SASL
(St. Andrews Static Language)
Reference Manual

TM-26

D.A. Turner
Department of Computer Science
University of Kent at Canterbury

edited and adapted for MTS
by
Harvey Abramson

October 1981

Department of Computer Science
The University of British Columbia
Vancouver, British Columbia V6T 1W5

Contents

Notice

SASL, the language and system described here, was first implemented in C under UNIX by Dr. William F. Campbell at the University of St. Andrews in Scotland. It has been translated into BCPL and modified for MTS by Prof. Harvey Abramson of the Department of Computer Science at the University of British Columbia. Please bring any bugs in the BCPL-MTS implementation of InterSASL, or any discrepancies with this manual, to my attention.

Prof. Harvey Abramson

I INTRODUCTION

SASL is a mathematical notation for describing certain kinds of data structure. The name SASL stands for "St Andrews Static Language". "Static" because unlike a conventional programming language, SASL contains no commands and a data structure, once defined, cannot be altered. For more information about the advantages of this kind of language and its relationship to other languages the reader is referred to the list of references on page 4.[1].

This manual is intended simply to describe SASL notation , to motivate its use by some elementary programming examples and to serve as a reference manual for the SASL user. The reader using this document as a reference manual should turn to section VIII where the syntax of SASL is summarised. The first time reader is advised to read sections I to VII in order at least once and then attempt some of the exercises in Appendix I.

In SASL a "program" is an expression describing some data object - the response of the system is to print a representation of the object. For example

```
2 + 3
```

is a SASL program, to which the system responds by printing

```
5
```

A slightly more complicated example will serve better to convey the flavour of the language

```
fac 4

where fac 0 = 1

      fac n = n * fac (n - 1)
```

The system responds to this expression by printing 24 (the factorial of 4). The value of such a construction (called a where-expression) is the value of the expression before the "where". Unlike the case of "+" in the first example, a knowledge of the factorial function is not built into the SASL system so the expression "fac 4" is followed by a definition of the meaning of the name "fac". Definitions of the meanings of any number of names can be given following a "where". Each definition consists of one or more clauses.

Each clause of a definition can be read as a true statement, an equation, involving the entity being defined. In the above example there are two equations involving the factorial function. The "n" of the second clause stands for an

arbitrary number (excluding zero which has been dealt with specially by the first equation). Mathematically the two equations are sufficient to define the factorial function - indeed it can be regarded as the solution of the equations.

Computationally each clause can be read as a substitution rule asserting that the form on the left, wherever it occurs, should be replaced by the form on the right. Using the clauses in this way on the expression "fac 4" yields in succession the expressions:

```
4 * fac 3

4 * (3 * fac 2)

4 * (3 * (2 * fac 1))

4 * (3 * (2 * (1 * fac 0)))

4 * (3 * (2 * (1 * 1)))
```

which gives the value 24. It can be shown that the result obtained by using such clauses as substitution rules is always the same as that obtained by finding the mathematical solution of the clauses considered as equations.

In SASL by the use of where any expression can be followed by the definitions of the meanings of one or more names. So for example the expression to the right of an "=" in a definition can itself be a where-expression. In this way expressions of arbitrary complexity can be constructed. For example

```
binomial (n,3) + binomial(n,4)

      where n = 10

      binomial (n,r) = fac n / (fac (n - r) * fac r)

                  where fac 0 = 1

                        fac n = n * fac (n - 1)
```

gives the sum of the binomial coefficients $^{10}C_{3} + {}^{10}C_{4}$

Acknowledgements

No particular originality is claimed for SASL - indeed it was a design aim that the notation should be as "standard" as possible - the debt to Landin and Strachey will be particularly apparent.

I would like to express my thanks to my colleagues Antony Davie and Michael Weatherill for their advice and encouragement, numerous St Andrews Honours students for their patience and constructive criticism in trying out earlier versions of SASL and Maureen Saunders for typing this manual.

D A Turner December 1976

References

[1] W.H. Burge, Recursive Programming Techniques, Addison-Wesley, 1975.

[2] William Campbell, An Abstract Machine for a Purely Functional Language, Dept. Of Computer Science - University of St. Andrews, 1979.

[3] Peter Henderson, Functional Programming - Application and Implementation, Prentice-Hall, 1980.

[4] D.A. Turner, A New Implementation Technique for Applicative Languages, Software-Practice and Experience, vol. 9,31-49,1979.

[5] D.A. Turner, The Semantic Elegance of Applicative Languages, Proceedings - ACM Conference on Functional Programming Languages and Architecture, 1981.

II OBJECTS

The data items which SASL expressions describe are called throughout this manual objects. Every SASL expression has an object for its value. An expression has no other significance than as a way of talking about this object - it can be replaced by any other expression which has the same value without affecting the value of any larger expression of which it is a part. This property of expressions is called referential transparency. A SASL program is an expression and the outcome of the program is to print the object which it has for its value.

There are 6 types of object in SASL's universe of discourse:

(a) numbers - ie the integers, positive, negative and zero

eg 0 13 -6 128

(b) truthvalues - there are two of these,

true and false

(c) characters - these are written using "%" as a device that quotes the character immediately following it

eg %A %1 %%

Not conveniently written in this form are the control characters for space, newline, newpage and tab, written instead

sp nl np tab

respectively.

(d) lists - a list is an ordered set of objects called its components.

eg 1, 2, 3

is a list of length 3, all of whose components are numbers. It is also permitted to have infinite lists - for example the list of all prime numbers

2, 3, 5, 7, 11, 13, ...

is a valid SASL object. (See Appendix I, exercises 1, 8, 9).

(e) Functions - a function is a law of correspondence, assigning to each object in SASL's universe of discourse a corresponding object called the value of the function on that object, or if you prefer, the "output" of the function given the corresponding object as "input". For example the "fac" of the introduction is a function which assigns to each non-negative integer its factorial and gives the value undefined on all other objects.

(f) undefined - finally there is a unique object undefined which is the value of ill formed or non-terminating expressions like

```
2 + false
```

or

```
fac (-3)
```

It is also produced by definitions which give inadequate information. For example the definition

```
x = x
```

gives the name "x" the value undefined.

Completeness

All six types of object have the same "civil rights" :

Any object can be named
Any object can be the value of an expression
Any object can be a component of a list
Any object can be given to a function as its input
Any object can be returned by a function as its output

So among the possibilities are - a list of lists, a list of functions, a function which returns a list, a function which returns a function.

Note that the different types of object can be mixed freely. For example

```
a > b -> true ; 33
```

equivalent to the (illegal) Algol

```
if a > b then true else 33
```

is a perfectly legal SASL expression. In the same spirit the components of a list need not all be of the same type. So for

example

```
1, true, fac(-3), (1, 2, 3)
```

is a list whose four components are respectively - a number, a truthvalue, undefined and a list. Similarly a given function need not always return the same type of object as output. (Though given the same object as input a function must always return exactly the same object as output, because of the static nature of the language.)

Functions with more than one input

Every function expects one object as input and gives one object as output. Either of these objects, however, could be a list. This gives us one way (due to P J Landin) of representing a function of several arguments - for example the definition

```
f(x,y) = x + y
```

makes f a function which expects as input a 2-list and returns the sum of its components. So

```
f(2 , 3) is 5
```

Notice by the way that we could give the input list a name by a definition like

```
L = 2, 3
```

and then write instead the expression

```
f L
```

and this also would have the value 5.

Note on syntax - The reader should understand why we need the brackets in f(2, 3). Not to make a list - that is done by the comma. Nor to denote functional application - mere juxtaposition does that. They are there because functional application binds more tightly than comma - without them we should be applying f to 2 only and not to the whole list.

This method of representing a function of several inputs allows us to represent a function with a variable number of inputs. For example we can define a function "sum" (see section VI) that takes a list of any length and sums its components. Notice also that we can represent a function with several outputs (even a variable number of outputs) by having it return a list. So for example, f, defined by

```
f(a, b) = a + b, a - b
```

returns as a pair the sum and difference of its inputs.

Curried Functions

Another method of representing a function of several inputs (named after the American logician H B Curry but due to Schonfinkel) uses a function-generating function. As mentioned earlier, any object can be returned by a function as its value. In particular, a function-generating function may be defined, ie, a function which returns another function as its value. For example, consider the definition

```
f x y = x + y
```

f is a function that when applied to an input, "x", returns another function that when applied to an input, "y", returns x + y. So

```
f 1 2 is 3
```

Note that f 1 2 is read as (f 1) 2 and that f 1 has a meaning in its own right - it is the function that adds one to things. In fact the definition of f could have been written more explicitly as

```
f x = g
      where g y = x + y
```

Curried functions like this f are extremely convenient and provide the normal method in SASL for representing functions of two or more arguments. For example given the definition

```
dooda x y z = (y + z)/x
```

dooda is a curried function of three arguments, with eg

```
dooda 3 5 7
```

giving the value 4. A curried function can always be applied to less than its full quota of arguments, to produce a "more specific" version of the function requiring fewer arguments. For example

```
dooda 2
```

is a (curried) function of two arguments that could be called "average".

III LEXICAL CONVENTIONS

Textually expressions are built up from units called basic symbols. There are four kinds of basic symbols: names (like "fac"), constants (like "4"), operators (like "+") and seven special symbols called delimiters

where = ( ) , -> ;

A name is any sequence of letters, digits and the "_" symbol beginning with a letter. Examples

x x1 fac a_rather_long_name

The operators and the various kinds of constant are listed in section VII.

A basic symbol can consist of more than one character but is regarded as a single textual entity. For a list of "reserved words", ie names especially reserved for use as basic symbols, the reader is referred to Appendix III.

Layout

Certain characters, called layout, are ignored by the system and can be placed freely between basic symbols to make programs readable. Layout consists of spaces, newlines and comments. A comment consists of two vertical bars and all symbols to the right on the same line, thus:

```
|| this is a comment
```

Layout cannot occur inside a basic symbol, except a string constant where it is not ignored but taken as part of the message being quoted. The presence of layout between basic symbols is optional, subject to the following constraints:

i) Adjacent symbols must be separated by at least one space wherever they would otherwise constitute an instance of a single, larger, basic symbol. For example in

```
fac 4
```

the space between the name and the constant is necessary because fac4 would not be read by the system as a name followed by a constant but taken as a single name.

ii) The following offside rule must be observed: "Every symbol of an expression must lie below or to the right of the first symbol of the expression" So for example 2 + 3 can be written as

```
2+3     || all squashed up, or as

2       || all
  +     || spread
 3      || out, but not as

 2 +
3       || the 3 is OFFSIDE and will be
        || rejected by the system.
```

Finally note the convention that the delimiter ; can be omitted provided a newline is taken instead. So a -> b ; c will often be written

```
a -> b
c
```

Similarly the clauses following a where, which are supposed to be separated by semicolons (when there is more than one clause) are usually written one per line with the semicolons omitted.

IV EXPRESSIONS

Simple Expressions

A simple expression is a name or a constant. Also any expression no matter how large and complicated (for example a where-expression) can be enclosed in brackets without altering its meaning and then becomes a simple expression. Brackets are not used for any other purpose.

Note on syntax The first time reader of a language manual often finds himself asking the question "can I ...?" eg "can I write a conditional expression as a component of a list?", "can I write a where-expression as the argument to a function?". The above rule about bracketing says that the answer to such questions in the case of SASL is always yes. If the syntax (summarised in section VIII) appears at first glance to forbid it, this merely means that the offending sub-expression needs to be enclosed in brackets.

Combinations

If two simple expressions are juxtaposed, this represents the application of a function to an argument (input). For example

```
f x
```

denotes the result of applying the function f to the object x. Notice that no brackets are needed - f(x) is also legal but so is (f)x or (f x) they all mean the same - extra brackets can always be inserted in an expression to emphasise grouping without altering the meaning. In f(x + 1) however the brackets are necessary because x + 1 is not a simple expression. Functional application always associates to the left, so

```
f x y
```

means that the function f is applied to the object x, yielding as a result a function which is then applied to y. Thus:

```
(f x)(y)
```

to put in some unnecessary but harmless brackets.

Operator Expressions

example 13 * (x + f y)

Operator expressions are built up out of simple expressions and combinations using various operators. Different operators have different binding powers, as is customary in mathematical notation. For example "*" is more binding than "+" whence the need for brackets in the above example. The operators are

listed in section VIII together with their binding power and include the arithmetic operators + - * / rem ("/" is integer division), the relational operators > >= = ¬= < <= and the logical operators ¬ & | ("|" is inclusive "or"). There is a dot operator for functional composition; i.e., (f.g) x is the same as f(g x) - the dot is the most binding operator. In addition there are two special operators on lists, : and ++ , which are discussed later. With the exception of these last two, all operators associate to the left, so for example a - b - c means (a - b) - c. Note also that functional application is more binding than any operator so in

```
f x + 1
```

f is being applied to x, not to x + 1.

Equality

Notice that the sign "=" which has already been encountered as a delimiter in definitions is also used as an operator in expressions. (This is an example of two different basic symbols being represented by the same character - fortunately the system is always able to tell by context which use is intended.) Thus in the definition

```
delta = i = j -> 0 ; 1
```

the second "=" is clearly being used as a relational operator. In general each operator expects operands of a particular type - eg numbers for the arithmetic and relational operators, truthvalues for the logical operators - and yields the value undefined otherwise. The equality operators = and ¬=, however, are defined between arbitrary pairs of objects. Objects of different type are always unequal. E.g., - the following expressions all take the value true:

```
2 + 2 = 4          1 ¬= 2          false ¬= true

1 ¬= false         %A ¬= %a        (1,2,3) = (1,1+1,1+1+1)
```

Notice in the last example that equality between lists means element by element equality. Lists of different length are always unequal.

Logically, two functions are equal if, for every object in SASL's universe, when they are both given the object as input they both give the same output. Unfortunately this is not a decidable question, since there is an infinity of objects to be tested as inputs. Therefore the expression

```
f = g
```

where f and g are both functions takes the value undefined, (but

of course if only one of them is a function the result will be false).

Finally note that = and ¬= cannot be used to test for undefined, i.e.,

```
a = b          a ¬= b
```

are both undefined if either a or b is undefined. It is a fundamental result in the theory of computation (the Halting theorem) that there can be no effective test for undefined.

List Expressions

Operator expressions may be separated by commas to denote lists.

Example

```
a, b, a + b, a - b
```

Note that the list is created by commas and that brackets are not needed. In SASL brackets are used only to force (or to emphasise) grouping. Note also that commas are less binding than any operator.

The components of a list are accessed by applying the list to a number; 1 for the first component, 2 for the second, etc. So if "L" names the list

```
1 , (2, 3, 4), (5, 6)

L 1 is 1

L 2 is 2, 3, 4

L 3 is 5

L 4 is undefined

L 3 2 is 6  (remember the left association rule)
```

The list with no components - the empty list - is written specially:

```
()      || Pronounced "nil"
```

A special notation is also needed for singleton lists. Thus

```
2,
```

is a list of length one, whose first and only member is the

number 2.

A string is just a list all of whose components happen to be characters. So

```
'hello"
```

is just shorthand for

```
%h, %e, %l, %l, %o
```

and the empty string '" is just the same as the empty list ().

There are two useful operators on lists, namely : ("prefix") and ++ ("concatenate"). Examples of prefixing:

```
1 : (2, 3, 4)    gives  1, 2, 3, 4

%h : 'ello"      gives  'hello"

a : ()           gives  a,

() : (1, 2, 3)   gives  () , 1, 2, 3

(1,2) : (3,4,5)  gives  (1,2),3,4,5
```

Note that the left operand of ":" can be any object, that the right operand must be a list, and that its action is to make the list one component longer by prefixing the given object as its first component.

Examples of concatenation

```
(1,2) ++ (3,4,5)  gives 1,2,3,4,5

(1,) ++ (2,3)     gives 1,2,3

'hel" ++ 'lo"     gives 'hello"

() ++ L = L ++ () = L  provided L is a list

() ++ () = ()
```

Notice that both operands of "++" must be lists, or else the result is undefined.

Both these operators associate to the right (no other operator does). So

```
1 : 2 : (3 , 4)   gives 1,2,3,4

'ab" ++ %c : 'de" gives 'abcde"
```

Conditional Expressions

These are of the form

```
condition -> object1 ; object2
```

and the value is object1 if condition is true, object2 if condition is false, and undefined otherwise. So for example

```
x < 0 -> - x ; x
```

denotes the "absolute value" of x.

Syntactically, the condition can be represented by an operator expression and each alternative by any expression except a where-expression (i.e., a where-expression would have to be enclosed in brackets if it occurred as the arm of a conditional). So for example the second arm of the conditional could be another conditional and so on, allowing the often useful form:

```
condition1  ->  object1
condition2  ->  object2
                .
                .
                .
conditionN  ->  objectN
object
```

Note that the layout here obviates the need for any semicolons.

where-Expressions

Finally, any expression can be followed by "where definitions" as explained in the introduction. There can be any number of definitions, each consisting of any number of clauses. Each name defined can be used throughout the where-expression, with the given meaning, unless it is redefined in an inner where-expression, in which case its use for a different purpose in the inner where-expression in no way interferes with its use in the rest of the expression. So for example

```
a + (a + a
        where a = 2)
where a = 1
```

has value 5, since the first "a" has the value 1 and the next

two have the value 2. The first "a" is said to be outside the scope of the inner where. It should be stressed that the meaning of a given occurrence of a name depends on its textual position. So

```
1 + (f 1 where y = 10)

where

f x = x + y

      where y = 1
```

has the value 3 (no, not 12).

Outside the scope of any where all names have the value undefined, except those defined in the so-called "predefined environment", which denote standard functions (see section VI). By means of directives of the form "def definitions ?" the user can extend the predefined environment (see Appendix III).

When more than one clause is used to define a single function the clauses must be grouped together; furthermore their order is significant in that during substitution clausal forms are matched in the order they are given.

The order of definitions of distinct objects in a where-expression is of no significance.

V DEFINITIONS

Multiple Definitions

The simplest kind of definition sets a single name equal to an expression, as in

```
x = 13
```

This is a special case of a more general form

```
"namelist" = "expression"
```

where "namelist" is a construction of arbitrary complexity built from names and constants using commas, brackets and the operator ":". The effect of the definition is that the names on the left are given values such that the equation becomes a true one. (If there are no such values the names are given the value undefined).

Examples

1)
```
x, y, z = 1, 2, 3
```

has the same effect as the three simple definitions

```
x = 1; y = 2; z = 3
```

2)
```
x, y  =  a < b -> 1, 2 ; -1 , -2
```

has the same effect as

```
x  =  a < b -> 1 ; -1
y  =  a < b -> 2 ; -2
```

3)
```
(a,b), c, d = L   || L must be a 3-list
                  || with L 1 a 2 - list
```

has the same effect as

```
a = L 1 1; b = L 1 2; c = L 2; d = L 3
```

4)
```
a : b : c = 'hello"
```

has the same effect as

```
a = %h;  b = %e;  c = 'llo"
```

5)
```
1, x, x  =  L
```

here L must be a 3-list with L 1 = 1 and L 2 = L 3 in which case the definition has the effect of

```
x = L 2
```

Function-form Definitions

These consist of one or more clauses of the form

"function form" = "expression"

where "function form" consists of the name of the function being defined followed by one or more formal parameters. Each formal parameter is a name, a constant or a "namelist" enclosed in brackets. The names occurring in the formal parameters are not being defined but are purely local to the clause, standing for arbitrary input objects.
Examples

1)
```
I x = x
```

Defines the "identity function" - applied to any object (even itself!) It gives the same object back as output.

2)
```
f 0 = 1
f 1 = 2
f 2 = 0
```

Defines f to be "add 1 modulo 3".

3)
```
A(0,n) = n + 1
A(m,0) = A(m-1,1)
A(m,n) = A(m-1,A(m,n-1))
```

Defines "Ackerman's function". Note that the last clause "(m,n)" stand for an arbitrary pair of inputs excluding those which are dealt with specially by the other two clauses. (The order in which the clauses are written is relevant.)

Section VI, where the standard functions are defined, should be read carefully for many more examples of SASL function definitions.

A Note on Recursion

As the reader will have gathered from previous examples in SASL definitions can be recursive. Mutual recursion is also permitted; eg,

```
odd x = x = 1 | even (x-1)

even x = x = 0 | odd (x-1)
```

Not only functions but also lists can be defined recursively. For example

```
x = 1 : x
```

makes x the infinite list 1,1,1,... (Any infinite list whose structure can be described recursively is permitted in SASL.) Another example of this kind of "immediate" recursion is to define a function f by

```
f = H f
```

where H is a function-generating function. For example if H was defined by

```
H g n = n = 0 -> 1 ; g (n-1) * n
```

then f would be the factorial function. Immediate recursion (and immediate mutual recursion) is always permitted in SASL.

VI PREDEFINED FUNCTIONS

The following definitions establish the predefined environment of standard functions. Where a function cannot conveniently be defined in SASL its defining expression is given as a comment in English.

```
hd (a : x) = a  || so hd x is just x 1 (provided x is a list)

tl (a : x) = x

|| So hd gives the first component of a list and tl removes the
|| first component from the list.  For any non-empty list x,
|| (hd x : tl x) = x.

number x   =  || if x is a number then true else false

char x     =  || likewise, characters

logical x  =  || likewise, truthvalues

list x     =  || likewise, lists

function x =  || likewise, functions

|| So for any x which is not undefined exactly one of the above
|| five functions gives true and the other four give false.

letter x =  x = %A | x = %B | ... | x = %Z |
            x = %a | x = %b | ... | x = %z

digit x  =  x = %0 | x = %1 | ... | x = %9

abs x =  x < 0 -> -x ; x

length () = 0
length (a : x) = 1 + length x

reverse () = ()
reverse (a:x) = reverse x ++ (a,)

sum () = 0
sum (a : x) = a + sum x

product () = 1
product (a : x) = a * product x

and () = true
and (a : x) = a & and x

or () = false
or (a : x) = a | or x
```

```
count a b =  a > b -> ()
             a : count (a + 1) b

|| So count a b is the list a, a + 1, ..., b

from n =  n : from (n + 1)  || ie the infinite list n, n+1, n+2, ...

map f () = ()
map f (a : x) = f a : map f x

   ||  So  map f (a1, ..., an) gives  f a1, ..., f an

for a b f = map f (count a b)

zip x =  hd x = () -> ()
         map hd x : zip (map tl x)

   ||  So  zip ( (a1, ..., an), (b1, ..., bn), ..., (z1, ..., zn) )
   ||  gives  ( (a1, b1, ..., z1), ..., (an, bn, ..., zn) )
   ||  eq  zip( 'abc", '+++", '123" ) gives ( 'a+1", 'b+2", 'c+3" )

while f g x =  f x -> while f g (g x)
               x

until f g x =  f x -> x
               until f g (g x)

||                    n                                    n
||  while f g x = g x  for the least n such that  f (g x) is false.

||  The next three functions rest on the idea that a list in which
||  there are no repetitions can conveniently be used to
||  represent a set.

member () a = false
member (a : x) a = true   || note the repetition of "a" in the form.
member (a : x) b = member x b

   ||  member set x is true iff x occurs in the list, set.

union () y = y
union (a : x) y =  member y a -> union y x
                   || insures against repetitions
                   a : union x y

intersection () y = ()
intersection (a : x) y =  member y a -> a : intersection x y
                          intersection x y


code ch =    || the numeric code for character ch
             || implementation dependent
```

```
decode ch = || the character whose code is n
            || implementation dependent

digitval d = || the digit corresponding to the character d,
             || eg, digitval %3 gives 3

spaces 0 = ()
spaces n = sp : spaces (n - 1)

width x =  x = true -> 4
           x = false -> 5
           x = () -> 0
           list x -> width ( hd x ) + width ( tl x )
           char x -> 1
           number x ->  x < 0 -> 1 + width ( -x )
                        x > 9 -> 1 + width ( x / 10 )
                        1
           'unknown width"

    || So width x the number of characters needed for printing x.
    || See section VII.

Ljustify n x = x, spaces (n - width x)

Rjustify n x = spaces(n - width x), x

Cjustify n x = spaces L, x, spaces R
               where L, R = D/2 , D - L
                     D = n - width x

|| Ljustify n x left-justifies x when printed in a field of total
|| print-width n; Rjustify and Cjustify likewise but right- and
|| centre-justified respectively.

show () = '()"   || show x? lets you see
show nl = 'nl"   || the structure of x.
show sp = 'sp"   || See Section VII - OUTPUT.
show tb = 'tb"
show x = char x -> %% , x
         list x ->
              haschars 6 x -> %' : showstr x
              %( : show (hd x) : showlist (tl x)
         x
         where
              haschars 0 x = true
              haschars n () = true
              haschars n (a : x) = char a & haschars (n - 1) x
              showlist () = %),
              showlist (a : x) = %, : show a : showlist x
              showlist x = ')++" : show x
              showstr () = %',
              showstr x = char (hd x) -> (hd x) : showstr (tl x)
                          %" : '++" : show x
```

VII OUTPUT

The object which is the value of a program is printed using the following conventions.

If it is a character it is set down in the first print position of a single, otherwise blank, line of output. If it is a number it is printed as a string of decimal digits, preceded by a "-" if negative. If it is a truthvalue it is printed as true or false. Each of these objects is printed occupying the minimal number of print positions with no preceding or following layout.

An attempt to print a function results in a warning message (consisting of the name of the function in angle brackets <...>) since a function is an infinite object with no standard external representation.

An attempt to print the undefined object results either in an error message or in a failure to produce any output (because the system has gone into a loop trying to evaluate the expression whose value is undefined).

A list is printed by printing each of its components in turn, from left to right, starting with the first and without inserting spaces, commas or any other delimiters between the components. This has the effect of "flattening out" all structure, so each of

```
1, 2, 3, 4

(1,2), (), (3,4)

'1234"
```

produce the same output when printed, namely:

```
1234
```

So spaces and newlines are not automatically inserted in the output. It is up to the user to include the layout characters he wants at relevant points in the structure that is the final value of his program.

The user will find, after a little practice, that this scheme gives great flexibility in the control of layout. The functions Ljustify, Rjustify and Cjustify (see section VI) are useful here.

Example. The following program prints a table of factorials

```
title, for 1 7 line
where
title = 'A TABLE OF FACTORIALS", nl
line n = 'factorial ", n, ' is ", fac n, nl
fac 0 = 1
fac n = n * fac (n - 1)
```

This produces the output

```
A TABLE OF FACTORIALS
factorial 1 is 1
factorial 2 is 2
          .
          .
          .
factorial 7 is 5040
```

show

The user can prevent any particular structure from being "flattened" on printing by using the standard function show - for any list structure x, show x prints out a full description of x (which could be read back in again to create the object x). In fact it works (see section VI) by inserting extra characters into the structure of x so that when show x is flattened the result is an expression describing the structure of x.

In general all output is directed to the terminal at which the SASL program was typed in. The user has the option of redirecting the output for the SASL program to a named file (see Appendix III).

VIII SUMMARY OF SYNTAX ETC

<expr> ::= <expr> where <defs> | <condexp>

<condexp> ::= <opexp> -> <condexp> ; <condexp> | <listexp>

<listexp> ::= <opexp>,...,<opexp> | <opexp>, | <opexp>

<opexp> ::= <prefix><opexp> | <opexp><infix><opexp> | <comb>

<comb> ::= <comb><simple> | <simple>

<simple> ::= <name> | <constant> | (<expr>)

<defs> ::= <clause> ; <defs> | <clause>

<clause> ::= <namelist> = <expr> | <name> <rhs>

<rhs> ::= <formal><rhs> | <formal> = <expr>

<namelist> ::= <struct>,...,<struct> | <struct>, | <struct>

<struct> ::= <formal> : <struct> | <formal>

<formal> ::= <name> | <constant> | (<namelist>)

<constant> ::= <numeral> | <char-const> | true | false | () |<string>

<numeral> ::= <digit>* | - <digit>* || * means 1 or more

<char-const> ::= %<any char> | sp | nl | tab

<string> ::= '<any message not containing unmatched string quotes>"

Note: a semi-colon ; may be replaced by a newline in
<condexp> and <defs>.

Operators in order of increasing binding power

:	++	infix (right associative)
|		infix
&		infix
¬		prefix
> >= = ¬= <= <		infix
+ -		infix
+ -		prefix
* /	rem	infix
.		infix

List of predefined functions

hd	tl	number	char	logical	list
function	letter	digit	abs	length	reverse
sum	product	and	or	count	from
map	zip	while	until	member	union
intersection	code	decode	digitval	spaces	Ljustify
Rjustify	Cjustify	show			

APPENDIX I SOME SASL EXERCISES

Solutions to these problems are given in Appendix II. None of them require more than a dozen lines of SASL for their solution. They have not been chosen to be specially difficult or tricky but simply to illustrate some of the basic techniques used in SASL programming.

1) Define an infinite list of the Fibonacci numbers

```
fib = 1, 1, 2, 3, 5, ... || except for the first two,
                         || each number is the sum of
                         || the previous two
```

2) Define a function "sort" which takes a list of numbers and sorts it into ascending order.

3) Write a program to print the moves for Towers of Hanoi (ask someone if you do not know the rules) with 3 discs.

4) Write a program to print the following table:

```
A TABLE OF POWERS

N     N**2     N**3     N**4     N**5
1     1        1        1        1
2     4        8       16       32
```

... etc where N runs up to 10 and the table is to fill a page of width 100.

5) Define a function "perms" which lists all the permutations of a given list.

6) A (curried) function of n arguments can be called tautologous if it returns true for every one of the 2**n possible combinations of truth-valued arguments. Define a function "Taut" to test for this.

7) If f is a function that expects a 2-list, then

```
curry f

where curry f x y = f(x , y)
```

is a curried function of two arguments with the same output. Write a more general function, CURRY, such that if f is a function that expects an n-list then

```
CURRY n f
```

will be the corresponding function of n arguments.

8) Write a program which will print all the prime numbers in order, starting with 2. (Use the sieve of Eratosthenes.)

9) Write a program to produce the following output:

```
The 1st line is :
'The 1st line is :"
The 2nd line is :
''The 1st line is :""
The 3rd line is :
'The 2nd line is :"

... etc
```

10) Write a program to find a way of placing 8 queens on a chess board so that no queen is in check from another.

APPENDIX II SOLUTIONS TO EXERCISES

In some cases several solutions are given to illustrate different techniques.

1) Fibonacci

a) || an easy way, using an auxilliary function f

```
fib = f 1 1
f a b = a : f b (a + b)
```

b) || by immediate recursion

```
fib = 1 : 1 : map sum (zip (fib, tl fib))
```

2) Sorting

a) || insertion sort (simple but inefficient)

```
sort () = ()
sort (a : x) = insert a (sort x)
insert a () = a,
insert a (b : x) =  a < b -> a : b : x
                    b : insert a x
```

b) || quicksort

```
sort () = ()
sort (a : x) = sort m ++ a : sort n
               where m, n = split a x () ()
split a () m n = m, n
split a (b : x) m n =  b < a -> split a x (b : m) n
                       split a x m (b : n)
```

c) || treesort

```
sort x = flatten (maketree x () )
maketree () t = t
maketree (a : x) t = maketree x (add a t)
add a () = a, (), ()
add a (b, L, R) =  a < b -> b, add a L, R
                   b, L, add a R
flatten () = ()
flatten (a , L , R) = flatten L ++ a : flatten R
```

3) Towers of Hanoi

```
hanoi 3 'abc"
where
hanoi 0 (a,b,c) = ()
hanoi n ( a,b,c) = hanoi (n-1) (a,c,b),
                   'move a disc from ",a,' to ",b,nl,
                   hanoi (n-1) (c,b,a)
```

4) Table of Powers

```
title, captions, for 1 10 line
where
title = 'A TABLE OF POWERS", nl
captions = map f (%N : for 2 5 caption), nl
caption i = 'N**", i
f = Ljustify 20
line n = map f (for 1 5 (powers n)), nl
powers n = f n
           where f x = x : f (n * x)
```

5) Permutations

```
|| There are umpteen ways of defining this, but here is one
|| of the shorter ways.

perms () = (),
perms x = f x
where
f (a : y) = map (cons a) (perms y) ++ g (y ++ (a,))
g y =  x = y -> () ; f y
cons x y = x : y
```

6) Taut

a)
```
|| This would be a much harder problem if f wasn't
|| curried.

Taut 0 t = t
Taut n f = Taut (n-1) (f true) & Taut (n-1) (f false)
```

b)
```
|| a refinement - we don't even need to know n

Taut f = logical f -> f
         Taut (f true) & Taut (f false )
```

7) CURRY

```
|| This is a bit subtle, although its only 3 lines long

CURRY 0 f = f ()
CURRY n f x = CURRY (n - 1) (pa f x)
pa f a x = f (a : x)
```

|| pa means "partially apply". The key step here was inventing "pa" which takes a function which expects an n-list and freezes its first argument creating a function that wants an n-1 list. The reader should satisfy himself that CURRY as defined obeys the equation CURRY n f x1...xn = f (x1...xn) for arbitrary n, as required.

8) Primes

```
show primes
where
primes = sieve (from 2)
sieve (p : x) = p : sieve (filter p x)
filter p (a : x) =  a rem p = 0 -> filter p x
                    a : filter p x
```

9) Recursive display of lines

```
zip (L, newlines)
where
L = f 1
newlines = nl : newlines
f n = ('the ", n, ord n, ' line is :") :
      (%', L n, %") : f (n + 1)
ord 1 = 'st";  ord 2 = 'nd";  ord 3 = 'rd";
ord n =  n >= 10 -> ord (n rem 10)
         'th"
```

10) Eight Queens

```
|| One method

displ soln
where
displ b = zip ( 'rnbqkbnr" , b , spaces 8)
soln = until full extend ()
extend b = until safe alter (addqueen b)  || b is a board
addqueen b = 1 : b
full b = length b = 8
alter(8 : b) = alter b || backtrack
alter(q : b) = q + 1 : b
safe(q : b) = and (for 1 (length b) nocheck)
              where nocheck i = q ¬= b i & abs(q - b i) ¬= i
```

```
|| another method

hd (solns 1 ()) || hd because we want only the first solution
where
solns q b = q > 8 -> ()
            safe q b -> length (q : b) = 8 -> displ (q : b),
                        solns 1 (q : b) ++ solns (q+1) b
            solns (q+1) b
safe q b = and (for 1 (length b) nocheck)
           where nocheck i = q ¬= b i  & abs (q - b i) ¬= i
displ b = zip ( 'rnbqkbnr" , b , spaces 8) , nl
```

APPENDIX III RUNNING SASL UNDER MTS

The SASL system that runs under MTS is interactive and allows the user to maintain a global environment of defined names and to evaluate SASL expressions interactively within this environment.

1) Entering the SASL System

To "login" to sasl type

```
$run sasl:intersasl
```

Normally, SASL establishes a skeleton environment that contains just a few system defined names. Should the user wish to start with an environment containing all of the SASL predefined functions (see section VI) he may type

```
$run sasl:intersasl par=sasl:sasl.prelude
```

instead. In fact, he may start with any predefined environment that he has prepared using a dump command (see below) by typing

```
$run sasl:intersasl par=filename
```

where the named file contains the prepared environment.

InterSASL currently runs in a space of 250,000 "SASL cells", or in 750,000 words of memory.

2) Typing Rules

Reserved words (eg where) are typed in without underlining. The following words all have special meaning either in SASL system commands or in the SASL language proper. Therefore they may not be used as names.

where	rem	true	false	nl
tb	def	display	echo	noecho
off	mess	nomess	delete	dump
trace	gc	nogc	help	to
sp	get	load	definitions	

3) SASL System Commands

Once logged into SASL the user may type in any of the following commands.

a) help
- list a summary of SASL commands.

b) Maintaining the global environment.

definitions
- list all names defined in the current global environment.

display <name>
- display the definition of the named object.

def <defs> ?
- add the definitions to the environment. New definitions of existing names replace old definitions. (See section VIII for syntax of <defs>.)

delete <name>
- delete the named object from the global environment.

dump <filename>
- dump a copy of the current global environment onto the named MTS file; the environment is dumped to -DUMP if <filename> does not exist.

load <filename>
- load a new global environment from the named MTS file; this replaces the old one. The file must have been prepared by the use of a dump command.

c) Evaluating SASL expressions

<expr> ?
- evaluate the SASL expression and print its value as output.

<expr> to <filename>
- as before but output is appended to the end of the named MTS file. The output is written to -OUT if <filename> does not exist. The user can cancel the evaluation of an expression by pressing the ATTN key

d) Controlling messages

mess (nomess)
- enable (inhibit) the output of resource usage messages at the end of subsequent evaluations.

gc (nogc)
- enable (inhibit) the output of heap usage messages at each subsequent garbage collection.

trace n
- where n is an integer

e) Reading commands from files

get <filename>
- treat the named MTS file as subsequent source input up until its EOF. The file may contain any number of commands but get's may be nested only nine deep.

echo (noecho)
- enable (inhibit) the echoing of subsequent input from MTS files. These two commands are most often placed within MTS files.

f) $<MTS command>
- execute MTS command (like ! in UNIX ed)

g) off
- "log off" from SASL

4) Error Messages

The system deals with expressions in two stages, so there are two kinds of error message.

a) Compiler messages

These are normally output together with a copy of the offending line in the source text and are self-explanatory. Eg

```
... (2 + *) ...

        Syntax : expression expected where * found.
```

After an error the compiler recovers and continues through the source looking for further errors.

b) Evaluator Messages

These arise when the value of a meaningless expression like 2+false has to be printed. Again the messages are normally self-explanatory,

example

```
map double (1, 2, 3, 4, 5, false, 7, 8, 9)
where
double x = 2 * x, sp
```

produces the output

```
2 4 6 8 10
Illegal Expression: 2 * false
 14 16 18
```

The form of the illegal expression together with its relative position in the output normally suffice to locate the error in the program. In desperation you can resort to the trace n command for tracing the evaluation after n cycles. There is one case where this is obviously helpful - if your program is sitting in a tight loop and not printing anything. See 3d above.

(Feedback on ways to improve the error reports is especially welcome. In particular if the SASL system says something that you think is just wrong or downright misleading please tell us about it - DAT, WRC, HDA.)