2

Data

Expressions in occam have values which represent, in the usual way for programming languages, integers, floating-point numbers, Boolean (truth) values, characters, and arrays (sequences) of these values.

Every expression in occam has a mechanically determinable type. There are no automatic translations of values from one type to another, as there are in some other programming notations. Whenever a value of one type is turned – coerced – into a corresponding value of another type, the coercion has to be made explicit.

Integers

The usual type used for discrete arithmetic is `INT`, which is a set of integers represented in twos-complement. Numerals like `42`, or `0`, represent the corresponding values of type `INT`. (When it comes to putting parentheses into expressions, it will transpire that for pragmatic reasons, negative numerals like `-137` are not literal numerals, but are expressions.) For definiteness, the type of an `INT` literal may also be noted along with the numeral, `27(INT)`, and so on.

It is not defined which twos-complement range of integers have to be representable as `INT`s, but in any particular implementation arithmetic with `INT`s is meant to be the most efficient. To be sure of representing a particular large value, or to be definite about a value communicated with another program, there are fixed length integer types which include `INT16`, `INT32` and `INT64`. Values of these types are integers with twos-complement representations in sixteen, thirty-two and sixty-four bits respectively. The type of a literal of fixed length always has to be given explicitly, for example `9223372036854775807(INT64)`.

It is sometimes handy to construct integer constants, especially very large or very small ones, which happen to be represented by a convenient bit pattern. In this case you can use hexadecimal numerals preceded by a hash sign, `#`, or a dollar sign, `$`. Thus, `#010A` is the same as `266(INT)`, and the constant just a little under nine and a quarter trillion at the end of the previous paragraph is `#7FFFFFFFFFFFFFFF(INT64)`.

In fact, the sensible way of describing that particular constant is as `MOSTPOS INT64`, since it is the most positive (largest) possible value of an `INT64`. The same holds for the most positive value of each of the integer types, and `MOSTNEG` for the smallest values, like `MOSTNEG INT64` which is `#8000000000000000(INT64)`.

Characters

There is a type `BYTE` of small non-negative integers, usually used for characters and similar data. The values of type `BYTE` are the non-negative integers less than 256. Although `BYTE` literals can be written as numerals, like `0(BYTE)`, `255(BYTE)`, there are also character denotations, like `'a'`, `'b'`, whose values are the ASCII codes of the corresponding characters represented as `BYTE`s.

In fact, the `BYTE` type is usually used for characters, and the denotation you would normally use for the code of most printable characters – `33(BYTE)` to `126(BYTE)` – is that character between single quotes. To avoid confusion, you cannot put the single- or double-quote characters between quotes; for these characters there is an escape notation, which uses up the asterisk character as well: `'*''` is `39(BYTE)`, a single-quote character; `'*"'` is `34(BYTE)`, a double-quote character; `'**'` is `42(BYTE)`, an asterisk character.

There are escape denotations for some layout characters: `'*s'` and `'*S'` are clearer alternatives for `' '`, a space character; similarly `'*c'` and `'*C'` for carriage-return, `'*n'` and `'*N'` for the newline character, `'*t'` and `'*T'` for the tab character. You can also denote any `BYTE` by a two-character hex numeral in a character escape, so that for example `'*#0A'` is the same as `#0A(BYTE)`.

Notice that the character type is different from integer types; that whilst `#FF(BYTE)` is positive, `#FFFF(INT16)` is negative.

Floating-point numbers

Approximate reals can be represented as values of type `REAL32` or `REAL64`. These types are defined by ANSI/IEEE Standard 754-1985[13], and are sometimes called *single-* and *double-length real numbers*. They are, of course, not *really* real numbers in the sense that mathematicians use the words, and arithmetic with `REAL32`s and `REAL64`s is only approximate.

A literal of one these types is written as a mantissa, optionally followed by an exponent, and necessarily qualified by its type. The mantissa is a sequence of digits with a decimal point; the exponent, if there is one, is written with an `E`, a sign, and a sequence of digits. Thus, `2.71828(REAL32)` and `0.000314159265359E+4(REAL64)` are each floating-point literals. The value of the second is 3.14159265359 (which is $0.000314159265359 \times 10^4$) rounded to a value representable as a double-length number according to the Standard.

Notice that the approximation inherent in floating-point numbers means that there really is not a representation as a (say) `REAL64` for some numbers that you can represent with a few decimal digits. There are some literals, such as `0.5(REAL64)`, which have the values you would expect: one-half is exactly representable as a binary fraction. Others, such as `0.2(REAL64)`, have values which are only approximations to what you might expect. Since one-fifth – 0.2_{10} – when represented as a binary fraction is necessarily a recurring sequence – $0.0011\dot{0}01\dot{1}_2$ – the value of `0.2(REAL64)` is $0.2_{10} \times (1 + 2^{-54})$ which is the rounded approximation with a fractional part of fifty-two bits.

Booleans

The literals `TRUE` and `FALSE` have type `BOOL`, a type which has only those two different values. Expressions with Boolean values are usually used in decisions that control the sequencing of programs.

Arrays

A finite sequence of values all of the same type can be organized as an array. The type of an array of n values of type T is written `[n]T`. A constant value of such a type, a *table*, can be denoted by a sequence of constant expressions between brackets, separated by commas. So

```
[3, 2, 1]
```

is an `[3]INT` (read as 'an array of three `INT`s', or 'a row of three `INT`s') and the unlikely

```
[[[MOSTNEG INT64]]]
```

is an `[1][1][1]INT64`. For no particular reason, when you write down the type of an array, the expressions that determine the number of components must be of type `INT`, and of no other integer type.

Perhaps it is worth emphasizing that arrays have to be sequences of components all of the same type: `[1(INT32),1.0(REAL32)]` is not an array, nor is `[1,[2,3]]`. Since the type of an array includes the number of components in the sequence, an array of arrays has to have components of uniform length, so

```
["That man,", "I wish him well.", "I wish him grass."]
```

is not a valid `[][]BYTE`. On the other hand, while arrays of arrays do have to be 'rectangular', they need not be 'square'. An array of one length can have components of another length:

```
[[[TRUE, TRUE, FALSE], [FALSE, FALSE, TRUE]]]
```

is an `[1][2][3]BOOL`; both of the components of the array with two components are of length three.

Arrays of characters can be denoted by *strings* of characters between double quotes, like

```
"Jetzt geht*'s um die Wurst."
```

which is a much clearer way of denoting the array of twenty-six `BYTE`s

```
['J', 'e', 't', 'z', 't', '*s',
 'g', 'e', 'h', 't', 39(BYTE), 's', '*s',
 'u', 'm', '*s',
 'd', 'i', 'e', '*s',
 'W', 'u', 'r', 's', 't', '.']
```

Any of the character escape sequences can also be used in a string.

Expressions

All of the expressions described so far denote constants; there are also various ways of giving names to values, some of which are fixed, some of which change. Every named value also has a determinable type.

A name can be given to the value of an expression by *abbreviating* the expression. An abbreviation like

```
VAL REAL64 fine.structure IS 1.0(REAL64)/137.0388(REAL64) :
```

identifies the name *fine.structure* with the value of the expression

```
1.0(REAL64) / 137.0388(REAL64)
```

in the *scope* of the abbreviation. (Roughly speaking, this is the subsequent process.)

Not only can you use the name, rather than having to write out the value of the expression, but you can expect that an implementation might evaluate the expression only once, irrespective of how often you use the name to stand for its value. This means, of course, that it would be wrong for an abbreviated expression to be invalid, even if the subsequent process happens not to use the name. On the other hand, an implementation is not *obliged* to evaluate and store the values of abbreviated expressions. There are restrictions on the use of abbreviations which mean that whenever an abbreviation is valid, the meaning of the program in which it appears is unaffected by whether the expression is evaluated once and a stored value used, or the expression is evaluated every time its name is used.

Because the type of an expression can be determined, it is allowed to omit the sizes of arrays from an abbreviation, or even to leave out the type altogether. Each of

```
VAL [30]BYTE detail IS "Who is? Sylvia! What? Is she!*#00" :
VAL []BYTE less.detail IS detail :
VAL do.it.yourself IS "Who is? Sylvia! What? Is she!*#00" :
```

identifies each of the three names as an abbreviation for the same `[30]BYTE` value.

It is normally good practice to quote the type of an abbreviation, if only to remind the reader; it would also help a mechanical checker to produce sensible error messages in case of type errors. On the other hand if the length of an array is relatively unimportant, as in the case of a string that will just be typed out as a message, it would be tedious to require the type to be quoted in full.

Integer arithmetic

The usual arithmetic operators: `+` for addition, `-` for subtraction, `*` for multiplication, `/` for division apply to any pair of integers of the same type, returning a value of the same type, provided that the result is representable as a value of that type. A process which causes the evaluation of an expression that overflows the range of representable values, or which does something invalid such as dividing by zero, behaves like `STOP`.

Provided that they are in the representable range, the results of addition, subtraction and multiplication are what you would expect for arithmetic on the signed integers. Integer division returns a result which is rounded towards zero, so that for example the value of `9/4` is 2, and that of `(-9)/4` is -2. The corresponding remainder is given by `\`, which can also be written `REM`; again it is applicable to any pair of integers of the same type and returns an integer of the same type. So, the value of `(-9)\4` is -1, and in general the sign of a non-zero remainder is the same as the left operand, and $((x \,/\, y) * y) + (x \setminus y)$ has the same value as x provided no sub-expression leads to an overflow.

The operators PLUS, MINUS, TIMES represent cyclic addition, subtraction, and multiplication of integers. Cyclic, or modulo, arithmetic cannot lead to overflows; the result of an operation is that value in the representable range which differs from the result that you might expect by a multiple of the range of representable values. For example, the value of

```
(MOSTNEG INT32) MINUS 1(INT32)
```

is the same as MOSTPOS INT32; the value of

```
(MOSTPOS INT64) TIMES (MOSTPOS INT64)
```

is 1(INT64), which is the INT64 which differs from $(2^{63}-1)^2$ by a multiple of 2^{64}. Although all of the other operators treat integer representations as signed twos-complement numbers, you will get the same result for the three cyclic operations if you think of them as unsigned modulo arithmetic. (Of course, there is no such thing as cyclic division or cyclic remainder!)

Both the normal and cyclic subtraction operators can be used as a unary prefix, so that -x is the same as 0 - x, and MINUS x as 0 MINUS x. (There is exactly one value x of each integer type for which -x does not mean the same as MINUS x.)

Bit-pattern arithmetic

It is sometimes handy for a programmer to treat a word of storage as if it were just a sequence of bits. In occam this is done with the bit-pattern operators which operate on the representations of integers. The bit-by-bit *and*, *or*, and *exclusive-or* operations produce results which are integers of the same type as both of their operands, each bit of the representation of the result depending on the corresponding bits in the representations of the operands.

The bit-by-bit *and*, written either as /\ or as BITAND, has a one bit in its result precisely where both of its operands have one bits, so

```
#00FF(INT16) /\ #1234(INT16)
```

is #0034(INT16). Notice that this really is an operation on perfectly ordinary integers, and you can write expressions like x/\(x MINUS 1). (If you are unfamiliar with bit-manipulation operations, check that you understand why, for any integer x that has a non-zero value, there is one less bit set to one in x/\(x MINUS 1) than in x.)

The bit-by-bit *or*, written as \/ or as BITOR, has a one bit in its result precisely where either or both of its operands has a one bit, so

```
#00FF(INT16) \/ #1234(INT16)
```

is #12FF(INT16). The bit-by-bit exclusive-or, written ><, has a one bit in its result precisely where exactly one of its operands has a one bit, so

```
#00FF(INT16) >< #1234(INT16)
```

is `#12CB(INT16)`.

The bit-by-bit negation or *inversion* operation, written as ~ (a tilde sign) or as `BITNOT`, is a monadic operation, applicable to integers and returning an integer of the same type. The result has a representation with one bits exactly where the representation of the operand has zero bits. For example `~#1234(INT16)` is `#EDCB(INT16)`, and in general if x is an `INT`, ~x is the same as (-1) >< x, and similarly for other integer types.

The shifting operations, << and >>, operate on a bit pattern by shifting it left or right, respectively, by a specified number of bits. Bits that are shifted out of the pattern are lost, and the vacated bit positions are filled by zeros, so both `1(INT16)<<15` and `#FFFF(INT16)<<15` have the same value as `MOSTNEG INT16`; and `#87654321(INT32)>>16` is `#00008765(INT32)`.

The result of a shift is of the same integer type as the left argument, but notice that the number of bits over which to shift is of type `INT` in each case. The operation is valid only when this number of bits over which to shift is between zero and the length of the representation being shifted; other values can be treated as overflows.

A particularly common use of << is in raising two to an integer power, because the value of 1 << n is 2^n provided this value is in the representable range. Similarly, for small enough n, (1 << n) `MINUS` 1 is the integer $2^n - 1$, having a representation in which the n least significant bits are set; although ~((~ 0) << n) is perhaps a neater way of saying the same thing.

Floating-point arithmetic

Floating-point arithmetic is quite similar to integer arithmetic: +, -, *, and / can be used between any two floating-point numbers of the same length and return a result of the same type. Of course, it is quite likely that the result of a floating-point operation cannot be represented exactly by a value of its type, and it is defined that the result is *rounded* to the nearest representable value. Rounding returns a value which is wrong by at most a half of the value of the least significant bit in the representation, choosing a zero in the least significant bit if the required value is out by exactly half a bit value.

The remainder operator \ or `REM` can also be used between pairs of floating-point expressions. Its result is not related to division in the same way as for integers; in effect, the result of a division is rounded to an infinite precision integer, and the result of a remainder operation is the difference between the numerator and that integer number of times the denominator. Only then is the result rounded to the length of the result, so it is more accurate than a result which could be calculated from the quotient.

Consider `6.0(REAL64) REM 4.0(REAL64)`; in this case both the numbers have exact representations, and the result of the corresponding division is exactly 1.5. Rounding this to the nearest integer yields 2, which is the nearest integer with zero as its least significant bit. It follows that the result of `6.0(REAL64) REM 4.0(REAL64)` is $6-(2\times4)$ as a `REAL64`, that is it is the same as `-2.0(REAL64)`.

In some programming notations, all floating-point arithmetic is carried out to double-length or longer and the result of the calculation reduced to the required precision. To do this might mean rounding the result of an operation to double-length, and then rounding that value to a single-length value. Notice that it is defined that expressions in occam are evaluated in the arithmetic of the type of the arguments. This means that fewer rounding operations are needed, giving results that may be more accurate. In particular, since storing an intermediate value cannot cause rounding, it is safe for a compiler to rearrange evaluation of floating-point expressions to optimize performance.

It is, of course, possible that an expression of a `REAL` type does not have a numeric value representable in that type: the value may be too large or small, or too near zero without being exactly zero. Although the Standard defines bit patterns to represent these overflows and underflows, a process which evaluates floating-point expressions that overflow or underflow is normally invalid, and stops. Implementations of occam may provide a library of predefined routines which implement the extensions of floating-point operations to these non-numbers.

Relational expressions

Pairs of expressions of the same integer, byte, floating-point, or Boolean type can be compared for equality, or inequality: $x=y$ is `TRUE` if x is exactly equal to y, and `FALSE` otherwise; x `<>` y is `TRUE` exactly when x is not equal to y. Notice that, whilst you can test whether or not two floating-point numbers are equal it is unlikely to be exactly what you intend. For example,

```
(1.0E+16(REAL64) + 1.0(REAL64)) = 1.0E+16(REAL64)
```

and although

```
68.0(REAL32) * (1.0(REAL32) / 127.0(REAL32)) =
                            (68.0(REAL32) / 127.0(REAL32))
```

is `TRUE`, the corresponding equality does not hold in double-length arithmetic, because

```
68.0(REAL64) * (1.0(REAL64) / 127.0(REAL64)) <
                            (68.0(REAL64) / 127.0(REAL64))
```

The difference is caused by the rounding errors happening to be equal in single-length arithmetic but differing by 2^{-53} in the two double-length expressions.

Pairs of expressions of the same integer, byte, or floating-point type can be compared by `<`, `>`, `<=`, `>=`. The result is `TRUE` when the left operand is less than, greater than, no greater than, or no less than, respectively, the right operand. Again, relying on the distinction between `<` and `<=` on floating-point values is unwise.

There is also a cyclic comparison, `AFTER`, defined on pairs of integers of the same type so that

```
(a AFTER b)  =  ((a MINUS b) > 0)
```

It is intended for comparing the readings on a clock or timer: a clock counts cyclically, like the turning hands of a physical clock, and almost half of all integers are `AFTER` a given integer of the same type.

Boolean expressions

The operations `AND`, and `OR` operate on a pair of truth values and return a truth value. The result of an expression in `AND` is `FALSE` if its left operand is `FALSE`, and is the same as the right operand otherwise. Similarly, the result of an expression in `OR` is `TRUE` if its left operand is `TRUE`, and is the same as the right operand otherwise.

Notice that these operations are distinguished from others in that because of the left-to-right asymmetry it is safe to use them with invalid sub-expressions, such as divisions by zero or representation overflow. It is not possible for the expression

```
(x = 0) OR ((y / x) > 1) OR ((y / x) < (-1))
```

to be invalid because of a division by zero. Notice as well that you are allowed to leave brackets out of expressions in `AND` and `OR` because of their associativity.

Monadic negation, `NOT`, returns `TRUE` if its operand is `FALSE` and vice versa, so for example `NOT`($x = y$) is the same as x `<>` y.

Type coercion

Because the types of the arguments to an operator must exactly match the requirements of the operator, it is often necessary to replace an expression of one type by another of a different type but a corresponding value. Generally, writing the name of a type in front of an expression returns a result of the specified type and the 'corresponding' value.

In the case of integer types and characters, the corresponding value is the same integer, and the coercion is valid only if the integer is within the range representable by the result type. For the purpose of coercion, Booleans are also treated as integers with `INT FALSE` being 0, and `BOOL 1` being `TRUE`.

Similarly, a `REAL32` can be recast as the `REAL64` which represents the same number. Thus `REAL64 0.5(REAL32)` is the same double-length number as `0.5(REAL64)`. As ever, beware rounding errors: `REAL64 0.2(REAL32)` is not quite the same as `0.2(REAL64)`, the latter being a fifty-three bit approximation to one fifth, the former only a twenty-seven bit approximation.

In the case of other conversions, between integers and floating-point values, between floating-point values and integers, or from double-length to single-length floating-point values, you have to specify whether to round or truncate the value on conversion. It should be obvious that turning a `REAL64` into an integer might require rounding because there might be a fractional part to its value. Notice, additionally, that since there are more bits in an `INT64` than in the mantissa of a `REAL64`, you sometimes have to round or truncate integers to get representable floating-point values. In occam, you always have to specify the rounding when converting an integer to a floating-point value, even between `INT16` and `REAL64`.

Rounding has already been described: the result value differs from the desired value by at most a half of the value of the least significant bit; the representation with a zero in the least significant bit is chosen where there are two alternatives. You can also specify *truncation* of the result: any bits in excess of the size of the representation are simply lost. For example, the value of `INT ROUND 1.5(REAL32)` is 2, but the value of `INT TRUNC 1.5(REAL32)` is 1. Again, since there are only twenty-three bits in the mantissa of a `REAL32`,

```
REAL32 TRUNC ((1(INT32) << 31) \/ #FF(INT32))
```

is 2^{31} as a `REAL32`, whereas

```
REAL32 ROUND ((1(INT32) << 31) \/ #FF(INT32))
```

is $2^{31} + 2^{8}$ as a `REAL32`.

Components of arrays

If the value of *exp* is of an array type, then `SIZE` *exp* is the number of components in the array, so if

```
VAL [39]BYTE circumflex IS
             "Qu*'est-ce qu*'il y a sur l*'eau du Rhone?" :
```

then `SIZE` *circumflex* is 39. Do not confuse the `SIZE` of an array with the amount of space it occupies. The `SIZE` of a `[39]INT` is also 39, as is the `SIZE` of a [39][39][39]REAL64, but the SIZE of a [1][39]BYTE is only 1. Like other expressions to do with the indexing of arrays, SIZE e is always of type INT.

The components of a value which is an array can be selected individually, by *subscripting* or indexing them. They are indexed by consecutive non-negative integers (of type INT), counting from zero; for example the value of *circumflex*[0] is 'Q', and that of *circumflex*[38] is '?'. Of course, the components of an array of arrays are themselves arrays, so if a is an [3][2][1]BYTE, $a[2]$ is an [2][1]BYTE, $a[2][1]$ is an [1]BYTE, and $a[2][1][0]$ is a BYTE. For an indexed expression to be valid, the index must be non-negative and smaller than the size of the array.

Whole sub-arrays of contiguous components from an array can be selected together; for example

```
["Lost lanes of Queen Anne*'s lace" FROM 14 FOR 10]
```

is the [10]BYTE value "Queen Anne" which is the same as the character of the string indexed by fourteen, the character indexed by fifteen, and so on for ten characters. A *segment* expression like this is valid only if it lies entirely within the array of which it is a segment: any valid subscription of a segment has to correspond to a valid subscription of the array of which it is a segment.

Notice that a segment is indexed from zero, just like any other array value, so

```
["Lost lanes of Queen Anne*'s lace" FROM 14 FOR 10][6]
```

is "Queen Anne"[6], which is 'A'.

Constant expressions

You should expect an implementation of occam to require that certain expressions in a valid program have to be ‘constants’ that could reasonably be evaluated by a compiler, without having to execute the program. In particular, the sizes of various arrays will usually have to be constant. Roughly speaking, an expression made only of sub-expressions described in this chapter is a constant expression.