Dynamic tests of the Desi JIT Compiler

Brian Shearing

Peter Grogono

31 October 2015

This document comprises a collection of Desi programs that test the dynamic behaviour of the Desi JIT Compiler. The intent is to verify every combination of DIL command and type of operand. The scripts are followed by the log of the test run and by some informal exercises.

Contents

1 Introduction

This document corresponds to Release 9 of the JIT compiler, and supports scalar types, including texts, and a pre-release of maps. Communication between processes is supported, as are signals, and conditional communication.

Tests of library routines that support mathematical functions and random-numbers are not comprehensive and do little but check the linkage from Desi invocation to actual external routine. Library routines for input and output are exercised rather than tested.

2 Booleans

```
beginTestT = routine t: Text;                          external "C", "beginTestT";
i32EQ      = routine c: Word; v: Word;                 external "C", "i32EQ";
textEQ     = routine c: Text; v: Text;                 external "C", "textEQ";
boolEQ     = routine c: Boolean; v: Boolean;           external "C", "boolEQ";
endTest    = routine                                   external "C", "endTest";
```

2.1 Process for communication tests

```
BoolExchange = protocol { given: Boolean; ↑result: Boolean }

OneShotContradictionService = process p: +BoolExchange; {
    p. result := not p.given;
}
```

2.2 Routine for routine tests

```
Swapper = routine p, q: Boolean → t, v: Boolean; {
    boolEQ(t, false);
    boolEQ(v, false);
    t := q;   v := p;
}
```

2.3 Main cell

```
main = cell {
    beginTestT("Booleans");
```

2.4 Initialisation

```
f: Boolean;   boolEQ(f, false);
t: Boolean := true;   boolEQ(t, true);
```

2.5 Unary operations

```
boolEQ(not f, true);
boolEQ(not t, false);
```

2.6 Comparisons

We start with tests that permute binary combinations of constants and variables. Then we settle down to using just variables to exercise each operator.

```
boolEQ(false < false, false);
boolEQ(false < true, true);
boolEQ(true < false, false);
boolEQ(true < true, false);

boolEQ(f < false, false);
boolEQ(false < t, true);
boolEQ(true < f, false);
boolEQ(t < true, false);
```

$boolEQ(f < f,\ \mathbf{false});$
$boolEQ(f < t,\ \mathbf{true});$
$boolEQ(t < f,\ \mathbf{false});$
$boolEQ(t < t,\ \mathbf{false});$

$boolEQ(f \leq\ f,\ \mathbf{true});$
$boolEQ(f \leq\ t,\ \mathbf{true});$
$boolEQ(t \leq\ f,\ \mathbf{false});$
$boolEQ(t \leq\ t,\ \mathbf{true});$

$boolEQ(f \geq\ f,\ \mathbf{true});$
$boolEQ(f \geq\ t,\ \mathbf{false});$
$boolEQ(t \geq\ f,\ \mathbf{true});$
$boolEQ(t \geq\ t,\ \mathbf{true});$

$boolEQ(f > f,\ \mathbf{false});$
$boolEQ(f > t,\ \mathbf{false});$
$boolEQ(t > f,\ \mathbf{true});$
$boolEQ(t > t,\ \mathbf{false});$

$boolEQ(f = f,\ \mathbf{true});$
$boolEQ(f = t,\ \mathbf{false});$
$boolEQ(t = f,\ \mathbf{false});$
$boolEQ(t = t,\ \mathbf{true});$

$boolEQ(f <> f, \mathbf{false});$
$boolEQ(f <> t, \mathbf{true});$
$boolEQ(t <> f, \mathbf{true});$
$boolEQ(t <> t, \mathbf{false});$

2.7 Logical operations

$boolEQ(f\ \mathbf{iff}\ \ f,\ \mathbf{true});$
$boolEQ(f\ \mathbf{iff}\ \ t,\ \mathbf{false});$
$boolEQ(t\ \mathbf{iff}\ \ f,\ \mathbf{false});$
$boolEQ(t\ \mathbf{iff}\ \ t,\ \mathbf{true});$

$boolEQ(f \Longleftrightarrow f, \mathbf{true});$
$boolEQ(f \Longleftrightarrow t, \mathbf{false});$
$boolEQ(t \Longleftrightarrow f, \mathbf{false});$
$boolEQ(t \Longleftrightarrow t, \mathbf{true});$

$boolEQ(f\ \mathbf{xor}\ f,\ \mathbf{false});$
$boolEQ(f\ \mathbf{xor}\ t,\ \mathbf{true});$
$boolEQ(t\ \mathbf{xor}\ f,\ \mathbf{true});$
$boolEQ(t\ \mathbf{xor}\ t,\ \mathbf{false});$

$boolEQ(f\ \mathbf{and}\ f, \mathbf{false});$
$boolEQ(f\ \mathbf{and}\ t, \mathbf{false});$
$boolEQ(t\ \mathbf{and}\ f, \mathbf{false});$
$boolEQ(t\ \mathbf{and}\ t, \mathbf{true});$

boolEQ(*f* **nand** *f*, **true**);
boolEQ(*f* **nand** *t*, **true**);
boolEQ(*t* **nand** *f*, **true**);
boolEQ(*t* **nand** *t*, **false**);

boolEQ(*f* **or** *f*, **false**);
boolEQ(*f* **or** *t*, **true**);
boolEQ(*t* **or** *f*, **true**);
boolEQ(*t* **or** *t*, **true**);

boolEQ(*f* **nor** *f*, **true**);
boolEQ(*f* **nor** *t*, **false**);
boolEQ(*t* **nor** *f*, **false**);
boolEQ(*t* **nor** *t*, **false**);

boolEQ(*f* **implies** *f*, **true**);
boolEQ(*f* **implies** *t*, **true**);
boolEQ(*t* **implies** *f*, **false**);
boolEQ(*t* **implies** *t*, **true**);

boolEQ(*f* $\Longrightarrow$ *f*, **true**);
boolEQ(*f* $\Longrightarrow$ *t*, **true**);
boolEQ(*t* $\Longrightarrow$ *f*, **false**);
boolEQ(*t* $\Longrightarrow$ *t*, **true**);

boolEQ(*f* **revimp** *f*, **true**);
boolEQ(*f* **revimp** *t*, **false**);
boolEQ(*t* **revimp** *f*, **true**);
boolEQ(*t* **revimp** *t*, **true**);

boolEQ(*f* $\Longleftarrow$ *f*, **true**);
boolEQ(*f* $\Longleftarrow$ *t*, **false**);
boolEQ(*t* $\Longleftarrow$ *f*, **true**);
boolEQ(*t* $\Longleftarrow$ *t*, **true**);

2.8 Assignment

b: *Boolean*;

b := *f*; *b* **and**= *f*; *boolEQ*(*b*, **false**);
b := *f*; *b* **and**= *t*; *boolEQ*(*b*, **false**);
b := *t*; *b* **and**= *f*; *boolEQ*(*b*, **false**);
b := *t*; *b* **and**= *t*; *boolEQ*(*b*, **true**);

b := *f*; *b* **nand**= *f*; *boolEQ*(*b*, **true**);
b := *f*; *b* **nand**= *t*; *boolEQ*(*b*, **true**);
b := *t*; *b* **nand**= *f*; *boolEQ*(*b*, **true**);
b := *t*; *b* **nand**= *t*; *boolEQ*(*b*, **false**);

b := *f*; *b* **or**= *f*; *boolEQ*(*b*, **false**);
b := *f*; *b* **or**= *t*; *boolEQ*(*b*, **true**);

$b := t;\quad b$ **or**$= f;\quad boolEQ(b,$ **true**$);$
$b := t;\quad b$ **or**$= t;\quad boolEQ(b,$ **true**$);$

$b := f;\quad b$ **nor**$= f;\quad boolEQ(b,$ **true**$);$
$b := f;\quad b$ **nor**$= t;\quad boolEQ(b,$ **false**$);$
$b := t;\quad b$ **nor**$= f;\quad boolEQ(b,$ **false**$);$
$b := t;\quad b$ **nor**$= t;\quad boolEQ(b,$ **false**$);$

$b := f;\quad b$ **xor**$= f;\quad boolEQ(b,$ **false**$);$
$b := f;\quad b$ **xor**$= t;\quad boolEQ(b,$ **true**$);$
$b := t;\quad b$ **xor**$= f;\quad boolEQ(b,$ **true**$);$
$b := t;\quad b$ **xor**$= t;\quad boolEQ(b,$ **false**$);$

$b := f;\quad b$ **implies**$= f;\quad boolEQ(b,$ **true**$);$
$b := f;\quad b$ **implies**$= t;\quad boolEQ(b,$ **true**$);$
$b := t;\quad b$ **implies**$= f;\quad boolEQ(b,$ **false**$);$
$b := t;\quad b$ **implies**$= t;\quad boolEQ(b,$ **true**$);$

$b := f;\quad b$ **revimp**$= f;\ boolEQ(b,$ **true**$);$
$b := f;\quad b$ **revimp**$= t;\ boolEQ(b,$ **false**$);$
$b := t;\quad b$ **revimp**$= f;\ boolEQ(b,$ **true**$);$
$b := t;\quad b$ **revimp**$= t;\ boolEQ(b,$ **true**$);$

2.9 Expressions

We employ logical identities to exercise the evaluation of expressions. The identities below are taken from the Desi user manual.

$boolEQ($**not**$(f$ **or** $f) \iff$ **not** f **and not** $f,$ **true**$);$
$boolEQ($**not**$(f$ **or** $t) \iff$ **not** f **and not** $t,$ **true**$);$
$boolEQ($**not**$(t$ **or** $f) \iff$ **not** t **and not** $f,$ **true**$);$
$boolEQ($**not**$(t$ **or** $t) \iff$ **not** t **and not** $t,$ **true**$);$

$boolEQ($**not**$(f$ **and** $f) \iff$ **not** f **or not** $f,$ **true**$);$
$boolEQ($**not**$(f$ **and** $t) \iff$ **not** f **or not** $t,$ **true**$);$
$boolEQ($**not**$(t$ **and** $f) \iff$ **not** t **or not** $f,$ **true**$);$
$boolEQ($**not**$(t$ **and** $t) \iff$ **not** t **or not** $t,$ **true**$);$

$boolEQ($**not**$(f$ **or** $f) \iff f$ **nor** $f,$ **true**$);$
$boolEQ($**not**$(f$ **or** $t) \iff f$ **nor** $t,$ **true**$);$
$boolEQ($**not**$(t$ **or** $f) \iff t$ **nor** $f,$ **true**$);$
$boolEQ($**not**$(t$ **or** $t) \iff t$ **nor** $t,$ **true**$);$

$boolEQ($**not**$(f$ **and** $f) \iff f$ **nand** $f,$ **true**$);$
$boolEQ($**not**$(f$ **and** $t) \iff f$ **nand** $t,$ **true**$);$
$boolEQ($**not**$(t$ **and** $f) \iff t$ **nand** $f,$ **true**$);$
$boolEQ($**not**$(t$ **and** $t) \iff t$ **nand** $t,$ **true**$);$

$boolEQ((f \iff f) \iff f = f,$ **true**$);$
$boolEQ((f \iff t) \iff f = t,$ **true**$);$
$boolEQ((t \iff f) \iff t = f,$ **true**$);$
$boolEQ((t \iff t) \iff t = t,$ **true**$);$

boolEQ(*f* **xor** *f* $\Longleftrightarrow$ *f* <> *f*, **true**);
boolEQ(*f* **xor** *t* $\Longleftrightarrow$ *f* <> *t*, **true**);
boolEQ(*t* **xor** *f* $\Longleftrightarrow$ *t* <> *f*, **true**);
boolEQ(*t* **xor** *t* $\Longleftrightarrow$ *t* <> *t*, **true**);

boolEQ(*f* **implies** *f* $\Longleftrightarrow$ **not** *f* **or** *f*, **true**);
boolEQ(*f* **implies** *t* $\Longleftrightarrow$ **not** *f* **or** *t*, **true**);
boolEQ(*t* **implies** *f* $\Longleftrightarrow$ **not** *t* **or** *f*, **true**);
boolEQ(*t* **implies** *t* $\Longleftrightarrow$ **not** *t* **or** *t*, **true**);

boolEQ(*f* **revimp** *f* $\Longleftrightarrow$ *f* **or not** *f*, **true**);
boolEQ(*f* **revimp** *t* $\Longleftrightarrow$ *f* **or not** *t*, **true**);
boolEQ(*t* **revimp** *f* $\Longleftrightarrow$ *t* **or not** *f*, **true**);
boolEQ(*t* **revimp** *t* $\Longleftrightarrow$ *t* **or not** *t*, **true**);

boolEQ(*f* **and** *f* $\Longleftrightarrow$ (*f* **if** *f* **else false**), **true**);
boolEQ(*f* **and** *t* $\Longleftrightarrow$ (*t* **if** *f* **else false**), **true**);
boolEQ(*t* **and** *f* $\Longleftrightarrow$ (*f* **if** *t* **else false**), **true**);
boolEQ(*t* **and** *t* $\Longleftrightarrow$ (*t* **if** *t* **else false**), **true**);

boolEQ(*f* **nand** *f* $\Longleftrightarrow$ (**not** *f* **if** *f* **else true**), **true**);
boolEQ(*f* **nand** *t* $\Longleftrightarrow$ (**not** *t* **if** *f* **else true**), **true**);
boolEQ(*t* **nand** *f* $\Longleftrightarrow$ (**not** *f* **if** *t* **else true**), **true**);
boolEQ(*t* **nand** *t* $\Longleftrightarrow$ (**not** *t* **if** *t* **else true**), **true**);

boolEQ(*f* **or** *f* $\Longleftrightarrow$ (**true if** *f* **else** *f*), **true**);
boolEQ(*f* **or** *t* $\Longleftrightarrow$ (**true if** *f* **else** *t*), **true**);
boolEQ(*t* **or** *f* $\Longleftrightarrow$ (**true if** *t* **else** *f*), **true**);
boolEQ(*t* **or** *t* $\Longleftrightarrow$ (**true if** *t* **else** *t*), **true**);

boolEQ(*f* **implies** *f* $\Longleftrightarrow$ (**true if not** *f* **else** *f*), **true**);
boolEQ(*f* **implies** *t* $\Longleftrightarrow$ (**true if not** *f* **else** *t*), **true**);
boolEQ(*t* **implies** *f* $\Longleftrightarrow$ (**true if not** *t* **else** *f*), **true**);
boolEQ(*t* **implies** *t* $\Longleftrightarrow$ (**true if not** *t* **else** *t*), **true**);

boolEQ(*f* **revimp** *f* $\Longleftrightarrow$ (**true if** *f* **else not** *f*), **true**);
boolEQ(*f* **revimp** *t* $\Longleftrightarrow$ (**true if** *f* **else not** *t*), **true**);
boolEQ(*t* **revimp** *f* $\Longleftrightarrow$ (**true if** *t* **else not** *f*), **true**);
boolEQ(*t* **revimp** *t* $\Longleftrightarrow$ (**true if** *t* **else not** *t*), **true**);

2.10 Coercions

These tests exercise coercions from, rather than to, Booleans.

i32EQ(*Word*(*f*), 0);
i32EQ(*Word*(*t*), 1);
textEQ(*Text*(*f*), " false");
textEQ(*Text*(*t*), "true");

2.11 Communication

These tests exercise transfer of Booleans between processes.

```
x: −BoolExchange;
OneShotContradictionService(x);
x.given := true;
r: Boolean := x.result;
boolEQ(r, false);
```

2.12 Routine test

```
y, z: Boolean;
Swapper(true, false → y, z);
boolEQ(y, false);
boolEQ(z, true);
```

2.13 Prologue

```
    endTest();
}

main();
```

3 Bytes

```
beginTestT = routine t: Text;                          external "C", "beginTestT";
i8EQ       = routine c: Byte; v: Byte;                 external "C", "i8EQ";
i32EQ      = routine c: Word; v: Word;                 external "C", "i32EQ";
r64EQ      = routine c: Real; v: Real;                 external "C", "r64EQ";
boolEQ     = routine c: Boolean; v: Boolean;           external "C", "boolEQ";
textEQ     = routine c: Text; v: Text;                 external "C", "textEQ";
endTest    = routine                                   external "C", "endTest";
```

3.1 Process for communication tests

```
ByteExchange = protocol { given: Byte; ↑result: Byte }

ByteDoublingService = process p: +ByteExchange; {
  p. result := 2 * p.given;
}
```

3.2 Routine for routine tests

```
Swapper = routine p, q: Byte → t, v: Byte; {
    i8EQ(t, 0);
    i8EQ(v, 0);
    t := q;   v := p;
}
```

3.3 Main cell

```
main = cell {
    beginTestT("Bytes");
```

3.4 Initialisation

```
b: Byte;   i8EQ(b, 0);
four: Byte := 4;   five: Byte := 5;   six: Byte := 6;
```

3.5 Unary operation

Although bytes may be negated and inverted, they are considered to be unsigned.

```
i8EQ(+5, 5);
i8EQ(+five, 5);
i8EQ(−5, 251);
i8EQ(−five, 251);
i8EQ(~5, 250);
i8EQ(~five, 250);
```

3.6 Arithmetic operations

$i8EQ(5+6,\ 11);$
$i8EQ(5+six,\ 11);$
$i8EQ(five+6,\ 11);$
$i8EQ(five+six,\ 11);$

$i8EQ(6-5,\ 1);$
$i8EQ(six-5,\ 1);$
$i8EQ(6-five,\ 1);$
$i8EQ(six-five,\ 1);$

$i8EQ(5*6,\ 30);$
$i8EQ(5*six,\ 30);$
$i8EQ(five*6,\ 30);$
$i8EQ(five*six,\ 30);$

$i8EQ(6/5,\ 1);$
$i8EQ(six/5,\ 1);$
$i8EQ(6/five,\ 1);$
$i8EQ(six/five,\ 1);$

$i8EQ(6\%4,\ 2);$
$i8EQ(six\%4,\ 2);$
$i8EQ(6\%four,\ 2);$
$i8EQ(six\%four,\ 2);$

3.7 Comparisons

$boolEQ(5 < 6,\ \mathbf{true});$
$boolEQ(5 < six,\ \mathbf{true});$
$boolEQ(five < 6,\ \mathbf{true});$
$boolEQ(five < six,\ \mathbf{true});$

$boolEQ(five < five,\ \mathbf{false});$
$boolEQ(five < six,\ \mathbf{true});$
$boolEQ(six < five,\ \mathbf{false});$
$boolEQ(six < six,\ \mathbf{false});$

$boolEQ(five \leq five,\ \mathbf{true});$
$boolEQ(five \leq six,\ \mathbf{true});$
$boolEQ(six \leq five,\ \mathbf{false});$
$boolEQ(six \leq six,\ \mathbf{true});$

$boolEQ(five > five,\ \mathbf{false});$
$boolEQ(five > six,\ \mathbf{false});$
$boolEQ(six > five,\ \mathbf{true});$
$boolEQ(six > six,\ \mathbf{false});$

$boolEQ(five \geq five,\ \mathbf{true});$
$boolEQ(five \geq six,\ \mathbf{false});$
$boolEQ(six \geq five,\ \mathbf{true});$

```
boolEQ(six >= six, true);

boolEQ(five = five, true);
boolEQ(five = six, false);
boolEQ(six = five, false);
boolEQ(six = six, true);

boolEQ(five <> five, false);
boolEQ(five <> six, true);
boolEQ(six <> five, true);
boolEQ(six <> six, false);
```

3.8 Bits

```
cc: Byte := 0xCC; aa: Byte := 0xAA;
i8EQ(cc andb aa, 0x88);
i8EQ(cc orb aa, 0xEE);
i8EQ(cc xorb aa, 0x66);
i8EQ(cc impliesb aa, 0xBB);
i8EQ(cc nandb aa, 0x77);
i8EQ(cc norb aa, 0x11);
i8EQ(cc revimpb aa, 0x44);
```

3.9 Shifts

Because bytes are considered to be unsigned an arithmetical shift to the right produces the same effect as a logical shift to the right.

```
i8EQ(aa << 5, 0x40);
i8EQ(aa << five, 0x40);
i8EQ(aa >> 5, 0x05);
i8EQ(aa >> five, 0x05);
i8EQ(aa >>> 5, 0x05);
i8EQ(aa >>> five, 0x05);
```

3.10 Assignment

```
b := 5;   b += 6; i8EQ(b, 11);
b := six;   b -= 5; i8EQ(b, 1);
b := 5;   b *= six;   i8EQ(b, 30);
b := six;   b /= 5;   i8EQ(b, 1);
b := 6;   b %= four; i8EQ(b, 2);

b := cc;   b andb= aa; i8EQ(b, 0x88);
b := cc;   b orb= aa; i8EQ(b, 0xEE);
b := cc;   b xorb= aa; i8EQ(b, 0x66);
b := cc;   b impliesb= aa; i8EQ(b, 0xBB);
b := cc;   b nandb= aa; i8EQ(b, 0x77);
b := cc;   b norb= aa; i8EQ(b, 0x11);
b := cc;   b revimpb= aa; i8EQ(b, 0x44);

b := aa;   b <<= 5; i8EQ(b, 0x40);
```

$b := aa$; $b >>= five$; $i8EQ(b, 0x05)$;
$b := aa$; $b >>>= 5$; $i8EQ(b, 0x05)$;

3.11 Coercions

These tests exercise coercions from, rather than to, Bytes.

$i32EQ(Word(aa), 0xAA)$;
$r64EQ(Real(aa), 170.0)$;
$textEQ(Text(aa), "170")$;

3.12 Operations that overflow

We check that byte operands act like unsigned words constrained modulo 256. In this release we do not check for byte overflow. In addition to the tests below, consider also the left-shifts above.[1]

$i32EQ(Word(Byte(250)+Byte(7)), 1)$; $--$ *Should fault*
$i32EQ(Word(Byte(250)*Byte(7)), 214)$; $--$ *Should fault*

3.13 Communication test

x: $-ByteExchange$;
$ByteDoublingService(x)$;
$x.given := 2$;
r: $Byte := x.result$;
$i8EQ(r, 4)$;

3.14 Routine test

y, z: $Byte$;
$Swapper(2, 3 \rightarrow y, z)$;
$i8EQ(y, 3)$;
$i8EQ(z, 2)$;

3.15 Prologue

End of test, and launch of main process.

$endTest()$;
}

$main()$;

[1] Desi may crash your aeroplane, nuclear plant, or economy, but relax! It won't crash your computer. ;)

4 Words

```
beginTestT = routine t: Text;                          external "C", "beginTestT";
i8EQ       = routine c: Byte; v: Byte;                 external "C", "i8EQ";
i32EQ      = routine c: Word; v: Word;                 external "C", "i32EQ";
r64EQ      = routine c: Real; v: Real;                 external "C", "r64EQ";
textEQ     = routine c: Text; v: Text;                 external "C", "textEQ";
boolEQ     = routine c: Boolean; v: Boolean;           external "C", "boolEQ";
endTest    = routine                                   external "C", "endTest";
```

4.1 Processes for communication tests

Process *Waver* receives and sends a nominated number of signals. This test will become more rigorous when conditional communication is available.

```
Signal = protocol { signal }

Waver = process input: +Signal; output: −Signal; n: Word; {
   i32EQ(0, 0);
   for (i := 0; i <> n; i += 1) {
      input. signal;
      output. signal;
   }
   i32EQ(n, n);
}

WordExchange = protocol { given: Word; ↑result: Word }

WordDoublingService = process p: +WordExchange; {
  p. result := 2 * p.given;
}
```

Routine ChannelBouncer is used to check passing of channels into and out of routines.

```
--  ChannelBouncer = routine input: −WordExchange → output: −WordExchange {
--      output := input;
--  }
```

4.2 Routine for routine tests

```
Swapper = routine p, q: Word → t, v: Word; {
  i32EQ(t, 0);
  i32EQ(v, 0);
  t := q; v := p;
}
```

4.3 Main cell

```
main = cell {
  beginTestT("Words");
```

4.4 Initialisation

w: *Word*; *i32EQ*(*w*, 0);
assert $w = 0$, " Initialisation failure ";

4.5 Unary operations

i32EQ(−1, 0*xFFFFFFFF*);
i32EQ(˜0*xAAAAAAAA*, 0*x55555555*);

4.6 Arithmetic

four: *Word* := 4; *five*: *Word* := 5; *six*: *Word* := 6;
i32EQ(5 + 6, 11);
i32EQ(5 + *six*, 11);
i32EQ(*five* + 6, 11);
i32EQ(*five* + *six*, 11);

i32EQ(6 − 5, 1);
i32EQ(*six* − 5, 1);
i32EQ(6 − *five*, 1);
i32EQ(*six* − *five*, 1);
i32EQ(5 − 6, −1);
i32EQ(5 − *six*, −1);
i32EQ(*five* − 6, −1);
i32EQ(*five* − *six*, −1);

i32EQ(5 ∗ 6, 30);
i32EQ(5 ∗ *six*, 30);
i32EQ(*five* ∗ 6, 30);
i32EQ(*five* ∗ *six*, 30);

i32EQ(6 / 5, 1);
i32EQ(*six* / 5, 1);
i32EQ(6 / *five*, 1);
i32EQ(*six* / *five*, 1);

i32EQ(6 % 4, 2);
i32EQ(*six* % 4, 2);
i32EQ(6 % *four*, 2);
i32EQ(*six* % *four*, 2);

4.7 Comparisons

boolEQ(5 = 6, **false**);
boolEQ(5 = *six*, **false**);
boolEQ(*five* = 6, **false**);
boolEQ(*five* = *six*, **false**);
boolEQ(6 = 5, **false**);
boolEQ(6 = 6, **true**);
boolEQ(5 = 5, **true**);
boolEQ(5 = *five*, **true**);

$boolEQ(five = five,\ \mathbf{true});$
$boolEQ(five = 5,\ \mathbf{true});$

$boolEQ(5 <> 6,\ \mathbf{true});$
$boolEQ(5 <> six,\ \mathbf{true});$
$boolEQ(five <> 6,\ \mathbf{true});$
$boolEQ(five <> six,\ \mathbf{true});$
$boolEQ(6 <> 5,\ \mathbf{true});$
$boolEQ(6 <> 6,\ \mathbf{false});$
$boolEQ(5 <> 5,\ \mathbf{false});$
$boolEQ(5 <> five,\ \mathbf{false});$
$boolEQ(five <> five,\ \mathbf{false});$
$boolEQ(five <> 5,\ \mathbf{false});$

$boolEQ(5 < 6,\ \mathbf{true});$
$boolEQ(5 < six,\ \mathbf{true});$
$boolEQ(five < 6,\ \mathbf{true});$
$boolEQ(five < six,\ \mathbf{true});$
$boolEQ(6 < 5,\ \mathbf{false});$
$boolEQ(6 < 6,\ \mathbf{false});$

$boolEQ(5 \geq 6,\ \mathbf{false});$
$boolEQ(5 \geq six,\ \mathbf{false});$
$boolEQ(five \geq 6,\ \mathbf{false});$
$boolEQ(five \geq six,\ \mathbf{false});$
$boolEQ(6 \geq 5,\ \mathbf{true});$
$boolEQ(6 \geq 6,\ \mathbf{true});$

$boolEQ(5 \leq 6,\ \mathbf{true});$
$boolEQ(5 \leq six,\ \mathbf{true});$
$boolEQ(five \leq 6,\ \mathbf{true});$
$boolEQ(five \leq six,\ \mathbf{true});$
$boolEQ(6 \leq 5,\ \mathbf{false});$
$boolEQ(6 \leq 6,\ \mathbf{true});$
$boolEQ(5 \leq 5,\ \mathbf{true});$
$boolEQ(5 \leq five,\ \mathbf{true});$
$boolEQ(five \leq five,\ \mathbf{true});$
$boolEQ(five \leq 5,\ \mathbf{true});$

$boolEQ(5 > 6,\ \mathbf{false});$
$boolEQ(5 > six,\ \mathbf{false});$
$boolEQ(five > 6,\ \mathbf{false});$
$boolEQ(five > six,\ \mathbf{false});$
$boolEQ(6 > 5,\ \mathbf{true});$
$boolEQ(6 > 6,\ \mathbf{false});$
$boolEQ(5 > 5,\ \mathbf{false});$
$boolEQ(5 > five,\ \mathbf{false});$
$boolEQ(five > five,\ \mathbf{false});$
$boolEQ(five > 5,\ \mathbf{false});$

4.8 Bits

aa: *Word* := 0*xAAAAAAAA*; *cc*: *Word* := 0*xCCCCCCCC*;
i32EQ(*cc* **andb** *aa*, 0*x88888888*);
i32EQ(*cc* **orb** *aa*, 0*xEEEEEEEE*);
i32EQ(*cc* **xorb** *aa*, 0*x66666666*);
i32EQ(*cc* **impliesb** *aa*, 0*xBBBBBBBB*);
i32EQ(*cc* **nandb** *aa*, 0*x77777777*);
i32EQ(*cc* **norb** *aa*, 0*x11111111*);
i32EQ(*cc* **revimpb** *aa*, 0*x44444444*);

4.9 Shifts

i32EQ(*aa* << 5, 0*x55555540*);
i32EQ(*aa* << *five*, 0*x55555540*);
i32EQ(*aa* >> 5, 0*xFD555555*);
i32EQ(*aa* >> *five*, 0*xFD555555*);
i32EQ(*aa* >>> 5, 0*x05555555*);
i32EQ(*aa* >>> *five*, 0*x05555555*);

4.10 Assignments

w := 5; *w* += 6; *i32EQ*(*w*, 11);
w := *six*; *w* −= 5; *i32EQ*(*w*, 1);
w := 5; *w* ∗= *six*; *i32EQ*(*w*, 30);
w := *six*; *w* /= 5; *i32EQ*(*w*, 1);
w := 6; *w* %= *four*; *i32EQ*(*w*, 2);

w := *cc*; *w* **andb**= *aa*; *i32EQ*(*w*, 0*x88888888*);
w := *cc*; *w* **orb**= *aa*; *i32EQ*(*w*, 0*xEEEEEEEE*);
w := *cc*; *w* **xorb**= *aa*; *i32EQ*(*w*, 0*x66666666*);
w := *cc*; *w* **impliesb**= *aa*; *i32EQ*(*w*, 0*xBBBBBBBB*);
w := *cc*; *w* **nandb**= *aa*; *i32EQ*(*w*, 0*x77777777*);
w := *cc*; *w* **norb**= *aa*; *i32EQ*(*w*, 0*x11111111*);
w := *cc*; *w* **revimpb**= *aa*; *i32EQ*(*w*, 0*x44444444*);

w := *aa*; *w* <<= 5; *i32EQ*(*w*, 0*x55555540*);
w := *aa*; *w* >>= *five*; *i32EQ*(*w*, 0*xFD555555*);
w := *aa*; *w* >>>= 5; *i32EQ*(*w*, 0*x05555555*);

4.11 Expressions

i32EQ((*four* + 5) ∗ 3, 27);
i32EQ((*four* − 6) ∗ 3, −*six*);
i32EQ((*four* + 5) ∗ (*five* + 6), 99);
i32EQ((*four* + 5) ∗ (*five* + 6) / 9, 11);
i32EQ((*four* + 5) ∗ (*five* + 6) % *six*, 3);
i32EQ((*four* + 5 ∗ *six*) / (4 + *five*), 3);

i32EQ(25 **if** 5 = 6 **else** 26, 26);
i32EQ(25 **if** 5 <> 6 **else** 26, 25);
i32EQ(25 **if** 5 > 6 **else** 26, 26);
i32EQ(25 **if** 5 ≤ 6 **else** 26, 25);
i32EQ(25 **if** 5 ≥ 6 **else** 26, 26);
i32EQ(25 **if** 5 < 6 **else** 26, 25);

4.12 Coercions

These tests exercise coercions from, rather than to, Words

```
i8EQ(Byte(0), 0);
i8EQ(Byte(255), 255);
i8EQ(Byte(−1), 255); −− Should fault
i8EQ(Byte(256), 0);  −− Should fault

r64EQ(Real(0), 0.0);
r64EQ(Real(1), 1.0);
r64EQ(Real(−1), −1.0);
r64EQ(Real(1234), 1234.0);
r64EQ(Real(0x80000000), −2147483648.0);

textEQ(Text(0), "0");
textEQ(Text(1), "1");
textEQ(Text(−1), "−1");
textEQ(Text(1234), "1234");
textEQ(Text(0x80000000), "−2147483648");
```

4.13 Communication tests

```
output: −Signal;
input: +Signal;
Waver(output, input, 5);
for (i := 0;  i <> 5;  i += 1) {
    output.signal;
    input.signal;
}

x: −WordExchange;
WordDoublingService(x);
x.given := 2;
i32EQ(x.result, 4);
```

4.14 Routine test

```
y, z: Word;
Swapper(2, 3 → y, z);
i32EQ(y, 3);
i32EQ(z, 2);
```

4.15 Prologue

End of test, and launch of main process.

```
    endTest();
}

main();
```

5 Reals

```
beginTestT = routine t: Text;                          external "C", "beginTestT";
i8EQ       = routine c: Byte; v: Byte;                 external "C", "i8EQ";
i32EQ      = routine c: Word; v: Word;                 external "C", "i32EQ";
r64EQ      = routine c: Real; v: Real;                 external "C", "r64EQ";
textEQ     = routine c: Text; v: Text;                 external "C", "textEQ";
boolEQ     = routine c: Boolean; v: Boolean;           external "C", "boolEQ";
endTest    = routine                                   external "C", "endTest";
```

5.1 Process for communication tests

```
RealExchange = protocol { given: Real; ↑result: Real }

RealDoublingService = process p: +RealExchange; {
    p.result := 2 * p.given;
}
```

5.2 Routine for routine tests

```
Swapper = routine p, q: Real → t, v: Real; {
    r64EQ(t, 0.0);
    r64EQ(v, 0.0);
    t := q;   v := p;
}
```

5.3 Routine to test real equality

r64ApproxEQ(x,y) tests the equality of x and y to nine significant decimal places. x and y must be strictly positive.

```
r64ApproxEQ = routine x, y: Real; {
    assert x > 0.0 and y > 0.0,
            "x and y in r64ApproxEQ(x,y) must be strictly positive";
    loop { |x ≤ 0.1| x *= 10.0;   y *= 10.0 }
    loop { |x > 1.0| x /= 10.0;   y /= 10.0 }
    assert 0.1 < x and x ≤ 1.0,
            "x in r64ApproxEQ(x,y) failed to scale";
    for ( i := 0;   i <> 9; i += 1) { x *= 10.0; y *= 10.0 }
    i32EQ(round(x), round(y))
}
```

The tests that check real library routines employ various well-known constants.

```
e = constant 2.718281828459045;
pi = constant 3.141592653589793;
root2 = constant 1.414213562373095;
sinOfRadian = constant 0.841470984807896;
cosOfRadian = constant 0.540302305868139;
```

5.4 Main cell

main = **cell** {
beginTestT("Reals");

5.5 Initialisation

r: *Real*; *r64EQ*(*r*, 0.0);

5.6 Arithmetic

four: *Real* := 4.0; *fiv*: *Real* := 5.0; *six*: *Real* := 6.5;
r64EQ(6.5+5.0, 11.5);
r64EQ(6.5+*fiv*, 11.5);
r64EQ(*six*+5.0, 11.5);
r64EQ(*six*+*fiv*, 11.5);

r64EQ(5.0−6.5, −1.5);
r64EQ(*fiv*−6.5, −1.5);
r64EQ(5.0−*six*, −1.5);
r64EQ(*fiv*−*six*, −1.5);

r64EQ(6.5∗5.0, 32.5);
r64EQ(6.5∗*fiv*, 32.5);
r64EQ(*six*∗5.0, 32.5);
r64EQ(*six*∗*fiv*, 32.5);

r64EQ(6.5/5.0, 1.3);
r64EQ(6.5/*fiv*, 1.3);
r64EQ(*six*/5.0, 1.3);
r64EQ(*six*/*fiv*, 1.3);

r64EQ(6.5%5.0, 1.5);
r64EQ(6.5%*fiv*, 1.5);
r64EQ(*six*%5.0, 1.5);
r64EQ(*six*%*fiv*, 1.5);

5.7 Comparisons

boolEQ(5.0 = 6.5, **false**);
boolEQ(5.0 = *six*, **false**);
boolEQ(*fiv* = 6.5, **false**);
boolEQ(*fiv* = *six*, **false**);
boolEQ(6.5 = 5.0, **false**);
boolEQ(6.5 = 6.5, **true**);
boolEQ(5.0 = 5.0, **true**);
boolEQ(5.0 = *fiv*, **true**);
boolEQ(*fiv* = *fiv*, **true**);
boolEQ(*fiv* = 5.0, **true**);

boolEQ(5.0 <> 6.5, **true**);
boolEQ(5.0 <> *six*, **true**);

$boolEQ(fiv <> 6.5, \textbf{true});$
$boolEQ(fiv <> six, \textbf{true});$
$boolEQ(6.5 <> 5.0, \textbf{true});$
$boolEQ(6.5 <> 6.5, \textbf{false});$
$boolEQ(5.0 <> 5.0, \textbf{false});$
$boolEQ(5.0 <> fiv, \textbf{false});$
$boolEQ(fiv <> fiv, \textbf{false});$
$boolEQ(fiv <> 5.0, \textbf{false});$

$boolEQ(5.0 < 6.5, \textbf{true});$
$boolEQ(5.0 < six, \textbf{true});$
$boolEQ(fiv < 6.5, \textbf{true});$
$boolEQ(fiv < six, \textbf{true});$
$boolEQ(6.5 < 5.0, \textbf{false});$
$boolEQ(6.5 < 6.5, \textbf{false});$

$boolEQ(5.0 \geq 6.5, \textbf{false});$
$boolEQ(5.0 \geq six, \textbf{false});$
$boolEQ(fiv \geq 6.5, \textbf{false});$
$boolEQ(fiv \geq six, \textbf{false});$
$boolEQ(6.5 \geq 5.0, \textbf{true});$
$boolEQ(6.5 \geq 6.5, \textbf{true});$

$boolEQ(5.0 \leq 6.5, \textbf{true});$
$boolEQ(5.0 \leq six, \textbf{true});$
$boolEQ(fiv \leq 6.5, \textbf{true});$
$boolEQ(fiv \leq six, \textbf{true});$
$boolEQ(6.5 \leq 5.0, \textbf{false});$
$boolEQ(6.5 \leq 6.5, \textbf{true});$
$boolEQ(5.0 \leq 5.0, \textbf{true});$
$boolEQ(5.0 \leq fiv, \textbf{true});$
$boolEQ(fiv \leq fiv, \textbf{true});$
$boolEQ(fiv \leq 5.0, \textbf{true});$

$boolEQ(5.0 > 6.5, \textbf{false});$
$boolEQ(5.0 > six, \textbf{false});$
$boolEQ(fiv > 6.5, \textbf{false});$
$boolEQ(fiv > six, \textbf{false});$
$boolEQ(6.5 > 5.0, \textbf{true});$
$boolEQ(6.5 > 6.5, \textbf{false});$
$boolEQ(5.0 > 5.0, \textbf{false});$
$boolEQ(5.0 > fiv, \textbf{false});$
$boolEQ(fiv > fiv, \textbf{false});$
$boolEQ(fiv > 5.0, \textbf{false});$

5.8 Expressions

$r64EQ((four + 5.25) * 3.0, 27.75);$
$r64EQ((four - 6.0) * 3.0, -6.0);$
$r64EQ((four + 5.0) * (fiv + 6.0), 99.0);$
$r64EQ((four + 5.0) * (fiv + 6.0) / 9.0, 11.0);$
$r64EQ(((four + 5.0) * (fiv + 6.0) + 1.0) \% fiv, 0.0);$

r64EQ((*four* + 5.0 ∗ (*six* − 0.5) − 1.0) / (6.0 + *fiv*), 3.0);

i32EQ(25 **if** 5.0 = 6.5 **else** 26, 26);
i32EQ(25 **if** 5.0 <> 6.5 **else** 26, 25);
i32EQ(25 **if** 5.0 > 6.5 **else** 26, 26);
i32EQ(25 **if** 5.0 ≤ 6.5 **else** 26, 25);
i32EQ(25 **if** 5.0 ≥ 6.5 **else** 26, 26);
i32EQ(25 **if** 5.0 < 6.5 **else** 26, 25);

5.9 Precision

The following checks confirm that 64-bit reals are being employed, which provide about fifteen significant decimal places of precision.

boolEQ(1.0*e6* + 0.5 = 1000000.5, **true**);
boolEQ(1.0*e7* + 0.5 = 10000000.5, **true**);
boolEQ(1.0*e8* + 0.5 = 100000000.5, **true**);
boolEQ(1.0*e9* + 0.5 = 1000000000.5, **true**);
boolEQ(1.0*e12* + 0.5 = 1000000000000.5, **true**);
boolEQ(1.0*e13* + 0.5 = 10000000000000.5, **true**);
boolEQ(1.0*e14* + 0.5 = 100000000000000.5, **true**);
boolEQ(1.0*e15* + 0.5 = 1000000000000000.5, **false**);

5.10 Coercions

These tests exercise coercions from, rather than to, Reals

i32EQ(*floor*(0.0), 0);
i32EQ(*round*(0.0), 0);
i32EQ(*ceil*(0.0), 0);

i32EQ(*floor*(1.0), 1);
i32EQ(*round*(1.0), 1);
i32EQ(*ceil*(1.0), 1);

i32EQ(*floor*(−1.0), −1);
i32EQ(*round*(−1.0), −1);
i32EQ(*ceil*(−1.0), −1);

i32EQ(*floor*(1.5), 1);
i32EQ(*round*(1.5), 2);
i32EQ(*ceil*(1.4), 2);
i32EQ(*ceil*(1.5), 2);

i32EQ(*floor*(−1.5), −2);
i32EQ(*round*(−1.5), −2);
i32EQ(*ceil*(−1.4), −1);
i32EQ(*ceil*(−1.5), −1);

We encode a real as follows:

- A real that is a whole number and whose magnitude is no greater than fifteen digits is encoded without an exponent in a form such as 123456789012345.0

- A real with no more than fifteen significant digits and whose digits span the decimal point is encoded without an exponent in a form such as 1234567.89012345
- Other reals are encoded in an exponent form to fifteen decimal places, as in 1.23456789012345e-003

textEQ(*Text*(0.0), "0.0");
textEQ(*Text*(−1.0), "−1.0");
textEQ(*Text*(12.0), "12.0");
textEQ(*Text*(−123.0), "−123.0");
textEQ(*Text*(1234.0), "1234.0");
textEQ(*Text*(−12345.0), "−12345.0");
textEQ(*Text*(123456.0), "123456.0");
textEQ(*Text*(−1234567.0), "−1234567.0");
textEQ(*Text*(12345678.0), "12345678.0");
textEQ(*Text*(−123456789.0), "−123456789.0");
textEQ(*Text*(1234567890.0), "1234567890.0");
textEQ(*Text*(−12345678901.0), "−12345678901.0");
textEQ(*Text*(123456789012.0), "123456789012.0");
textEQ(*Text*(−1234567890123.0), "−1234567890123.0");
textEQ(*Text*(12345678901234.0), "12345678901234.0");
textEQ(*Text*(−123456789012345.0), "−123456789012345.0");

textEQ(*Text*(12345678901234.5), "12345678901234.5");
textEQ(*Text*(−1234567890123.45), "−1234567890123.45");
textEQ(*Text*(123456789012.345), "123456789012.345");
textEQ(*Text*(−12345678901.2345), "−12345678901.2345");
textEQ(*Text*(1234567890.12345), "1234567890.12345");
textEQ(*Text*(−123456789.012345), "−123456789.012345");
textEQ(*Text*(12345678.9012345), "12345678.9012345");
textEQ(*Text*(−1234567.89012345), "−1234567.89012345");
textEQ(*Text*(123456.789012345), "123456.789012345");
textEQ(*Text*(−12345.6789012345), "−12345.6789012345");
textEQ(*Text*(1234.56789012345), "1234.56789012345");
textEQ(*Text*(−123.456789012345), "−123.456789012345");
textEQ(*Text*(12.3456789012345), "12.3456789012345");
textEQ(*Text*(−1.23456789012345), "−1.23456789012345");

textEQ(*Text*(1.234567890123456), "1.234567890123456e+000");
textEQ(*Text*(1.0*e*+300),"1.000000000000000e+300");
textEQ(*Text*(−2.5*e*−300),"−2.500000000000000e−300");

5.11 Maths library test

r64EQ(*abs*(−2), 2);
r64EQ(*abs*(−2.5), 2.5);
r64ApproxEQ(*sqrt*(2.0), *root2*);
r64ApproxEQ(*exp*(1.0), *e*);
r64ApproxEQ(*log*(*e*), 1.0);
r64ApproxEQ(*sin*(1.0), *sinOfRadian*);
r64ApproxEQ(*cos*(1.0), *cosOfRadian*);
r64ApproxEQ(*tan*(*pi*/4), 1.0);

```
r64ApproxEQ(atan(1.0), pi/4);
r64ApproxEQ(atan2(1.0, 1.0), pi/4);
r64ApproxEQ(atan2(1.0, 0.0), pi/2);
```

5.12 Random-number generation

```
for (i := 0;  i <> 1000; i += 1) {
    rnd: Word := rand(100);
    boolEQ(rnd ≥ 0 and rnd < 100, true);
}
seed(12345);
r1: Real := rand();
r2: Real := rand();
seed(12345);
r64EQ(rand(), r1);
r64EQ(rand(), r2);
seed(12345);
r64EQ(rand(1.0), r1);
r64EQ(rand(1.0), r2);
```

5.13 Communication test

```
x: −RealExchange;
RealDoublingService(x);
x.given := 2;
res: Real := x.result;
r64EQ(res, 4);
```

5.14 Routine test

```
y, z: Real;
Swapper(2.34, 3.45 → y, z);
r64EQ(y, 3.45);
r64EQ(z, 2.34);
```

5.15 Prologue

End of test, and launch of main process.

```
    endTest();
}

main();
```

6 Texts

```
beginTestT = routine t: Text;                          external "C", "beginTestT";
i8EQ       = routine c: Byte; v: Byte;                 external "C", "i8EQ";
i32EQ      = routine c: Word; v: Word;                 external "C", "i32EQ";
r64EQ      = routine c: Real; v: Real;                 external "C", "r64EQ";
textEQ     = routine c: Text; v: Text;                 external "C", "textEQ";
boolEQ     = routine c: Boolean; v: Boolean;           external "C", "boolEQ";
endTest    = routine                                   external "C", "endTest";
```

6.1 Process for communication tests

```
TextExchange = protocol { given: Text; ↑result: Text }

TextDoublingService = process p: +TextExchange; {
    x: Text := p.given;
    p. result := x // x;
}
```

6.2 Routine for routine tests

```
Swapper = routine p, q: Text → t, v: Text; {
    textEQ(t, "");
    textEQ(v, "");
    t := q;   v := p;
}
```

6.3 Main cell

```
main = cell {
    beginTestT("Texts");
```

6.4 Initialisation

```
    empty: Text;   textEQ(empty, "");
```

6.5 Parameters

```
    abc: Text := "abc";
    textEQ("abc", "abc");
    textEQ(abc, "abc");
```

6.6 Lengths

```
    i32EQ(#"", 0);
    i32EQ(#empty, 0);
    i32EQ(#"abc", 3);
    i32EQ(#abc, 3);
```

6.7 Concatenation

textEQ("abc"//"", "abc");
textEQ(""//"abc", "abc");
i32EQ(#(""//"abc"), 3);
i32EQ(#(""//*abc*), 3);
textEQ("ab"//"c", "abc");
textEQ(*abc*//"de", "abcde");
textEQ("de"//*abc*, "deabc");
textEQ(*abc*//*abc*, "abcabc");

6.8 Selection

textEQ(*abc*[0], "a");
textEQ(*abc*[1], "b");
textEQ(*abc*[2], "c");
textEQ(*abc* [0..0], "");
textEQ(*abc* [0..1], "a");
textEQ(*abc* [0..2], "ab");
textEQ(*abc* [0..3], "abc");
textEQ(*abc* [1..1], "");
textEQ(*abc* [1..2], "b");
textEQ(*abc* [1..3], "bc");
textEQ(*abc* [2..2], "");
textEQ(*abc* [2..3], "c");
textEQ(*abc* [3..3], "");

textEQ("abc"[0], "a");
textEQ("abc"[1], "b");
textEQ("abc"[2], "c");
textEQ("abc" [0..0], "");
textEQ("abc" [0..1], "a");
textEQ("abc" [0..2], "ab");
textEQ("abc" [0..3], "abc");
textEQ("abc" [1..1], "");
textEQ("abc" [1..2], "b");
textEQ("abc" [1..3], "bc");
textEQ("abc" [2..2], "");
textEQ("abc" [2..3], "c");
textEQ("abc" [3..3], "");

6.9 Assignment

abc[0] := ""; *textEQ*(*abc*, "bc");
abc[0] := "a"; *textEQ*(*abc*, "ac");
abc[0] := "ab"; *textEQ*(*abc*, "abc");
abc[1] := ""; *textEQ*(*abc*, "ac");
abc[1] := "b"; *textEQ*(*abc*, "ab");
abc[1] := "bc"; *textEQ*(*abc*, "abc");
abc[2] := ""; *textEQ*(*abc*, "ab");
abc[1] := "bb"; *textEQ*(*abc*, "abb");
abc[2] := "c"; *textEQ*(*abc*, "abc");
abc [0..0] := ""; *textEQ*(*abc*, "abc");
abc [0..0] := "c"; *textEQ*(*abc*, "cabc");

```
abc [0..0]  := "ab";   textEQ(abc, "abcabc");
abc [0..1]  := "";   textEQ(abc, "bcabc");
abc [0..1]  := "a";   textEQ(abc, "acabc");
abc [0..1]  := "ab";   textEQ(abc, "abcabc");
abc [1..3]  := "";   textEQ(abc, "aabc");
abc [1..3]  := "a";   textEQ(abc, "aac");
abc [1..3]  := "bc";   textEQ(abc, "abc");
abc [3..3]  := "def";   textEQ(abc, "abcdef");
abc[7]  := "h";   textEQ(abc, "abcdef h");
```

In the following assignment the replaced region overlaps and extends the original text.

```
abc [6..8]  := "ghi";   textEQ(abc, "abcdefghi");
```

In the following assignments the 'replaced' region is entirely outside the original text, causing blanks to be inserted. Note the third example here, which demonstrates an elegant way to fill a text with a nominated number of blanks.

```
abc[12]  := "";   textEQ(abc, "abcdefghi    ");
abc[18]  := "stu";   textEQ(abc, "abcdefghi          stu");
e: Text;   e[20]  := "";   textEQ(e, "                    ");
```

The following assignments look benign but are tricky to implement *in situ* without employing a temporary buffer—a luxury that text operators take pride in eschewing.

```
abc := "abc";   abc := abc//abc;   textEQ(abc, "abcabc");
abc := "abc";   abc := "abc"//abc;  textEQ(abc, "abcabc");
```

The final example of this section is taken from the Desi documentation.

```
t: Text := "unable";   t [2..2]  := "maintain";  textEQ(t, "unmaintainable");
```

6.10 Comparisons

```
boolEQ("" = "", true);
boolEQ("x" = "", false);
boolEQ("" = "x", false);
boolEQ("x" = "x", true);
boolEQ("x" = "y", false);
boolEQ("y" = "x", false);
boolEQ("xx" = "xx", true);
boolEQ("xx" = "xy", false);
boolEQ("xy" = "xx", false);
boolEQ("xy" = "x", false);
boolEQ("x" = "xy", false);

boolEQ("" <> "", false);
boolEQ("x" <> "", true);
boolEQ("" <> "x", true);
boolEQ("x" <> "x", false);
boolEQ("x" <> "y", true);
boolEQ("y" <> "x", true);
boolEQ("xx" <> "xx", false);
boolEQ("xx" <> "xy", true);
boolEQ("xy" <> "xx", true);
```

boolEQ("xy" <> "x", **true**);
boolEQ("x" <> "xy", **true**);

boolEQ("" < "", **false**);
boolEQ("x" < "", **false**);
boolEQ("" < "x", **true**);
boolEQ("x" < "x", **false**);
boolEQ("x" < "y", **true**);
boolEQ("y" < "x", **false**);
boolEQ("xx" < "xx", **false**);
boolEQ("xx" < "xy", **true**);
boolEQ("xy" < "xx", **false**);
boolEQ("xy" < "x", **false**);
boolEQ("x" < "xy", **true**);

boolEQ("" ≥ "", **true**);
boolEQ("x" ≥ "", **true**);
boolEQ("" ≥ "x", **false**);
boolEQ("x" ≥ "x", **true**);
boolEQ("x" ≥ "y", **false**);
boolEQ("y" ≥ "x", **true**);
boolEQ("xx" ≥ "xx", **true**);
boolEQ("xx" ≥ "xy", **false**);
boolEQ("xy" ≥ "xx", **true**);
boolEQ("xy" ≥ "x", **true**);
boolEQ("x" ≥ "xy", **false**);

boolEQ("" ≤ "", **true**);
boolEQ("x" ≤ "", **false**);
boolEQ("" ≤ "x", **true**);
boolEQ("x" ≤ "x", **true**);
boolEQ("x" ≤ "y", **true**);
boolEQ("y" ≤ "x", **false**);
boolEQ("xx" ≤ "xx", **true**);
boolEQ("xx" ≤ "xy", **true**);
boolEQ("xy" ≤ "xx", **false**);
boolEQ("xy" ≤ "x", **false**);
boolEQ("x" ≤ "xy", **true**);

boolEQ("" > "", **false**);
boolEQ("x" > "", **true**);
boolEQ("" > "x", **false**);
boolEQ("x" > "x", **false**);
boolEQ("x" > "y", **false**);
boolEQ("y" > "x", **true**);
boolEQ("xx" > "xx", **false**);
boolEQ("xx" > "xy", **false**);
boolEQ("xy" > "xx", **true**);
boolEQ("xy" > "x", **true**);
boolEQ("x" > "xy", **false**);

6.11 Conditional expressions

i32EQ(25 **if** "x" = "y" **else** 26, 26);
i32EQ(25 **if** "x" <> "y" **else** 26, 25);
i32EQ(25 **if** "x" > "y" **else** 26, 26);
i32EQ(25 **if** "x" ≤ "y" **else** 26, 25);
i32EQ(25 **if** "x" ≥ "y" **else** 26, 26);
i32EQ(25 **if** "x" < "y" **else** 26, 25);

6.12 Communication test

x: −*TextExchange*;
TextDoublingService(*x*);
x.*given* := "abc";
r: *Text* := *x*. *result*;
textEQ(*r*, "abcabc");

6.13 Routine test

y, *z*: *Text*;
Swapper("abc", "de" → *y*, *z*);
textEQ(*y*, "de");
textEQ(*z*, "abc");

6.14 Prologue

End of test, and launch of main process.

endTest();
}

main();

7 Maps

```
beginTestT = routine t: Text;                          external "C", "beginTestT";
i8EQ       = routine c: Byte; v: Byte;                 external "C", "i8EQ";
i32EQ      = routine c: Word; v: Word;                 external "C", "i32EQ";
r64EQ      = routine c: Real; v: Real;                 external "C", "r64EQ";
textEQ     = routine c: Text; v: Text;                 external "C", "textEQ";
boolEQ     = routine c: Boolean; v: Boolean;           external "C", "boolEQ";
endTest    = routine                                   external "C", "endTest";

WordArray = type Word indexes Word;
```

7.1 Process for communication tests

```
--  TextExchange = protocol { given: Text; ↑result: Text }
--
--  TextDoublingService = process p: +TextExchange; {
--      x: Text := p.given;
--      p. result := x // x;
--  }
```

7.2 Routine for routine tests

```
--  Swapper = routine p, q: Text → t, v: Text; {
--      textEQ(t, "");
--      textEQ(v, "");
--      t := q;   v := p;
--  }
```

7.3 Word → Boolean

```
BooleanMaps = routine {
    wb: Word indexes Boolean;

    i32EQ(#wb, 0);

    for (i := 0;   i <> 20; i += 1) {
        wb[i] := (i % 2) = 0;
        i32EQ(#wb, i + 1);
        boolEQ(wb[i], (i % 2) = 0)
    }

    for i in domain wb {
        boolEQ(wb[i], (i % 2) = 0)
    }
}
```

7.4 Word → Byte

```
ByteMaps = routine {
    wo: Word indexes Byte;

    i32EQ(#wo, 0);

    for (i := 0;  i <> 20; i += 1) {
        wo[i] := i * 2;
        i32EQ(#wo, i + 1);
        i8EQ(wo[i], i * 2)
    }

    for (i := 0;  i <> 20; i += 1) {
        i8EQ(wo[i], i * 2)
    }
}
```

7.5 Word → Word

```
WordMaps = routine {
    ww: WordArray;

    i32EQ(#ww, 0);

    for (i := 0;  i <> 20; i += 1) {
        ww[i] := i * 2;
        i32EQ(#ww, i + 1);
        i32EQ(ww[i], i * 2)
    }

    for (i := 0;  i <> 20; i += 1) {
        i32EQ(ww[i], i * 2)
    }

    ww2: WordArray := ww;
    for (i := 0;  i <> 20; i += 1) {
        i32EQ(ww2[i], i * 2)
    }
}
```

7.6 Word → Real

```
RealMaps = routine {
    wr: Word indexes Real;

    i32EQ(#wr, 0);

    for (i := 0;  i <> 20; i += 1) {
        wr[i] := i * 2.0;
        i32EQ(#wr, i + 1);
        r64EQ(wr[i], i * 2.0)
    }
```

```
        for (i := 0;  i <> 20; i += 1) {
            r64EQ(wr[i], i * 2.0)
        }
    }
```

7.7 Word → Text

```
    TextMaps = routine {
        wt: Word indexes Text;

        i32EQ(#wt, 0);

        for (i := 0;  i <> 20; i += 1) {
            wt[i] := Text(i * 2);
            i32EQ(#wt, i + 1);
            textEQ(wt[i], Text(i * 2))
        }

        for (i := 0;  i <> 20; i += 1) {
            textEQ(wt[i], Text(i * 2))
        }
    }
```

7.8 Word → Map

```
    MapMaps = routine {
        -- www: Word indexes Word indexes Word;
    }
```

7.9 Main cell

```
main = cell {
    beginTestT("Maps");
    BooleanMaps();
    ByteMaps();
    WordMaps();
    RealMaps();
    TextMaps();
    MapMaps();
    endTest()
}

main();
```

8 Test Log

```
test Booleans: tests = 169; successes = 169; failures = 0; 100%

secondary storage:
 acquired 3035 released 3035 residual 0 most at once 2891 average size 339
  Distribution                count   %  +%
            1-1                   0   0   0
            2-3                   0   0   0
            4-7                   0   0   0
            8-15                 22   0   0 *
           16-31               2144  70  71 ****************************************
           32-63                290   9  80 ******
           64-127               265   8  89 *****
          128-255                42   1  91 *
          256-511                22   0  91 *
          512-1023               91   2  94 **
         1024-2047                0   0  94
         2048-4095               99   3  98 **
         4096-8191               50   1  99 *
         8192-16383               6   0  99 *
        16384-32767               4   0 100 *
```

9 Exercises

9.1 Console output exercise

Program *Ex006ConOut* exercises the console output library.[2]

```
PiTable = process {
    for (i := 1;   i <> 6;  i += 1) {
      scrln(i // " \u00D7 \u03C0 = " // 3.141592653589793 * i);
    }
}

main = cell {
    PiTable();
    PiTable();
    PiTable();
    PiTable();
}

main();
```

```
1 × π = 3.141592653589793e+000
2 × π = 6.283185307179586e+000
3 × π = 9.424777960769379e+000
4 × π = 1.256637061435917e+001
5 × π = 1.570796326794897e+001
1 × π = 3.141592653589793e+000
1 × π = 3.141592653589793e+000
2 × π = 6.283185307179586e+000
2 × π = 6.283185307179586e+000
3 × π = 9.424777960769379e+000
3 × π = 9.424777960769379e+000
4 × π = 1.256637061435917e+001
4 × π = 1.256637061435917e+001
5 × π = 1.570796326794897e+001
5 × π = 1.570796326794897e+001
1 × π = 3.141592653589793e+000
2 × π = 6.283185307179586e+000
3 × π = 9.424777960769379e+000
4 × π = 1.256637061435917e+001
5 × π = 1.570796326794897e+001
```

[2] Desi had no trouble with the non-Ascii characters in this test but LaTeX did. To get the characters to appear it was necessary to employ two additional commands: '`\usepackage[mathletters]{ucs}`' and '`\usepackage[utf8x]{inputenc}`'.

9.2 Console input exercise

Program *Ex007ConInp* exercises the console input library by echoing input to output. The full spectrum of characters can be entered at the console, using the conventions of a Desi text. This technique is preferred over the limited traditional means of entering Ascii characters with codes above 127 with the help of the Alt key.

```
demand_a_line = routine → t: Text; {
    scrch("Type a line (ENTER on its own to exit) >>");
    t := kbdln()
}

main = cell {
   t: Text := demand_a_line();
   loop {
     | #t <> 0 | scrln(t);   t := demand_a_line()
   }
}

main();
```

```
Type a line (ENTER on its own to exit) >>abc
abc
Type a line (ENTER on its own to exit) >>abc\ndef
abc
def
Type a line (ENTER on its own to exit) >>a\n\tb\n\t\tc
a
        b
                c
Type a line (ENTER on its own to exit) >>Area = \u03C0r\u00B2
Area = πr²
Type a line (ENTER on its own to exit) >>
```

9.3 Selection

Program *Ex008Select.tex* derives the first few odd numbers in an obscure manner.

```
CYCLES = constant 3;
ADDITIONS = constant 4;
P = protocol { w: Word }
```

Process *Gen(z,n)* writes a sequence of integers to channel z, starting with integer n. The length of the sequence is given by the constant *ADDITIONS*.

```
Gen = process z: −P; n: Word {
    for (i := 0;  i <> ADDITIONS; i += 1) {
        z.w := i + n;
    }
}
```

Process *Add(x,y)* carries out *ADDITIONS* additions. It takes one operand from channel x and the other from channel y, reading from whichever channel is ready first and then waiting for the other to become available. It reports on which channel it read first.

```
Add = process x, y: +P {
    for (i := 0;  i <> ADDITIONS; i += 1) {
        select {
        || v: Word := x.w; v += y.w;
            scrln("First choice, from channel " // Word(x) // ": " // v)
        || v: Word := y.w; v += x.w;
            scrln("Second choice, from channel " // Word(y) // ": " // v)
        }
    }
}

Main = cell {
    for (i := 0;  i <> ADDITIONS*CYCLES; i += ADDITIONS) {
        c1, c2: P;
        Add(c1, c2);
        Gen(c2, i + 1);
        Gen(c1, i);
    }
}

Main();
```

```
First choice, from channel 24: 9
Second choice, from channel 48: 17
First choice, from channel 24: 11
First choice, from channel 40: 19
Second choice, from channel 16: 1
First choice, from channel 40: 21
First choice, from channel 8: 3
First choice, from channel 40: 23
Second choice, from channel 16: 5
First choice, from channel 24: 13
First choice, from channel 8: 7
First choice, from channel 24: 15
```