two immutable objects of type **bool**, denoted **true** and **false**. These objects represent logical truth values.

and: **proctype (bool, bool) returns (bool)**
or: **proctype (bool, bool) returns (bool)**
not: **proctype (bool) returns (bool)**

These are the standard logical operations.

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

These two operations return **true** if and only if both arguments are the same object.

copy: **proctype (bool) returns (bool)**

Copy simply returns its argument.

4. Int

Objects of type int are immutable, and are intended to model the mathematical integers. However, the only restriction placed on an implementation is that some closed interval [Int_Min, Int_Max] be represented, with Int_Min < 0 and Int_Max > 0. An overflow exception is signalled by an operation if the result of that operation would lie outside this interval.

add: **proctype** (int, int) **returns** (int) **signals** (overflow)
sub: **proctype** (int, int) **returns** (int) **signals** (overflow)
mul: **proctype** (int, int) **returns** (int) **signals** (overflow)

The standard integer addition, subtraction, and multiplication operations.

minus: **proctype** (int) **returns** (int) **signals** (overflow)

Minus returns the negative of its argument.

div: **proctype** (int, int) **returns** (int) **signals** (zero_divide, overflow)

Div computes the integer quotient of *arg1* and *arg2*: $\exists r\ [(0 \le r < |arg2|) \wedge (arg1 = arg2*res + r)]$. Zero_divide occurs if $arg2 = 0$.

mod: **proctype** (int, int) **returns** (int) **signals** (zero_divide, overflow)

Mod computes the integer remainder of dividing *arg1* by *arg2*. That is, $\exists q\ [(0 \le res < |arg2|) \wedge (arg1 = arg2*q + res)]$. Zero_divide occurs if $arg2 = 0$.

power: **proctype** (int, int) **returns** (int) **signals** (negative_exponent, overflow)

This operation computes *arg1* raised to the *arg2* power. Power(0, 0) = 1. Negative_exponent occurs if $arg2 < 0$.

from_to_by: **itertype** (int, int, int) **yields** (int)

This iterator yields, in succession, *arg1*, *arg1* + *arg3*, *arg1* + 2**arg3*, etc., as long as the value to yield, x, satisfies $arg1 \le x \le arg2$ when $arg3 > 0$, or $arg2 \le x \le arg1$ when $arg3 < 0$. The iterator continually yields *arg1* if $arg3 = 0$.

lt: **proctype** (int, int) **returns** (bool)
le: **proctype** (int, int) **returns** (bool)
ge: **proctype** (int, int) **returns** (bool)
gt: **proctype** (int, int) **returns** (bool)

The standard ordering relations.

equal: **proctype** (int, int) **returns** (bool)
similar: **proctype** (int, int) **returns** (bool)

These two operations return **true** if and only if both arguments are the same object.

copy: **proctype** (int) **returns** (int)

Copy simply returns its argument.

5. Real

Objects of type **real** are immutable, and are intended to model the mathematical real numbers. However, only a subset of

$$D = [-Real_Max, -Real_Min] \cup \{0\} \cup [Real_Min, Real_Max]$$

need be represented, where $0 < Real_Min < 1 < Real_Max$. Call this subset Real. We require that both 0 and 1 be elements of Real. If the exact value of a real literal lies in D, then the value in CLU is given by a function Approx, which satisfies the following axioms.

$\forall\ r \in D$	$Approx(r) \in Real$	
$\forall\ r \in Real$	$Approx(r) = r$	
$\forall\ r \in D$	$\lvert(Approx(r) - r)/r\rvert < 10^{-p}$	$p \geq 6$
$\forall\ r,s \in D$	$r \leq s \rightarrow Approx(r) \leq Approx(s)$	
$\forall\ r \in D$	$Approx(-r) = Approx(r)$	

The constant p is the *precision* of the approximation.

add: **proctype** (real, real) **returns** (real) **signals** (overflow, underflow)
sub: **proctype** (real, real) **returns** (real) **signals** (overflow, underflow)
mul: **proctype** (real, real) **returns** (real) **signals** (overflow, underflow)
minus: **proctype** (real) **returns** (real)
div: **proctype** (real, real) **returns** (real) **signals** (zero_divide, overflow, underflow)

These operations satisfy the following axioms:

1) $(a,b \geq 0 \lor a,b \leq 0) \rightarrow add(a, b) = Approx(a + b)$
2) $add(a, b) = (1 + \epsilon)(a + b) \qquad \lvert\epsilon\rvert < 10^{-p}$
3) $add(a, 0) = a$
4) $add(a, b) = add(b, a)$
5) $a \leq a' \rightarrow add(a, b) \leq add(a', b)$
6) $minus(a) = -a$
7) $sub(a, b) = add(a, -b)$
8) $mul(a, b) = Approx(a * b)$
9) $div(a, b) = Approx(a / b)$

In axiom 2, the value of p is the same as that used in defining Approx. Note that the infix and prefix expressions above are computed over the mathematical real numbers.

The axioms only hold if no exceptions occur. An exception occurs if the result of an exact computation lies outside of D; overflow occurs if the magnitude exceeds Real_Max, and underflow occurs if the magnitude is less than Real_Min. Zero_divide occurs if $arg2 = 0$.

power: **proctype** (real, real) **returns** (real) **signals** (zero_divide, complex_result, overflow, underflow)

This operation computes *arg1* raised to the *arg2* power. Zero_divide occurs if $arg1 = 0 \wedge arg2 < 0$. Complex_result occurs if $arg1 < 0$ and *arg2* is non-integral. Overflow and underflow occur as explained above.

i2r: **proctype** (int) **returns** (real) **signals** (overflow)

I2r returns a real number corresponding to the argument: $res = \mathrm{Approx}(arg1)$. Overflow occurs if *arg1* lies outside the domain D.

r2i: **proctype** (real) **returns** (int) **signals** (overflow)

R2i rounds to the nearest integer, and toward zero in case of a tie: $(|res - arg1| \le 1/2) \wedge (|res| < |arg1| + 1/2)$. Overflow occurs if the result lies outside the domain for CLU integers.

trunc: **proctype** (real) **returns** (int) **signals** (overflow)

Trunc truncates its argument toward zero: $(|res - arg1| < 1) \wedge (|res| \le |arg1|)$. Overflow occurs if the result lies outside the domain for CLU integers.

lt: **proctype** (real, real) **returns** (bool)
le: **proctype** (real, real) **returns** (bool)
ge: **proctype** (real, real) **returns** (bool)
gt: **proctype** (real, real) **returns** (bool)

The standard ordering relations.

equal: **proctype** (real, real) **returns** (bool)
similar: **proctype** (real, real) **returns** (bool)

These two operations return **true** if and only if both arguments are the same object.

copy: **proctype** (real) **returns** (real)

Copy simply returns its argument.

6. Char

Objects of type **char** are immutable, and represent characters. Every implementation must provide at least 128, but no more than 512, characters. Characters are numbered from 0 to some Char_Top, and this numbering defines the ordering for the type. The first 128 characters are the ASCII characters in their standard order.

i2c: **proctype** (int) **returns** (char) **signals** (illegal_char)

I2c returns the character corresponding to the argument. Illegal_char occurs if the argument is not in the range [0, Char_Top].

c2i: **proctype** (char) **returns** (int)

This operation returns the number corresponding to the argument.

lt: **proctype** (char, char) **returns** (bool)
le: **proctype** (char, char) **returns** (bool)
ge: **proctype** (char, char) **returns** (bool)
gt: **proctype** (char, char) **returns** (bool)

The ordering relations consistent with the numbering of characters.

equal: **proctype** (char, char) **returns** (bool)
similar: **proctype** (char, char) **returns** (bool)

These two operations return **true** if and only if the two arguments are the same object.

copy: **proctype** (char) **returns** (char)

Copy simply returns its argument.

7. String

Objects of type **string** are immutable. Each string represents a tuple of characters. The ith character of the string is the ith element of the tuple. There are an infinite number of strings, but an implementation need only support a finite number. Attempts to construct illegal strings result in a failure exception.

size: **proctype** (string) **returns** (int)

This operation simply returns the size of the tuple respresented by the argument.

indexs: **proctype (string, string) returns (int)**

If *arg1* occurs in *arg2*, this operation returns the least index at which *arg1* occurs: *res* = min{i | Occurs(*arg1*, *arg2*, i)}. Note that the result is 1 if *arg1* is the 0–tuple. The result is 0 if *arg1* does not occur.

indexc: **proctype (char, string) returns (int)**

If <*arg1*> occurs in *arg2*, the result is the least index at which <*arg1*> occurs: *res* = min{i | Occurs(<*arg1*>, *arg2*, i)}. The result is 0 if <*arg1*> does not occur.

c2s: **proctype (char) returns (string)**

This operation returns the string representing the 1–tuple <*arg1*>.

concat: **proctype (string, string) returns (string)**

Concat returns the string representing the tuple $arg1 \parallel arg2$.

append: **proctype (string, char) returns (string)**

This operation returns the string representing the tuple *arg1* ‖ <*arg2*>.

fetch: **proctype (string, int) returns (char) signals (bounds)**

Fetch returns the $arg2^{th}$ character of *arg1*. Bounds occurs if $(arg2 < 1) \vee (arg2 > size(arg1))$.

rest: **proctype (string, int) returns (string) signals (bounds)**

The result of this operation is $Tail^{arg2-1}(arg1)$. Bounds occurs if $(arg2 < 1) \vee (arg2 > size(arg1) + 1)$.

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

If $arg3 \leq size(rest(arg1, arg2))$, the result is the string representing the tuple of size *arg3* which occurs in *arg1* at index *arg2*. Otherwise, the result is rest(*arg1*, *arg2*). Bounds occurs if $(arg2 < 1) \vee (arg2 > size(arg1) + 1)$. Negative_size occurs if $arg3 < 0$.

s2ac: **proctype (string) returns (array[char])**

This operation places the characters of the argument as elements of a new array of characters. The low bound of the array is 1, and the size of the array is size(*arg1*). The ith element of the array is the ith character of the string.

ac2s: **proctype (array[char]) returns (string)**

Ac2s serves as the inverse of s2ac. The result is the string with characters in the same order as in the argument. That is, the ith character of the result is the $(i + low(arg1) - 1)^{th}$ element of the argument.

chars: **itertype (string) yields (char)**

This iterator yields, in order, each character of the argument.

lt: **proctype (string, string) returns (bool)**
le: **proctype (string, string) returns (bool)**
ge: **proctype (string, string) returns (bool)**
gt: **proctype (string, string) returns (bool)**

These are the usual lexicographic orderings based on the ordering for characters. The lt operation is equivalent to the following:

```
lt = proc (x, y: string) returns (bool);
        size_x: int := string$size(x);
        size_y: int := string$size(y);
        min: int;
        if size_x <= size_y
           then min := size_x;
           else min := size_y;
           end;
        for i: int in int$from_to_by(1, min, 1) do
                if x[i] < y[i]
                   then return(true);
                   end;
                end;
        return(size_x < size_y);
        end lt;
```

equal: **proctype (string, string) returns (bool)**
similar: **proctype (string, string) returns (bool)**

These two operations return **true** if and only if both arguments are the same object.

copy: **proctype (string) returns (string)**

Copy simply returns its argument.

8. Array Types

The **array** type generator defines an infinite class of types. For every type T there is a type **array**[T]. Array objects are mutable. The state[1] of an object of type **array**[T] consists of:

a) an integer Low, called the low bound, and
b) a tuple Elts of objects of type T, called the elements.

---

1. For an array A, we should properly write Low_A, etc., to refer to the state of that particular object, but subscripts will be dropped when the association seems clear.

We also define Size = Size(Elts), and High = Low + Size - 1. We want to think of the elements of Elts as being numbered from Low, so we define the array_index of the ith element to be (i + Low - 1).

For any array, Low, High, and Size must be legal integers. Any attempts to create or modify an array in violation of this rule results in a failure exception. Note that for all array operations, if an exception other than failure occurs, the states of all array arguments are unchanged from those at the time of invocation.

create: **proctype** (int) **returns** (array[T])

This operation returns a new array for which Low is *arg1* and Elts is the 0-tuple.

new: **proctype** () **returns** (array[T])

This is equivalent to create(1).

predict: **proctype** (int, int) **returns** (array[T])

Predict is essentially the same as create(*arg1*), in that it returns a new array for which Low is *arg1* and Elts is the 0-tuple. However, if *arg2* is greater than (less than) 0, it is assumed that at least |*arg2*| addh's (addl's) will be performed on the array. These subsequent operations may execute somewhat faster.

low: **proctype** (array[T]) **returns** (int)
high: **proctype** (array[T]) **returns** (int)
size: **proctype** (array[T]) **returns** (int)

These operations return Low, High, and Size, respectively.

set_low: **proctype** (array[T], int) PSE

Set_low makes Low equal to *arg2*.

trim: **proctype** (array[T], int, int) **signals** (bounds, negative_size) PSE

This operation makes Low equal to *arg2*, and makes Elts equal to the tuple of size min(*arg3*, High' - *arg2* + 1) which occurs in Elts' at index *arg2* - Low' + 1.[1] That is, every element with array_index less than *arg2*, or greater than or equal to *arg2* + *arg3*, is removed. Bounds occurs if (*arg2* < Low') ∨ (*arg2* > High' + 1). Negative_size occurs if *arg3* < 0. Note that this operation is somewhat like **string**$substr.

1. Elts', High', etc. refer to the state just prior to invoking the operation.

fill: **proctype** (int, int, T) **returns** (array[T]) **signals** (negative_size)

Fill creates a new array for which **Low** is *arg1* and **Elts** is an *arg2*-tuple in which every element is *arg3*. Negative_size occurs if *arg2* < 0.

fill_copy: **proctype** (int, int, T) **returns** (array[T]) **signals** (negative_size)
where T **has** copy: **proctype** (T) **returns** (T)

This operation is equivalent to the following:

```
fill_copy = proc (nlow, nsize: int, elt: T) returns (at) signals (negative_size);
        at = array[T];
        if nsize < 0
           then signal negative_size;
           end
        x: at := at$predict(nlow, nsize);
        for i: int in int$from_to_by(1, nsize, 1) do
                at$addh(x, T$copy(elt));
                end;
        return(x);
        end fill_copy;
```

fetch: **proctype** (array[T], int) **returns** (T) **signals** (bounds)

Fetch returns the element of *arg1* with array_index *arg2*. Bounds occurs if (*arg2* < **Low**) ∨ (*arg2* > **High**).

bottom: **proctype** (array[T]) **returns** (T) **signals** (bounds)
top: **proctype** (array[T]) **returns** (T) **signals** (bounds)

These operations return the elements with array_indexes **Low** and **High**, respectively. Bounds occurs if **Size** = 0.

store: **proctype** (array[T], int, T) **signals** (bounds) PSE

Store makes **Elts** a new tuple which differs from the old in that *arg3* is the element with array_index *arg2*. Bounds occurs if (*arg2* < **Low**) ∨ (*arg2* > **High**).

addh: **proctype** (array[T], T) PSE

This operation makes **Elts** the new tuple **Elts'** ‖ <*arg2*>.

addl: **proctype** (array[T], T) PSE

This operation makes **Low** equal to **Low'** – 1, and makes **Elts** the tuple <*arg2*> ‖ **Elts'**. Decrementing **Low** keeps the array_indexes of the previous elements the same.

remh: **proctype** (array[T]) **returns** (T) **signals** (bounds) PSE

Remh makes **Elts** the tuple **Front(Elts')**, and returns the deleted element. Bounds occurs if **Size'** = 0.

reml: **proctype (array[T]) returns (T) signals (bounds)** PSE

Reml makes Low equal to Low' + 1, makes Elts the tuple Tail(Elts'), and returns the deleted element. Incrementing Low keeps the array_indexes of the remaining elements the same. Bounds occurs if Size' = 0.

elements: **itertype (array[T]) yields (T) signals (bounds)**

Elements with array_indexes in the range [Low', High'] are yielded in order. If the state of *arg1* is changed after the iterator has yielded an element, it is possible that when the iterator is resumed there is no element for the next array_index. Bounds occurs in such a case.

indexes: **itertype (array[T]) yields (int)**

This iterator is equivalent to int$from_to_by(Low', High', 1).

equal: **proctype (array[T], array[T]) returns (bool)**

Equal returns **true** if and only if both arguments are the same object.

similar: **proctype (array[T], array[T]) returns (bool)** SSE
where T has similar: **proctype (T, T) returns (bool)**

This operation is equivalent to the following:

```
similar = proc (x, y: at) returns (bool)
            where T has similar: proctype (T, T) returns (bool);
    at = array[T];
    if at$low(x) ~= at$low(y) cor at$size(x) ~= at$size(y)
      then return(false);
      end;
    for i: int in at$indexes(x) do
            if ~T$similar(x[i], y[i])
              then return(false);
              end;
            end;
    return(true);
    end similar;
```

similar1: **proctype (array[T], array[T]) returns (bool)** SSE
where T has equal: **proctype (T, T) returns (bool)**

Similar1 works in the same way as similar, except that T$equal is used instead of T$similar.

copy1: **proctype (array[T]) returns (array[T])**

Copy1 creates a new array with the same state as the argument.

copy: **proctype** (array[T]) **returns** (array[T]) SSE
where T **has** copy: **proctype** (T) **returns** (T)

This operation is equivalent to the following:

```
copy = proc (x: at) returns (at)
            where T has copy: proctype (T) returns (T);
    at = array[T];
    x := at$copy1(x);
    for i: int in at$indexes(x) do
            x[i] := T$copy(x[i]);
            end;
    return(x);
    end copy;
```

9. Record Types

The **record** type generator defines an infinite class of types. For every tuple of name/type pairs <(N_1, T_1), ..., (N_n, T_n)>, where all the names are distinct, in lower case, and in lexicographic order, there is a type **record**[N_1:T_1, ..., N_n:T_n]. (However the user may write this type with the pairs permuted, and may use upper case letters in names.) Records are mutable objects. The state of a record of type **record**[N_1:T_1, ..., N_n:T_n] is an n-tuple; the ith element of the tuple is of type T_i. The ith element is also called the N_i-component.

create: **proctype** (T_1, ..., T_n) **returns** (**record**[N_1:T_1, ..., N_n:T_n])

This operation returns a new record with the tuple <*arg1*, ..., *argN*> as its state. This operation is not available to the user; its use is implicit in the record constructor.

get_N_i : **proctype** (**record**[N_1:T_1, ..., N_n:T_n]) **returns** (T_i)

This operation returns the N_i-component of the argument. There is a get_N_i operation for each N_i.

put_N_i : **proctype** (**record**[N_1:T_1, ..., N_n:T_n], T_i) PSE

This operation makes the state of *arg1* a new tuple which differs from the old in that the N_i-component is *arg2*. There is a put_N_i operation for each N_i.

equal: **proctype** (**record**[N_1:T_1, ..., N_n:T_n], **record**[N_1:T_1, ..., N_n:T_n]) **returns** (**bool**)

Equal returns **true** if and only if both arguments are the same object.

similar: **proctype** (**record**[N_1:T_1, ..., N_n:T_n], **record**[N_1:T_1, ..., N_n:T_n]) **returns** (bool) SSE
where each T_i **has** similar: **proctype** (T_i, T_i) **returns** (bool)

Corresponding components of *arg1* and *arg2* are compared in (lexicographic) order, using T_i\$similar for the N_i-components. (The N_i-component of *arg1* becomes the first argument.) If a comparison results in **false**, the result of the operation is **false**, and no further comparisons are made. If all comparisons return **true**, the result is **true**.

similar1: **proctype** (**record**[N_1:T_1, ..., N_n:T_n], **record**[N_1:T_1, ..., N_n:T_n]) **returns** (bool) SSE
where each T_i **has** equal: **proctype** (T_i, T_i) **returns** (bool)

Similar1 works in the same way as similar, except that T_i\$equal is used instead of T_i\$similar.

copy1: **proctype** (**record**[N_1:T_1, ..., N_n:T_n]) **returns** (**record**[N_1:T_1, ..., N_n:T_n])

Copy1 returns a new record with the same state as the argument.

copy: **proctype** (**record**[N_1:T_1, ..., N_n:T_n]) **returns** (**record**[N_1:T_1, ..., N_n:T_n]) SSE
where each T_i **has** copy: **proctype** (T_i) **returns** (T_i)

This operation is equivalent to the following (note that the N_i are in lexicographic order):

```
copy = proc (x: rt) returns (rt)
            where T1 has copy: proctype (T1) returns (T1),
                  ...
                  Tn has copy: proctype (Tn) returns (Tn);
    rt = record[N1:T1, ..., Nn:Tn];
    x := rt$copy1(x);
    x.N1 := T1$copy(x.N1);
        ...
    x.Nn := Tn$copy(x.Nn);
    return(x);
    end copy;
```

10. Oneof Types

The **oneof** type generator defines an infinite class of types. For every tuple of name/type pairs <(N_1, T_1), ..., (N_n, T_n)>, where all of the names are distinct, in lower case, and in lexicographic order, there is a type **oneof**[N_1:T_1, ..., N_n:T_n]. (However the user may write this type with the pairs permuted, and may use upper case letters in names.) Oneof objects are immutable. Each object represents a name/object pair (N_i, X), where X is of type T_i. For each object X of type T_i there is an object for the pair (N_i, X). N_i is called the tag of the oneof, and X is called the value.

make_N_i : **proctype** (T_i) **returns** (**oneof**[N_1:T_1, ..., N_n:T_n])

This operation returns the oneof object for the pair (N_i, *arg1*). There is a make_N_i operation for each N_i.

is_N_i : **proctype** (**oneof**[N_1:T_1, ..., N_n:T_n]) **returns** (**bool**)

This operation returns **true** if and only if the tag of the argument is N_i. There is an is_N_i operation for each N_i.

value_N_i : **proctype** (**oneof**[N_1:T_1, ..., N_n:T_n]) **returns** (T_i) **signals** (wrong_tag)

If the argument has tag N_i, the result is the value part of the argument. Wrong_tag occurs if the tag is other than N_i. There is a value_N_i operation for each N_i.

equal: **proctype** (**oneof**[N_1:T_1, ..., N_n:T_n], **oneof**[N_1:T_1, ..., N_n:T_n]) **returns** (**bool**) SSE
where each T_i **has** equal: **proctype** (T_i, T_i) **returns** (**bool**)

If *arg1* and *arg2* have different tags, the result is **false**. If both tags are N_i, the result is that of invoking T_i\$equal with the two value parts.

similar: **proctype** (**oneof**[N_1:T_1, ..., N_n:T_n], **oneof**[N_1:T_1, ..., N_n:T_n]) **returns** (**bool**) SSE
where each T_i **has** similar: **proctype** (T_i, T_i) **returns** (**bool**)

If *arg1* and *arg2* have different tags, the result is **false**. If both tags are N_i, the result is that of invoking T_i\$similar with the two value parts.

copy: **proctype** (**oneof**[N_1:T_1, ..., N_n:T_n]) **returns** (**oneof**[N_1:T_1, ..., N_n:T_n]) SSE
where each T_i **has** copy: **proctype** (T_i) **returns** (T_i)

If *arg1* represents the pair (N_i, X), then the result is the oneof object for the pair (N_i, T_i\$copy(X)).

11. Procedure and Iterator Types

Let A, R, L_1, ..., L_n be ordered lists of types, and let N_1, ..., N_n be distinct names in lower case and in lexicographic order. Then there is a type

proctype (A) **returns** (R) **signals** (N_1(L_1), ..., N_n(L_n))

and a type

itertype (A) **yields** (R) **signals** (N_1(L_1), ..., N_n(L_n)).

(The user may permute the N_i(L_i)'s, and may use upper case letters in names. If R is empty then "**returns** (R)" is not written, "(L_i)" is not written if L_i is empty, and "**signals** (...)" is not written if n = 0.)

The create operations are not available to the user; their use is implicit in the procedure and iterator constructors.

Let T be a procedure (or iterator) type in the following.

equal: **proctype (T, T) returns (bool)**
similar: **proctype (T, T) returns (bool)**

These operations return **true** if and only if both arguments are the same implementation of the same abstraction, with the same parameters.

copy: **proctype (T) returns (T)**

Copy simply returns its argument.

12. Any

The type **any** is the union of all types. There are no operations for the type **any**. Thus, for example, no **array[any]**$copy operation exists.

```
remove = proc (x: cvt) returns (t) signals (empty);
        a: at := x.a;
        p: pt := x.p;
        r: t := at$bottom(a);                              % Save best for later return
            except when bounds: signal empty; end;
        v: t := at$remh(a);                                % Remove last element
        max_son: int := at$size(a);                        % Get new size
        if max_son = 0 then return (r); end;               % If now empty, we're done
        max_dad: int := max_son/2;                         % Last node with a son
        dad: int := 1;                                     % Node to place v if it beats sons
        while dad <= max_dad do                            % While node has a son
            son: int := dad*2;                             % Get the first son
            s: t := a[son];
            if son < max_son                               % If there is a second son
               then                                        % Find the best son
                    ns: t := a[son + 1];
                    if p(ns, s) then son, s := son + 1, ns; end;
               end;
            if p(v, s) then break; end;                    % If v beats son, we're done
            a[dad] := s;                                   % Move son up
            dad := son;                                    % Move v down
            end;
        a[dad] := v;                                       % Insert the element into place
        return (r);                                        % Return the previous best element
        end remove;

end p_queue;
```

Part D - Text Formatter

The following program is a simple text formatter. The input consists of a sequence of unformatted text lines mixed with commands lines. Each line is terminated by a newline character, and command lines begin with a period to distinguish them from text lines. For example:

```
Justification only occurs in "fill" mode.
In "nofill" mode, each input text line is output without modification.
The .br command causes a line-break.
.br
Just like this.
```

The program produces justified, indented, and paginated text. For example:

```
Justification  only occurs in "fill" mode. In "nofill" mode,
each input text line is output without modification. The .br
command causes a line-break.
Just like this.
```

The output text is indented 10 spaces from the left margin, and is divided into pages of 50 text lines each. A header, giving the page number, is output at the beginning of each page.

An input text line consists of a sequence of words and word-break characters. The word-break characters are space, tab, and newline; all other characters are constituents of words. Tab stops are considered to be every eight spaces.

The formatter has two basic modes of operation. In "nofill" mode, each input text line is output without modification. In "fill" mode, input is accepted until no more words can fit on the current output line. (An output line has 60 characters.) Newline characters are treated essentially as spaces. Extra spaces are then added between words until the last word has its last character in the rightmost position of the line.

In fill mode, any input line that starts with a word-break character causes a line-break: the current output line is neither filled nor adjusted, but is output as is. An "empty" input line (one starting with a newline character) causes a line-break and then causes a blank line to be output.

The formatter accepts three different commands:

.br causes a line-break

Part C - Priority Queue Cluster

This cluster is an implementation of priority queues. It inserts elements in $O(\log_2 n)$ time, and removes the 'best' element in $O(\log_2 n)$ time, where n is the number of items in the queue, and 'best' is determined by a total ordering predicate which the queue is created with.

The queue is implemented with a binary tree which is balanced such that every element is 'better' than its descendents, and the minimum depth of the tree differs from the maximum depth by one. The tree is implemented by keeping the elements in an array, with the left son of a[i] in a[i*2], and the right son in a[i*2+1]. The root of the tree, a[1], is the 'best' element.

Each insertion or deletion must rebalance the tree. Since the tree is of depth strictly less than $\log_2 n$, the number of comparisons is less than $\log_2 n$ for insertion and less than $2 \log_2 n$ for removal of an element. Consequently, a sort using this technique takes less than $3 n \log_2 n$ comparisons.

This cluster illustrates the use of a type parameter, and the use of a procedure as an object.

```
p_queue = cluster [t: type] is
          create,          % Create a p_queue with a particular sorting predicate
          top,             % Return the best element
          size,            % Return the number of elements
          empty,           % Return true if there are no elements
          insert,          % Insert an element of type t
          remove;          % Remove the best element and return it

          pt = proctype (t, t) returns (bool);
          at = array[t];
          rep = record [a: at, p: pt];

create =  proc (p: pt) returns (cvt);
          return (rep${a: at$create(1), p: p});          % Low index of array must be 1 !
          end create;

top =     proc (x: cvt) returns (t) signals (empty);
          return (at$bottom(x.a));
                except when bounds: signal empty; end;
          end top;

size =    proc (x: cvt) returns (int);
          return (at$size(x.a));
          end size;

empty =   proc (x: cvt) returns (bool);
          return (at$size(x.a) = 0);
          end empty;

insert =  proc (x: cvt, v: t);
          a: at := x.a;
          p: pt := x.p;
          at$addh(a, v);                                 % Make room for new item
          son: int := at$high(a);                        % Node to place v if father wins
          dad: int := son/2;                             % Get index of father
          while dad > 0 cand p(v, a[dad]) do             % While father loses
                a[son] := a[dad];                        % Move father down
                son, dad := dad, dad/2;                  % Get new son, father
                end;
          a[son] := v;                                   % Insert the element into place
          end insert;
```

```
remove = proc (x: cvt) returns (t) signals (empty);
        a: at := x.a;
        p: pt := x.p;
        r: t := at$bottom(a);                           % Save best for later return
            except when bounds: signal empty; end;
        v: t := at$remh(a);                             % Remove last element
        max_son: int := at$size(a);                     % Get new size
        if max_son = 0 then return (r); end;            % If now empty, we're done
        max_dad: int := max_son/2;                      % Last node with a son
        dad: int := 1;                                  % Node to place v if it beats sons
        while dad <= max_dad do                         % While node has a son
            son: int := dad*2;                          % Get the first son
            s: t := a[son];
            if son < max_son                            % If there is a second son
               then                                     % Find the best son
                    ns: t := a[son + 1];
                    if p(ns, s) then son, s := son + 1, ns; end;
               end;
            if p(v, s) then break; end;                 % If v beats son, we're done
            a[dad] := s;                                % Move son up
            dad := son;                                 % Move v down
            end;
        a[dad] := v;                                    % Insert the element into place
        return (r);                                     % Return the previous best element
        end remove;

end p_queue;
```

Part D - Text Formatter

The following program is a simple text formatter. The input consists of a sequence of unformatted text lines mixed with commands lines. Each line is terminated by a newline character, and command lines begin with a period to distinguish them from text lines. For example:

```
Justification only occurs in "fill" mode.
In "nofill" mode, each input text line is output without modification.
The .br command causes a line-break.
.br
Just like this.
```

The program produces justified, indented, and paginated text. For example:

```
Justification  only occurs in "fill" mode. In "nofill" mode,
each input text line is output without modification. The .br
command causes a line-break.
Just like this.
```

The output text is indented 10 spaces from the left margin, and is divided into pages of 50 text lines each. A header, giving the page number, is output at the beginning of each page.

An input text line consists of a sequence of words and word-break characters. The word-break characters are space, tab, and newline; all other characters are constituents of words. Tab stops are considered to be every eight spaces.

The formatter has two basic modes of operation. In "nofill" mode, each input text line is output without modification. In "fill" mode, input is accepted until no more words can fit on the current output line. (An output line has 60 characters.) Newline characters are treated essentially as spaces. Extra spaces are then added between words until the last word has its last character in the rightmost position of the line.

In fill mode, any input line that starts with a word-break character causes a line-break: the current output line is neither filled nor adjusted, but is output as is. An "empty" input line (one starting with a newline character) causes a line-break and then causes a blank line to be output.

The formatter accepts three different commands:

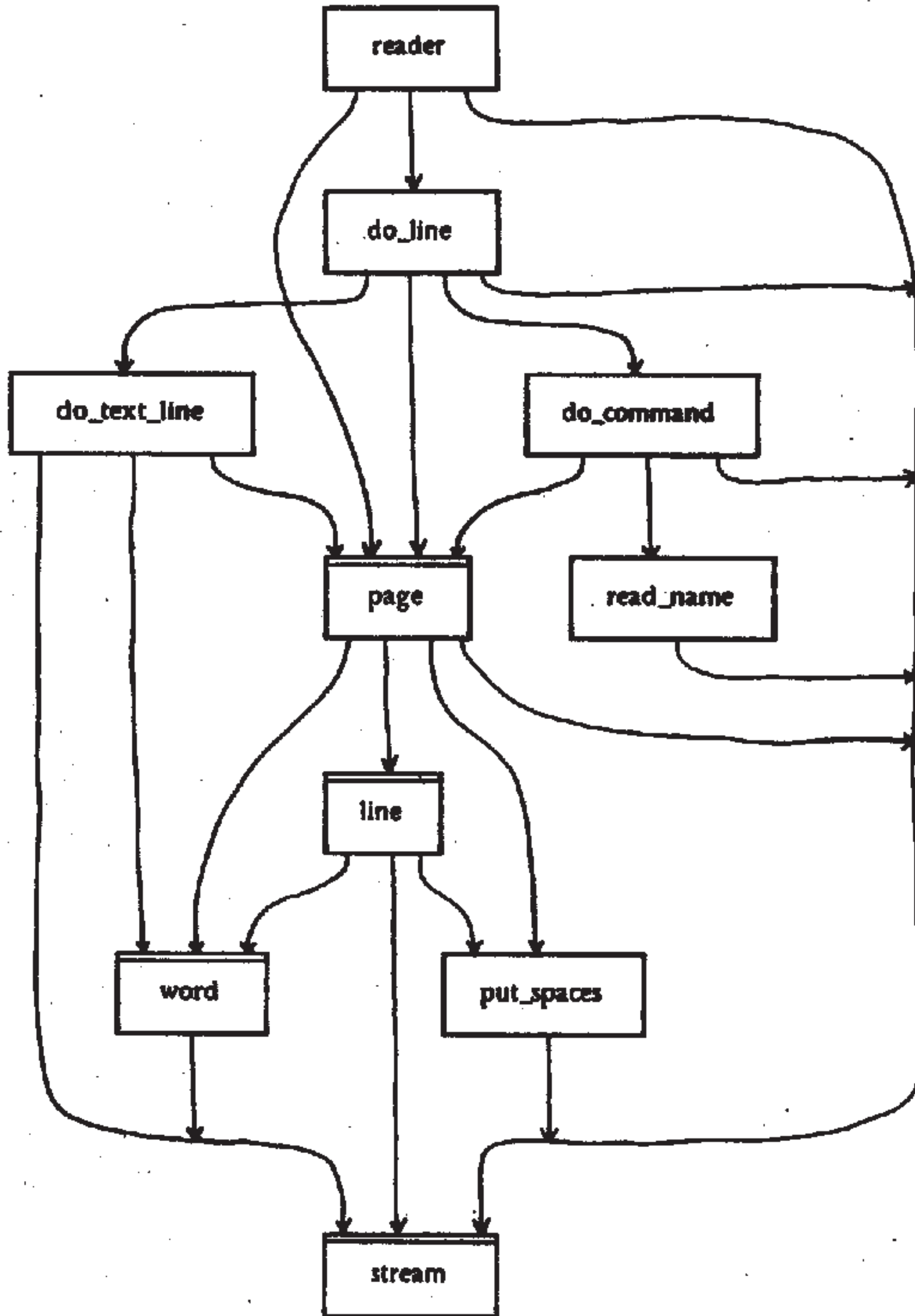
.br causes a line-break

.nf	causes a line-break, and changes the mode to "nofill"
.fi	causes a line-break, and changes the mode to "fill"

The program performs input and output on streams, which are connections (channels) to text files. The following operations on streams are used:

empty	tests if the end of the file has been reached
getc	removes and returns the next character from the stream
peekc	like *getc*, but the character is not removed
getl	removes and returns (the remainder of) the input line and removes but does not return the terminating newline character
putc	outputs a character, with newline indicating end of line
puts	outputs the characters of a string using *putc*
close	closes the stream and associated output file, if any

Module Dependency Diagram



Note: boxes with a double line at the top indicate clusters.

```
reader = proc (instream, outstream, errstream: stream)

         % Read the instream, processing it and placing the output on
         % outstream and writing error messages on errstream.

         p: page := page$create (outstream)
         while ~stream$empty (instream) do
                  do_line (instream, p, errstream)
                  end
         page$terminate (p)
         end reader

do_line = proc (instream: stream, p: page, errstream: stream).

         % Process an input line.  This procedure reads one line from
         % instream.  It is then processed either as a text line or as
         % a command line, depending upon whether or not the first
         % character of the line is a period.

         c: char := stream$peekc (instream)
         if c = '.' then
                  do_command (instream, p, errstream)
             else
                  do_text_line (instream, p)
             end
         end do_line
```

```
do_text_line = proc (instream: stream, p: page)

        % Process a text line.  This procedure reads one line from
        % instream and processes it as a text line.  If the first
        % character is a word-break character, then a line-break is
        % caused.  If the line is empty, then a blank line is output.
        % Otherwise, the words and word-break characters in the line
        % are processed in turn.

        c: char := stream$getc (instream)
        if c = '\n' then                          % empty input line
                page$skip_line (p)
                return
                end
        if c = ' ' cor c = '\t' then
                page$break_line (p)
                end
        while c ~= '\n' do
                if c = ' ' then
                        page$add_space (p)
                   elseif c = '\t' then
                        page$add_tab (p)
                   else
                        w: word := word$scan (c, instream)
                        page$add_word (p, w)
                        end
                c := stream$getc (instream)
                end
        page$add_newline (p)
        end do_text_line
```

```
do_command = proc (instream: stream, p: page, errstream: stream)

        % Process a command line.  This procedure reads one line from
        % instream and processes it as a command.

        stream$getc (instream)                  % skip the period
        n: string := read_name (instream)
        if n = "br" then
                page$break_line (p)
        elseif n = "fi" then
                page$break_line (p)
                page$set_fill (p)
        elseif n = "nf" then
                page$break_line (p)
                page$set_nofill (p)
        else
                stream$puts ("'", errstream)
                stream$puts (n, errstream)
                stream$puts ("' not a command.\n", errstream)
                end
        stream$getl (instream)          % read remainder of input line
        end do_command

read_name = proc (instream: stream) returns (string)

        % This procedure reads a command name from instream.  The
        % command name is terminated by a space or a newline.  The
        % command name is removed from instream; the terminating space
        % or newline is not.

        s: string := ""
        while true do
                c: char := stream$peekc (instream)
                    except when end_of_file: return (s) end
                if c = ' ' cor c = '\n' then
                        return (s)
                        end
                s := string$append (s, stream$getc (instream))
                end
        end read_name
```

```
page = cluster is create, add_word, add_space, add_tab, add_newline,
            break_line, skip_line, set_fill, set_nofill, terminate

            % The page cluster does the basic formatting.  It supports the
            % basic actions: BREAK_LINE, SKIP_LINE, SET_FILL, SET_NOFILL,
            % TERMINATE.  It performs the appropriate actions for the
            % basic components of the input: WORDs, SPACEs, TABs, and
            % NEWLINEs.  It maintains a current output line for the
            % purposes of performing justification.  It performs
            % pagination and the production of headings.  For this purpose
            % it maintains the current line number and the current page
            % number.

            rep = record [
                        line: line,                 % The current line.
                        fill: bool,                 % True <==> in fill mode.
                        lineno: int,                % The number of lines output
                                                    % so far on this page (not
                                                    % including any header lines).
                        pageno: int,                % The number of the current
                                                    % output page.
                        outstream: stream           % The output stream.
                        ]

create = proc (outstream: stream) returns (cvt)

            % Create a page object.  The first page is number 1, there are
            % no lines yet output on it.  Fill mode is in effect.

            return ( rep${       line: line$create (),
                                 fill: true,
                                 lineno: 0,
                                 pageno: 1,
                                 outstream: outstream})

            end create
```

```
add_word = proc (p: cvt, w: word)

        % Process a word.  This procedure adds the word w to the
        % output document.  If in nofill mode, then the word is simply
        % added to the end of the current line (there is no
        % line-length checking in nofill mode).  If in fill mode, then
        % we first check to see if there is room for the word on the
        % current line.  If the word will not fit on the current line,
        % we first justify and output the line and then start a new
        % one.  However, if the line is empty and the word won't fit
        % on it, then we just add the word to the end of the line; if
        % the word won't fit on an empty line, then it won't fit on
        % any line, so we have no choice but to put it on the current
        % line, even if it doesn't fit.

        if p.fill cand ~line$empty (p.line) then
                h: int := word$width (w)
                if line$length (p.line) + h > 60 then
                        line$justify (p.line, 60)
                        output_line (p)
                        end
                end
        line$add_word (p.line, w)
        end add_word

add_space = proc (p: cvt)

        % Process a space -- just add it to the current line.

        line$add_space (p.line)
        end add_space

add_tab = proc (p: cvt)

        % Process a tab -- just add it to the current line.

        line$add_tab (p.line)
        end add_tab

add_newline = proc (p: cvt)

        % Process a newline.  If in nofill mode, then the current line
        % is output as is.  Otherwise, a newline is treated just like
        % a space.

        if ~p.fill
                then output_line (p)
                else line$add_space (p.line)
                end
        end add_newline
```

```
break_line = proc (p: cvt)

          % Cause a line break.  If the line is not empty, then it is
          % output as is.  Line breaks have no effect on empty lines --
          % multiple line breaks are the same as one.

          if ~line$empty (p.line) then output_line (p) end
          end break_line

skip_line = proc (p: cvt)

          % Cause a line break and output a blank line.

          break_line (up (p))
          output_line (p)                    % line is empty
          end skip_line

set_fill = proc (p: cvt)

          % Enter fill mode.

          p.fill := true
          end set_fill

set_nofill = proc (p: cvt)

          % Enter nofill mode.

          p.fill := false
          end set_nofill

terminate = proc (p: cvt)

          % Terminate the output document.

          break_line (up (p))
          if p.lineno > 0 then
                    stream$putc ('\p', p.outstream)
                    end
          stream$close (p.outstream)
          end terminate
```

```
%       Internal procedure.

output_line = proc (p: rep)

        % Output line is used to keep track of the line number and the
        % page number and to put out the header at the top of each
        % page.

        if p.lineno = 0 then                        % print header
                stream$puts ("\n\n", p.outstream)
                put_spaces (10, p.outstream)
                stream$puts ("Page ", p.outstream)
                stream$puts (int2string (p.pageno), p.outstream)
                stream$puts ("\n\n\n", p.outstream)
                end
        p.lineno := p.lineno + 1
        line$output (p.line, p.outstream)
        if p.lineno = 50 then
                stream$putc ('\p', p.outstream)
                p.lineno := 0
                p.pageno := p.pageno + 1
                end
        end output_line

end page

put_spaces = proc (n: int, outstream: stream)

        % This procedure outputs N spaces to outstream.

        for i: int in int$from_to_by (1, n, 1) do
                stream$putc (' ', outstream)
                end
        end put_spaces
```

```
line = cluster is create, add_word, add_space, add_tab, length,
          empty, justify, output

          % A line is a mutable sequence of words, spaces, and tabs.
          % The length of a line is the amount of character positions
          % that would be used if the line were output.  One may output
          % a line onto a stream, in which case the line is made empty
          % after printing.  One may also justify a line to a given
          % length, which means that some spaces in the line will be
          % enlarged to make the length of the line equal to the desired
          % length.  Only spaces to the right of all tabs are subject to
          % justification.  Furthermore, spaces preceding the first word
          % in the output line or preceding the first word following a
          % tab are not subject to justification.  If there are no
          % spaces subject to justification or if the line is too long,
          % then no justification is performed and no error message is
          % produced.

          token = oneof [
                    space: int,          % the int is the width of the space
                    tab: int,            % the int is the width of the tab
                    word: word
                    ]
          at = array [token]
          rep = record [
                    length: int,         % the current length of the line
                    stuff: at            % the contents of the line
                    ]

create = proc () returns (cvt)

          % Create an empty line.

          return (rep${
                    length: 0,
                    stuff: at$new ()
                    })
          end create

add_word = proc (l: cvt, w: word)

          % Add a word at the end of the line.

          at$addh (l.stuff, token$make_word (w))
          l.length := l.length + word$width (w)
          end add_word
```

```
add_space = proc (l: cvt)

        % Add a space at the end of the line.

        at$addh (l.stuff, token$make_space (1))
        l.length := l.length + 1
        end add_space

add_tab = proc (l: cvt)

        % Add a tab at the end of the line.

        width: int := 8 - (l.length//8)
        l.length := l.length + width
        at$addh (l.stuff, token$make_tab (width))
        end add_tab

length = proc (l: cvt) returns (int)

        % Return the current length of the line.

        return (l.length)
        end length

empty = proc (l: cvt) returns (bool)

        % Return true if the line is empty.

        return (at$size(l.stuff) = 0)
        end empty
```

```
justify = proc (l: cvt, len: int)

          % Justify the line, if possible, so that it's length is equal
          % to LEN. Before justification, any trailing spaces are
          % removed. If the line length at that point is greater or
          % equal to the desired length, then no action is taken.
          % Otherwise, the set of justifiable spaces is found, as
          % described above. If there are no justifiable spaces, then
          % no further action is taken. Otherwise, the justifiable
          % spaces are enlarged to make the line length the desired
          % length. Failure is signalled if justification is attempted
          % but the resulting line length is incorrect. This condition
          % indicates a bug in justify; it should never be signalled,
          % regardless of the arguments to justify.

          remove_trailing_spaces (l)
          if l.length >= len then return end
          diff: int := len - l.length
          first: int := find_first_justifiable_space (l)
                    except when none: return end
          enlarge_spaces (l, first, diff)
          if l.length ~= len then signal failure ("justification failed") end
          end justify

output = proc (l: cvt, outstream: stream)

          % Output the line and reset it.

          if ~empty (up (l)) then
                    put_spaces (10, outstream)
                    for t: token in at$elements (l.stuff) do
                              tagcase t
                                        tag word (w: word):
                                                  word$output (w, outstream)
                                        tag space, tab (width: int):
                                                  put_spaces (width, outstream)
                                        end
                              end
                    end
          stream$putc ('\n', outstream)
          l.length := 0
          at$trim (l.stuff, 1, 0)
          end output
```

```
%       Internal procedures.

remove_trailing_spaces = proc (l: rep)

        % Remove all trailing spaces from the line.

        while at$size (l.stuff) > 0 do
                tagcase at$top (l.stuff)
                        tag word, tab:
                                break
                        tag space (width: int):
                                at$remh (l.stuff)
                                l.length := l.length - width
                        end
                end
        end remove_trailing_spaces

find_first_justifiable_space = proc (l: rep) returns (int) signals (none)

        % Find the first justifiable space.  This space is the first
        % space after the first word after the last tab in the line.
        % Return the index of the space in the array.  Signal NONE if
        % there are no justifiable spaces.

        a: at := l.stuff
        if at$size (a) = 0 then signal none end
        lo: int := at$low (a)
        hi: int := at$high (a)
        i: int := hi

        % find the last tab in the line (if any)
        while i>lo cand ~token$is_tab (a[i]) do
                i := i - 1
                end

        % find the first word after it (or the first word in the line)
        while i<=hi cand ~token$is_word (a[i]) do
                i := i + 1
                end

        % find the first space after that
        while i<=hi cand ~token$is_space (a[i]) do
                i := i + 1
                end

        if i>hi then signal none end
        return (i)
        end find_first_justifiable_space
```

```
enlarge_spaces = proc (l: rep, first, diff: int)

        % Enlarge the spaces in the array whose indexes are at least
        % FIRST.  Add a total of DIFF extra character widths of space.

        nspaces: int := count_spaces (l, first)
        if nspaces = 0 then return end
        neach: int := diff/nspaces
        nextra: int := diff//nspaces
        for i: int in int$from_to_by (first, at$high (l.stuff), 1) do
                tagcase l.stuff[i]
                        tag space (width: int):
                                width := width + neach
                                l.length := l.length + neach
                                if nextra > 0 then
                                        width := width + 1
                                        l.length := l.length + 1
                                        nextra := nextra - 1
                                        end
                                l.stuff[i] := token$make_space (width)
                        others:
                        end
                end
        end enlarge_spaces

count_spaces = proc (l: rep, i: int) returns (int)

        % Return a count of the number of spaces in the line whose
        % indexes in the array are at least I.

        count: int := 0
        while i <= at$high (l.stuff) do
                tagcase l.stuff[i]
                        tag space:
                                count := count + 1
                        others:
                        end
                i := i + 1
                end
        return (count)
        end count_spaces

end line
```

```
word = cluster is scan, width, output

        % A word is an item of text.  It may be output to a stream.
        % It has a width, which is the number of character positions
        % that are taken up when the word is printed.

        rep = string

scan = proc (c: char, instream: stream) returns (cvt)

        % Construct a word whose first character is C and whose
        % remaining characters are to be removed from the instream.

        s: string := string$c2s (c)
        while true do
                c := stream$peekc (instream)
                   except when end_of_file: break end
                if c = ' ' cor c = '\t' cor c = '\n' then
                        break
                        end
                s := string$append (s, stream$getc (instream))
                end
        return (s)
        end scan

width = proc (w: cvt) returns (int)

        % Return the width of the word.

        return (string$size (w))
        end width

output = proc (w: cvt, outstream: stream)

        % Output the word.

        stream$puts (w, outstream)
        end output

end word
```