For example:

(2 . 3 . 4) * (2 , 2) =

(4 * (2 . 2)) + (3 * (2 . 2) * (1 . 0)) + (2 * (2 . 2) * (1 . 0 . 0)) =

(8 . 8) + (6 . 6 . 0) + (4 . 4 . 0 . 0) = (5 . 1 . 4 . 8)

3.4 Division and Remainder

There are two operators for division, '/' and 'div'. The latter uses successive subtractions while the former uses the same algorithm used by humans as follows:

```
if N < M then (N / M) = 0
else
if (N < M*10) then (N / M) = (N div M)
else
N /M =
        (left(N) / M) .
        to_digit(((left(N) mod M) . lsd(N)) / M);
```

We will leave the explanation of the recursive part of this algorithm as an exercise to the reader.

D0 = sub(4,7) = 7 and
((8 . 6) - borrow(4,7)) - (3) = D2 . D1 =>
((8 . 6) - (1)) - (3) = D2 . D1 =>
(8 . 5) - (3) - D2 . D1 =>

D1 = sub(5,3) =2 and

((8) - borrow(5,3)) - (0) = D2 =>
((8) - (0)) - (0) = D2 =>
(8) - (0) = D2 =>
D2 = sub(8,0) = 8

Then (8 . 6 . 4) - (3 . 7) = 8 . 2 . 7

Also before applying such an algorithm, we make sure that the signs are resolved. For example:

M - -(N) = M + N
-(M) - N = -(M + N)
-(M) - -(N) = N - M
if M < N => M - N = -(N - M)

3.3 Multiplication

Multiplication of two integers depends on an operation that multiplies a digit by an integer:

_***_ : Digit, Int -> Int

Here is an example that illustrates its recursive application:

4 *** 3 . 2 . 5 = D3 . D2 . D1 . D0

where:

D0 = mult(4, 5) = 0 and
(4 *** (3 . 2)) + mult_carry(4, 5) = D3 . D2 . D1 =>
(4 *** (3 . 2)) + 2 = D3 . D2 . D1 =>
(1 . 2 . 8) + 2 = 1 . 3 . 0 { recursive application }
4 *** 3 . 2 . 5 = 1 . 3 . 0 . 0

Now the multiplication of two integers, say M = Mi . ... M1 . M0 and N = Nj . ... N1 . N0 will be:

(M0 *** N) + ((M1 *** N) * (1 . 0)) + ... + ((Mi *** N) * ((1 . 0) ^ i))

Figure 4 shows the complete addition operation.

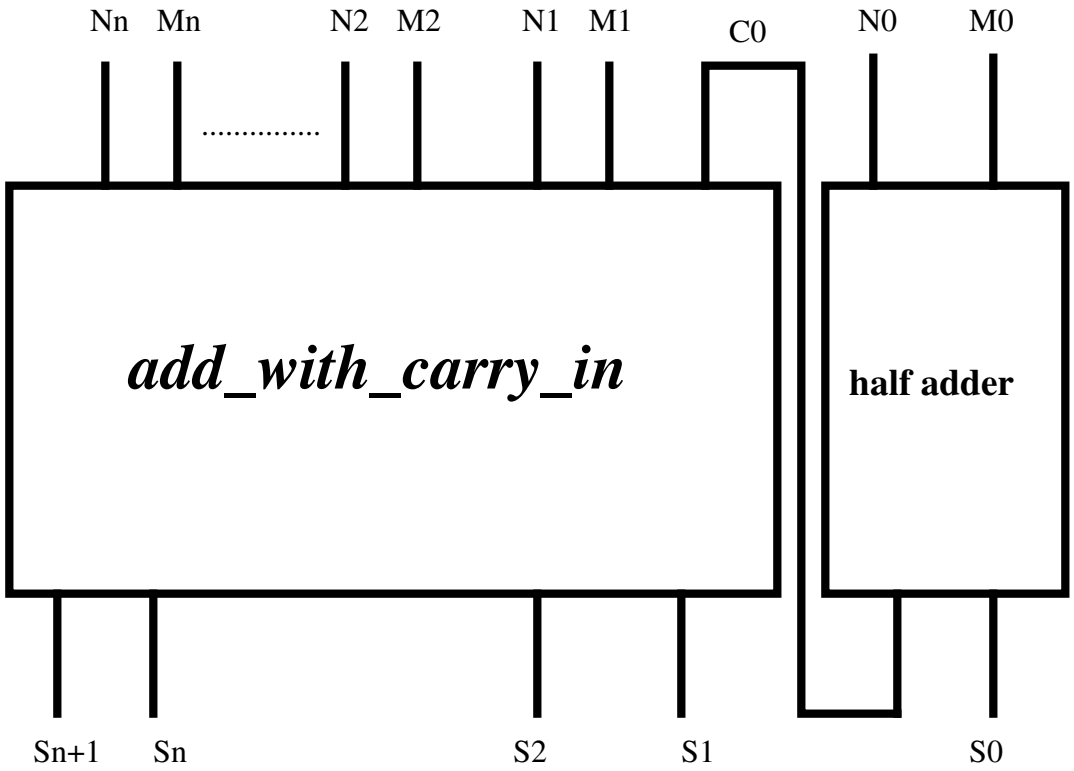

Figure 4. Addition of two integers

Before the addition is carried out, the signs of the two integers are checked. For instance:

-(M) + -(N) = -(M + N)
-(M) + N = N - M
M + -(N) = M - N

3.2 Subtraction

Subtraction is done in the same way humans subtract, i.e. by subtracting digits from right to left and using borrows when necessary. This application is recursive. For example

(8 . 6 . 4) - (3 . 7) = D2 . D1 . D0

where

Let n be the maximum of i and j Then

```
        Mn . ... . M1 . M0
+
        Nn . ... . N1 . N0
_____________________
Sn+1 .  Sn . ...   . S1 . S0
```

Where

- Mk = 0 for (i < k <= n)

- Nl = 0 for (j < l <= n)

- S0 is obtained with *half_adder*(M0, N0)

- Sn+1 . Sn . ... . S1 are constructed by
  add_with_carry_in((Mn . ... . M1), (Nn . ... . N1), *ha_carry*(M0, N0))

add_with_carry_in uses successive full adders as shown in figure 3.

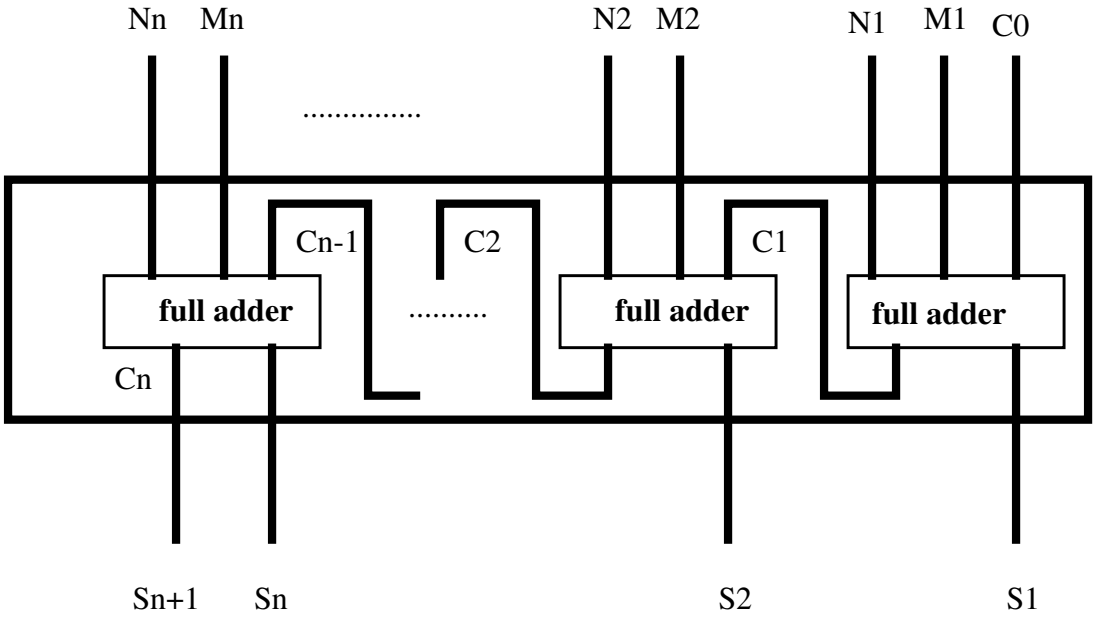

Figure 3. add_with_carry_in operation

.

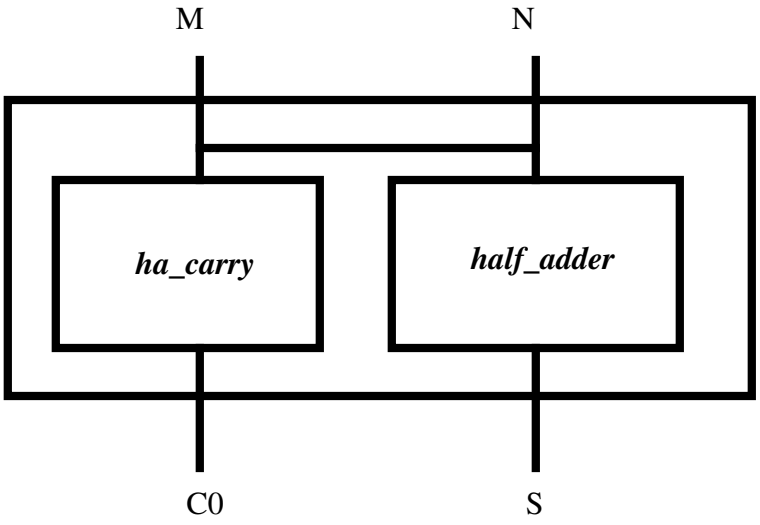

Figure 1. Half Adder

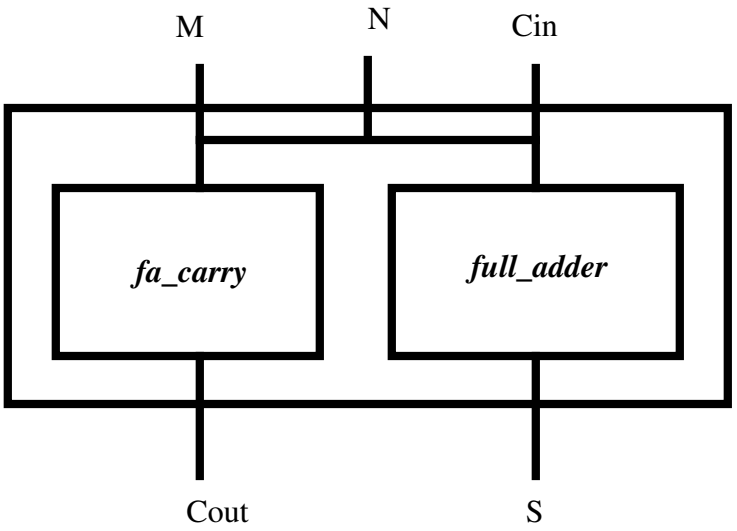

Figure 2. Full Adder

For example, the addition of two integers say

M = Mi . ... M1 . M0 and N = Nj . ... N1 . N0 is done as following:

_/_	: Int, Int -> Int
*_mod_*	: Int, Int -> Int
_^_	: Int, Int -> Int
add_with_carry_in	: Int, Int, Digit -> Int

add_with_carry_in: adds two integers and a carry in digit

Examples:

-(3 . 5 . 2 .1) + (2 . 5 . 7) gives -(3 . 2 . 6 . 4)

- (- (4 . 5 . 6 . 7)) + (4 . 5 . 6 . 6) gives 9 . 1 . 3 . 3

- (4 . 5 . 6 . 7) + (4 . 5 . 6 . 6) gives -(1)

(3 . 3 . 6 . 7) - (4 . 4 . 7 . 4) gives -(1 . 1 . 0 . 7)

3 - 1 gives 2

2 * (2 . 3 . 7 . 4) gives 4 . 7 . 4 . 8

- (2 . 0) * (2 . 2 . 2) gives 4 . 4 . 4 . 0

(4 . 8 . 8 . 8) / (4 . 4) gives 1 . 1 . 1

(4 . 8 . 8 . 8) mod (4 . 4) gives 4

- (0 . 2) ^ (0 . 3) gives -(8)

2 ^ 8 gives 2 . 5 . 6

add_with _carry_in ((3 . 4) , (1 . 5), 1) gives 5 . 0

3 ALGORITHMS FOR ARITHMETIC OPERATIONS

The algorithms that define the operational semantics of the arithmetic operations are chosen to be efficient. In this section we present the algorithms used for addition, substraction, multiplication, and division. All these algorithm are recursive in nature since the number of digits in an integer is not limited.

3.1 Addition

The addition of two non-negative integers is done using half adder and full adder operations shown in figure 1 and figure 2 respectively (ADT operations are shown in *italic* font)

The definitions for the unary operators + and - are as such

+(N) = N;
-(0) = 0;
-(-(N)) = N;

2.2.2 Comparison Operations

The comparison operators for equal, less than, less than or equal, greater than, greater than or equal, and not equal are defined as:

_ == _	: Int, Int -> Bool
_ < _	: Int, Int -> Bool
_ =< _	: Int, Int -> Bool
_ > _	: Int, Int -> Bool
_ >= _	: Int, Int -> Bool
_ <> _	: Int, Int -> Bool

Their operational algorithms are described in an efficient way, for example the algorithm for > is the following:

- if N2 contains only one digit and N1 contains more than one digit then
  N1 > N2 is true

- if N1 contains only one digit and N2 contains more than one digit then
  N1 > N2 is false

- if N1 has the form M1 . D1 (i.e. it contains more than one digit) and N2 has the form M2 . D2 then
  N1 > N2 is (M1 > M2) or (M1 == M2 and D1 > D2)

Examples:

-(-(4 . 5 . 6 . 7)) > +(4 . 5 . 6 . 6) gives true
- (4 . 5 . 6 . 7) > - (4 . 5 . 6 . 6) gives false
-(0) == 0 gives true

2.2.3 Arithmetic Operations

Many arithmetic operations on integers are defined such as addition, subtraction, multiplication, division, modulo, and exponent.:

_+_	: Int, Int -> Int
_-_	: Int, Int -> Int
_*_	: Int, Int -> Int

full_adder and *fa_carry* are similar but uses three digits as arguments

full_adder(5,9,9) gives 3 while *fa_carry*(5,9,9) gives 2.

sub(6,4) gives 2 while *borrow*(6,4) gives 0
sub(4,6) gives 8 while *borrow*(4,6) gives 1

mult(4,5) gives 0 while *mult_carry*(4,5) gives 2.

2.2 Integer Operations

Integer operations depend on the digit operations. The following is a description of some important operations.

2.2.1 General Purpose Operations

There are some auxiliary operations, used to define other operations. Here are some of such operations:

is_pos	: Int -> Bool
is_neg	: Int -> Bool
to_digit	: Int -> Digit
to_int	: Digit -> Int
is_digit	: Int -> Bool
lsd	: Int -> Digit
left	: Int -> Int

is_pos : checks if the given integer is positive or not.

is_neg : checks if a given integer is negative or not (i.e. the negation of is_pos).

to_digit : converts an integer that can be represented by only one digit to that digit (i.e. converts sort)

to_int : is the inverse of to_digit.

lsd : returns the least significant digit of an integer.

left : returns the integer divided by ten (i.e. the original integer excluding the lsd).

The following rule takes out all left leading 0s of any integer

0 . D = *to_int*(D);

borrow	: Digit, Digit -> Digit
mult	: Digit, Digit -> Digit
mult_carry	: Digit, Digit -> Digit

An explanation of the above definitions are presented below:

next: successor of digits modulo 10:
prev: predecessor of digits modulo 10.

More natural names for the previous two operations could be *succ* and *pred* respectively, but these would require overloading, which we wish to avoid.

to_nat: converts digits to natural numbers. This operation is used only in the definitions of the comparison operations <, =<, >, >=, ==, and <>.

half_adder: returns the resulting digit of adding two digits.

ha_carry: returns the carry out of adding two digits.

full_adder: returns the resulting digit of adding two digits and the carry in.

fa_carry: returns the carry out of adding two digits and the carry in.

sub: returns the digit resulting from subtracting one digit from another.

borrow: returns the borrow from subtracting one digit from another

mult: returns the resulting digit of multiplying two digits.

mult_carry: returns the carry out of multiplying two digits.

Examples:

succ(8) gives 9
succ(9) gives 0

pred(9) gives 8
pred(0) gives 9

to_nat(5) gives *succ*(*succ*(*succ*(*succ*(*succ*(0)))))

half_adder(4,5) gives 9 while *ha_carry*(4,5) gives 0
half_adder(5,7) gives 2 while *ha_carry*(5,7) gives 1

0	: -> Int
1	: -> Int
2	: -> Int
3	: -> Int
4	: -> Int
5	: -> Int
6	: -> Int
7	: -> Int
8	: -> Int
9	: -> Int
_._	: Int, Digit -> Int
+	: Int -> Int
-	: Int -> Int

Here are some examples:

7 represents the integer 7.

-(7) represents the integer -7.

3 . 4 . 5 . 6, +(3 . 4 . 5 . 6), or -(-(3 . 4 . 5 . 6)) represent the integer 3456.

-(1 . 0 . 0 . 0 . 0 . 0 . 0) represents the integer -1000000.

-(0), +(0), or 0 represent the integer 0

2.1 Digit Operations

The sort name for digits is **Digit**. The following are the digit operations provided:

next	: Digit -> Digit
prev	: Digit -> Digit
to_nat	: Digit -> Nat
_ < _	: Digit, Digit -> Bool
_ =< _	: Digit, Digit -> Bool
_ > _	: Digit, Digit -> Bool
_ >= _	: Digit, Digit -> Bool
_ == _	: Digit, Digit -> Bool
_ <> _	: Digit, Digit -> Bool
half_adder	: Digit, Digit -> Digit
ha_carry	: Digit, Digit -> Digit
full_adder	: Digit, Digit, Digit -> Digit
fa_carry	: Digit, Digit, Digit -> Digit
sub	: Digit, Digit -> Digit

A LOTOS Data Type Definition for Integers

Mazen Haj-Hussein
Protocols and Software Engineering Research Group
University of Ottawa, Computer Science Department
Ottawa, Ont., Canada K1N 9B4
E-mail: mazen@csi.uottawa.ca

ABSTRACT

LOTOS notation for natural numbers is inconvenient to write and read, and at the same time is computationally inefficient. Also, there is no notation for negative numbers. In this paper, we provide a notation and an abstract data type package that obviates these deficiencies.

1. INTRODUCTION

Every user of LOTOS must have been annoyed by the difficulty of writing large numerals, and by the absence of library facilities for negative numbers. Furthermore since LOTOS is (partially) executable, a specification, that includes the behaviour part and the abstract data types part, is effectively a “fast prototype” of the entity specified, thus it is possible to exercise a specification of a complex system at the design stage. Therefore, although these specifications are implementation independent, it is important to have them written in efficient executable styles.

In this paper we present a definition for integers that has the following advantages:

- Large integers can be easily represented using a simplified notation.
- Negative integers are allowed.
- The algorithms that define the properties of the integer operations are specified in an efficient executable style.

2. INTEGER REPRESENTATION

Integers are represented using a sequence of digits separated by the operator ‘.’. Their BNF definition is:
<integer> ::= <signed_integer> | <unsigned_integer>

<signed_integer> ::= <sign> (<integer>)

<unsigned_integer> ::= <digit> | <unsigned_integer> . <digit>

<sign> ::= - | +

The following is the definition of the basic integer operators (constructors) in ACT ONE notation: