S A S L

L A N G U A G E   M A N U A L

D A Turner

1 9 7 6

(Revised August 1979 for "Combinators" Version)

(Revised November 1983 for Inclusion of "ZF" Expressions)

C O N T E N T S

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 programs contain no commands and a data structure, once defined, cannot be altered by the subsequent execution of the program. For a fuller discussion of the advantages of this kind of language and its relationship to other languages the reader is referred elsewhere ([1], [2], [3]).

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 IX where the syntax of SASL is summarised. The first time reader is advised to read sections I to VIII in order at least once and then study the definitions of the standard functions in section X before attempting some of the exercises in Appendix I.

In SASL a ‘‘program’’ is an **expression** describing some data object. If an expression is presented to the SASL system (followed by a question mark), the response of the system is to print a representation of the object described by the expression.

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 ${}^{10}C_3 + {}^{10}C_4$

Instead of associating definitions directly with an expression by means of a **where**, it is often more convenient to add them to the environment in which all expressions are evaluated. This is done by means of a declaration of the form **def** definitions (also terminated by a question mark). So for example the effect of

```
def
fac 0 = 1
fac n = n * fac(n–1)
?
```

is to add the definition of the function ‘‘fac’’ to the global environment.

There is quite a rich environment of standard functions etc, which is set up automatically at the beginning of every SASL session by means of a set of definitions known as the SASL prelude. The SASL prelude is listed in section X of this manual.

History and Acknowledgements

The SASL language was developed at the University of St Andrews as a vehicle for teaching functional programming, reaching (more or less) its present form in December 1976.

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

Additional Remarks

Apart from minor tidying up and the correction of some errors, this edition of the SASL manual contains two main changes since the version dated December 1976. These are (i) the addition of floating point numbers, originally incorporated in the version of the manual issued at the University of Kent in August 1979, and (ii) the addition of ZF expressions and some associated operators, which are a more or less direct borrowing from the later language KRC (see the first chapter in [3]).

I would like to extend the list of thanks above to include Bill Campbell, who made several important contributions to the development of the various SASL implementations, Peter Welch, John Hammond and Silvio Meira, all of whom made valuable inputs to the development of the version of SASL described here, and Judith Farmer for typing this revised version of the manual.

D A Turner, November 1983

References

[1] P J Landin ‘‘The Next 700 Programming Languages’’ CACM Vol 9, No 3 (March 1966)

[2] W H Burge ‘‘Recursive Programming Techniques’’ Addison-Wesley 1975

[3] (eds) Darlington, Henderson, Turner ‘‘Functional Programming and its Applications’’ Cambridge University Press, 1982.

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 — these include the integers, positive, negative and zero

e.g. 0 13 –6 128

and also floating point numbers

e.g. 3.14159 1.12e–8

(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

e.g. %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.

e.g. 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 deemed to be the value of all ill formed or non-terminating expressions such as

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**.

Each SASL object falls into one of the above 6 categories. There are five types testing functions:

number, logical, char, list, function

and for any object which is not undefined, one of the above functions when applied to the object will return the value **true**, and the other four will return the value **false**. There is no test for ‘‘undefined’’ however. The result of applying a type testing operator to **undefined** is itself **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 V) 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. For example given 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 e.g.

```
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 ten special symbols called **delimiters**:

where = ( ) , → ; ← { }

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

x xl fac a_rather_long_name

Upper and lower case letters are distinct, so x and X are different names.

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

A basic symbol can consist of more than one character but is regarded as a single textual entity. For a list of ‘reserved words’, i.e. 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 ...?’’ e.g. ‘‘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 says that the answer to such questions in the case of SASL is **always yes**. If the syntax (summarised in section IX) 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 is section IX together with their binding power and include the arithmetic operators + – * / **div rem** (**div** 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) — dot is the most binding infix operator. In addition there are some special operators on lists including : and ++ and ––, which are discussed later. With the exception of these last three, all infix 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 — e.g. 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 alway 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 are 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 the = 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 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
```

L 1 is 1

L 2 is 2, 3, 4

L 3 is 5

L 4 is **undefined**

L 2 2 is 3   (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. We use single quote to open a string and double quote to close it. 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 $\rightarrow$ object1 ; object2

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

$x < 0 \rightarrow -x ; x$

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 one or more 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 in

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

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 X). By means of messages of the form ‘‘**def** definitions ?’’ the user can extend the predefined environment (see Appendix III).

When several clauses are used to define a single function the clauses must be grouped together; furthermore their order is significant in that when attempting to match actual parameters against formal parameters the system tries the different clauses in the order they are given.

The order of definitions of distinct objects in a **where**-expression is of no significance, however. There is for example, no requirement that names be defined before they are used, so mutual recursion is freely permitted.

V DEFINITIONS

A definition is the syntactic entity which associates one or more names with a value. In SASL definitions can occur in two contexts — following a **where**, in which case the definitions are local to one expression, or following a **def** in which case they remain in force for the whole of the rest of the session. The rules for writing definitions are exactly the same in both contexts. In both cases any number of definitions may be written one after another; each definition is either a structure definition, consisting of a single clause, or a function-form definition, which can consist of one or more clauses. In all cases the clauses are separated by semicolons or (equivalently) newlines.

Structure 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 = 1 < 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) l, 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 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 by the previous two clauses. (The order in which the clauses are written is relevant.)

(4)
```
sum () = 0
sum (a : x) = a + sum x
```

defines a function for summing the elements of a list.

Section X, 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; e.g.

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

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

```
x = 1 : x
```

makes x the infinite list l, l, l, ...

(Any infinite list or tree 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 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, including a decimal point and a scale factor if necessary and 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 some internal representation of the function enclosed 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 spaces, lay, layn, ljustify, rjustify and cjustify (see section X) are useful here.

example the following program prints a table of factorials

```
title, for 1 7 line
where
title = ‘A TABLE OF FACTORIALS’’ , n1
line n = ‘factorial ’’, n, ‘ is ’’, fac n, n1
fac 0 = 1
fac n = n * fact (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 X) 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).

VII FLOATING POINT NUMBERS

SASL originally supported only integer numbers but was subsequently extended to support floating point numbers as well. In fact all numbers are now stored internally in floating point form and when a number is output if it happens to be integer valued it is printed without the decimal point and trailing zeros. The associated changes to the syntax of the language are as follows.

(1) The definition of <numeral> has been extended to permit decimal points and scale factors, for example

```
2.67    le8    8.72e–13
```

are all valid numbers. Note that when a scale factor is used there must be at least one digit preceding the ‘‘e’’. Note also that the use of ‘‘.’’ inside numerals does not prevent it from continuing to be used elsewhere to mean functional composition.

(2) The following infix operators should be specially noted

>> <<	(relational operators, same binding power as ‘‘=’’)
/	(floating point division, same binding power as **div**)
**	(exponentation, same binding power as ‘‘.’’ functional composition)

Notes

(a) ‘‘>>’’ pronounced ‘‘much greater than’’ is defined so that a >> b is true if and only if we have a + b = a within the accuracy of the machine’s arithmetic. Together with the converse relation ‘‘<<’’ (‘‘much less than’’) it is used in testing for convergence.

(b) Since **div** is now provided for doing integer division the operator ‘‘/’’ has been taken over to mean floating point division.

(c) The exponentiation operator ‘‘**’’ is right associative, as in most languages.

(3) The predefined environment of standard functions now includes the following additional functions

```
arctan   cos   entier   exp   log   sin   sqrt
```

Also predefined are ‘‘e’’ and ‘‘pi’’ with their usual values. (In fact the predefined environment has been greatly extended to include many functions in addition to those in the original prelude. The new prelude is listed in section X.)

VIII ZF EXPRESSIONS

These were not originally in SASL but have been borrowed from the later language KRC (see ‘‘Recursion Equations as a Programming Language’’ in reference [3] of the introduction) because they are so convenient. We often wish to create a list by iterating over all the members of one or more existing lists — ZF expressions provide a way to express this without constructing an explicit recursion. The basic idea comes from Zermelo-Frankel set theory. The simplest form of ZF expression in SASL is

```
{ e1 ; n ← e2 }
```

where e1 and e2 are arbitrary expressions and n is a name. This is pronounced ‘‘list of all e1 such that n drawn from e2’’. The value of e2 should be a list. The name n is a local variable of the construction whose scope is delimited by the braces. The variable binding construct **name** ← **list** is called a ‘‘generator’’ — the name assumes in turn the value of each member of the list. An example is

```
{ a*a ; a ← x }
```

which yields a list of the squares of all the elements of x (in the same order that they occur in x). Note that the result is a list and not a set — repetitions are not eliminated.

The power of the notation arises from the fact that one is allowed more than one generator. The general form of a ZF expression in SASL is

```
{ exp ; qualifiers }
```

where there can be any number of qualifiers (separated by semicolons if there is more than one). Each qualifier is either a generator as in our first example, or else a ‘‘filter’’. A filter is an arbitrary boolean expression used to further restrict the range of a generator.

Examples

```
{ a,b; a ← x; b ← y }
```

is the Cartesian product of the lists x and y i.e. the list of all pairs of the form a,b where a is drawn from x and b is drawn from y (the order of occurrence is like that of nested ‘‘for-loops’’ with b varying more rapidly than a) and

```
{ n*n; n ← z; n > 0 }
```

is a list of the squares of all the positive members of z (still in the order in which they occurred in z). Note that semicolons after the first should be pronounced ‘‘and’’ (the first being pronounced ‘‘such that’’) and that each filter should occur somewhere to the right of the generator to which it refers. A ZF expression must of course contain at least one generator, but there is no upper bound on the number of generators.

Two abbreviations are permitted in the writing of ZF expressions. First, if two or more generators have the same right hand side, this can be abbreviated in the following style

```
i ← j ← k ← exp
```

is shorthand for i ← exp; j ← exp; k ← exp
and second, if the body is the same as the left hand side of the first generator it can be omitted, thus

```
{ a ← x; f a }
```

is short for { a; a ← x; f a }

The following additional operators have been introduced along with ZF expressions

––	(infix — same binding power as ‘‘:’’ and ‘‘++’’)
..	(infix) (binding power higher than ‘‘++’’ etc but below arithmetic operators)
...	(postfix) (binding power same as ..)
#	(prefix — more binding power than anything except function application)

Explanation ‘‘––’’ is list difference e.g.

    (1,2,3,4,5)––(1,4)

have the value (2,3,5). For exact definition see prelude function ‘‘list_diff’’. The operators ‘‘..’’ and ‘‘...’’ are used to produce subranges of the integers e.g.

    2..8 and 1...

have for their values (2,3,4,5,6,7,8) and the list of all positive numbers respectively. The operator ‘‘#’’ takes the length of lists.

Here are a few more examples of ZF expressions:

The function which takes a number and returns a list of its factors (including one but excluding itself) can be written

factors n = { a ← 1.. n/2; n **rem** a = 0 }

If we define a perfect number as one which is equal to the sum of its factors (for example 6 = 3 + 2 + 1 is perfect) we can write the list of all perfect numbers as

```
perfects = { n ← 1... ; n = sum(factors n) }
```

Another example is provided by the notion of the ‘‘partitions’’ of a number. By a partition we shall mean a list of positive integers adding up to the given number. For example the number 3 has 4 partitions, namely (1,1,1), (1,2), (2,1) and (3,); note that we take the order of the elements in a partition to be significant. A function for returning all the partitions of a number can be written

```
partitions 0 = (),
partitions n = { i:p; i ← 1..n; p ← partitions (n–i) }
```

Notice that generators come into scope in the order in which they are written, so the generator for ‘‘p’’ can involve ‘‘i’’, which was defined earlier.

Some more examples of ZF expressions will be found in appendix II (solutions to exercises).

IX SUMMARY OF SYNTAX ETC

```
<program> ::= <expr>? | def <defs>?
<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> ) | <zfexpr>

<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>|<charconst>|<boolconst> | () |<string>
<numeral> ::= <real><scale-factor> | —<real><scale-factor>
<real> ::= <digit>* | <digit>*.<digit>*          Note * means 1 or more
<scale-factor> ::= e<digit>* | e–digit>*
<boolconst> ::= true | false
<charconst> ::= %<any char> | sp | n1 | np | tab
<string> ::= ‘<any message not containing unmatched quotes>’’

<zfexpr> ::= {<expr>; <qualifiers>} | {<generator>; <qualifiers>}
<qualifiers> ::= <qualifier> | <qualifiers>; <qualifier>
<qualifier> ::= <generator> | <filter>
<generator> ::= <name>←<expr> | <name>←<generator>
<filter> ::= <expr>
```

Operators (in order of increasing binding power)

```
                : ++  ––                       infix (right associative)
                  ..                           infix (non-associative)
                  ...                          postfix
                  |                            infix
                  &                            infix
                  ~                            prefix
>>  >  >=  =  ~=  <=  <  <<                    infix
                  + –                          infix
                  + –                          prefix
             * /  div  rem                     infix
                  **                           infix (right associative)
                  #                            prefix
```

Predefined Names (alphabetical order, see ‘‘prelude’’ for their definitions)

abs all and append arctan

code cons converse char cjustify concat cost count
decode digit digitval drop
e entier eq exp
filter foldl foldr for from function
hd
I interleave intersection iterate
K
lay layn length letter list listdiff ljust log logical
map member mkset
not number
or
pi plus printwidth product
reverse rjustify
show sin some spaces sqrt sum
take times tl
union until
while
zip

X THE SASL PRELUDE

In this section we first give a brief explanation of the functions etc, defined in the prelude (in alphabetical order). This is followed by the source text of the prelude, for reference purposes.

abs
takes the absolute value of a number — e.g. abs(–3) is 3.

all
applied to a list of truthvalues, **all** returns their logical conjunction. See also ‘‘some’’.

and
another name for the boolean operator ‘‘&’’.

append
another name for the list concatenation operator ‘‘++’’.

arctan
inverse trigonometric function.

code
code is applied to a character and returns an integer representing its value in the ascii character sequence.

cons
another name for the list prefix operator ‘‘:’’.

char
type testing function — applied to any object it returns TRUE if it is a character and FALSE otherwise. See also ‘‘function’’, ‘‘list’’ ‘‘logical’’, ‘‘number’’.

cjustify
a formatting function — ‘‘cjustify n x’’ centre justifies x in a field of width n, where x can be any printable object. See also ‘‘rjustify’’, ‘‘ljustify’’.

concat
when applied to a list of lists, concat joins them all together with ‘‘++’’.

cos
trigonometric function.

count
another name for the ‘‘..’’ operator. For example ‘‘count 1 10’’ is a list of the numbers from 1 to 10 inclusive. This can also be written ‘‘1..10’’.

decode
applied to an integer between 0 and 127, ‘‘decode’’ returns the corresponding character from the ascii character set.

digit
a testing function which can be applied to characters, and returns TRUE or FALSE. See also ‘‘letter’’.

digitval
converts a digit character to the corresponding integer — e.g. digitval %3 is 3.

drop
drops a specified number of elements from the front of a list — i.e. ‘‘drop n x’’ returns x with the first n elements removed. If x has length less than n drop returns (). See also ‘‘take’’.

e
a real number — the base of natural logorithms.

entier
applied to a number returns its integer part — i.e. the largest integer not exceeding it.

eq
another name for the relational operator ‘‘=’’.

exp
exponential function on reals — i.e. ‘‘e’’ to the power of the argument.

filter
selects from a list only those elements which pass a given test — e.g. ‘‘filter number x’’ returns a list containing only those elements of x which are numbers.

foldl
folds up a list, using a given binary operator and given start value, in a left associative way. Best explained by means of an example:

```
foldl f r (a, b, c) = f c (f b (f a r))
```

for another example see the definition of ‘‘reverse’’. See also ‘‘foldr’’.

foldr
folds up a list, using a given binary operator and a given start value, in a right associative way. For example:

```
foldr f r (a, b, c) = f a (f b (f c r))
```

see also the definitions of ‘‘all’’, ‘‘concat’’, ‘‘product’’, ‘‘some’’, ‘‘sum’’ for examples of its use.

for
‘‘for a b f’’ means the same as ‘‘map f(a..b)’’.

from
another name for the ‘‘...’’ operator. For example ‘‘from 1’’ is a list of all the natural numbers starting at 1. This can also be written ‘‘1...’’.

function
type tester — TRUE on functions, FALSE otherwise.

hd
applied to a non empty list, ‘‘hd’’ returns the first element. It is an **error** to apply ‘‘hd’’ to the empty list, (). See also ‘‘tl’’.

I
the identity function — returns its argument as result.

interleave
applied to a list of lists, merges them into a single list by taking elements from each in rotation.

intersection
given two lists x and y, ‘‘intersection x y’’ is the set of elements common to both. (**Note** here and elsewhere in this section, the word ‘‘set’’ is used to refer to a list from which duplicates have been removed; sets do not exist as a separate data type in SASL.) See also ‘‘member’’, ‘‘union’’ and ‘‘listdiff’’.

iterate
‘‘iterate f x ’’ returns the infinite list ‘‘x, f x, f(f x), ...’’

K
a combinator — see its definition.

lay
a formatting function — applied to a list of printable objects, applies ‘‘show’’ to each one and inserts newlines between them.

layn
similar to ‘‘lay’’ but prints with numbered lines.

length
another name for the ‘‘#’’ operator on lists — length x returns an integer, which is the number of elements in list x. Undefined on infinite lists.

letter
testing function on characters — TRUE if argument is an upper or lower case letter, FALSE otherwise.

list
type tester — TRUE on lists, FALSE otherwise.

listdiff
another name for the ‘‘−−’’ operator on list. Given lists x and y, ‘‘listdiff x y’’ (also written ‘‘x−−y’’) returns a list of the elements of x but with any elements which also occur in y removed. Useful when using lists to represent sets, as it corresponds to set difference.

ljustify
formatting function — ‘‘ljustify n x ’’ left justifies any printable object x in field of total width n.

log
takes the natural logorithm of a real number.

logical
type tester — TRUE on truthvalues, FALSE otherwise.

map
given a function f and a list x, ‘‘map f x’’ applies f to every element of x.

member
‘‘member x a’’ returns TRUE if a occurs in the list x, FALSE otherwise.

mkset
removes duplicates from a list, so each element occurs once only in the result.

not
another name for the boolean operator ‘‘~’’.

number
type tester — TRUE on numbers, FALSE otherwise.

or
another name for the boolean operator ‘‘|’’.

pi
a real number — as in circles.

plus
another name for the arithmetic operator ‘‘+’’.

printwidth
for any printable object x, ‘‘printwidth x’’ is the width of the field which x will occupy when printed in the normal way.

product
takes the product of a list of numbers. See also ‘‘sum’’.

reverse
given any finite list x, ‘‘reverse x’’ is a list of the same elements in reverse order.

rjustify
formatting function — ‘‘rjustify n x’’ right justifies any printable object x in a field of width n.

show
list structures are normally flattened on printing, suppressing any information about their internal structure — ‘‘show’’ inserts extra characters (parentheses and commas) into the structure so that when it is printed its original structure will be visible.

sin
trigonometric function.

some
applied to a list of truthvalues, returns their logical disjunction.

spaces
‘‘spaces n’’ is a list of n spaces.

sqrt
arithmetic function.

sum
takes the sum of a list of numbers.

take
“take n x” is a list of the first n elements of list x. If x has length less than n, the result is just x.

times
another name for the arithmetic operator “*”.

tl
applied to a non empty list x, “tl x” is the rest of x after removal of the first element. It is an error to apply “tl” to the empty list, ().

union
given lists x and y, “union x y” is a set containing the elements of both (once each).

until
“until f g x” is the result of applying g to x repeatedly until f is TRUE on the result.

while
“while f g x” is the result of apply g to x repeatedly until f is FALSE on the result.

zip
applied to a list of lists, transposes it (in the sense of matrix transpose).

A Listing of the Prelude

```
DEF     ||THE SASL PRELUDE    DAT  August 1979
        ||last revised July 1984

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

  all = foldr and TRUE

  and x y = x&y

  append x y = x++y

||arctan      inverse trig function - primitive to sasl

||code ch = integer code for ch in local character set - primitive to sasl

  cons a x = a:x

||char              type testing function - primitive to sasl

  cjustify fieldwidth x
    =   spaces lmargin,x,spaces rmargin
        WHERE
        margin = fieldwidth - printwidth x
        lmargin = margin DIV 2
        rmargin = margin - lmargin

  concat = foldr append ()

||cos        trig function - primitive to sasl

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

```
||decode n          the character whose integer code is n - primitive to sasl

  digit x = code x>= code %0 & code x <= code %9

  digitval x = code x - code %0

  drop 0 x = x
  drop n() = ()
  drop n (a:x) = drop(n-1)x

  e = exp 1

||entier      primitive to sasl

  eq x y = x=y

||exp         exponential function - primitive to sasl

  filter f () = ()
  filter f(a:x) = f a -> a:filter f x ; filter f x

  foldl op r () = r
  foldl op r(a:x) = foldl op(op a r)x

  foldr op r
    =     f
          WHERE
          f () = r
          f(a:x) = op a(f x)

  for a b f = a>b -> () ; f a: for(a+1)b f

  from n= n:from(n+1)

||functiontype testing function - primitive to sasl

  hd(a:x)= a

  I x = x

  interleave = concat.zip

  intersection x y = filter(member y)x

  iterate f x = x:iterate f(f x)

  K x y = x

  lay () = ()
  lay (a:x) = show a:NL:lay x

  layn x  =       f 1 x
                  WHERE
                  f n () = ()
```

```
                      f n(a:x) = n:’) ":show a:NL:f(n+1) x

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

  letter x  =         code x >= code %a & code x <= code %z |
                      code x >= code %A & code x <= code %Z

||list      type testing function - primitive to sasl

  listdiff () y = ()   || listdiff defines the action of the "--" operator
  listdiff x () = x
  listdiff(a:x)(b:y)  =          a=b -> listdiff x y
                                 listdiff(a:listdiff x(b,))y

  ljustify fieldwidth x
   =  x ,spaces(fieldwidth - printwidth x)

||log        natural logarithm - primitive to sasl

||logical   type testing function - primitive to sasl

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

  member x a = some(map(eq a)x)

  mkset () = ()
  mkset (a:x) = a:filter(not.eq a)(mkset x)

  not x = ˜x

||number type testing function - primitive to sasl

  or x y = x | y

  pi = 4*arctan 1

  plus x y = x+y

  printwidth () = 0
  printwidth(a:x) = printwidth a + printwidth x
  printwidth TRUE = 4
  printwidth FALSE = 5
  printwidth x
    =       number x -> nwidth x; 1
            WHERE
            nwidth x
              =       x<0 -> nwidth(-x)+1
                      abs(x-entier(x+0.5))<2e-6 & abs x<=1e7 -> intwidth x
                      .001<=x & x<=1000-> intwidth x + 7
                      12
            intwidth x
              =       x<10-> 1
```

```
                        1 + intwidth(x DIV 10)

  product = foldr times 1

  reverse = foldl cons ()

  rjustify fieldwidth x = spaces(fieldwidth - printwidth x), x

  show x   =            x=() | ˜list x | haschars 6 x -> show1 x
                        show1(hd x):(tl x = ()-> %,; showlis(tl x))
                        WHERE
                        show1 () = ’()"
                        show1 NL = ’NL"
                        show1 NP = ’NP"
                        show1 TAB = ’TAB"
                        show1 x
                           =         char x -> %%,x
                                     list x
                                     ->          haschars 6 x -> %’:showstr x
                                                 %(:show1(hd x):(tl x=()->%,; showlis(tl x)):’)"
                                     x
                        haschars 0 x = TRUE
                        haschars n () = TRUE
                        haschars n(a:x) = char a & haschars (n-1) x
                        showlis () = ()
                        showlis(a:x) = %,:show1 a:showlis x
                        showlis x = ’)++":show1 x
                        showstr ()= %"
                        showstr x   =            char(hd x)-> hd x:showstr(tl x)
                                                 %":’++":show1 x

||sin       trig function - primitive to sasl

  some = foldr or FALSE

  spaces = map(K % ).count 1

||sqrt       primitive to sasl

  sum = foldr plus 0

  take 0 x = ()
  take n () = ()
  take n (a:x) = a:take(n-1)x

  times x y = x*y

  tl(a:x) = x

  union x y = filter(not.member y)x ++ y

  until f g x = while(not.f)g x

  while f g x
```

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

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


?   || END OF SASL PRELUDE
```

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 difficulty 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, ... || each number is the sum of the previous two
```

(2) Define a function ‘‘sort’’ which takes a list of number and sorts it into ascending order.

(3) Write a program (i.e. an expression followed by ?) to print the moves for Towers of Hanoi (ask someone if you do not know the rules) with 8 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 50 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. Define a more general functional, CURRY, such that if f is a function that expects an n-list then

```
CURRY n f
```

will be the corresponding curried 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 (self describing) 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 auxiliary function f
DEF
fib = f 1 1
f a b = a : f b (a + b)
?
```

(b)
```
|| by immediate recursion
DEF
fib = 1 : 1 : map sum (zip (fib, tl fib))
?
```

(2) Sorting

(a)
```
|| insertion sort (simple but inefficient

DEF
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

DEF  || this rather nice solution is due to Silvio Meira
sort () = ()
sort (a : x) = sort {b ← x; b ≤ a } ++ a : sort { b ← x; b>a}
?
```

(c)
```
|| treesort

DEF
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 8 '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 50 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

DEF
perms() = (),
perms x = {a : p; a ← x; p ← perms(x−−(a,))}
?
```

(6) Taut

(a)

```
|| This would be a much harder problem if f wasn’t
|| curried

DEF
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

DEF
Taut t =   logical f → f
           Taut (f TRUE) & Taut (f FALSE)
?
```

(7) CURRY

```
|| This is a bit subtle, although it’s only 3 lines long

DEF
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 (2...)
sieve (p : x ) = p : sieve {a ← x; a REM p > 0}
?
```

(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 REM 100) DIV 10 = 1 → ‘th’’  || because of 11,12,13
            n REM 10 > 3 | n REM 10 = 0 → ‘th’’
            ord (n REM 10)
?
```

(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 = #b = 8
alter(8 : b) = alter b  || backtrack
alter(q : b) = q + 1 : b
safe(q : b) =     all (for 1 (#b) nocheck)
                  WHERE
                  nocheck i = q ~= b i & abs(q – b i) ~= i
?

|| another method
DEF
displ b = zip (‘rnbqkbnr’’, b, spaces ()
queens 0 = ()
queens n = {q : b; q ← 1..8; b ← queen(n–1); safe q b}
safe q b = all {~checks q b i; q ← 1..#b}
checks q b i = q = b i | abs (q – b i) = i
?

displ(hd(queens 8))?  || hd because we only want one solution
```

APPENDIX III RUNNING SASL UNDER UNIX

(1) Lexical Conventions

(a) Reserved Words

The underlined words of the SASL language, namely

where def div rem true false nl np sp tab

are actually typed without underlining but in capital letters (to distinguish them from identifiers, which are normally in lower case). Thus

```
WHERE   DEF   DIV   REM   etc
```

For consistency SASL system command words and system directives (see sections 3, 4 below) are also all in upper case e.g. TO GET etc. (Actually it is permitted to use capital letters in identifiers but identifiers consisting entirely of capital letters are perhaps best avoided.) For reference a complete list of reserved words follows:

```
COUNT   DEF   DIV   FALSE   GC   GET
HELP   INTERACTIVE   NL   NP   OBJECT
OFF   READ   REM   RESET   SP   TAB
TO   TRUE   WHERE   WRITE
```

(b) String Quotes

SASL expects opening and closing quotes to be distinct (so that string quotes can be nested). We accomplish this by using single quote (apostrophe) to open strings and the double quote symbol to close them. Thus

‘this is a string constant’’

(c) Offside rule

The offside rule is enforced in the following form (which is slightly weaker than the one described in section III but which has been found more convenient in practice).

(i) In a definition every part of the expression following the ‘‘=’’ must be to the right of the column containing the ‘‘=’’

(ii) In a conditional expression every part of the expression following the ‘‘→’’ must be to the right of the column containing the ‘‘→’’

If the user is in doubt about what this implies he should study the prelude (section X of this manual) and the solutions to exercises in appendix II for some examples of correct SASL layout. One minor point about layout that should be noted is that the semicolons which separate the qualifiers in a ZF expression are special and **CANNOT** be omitted, even when they coincide with the end of line.

(2) Entering the SASL System

The SASL system that runs under UNIX is interactive. It accepts a series of ‘‘SASL system commands’’ from the user at the terminal, executing each one before asking the user for the next one.

To ‘‘login’’ to the SASL system type the UNIX shell command

```
sasl
```

There will be a pause while various bits and pieces are loaded, then the message

```
hello from sasl
```

will appear, indicating that the system is now ready to accept a command from the terminal. After each command has been processed the prompt

```
what now?
```

is given, requesting the next comment from the user.

The shell command ‘‘sasl’’ can also be supplied with the name of one or more files as arguments. These files will contain SASL commands that the user wishes to be processed before the hello message is given (typically these will be DEF commands). The file called ‘‘prelude’’ is included automatically without needing to be named. (The prelude is read before any other files which the user may supply as arguments.) To prevent this automatic inclusion of the prelude the user can call sasl with the flag ‘‘-p’’ as its first argument.

(3) SASL System Commands

The SASL system accepts 7 different forms of command

(a) <expr> ?

The system evaluates the SASL expression and prints its value in the usual way. (Don’t forget the question mark or nothing will happen.)

(b) DEF <defs> ?

The environment of predefined names in which subsequent expressions are evaluated is extended by the definitions given. (The file called ‘‘prelude’’ contains a command of this form.) If a name which already has a meaning is redefined then the new definition over-rides the old but a warning message is given.

Note that if DEF commands are entered on-line, the associated definitions will be lost at the end of the session (because they are compiled as they are entered and the source text is not saved). It is therefore normal practice, for other than very temporary definitions, to place your DEF commands in a file and then include them, either at the beginning of the session by making the file a parameter of the SASL program, or else by means of GET directive (see below) during the session.

(c)

<expr> TO <file>

Where <file> is specified by any UNIX pathname. Sets up a UNIX process to evaluate <expr> and write its value to <file>. The file will be created if it does not exist, if it does exist its previous contents will be overwritten. The SASL system does not keep the user waiting for the result of this command but simply gives the UNIX id-number of the process created and then immediately asks for the next command.

An example — after giving the definition

```
DEF
fac 0 = 1
fac n = n * fac(n–1)
?
```

the effect of the command

```
layn(for 1 50 fac)  TO  facout
```

will be to set up a process which leaves a table of factorials in the file called ‘‘facout’’. (The function ‘‘layn’’ is convenient for tabulating output — see the prelude for its definition.)

(d) <expr> INTERACTIVE

The value of <expr> should be a SASL function which expects for its input a list (potentially infinite) of characters. This function is then applied to a list composed of all the characters subsequently typed by the user at the terminal (up to but not including the next EOT character). The output of the function is printed at the terminal in the usual way. This enables the user to run interactive programs written in SASL such as desk calculations or games playing programs.

For example if we enter the following definitions

```
DEF
f x = ‘give me a number to be doubled’’:g x
g (SP:x)=g x
g(NL:x)= g x
g(a:x)= 2*digitval a:NL:f x
g()= ‘goodbye from doubler’’,NL
?
```

then the subsequent command

```
f INTERACTIVE
```

sets up an interactive program for doubling single-digit numbers supplied by the user at the terminal. It throws away spaces and newlines but assumes that any other character typed by the user is a number to be doubled — it gives the answer and then prompts the user for the next number.

(This process continues until the user types the end-of-transmission character. At this point the list of characters being input to the function will terminate, the ‘‘goodbye from doubler’’ message will be given and any subsequent characters typed by the user are taken as part of the next SASL system command.)

(e) !<shell command>

Enables any UNIX command to be executed from within the SASL system (exactly as in the UNIX editor).

(f) OFF

‘‘log off’’ from sasl. Returns you to the shell. Note that if the end-of-transmission character (control-D) is pressed when a command is expected this has the same effect as OFF.

The interrupt character (which is control-C on most UNIX systems) may be entered during the course of an evaluation and causes the evaluation to be abandoned, so the system then returns to a state in which it is ready to accept the next command.

If an interrupt is given outside an evaluation, i.e. when the system is expecting a command, the effect is to force a ‘‘SASL restart’’, which has the following effect. The state of the system is restored to what it was immediately before the hello message and the hello message is given afresh. Thus any definitions included initially will be preserved but those entered subsequently from the terminal will have been lost.

(g) HELP

Prints a summary of the available commands.

(4) System Directives

Certain messages, called system directives, may be included anywhere in the input to the SASL system (even in the middle of a SASL expression) and have effect immediately they are read.

```
GET <file>
```

causes the contents of <file> to be read as if they had been physically present in the input stream at this point in the place of the GET directive. (The included file may itself contain GET directives ... and so on to any depth.)

```
GC
```

causes store utilisation messages to be output during each subsequent call of the SASL ‘‘garbage collector’’.

```
OBJECT
```

causes the object code (consisting of combinators) to be displayed after each subsequent compilation of an expression or definition.

```
COUNT
```

causes statistics to be output after each subsequent evaluation of an expression. (In the case of evaluations set up ‘‘in background’’ by means of a TO command, COUNT is invoked automatically.)

```
RESET
```

cancels the effect of any previous GC, OBJECT or COUNT commands.

(5) **Multi-File I/O in SASL**

In order to take advantage of the UNIX file-handling environment the SASL language has been extended by the addition of the following expressions.

```
READ <file>
WRITE <file>
```

where <file> is an arbitrary UNIX pathname. Syntactically each of the above is a <simple> i.e. can occur in place of a name or constant without needing to be bracketed. The meanings are as follows.

The expression ‘‘READ <file>’’ returns for its value the contents of the file as a SASL string (i.e. a list of characters, the length of the list being the number of characters in the file). This string can then be manipulated like any other SASL data object. Note that there is no end-of-file character at the end of the string, the end of the file is simply indicated by the end of the string.

The expression ‘‘WRITE <file>’’ returns for its value a function which when applied to any SASL data object creates a special version of the object that has the property that when it comes to be printed it will be sent to <file> instead of the standard output stream. The file will be created if it does not already exist. Writing to a file in this way appends characters to the end of its existing contents.

As an example of the use of these facilities, it we enter the following SASL expression to be evaluated

```
f x, g x, ‘finished’’
WHERE
x = READ input
f = WRITE out1
g = WRITE out2
?
```

the effect will be that copies of the contents of the file ‘‘input’’ will be sent to both of the files ‘‘out1’’ and ‘‘out2’’ and that when this has been done the message

```
finished
```

will appear at the user’s terminal.

Note

The SASL system described here operates by translating SASL expressions to a form based on combinatory logic and then performing normal graph reduction on the combinators (*). This is an unconventional implementation method which offers significant performance advantages over traditional implementation techniques for applicative languages. It is still under development, however, and one as yet unresolved drawback is that the quality of run time error messages is very poor. The compile time messages (for syntax errors) are also rather unhelpful, but this is due to difficulties with YACC (the UNIX compiler-compiler). There are plans to fix both of these problems in a later release of the system. Bug reports should be sent to the author at the University of Kent.

D A Turner
Computer Laboratory

(*) See: ‘‘A new implementation technique for applicative languages’’
Software Practice & Experience, Jan 1979.
A more detailed account of the combinators implementation of SASL can be found in the author’s D.Phil thesis (‘‘Aspects of the implementation of programming languages’’, Oxford University, February 1981).

University of Kent
Canterbury

August 1979, November 1983