An Overview of ARC SASL

H. Richards
Burroughs Corporation
Austin Research Center
12201 Technology Blvd.
Austin, Texas 78759

The ARC SASL system consists of two parts: a functional programming language--SASL--based on higher-order recursion equations, and surrounding this (but not described here) an interactive program development system (ARCSYS) which responds to various commands for preparing, editing, and executing SASL programs.

The programming language is purely applicative, and contains no imperatives--for example, assignment operations--of any kind. A SASL program is an expression; executing a program amounts to evaluating the expression and displaying the result. For example, a simple program might be

```
2 + 7
```

to which ARCSYS responds by displaying "9". A slightly more elaborate example is

```
x / y
WHERE x = a + b
      y = a - b
      a = 10
      b = 5
```

ARCSYS evaluates an expression preceding a "WHERE" in the environment established by the definitions following the "WHERE"; in this example, the result is "3". A list of definitions can be regarded as a set of equations, whose solution establishes the values of the names on the left of the "=" signs. WHERE expressions may be nested, in which case ALGOL60-like static scope rules apply. Within each WHERE-clause, mutual recursion is permitted; consequently, the order in which definitions are written is not significant.

SASL's syntax has a two-dimensional aspect embodied in its so-called offside rule:

> None of an expression's tokens can lie to the left of its first token.

This rule leads to a welcome uniformity in indentation style, and allows certain explicit delimiters to be omitted in favor of line breaks. The example above could be written all on one line, the definitions being separated by semicolons:

```
x / y WHERE x = a + b; y = a - b; a = 10; b = 5
```

As originally written, the expression "a + b" is terminated by the appearance of "y" to its left on the next line.

There are five types of objects in SASL's universe of discourse--numbers (as above), characters, booleans, lists, and functions. The equality operator, denoted by "=", is defined over objects of all types. Two objects of different types are always unequal, and the equality of two functions is incomputable (because a function is an infinite mapping).

Literal numbers are written as conventional decimal numerals with optional minus sign, decimal point, and exponent; the usual arithmetic and numeric-comparison operators are defined on them. A literal character is denoted by preceding it with "%". The boolean literals are reserved words--TRUE and FALSE.

Enumerated lists are enclosed in brackets with commas separating their components, as in

```
[1,2,3,4,5]
```

Singleton lists require no commas, e.g., "[3]", and the empty list is denoted by "[]". Quoted strings are a shorthand for literal lists whose components are all characters; for example,

```
[%A,%u,%s,%t,%i,%n]
```

may be written

```
"Austin"
```

List components may be of any type, allowing the representation of matrices, for example, by lists of lists. The types of a list's components may be mixed, allowing the representation of general graph structures.

A list is indexed by applying it--like a function--to an integer. The other basic operations on lists are "#" (which gives the length), ":" (which adds a component to the front), "++" (concatenation) and "--" (list difference). So given, for example, the definitions

```
x = [1,2,3,4,5]
y = [1,3,5]
```

we would find the following correspondence between expressions and values ("=>" stands for "yields", and is not part of SASL):

```
y 2        =>    3
#x         =>    5
-1 : y     =>    [-1,1,3,5]
x++y       =>    [1,2,3,4,5,1,3,5]
x--y       =>    [2,4]
```

A useful piece of shorthand is the ".." notation for lists of integers; for example, "1..100" denotes the list of integers from 1 through 100. Lists need not be finite; some examples of the possibilities are

```
zeroes = 0:zeroes
bigtree = [0,bigtree,bigtree]
nats = 1...
```

(the suffix operator "..." maps its argument to an infinite list of integers). What makes it possible to program with infinite lists is that the implementation is consistently demand-driven in its evaluation strategy, and that ":" is non-strict (i.e., returns a result without evaluating its arguments). Infinite lists are no mere novelty; they are very helpful in separating lists' definitions from their uses, and they lead to a simple--and purely applicative--treatment of files and other input and output streams.

Functions defined in SASL can also be non-strict; consider, for example, the evaluation of

```
f x
WHERE x = x + 1
      f n = 3
```

The second definition establishes a function, **f**, with a formal parameter **n**. Clearly the definition of **x** is a bad one, in that there is no number that satisfies the equation (in the semantics of the language, we say that the value here attributed to **x** is $\perp$, the undefined object). Nevertheless, the value of "f x" is still 3. This nonstrictness considerably simplifies SASL's proof theory [2], because it means that equality is always substitutive.

Usually, of course, a function definition's formal parameter appears in the definition's right-hand side, as in the following definition of the factorial function:

```
fac n = product (1..n)
```

Here the ".." notation is used to avoid explicit recursion. The function product is defined in the ARCSYS definition library, which is the source of last resort for definitions of names used, but not defined, in programs evaluated by ARCSYS.

The definition of a function often requires a case analysis, and this can be accomplished by giving several alternative definitions distinguished by a simple form of pattern matching on formal parameters. As an example, consider the following definition of the library function **product** referred to above:

```
product  []    = 1
product (a:x) = a * product x
```

The pattern "(a:x)" serves two purposes in the second case: it not only restricts the case to non-empty lists, but gives distinct formal-parameter names to the first component (**a**) and the remainder (**x**) of the function's argument. In contrast to the order of definitions in a WHERE clause, the order of cases in a definition is significant: cases are tried in order of appearance.

An alternative to the multi-case definition is the conditional expression. An example is

```
power p x = p = 0 -> 1; x * power (p-1) x
```

A conditional expression begins with a tacit if; "->" is pronounced then, and ";" is pronounced else. The offside rule permits this example to be rewritten without the ";":

```
power p x = p = 0 -> 1
            x * power (p-1) x
```

The offside style is particularly advantageous in nested conditionals, as in this definition of the greatest common divisor function (using Euclid's algorithm):

```
gcd a b = a > b -> gcd (a-b) b
          a < b -> gcd a (b-a)
          a
```

Note that any SASL function of two or more arguments is always curried--that is, treated as a higher-order function of one argument--and can therefore be partially parameterized to yield more specialized functions. For example, given the above definition of **power** we could write

```
square = power 2
cube   = power 3
```

A skillful programmer can exploit this facility to achieve an extremely compressed programming style.

A recent addition to ARC SASL (borrowed from its immediate descendant, KRC [3]) is ZF-notation, which is based on Zermelo-Frankel set abstraction but in SASL used for lists rather than sets. Using ZF-notation, one can express very concisely a rather general type of iteration over lists. Syntactically, a ZF-expression has the form

```
[expression; qualifier; ... ; qualifier]
```

A qualifier is either a filter (i.e., a boolean expression) or a generator; the latter has the form

```
name <- list-expression
```

The name on the left of a "<-" is a local variable whose scope is delimited by the brackets.

A simple example is the following definition of the function **map** for mapping a function **f** over a list **x**:

```
map f x = [f a; a <- x]
```

The "[" is pronounced list of all, the ";" is pronounced such that, and the "<-" is pronounced is drawn from. An example involving two generators is the following definition of the cartesian product of two lists (each ";" after the first one is pronounced and):

```
cp x y = [[a,b]; a<-x; b<-y]
```

Using ZF expressions one can express quite complex thoughts lucidly.

We give a few more examples. The first is the definition of a function for generating all the permutations of a list:

```
perms [] = [[]]
perms x  = [a:p; a<-x; p<-perms(x--[a])]
```

The second example is an applicative version of C.A.R.Hoare's quicksort algorithm:

```
sort  []   = []
sort (a:x) = sort [b; b<-x; b≤a]
             ++ [a] ++
             sort [b; b<-x; b>a]
```

The third and fourth examples define a function named "fib", which maps an integer **n** to the **n**-th Fibonacci number. The first definition is the obvious one:

```
fib 1 = 1
fib 2 = 1
fib n = fib (n-1) + fib (n-2)
```

An alternative definition of fib is much more efficient:

```
fib 1 = 1
fib 2 = 1
fib n = fiblist (n-1) + fiblist (n-2)
fiblist = map fib (1...)
```

Our final example defines a list of all the prime numbers, using the method of Eratosthenes ("n REM p" yields the remainder after division of n by p):

```
sieve (2...)
WHERE sieve (p:x) = p : sieve [n; n<-x; n REM p > 0]
```

Acknowledgements: ARC SASL is virtually identical to the programming language SASL [1] developed by Prof. David Turner of the University of Kent, Canterbury, U.K. This brief overview borrows heavily from "An Overview of KRC", which Prof. Turner distributed to the audience as a supplement to his invited lecture at the 1982 ACM Conference on LISP and Functional Programming.

REFERENCES

[1] Turner, D. A. "SASL Language Manual". Computer Laboratory, University of Kent, Canterbury, U.K., 1976 (revised 1979).

[2] Turner, D. A. "Functional Programming and Proofs of Program Correctness." In Neel (ed.), Tools and Notions for Program Construction. Cambridge: Cambridge University Press, 1982.

[3] Turner, D. A. "KRC Language Manual". In preparation.