Project: DAZ PROJECT

Title: Calculator Example

Ref: ISS/HAT/DAZ/WRK507 *Issue:* 1.31 *Date:* 22 July 2011

Status: Informal *Type:* Technical

Author:

Name	*Location*	*Signature*	*Date*
R.D. Arthan	WIN01		

Abstract: This document gives an example of the Compliance Notation.

Distribution: Library

0 DOCUMENT CONTROL

0.1 Contents List

0.2 Document Cross References

[1] ISS/HAT/DAZ/USR501. *Compliance Tool — User Guide.* Lemma 1 Ltd., `http://www.lemma-one.com`.

[2] ISS/HAT/DAZ/USR503. *Compliance Tool — Proving VCs.* Lemma 1 Ltd., `http://www.lemma-one.com`.

[3] ISS/HAT/DAZ/WRK513. *Calculator Example VCs Proof Scripts.* R.D. Arthan and G.M. Prout, Lemma 1 Ltd., `http://www.lemma-one.com`.

1 INTRODUCTION

This document contains an example of the Compliance Notation. The example is concerned with the computational aspects of a simple calculator.

Part of the purpose of this example is to demonstrate the insertion of hypertext links in the script by the compliance tool (see [1]). For this reason, the example adopts the rather unusual policy of giving proofs of VCs immediately after the Compliance Notation clause which generates them (so that the interleaving of refinement steps and proofs is fairly complicated).

This example has also been used in the *Compliance Tool — Proving VCs* tutorial, [2]. For reference purposes, a proof script for all the VCs has been supplied in [3]. These proofs illustrate the techniques advocated in the tutorial, and differ slightly from those presented here.

2 PREAMBLE

The following Standard ML command sets up the Compliance Tool to process the rest of the script.

SML

```
force_delete_theory"BASICS'spec" handle Fail _ => ();
new_script {name="BASICS", unit_type="spec"};
```

3 BASIC DEFINITIONS

In this section, we define types and constants which will be of use throughout the rest of the script.

The SPARK package *BASICS* below helps record the following facts:

> The calculator deals with signed integers expressed using up to six decimal digits. It has a numeric keypad and 6 operation buttons labelled +, −, ×, +/−, !, √, and =.

Compliance Notation

```
package BASICS is

    BASE : constant INTEGER := 10;
    PRECISION : constant INTEGER := 6;
    MAX_NUMBER : constant INTEGER := BASE ** PRECISION − 1;
    MIN_NUMBER : constant INTEGER := −MAX_NUMBER;

    subtype DIGIT is INTEGER range 0 .. BASE − 1;

    subtype NUMBER is INTEGER range MIN_NUMBER .. MAX_NUMBER;

    type OPERATION is
      (PLUS, MINUS, TIMES, CHANGE_SIGN, SQUARE_ROOT, FACTORIAL, EQUALS);

end BASICS;
```

SML

```
output_ada_program{script="BASICS'spec", out_file="wrk507.ada"};
output_hypertext_edit_script{out_file="wrk507.ex"};
```

4 THE STATE

In this section, we define a package which contains all the state variables of the calculator.

The package *STATE* below defines the variables we will use to implement the following informal description of part of the calculator's behaviour:

> The calculator has two numeric state variables: the display, which contains the number currently being entered, and the accumulator, which contains the last result calculated.
>
> The user is considered to be in the process of entering a number whenever a digit button is pressed, and entry of a number is terminated by pressing one of the operation keys.
>
> When a binary operation key is pressed, the operation is remembered so that it can be calculated when the second operand has been entered.

SML

```
new_script {name="STATE", unit_type="spec"};
```

Compliance Notation

```
with BASICS;
package STATE is

    DISPLAY, ACCUMULATOR : BASICS.NUMBER;

    LAST_OP : BASICS.OPERATION;

    IN_NUMBER : BOOLEAN;

end STATE;
```

SML

```
output_ada_program{script="-", out_file="wrk507a.ada"};
output_hypertext_edit_script{out_file="wrk507a.ex"};
```

5 THE OPERATIONS

In this section, we define a package which contains procedures corresponding to pressing the calculator buttons.

5.1 Package Specification

We now want to introduce a package *OPERATIONS* which implements the following informal description ofhow the calculator responds to button presses:

> The behaviour when a digit button is pressed depends on whether a number is currently being entered into the display. If a number is being entered, then the digit is taken as part of the number. If a number is not being entered (e.g., if an operation button has just been pressed), then the digit is taken as the most significant digit of a new number in the display.
>
> When a binary operation button is pressed, any outstanding calculation is carried out and the answer (which will be the first operand of the operation) is displayed; the calculator is then ready for the user to enter the other operand of the operation.
>
> When a unary operation button is pressed, the result of performing that operation to the displayed number is computed and displayed; the accumulator is unchanged, but entry of the displayed number is considered to be complete.
>
> When the button marked = is pressed, any outstanding calculation is carried out and the answer is displayed.

The package implementing this is defined in section 5.2 below after we have dealt with some preliminaries.

5.1.1 Z Preliminaries

SML

```
open_theory "BASICS'spec";
new_theory "preliminaries";
```

To abbreviate the description of the package, we do some work in Z first, corresponding to the various sorts of button press.

Note that the use of $\mathbb{Z}$ rather than *BASICSoNUMBER* reflects the fact that we are ignoring questions of arithmetic overflow here. If we used the Z set which accurately represents the SPARK type, then we would have to add in pre-conditions saying that the operations do not overflow. The following schema defines what happens when a digit button is pressed.

z

DO_DIGIT

$DISPLAY_0,\ DISPLAY : \mathbb{Z};$
$IN\_NUMBER_0,\ IN\_NUMBER : BOOLEAN;$
$D : BASICSoDIGIT$

$$IN\_NUMBER_0 = TRUE \Rightarrow DISPLAY = DISPLAY_0 * BASICSoBASE + D;$$
$$IN\_NUMBER_0 = FALSE \Rightarrow DISPLAY = D;$$
$$IN\_NUMBER = TRUE$$

We now define sets *UNARY* and *BINARY* which partition the two sorts of operation key. Note that = can be considered as a sort of binary operation (which given operands x and y returns x).

z

$$UNARY \mathrel{\widehat{=}} \{BASICSoCHANGE\_SIGN,\ BASICSoFACTORIAL,\ BASICSoSQUARE\_ROOT\}$$

z

$$BINARY \mathrel{\widehat{=}} BASICSoOPERATION \setminus UNARY$$

We need to define a function for computing factorials in order to define the response to the factorial operation button.

z

$$fact : \mathbb{N} \rightarrow \mathbb{N}$$

$$fact\ 0 = 1\ ;$$
$$\forall m{:}\mathbb{N} \bullet fact(m+1) = (m + 1) * fact\ m$$

Unary operations behave as specified by the following schema. In which we do specify explicitly that the accumulator and last operation values are unchanged for clarity and for simplicity later on (when we group the unary and binary operations together).

z

DO_UNARY_OPERATION

$ACCUMULATOR_0,\ ACCUMULATOR : \mathbb{Z};$
$DISPLAY_0,\ DISPLAY : \mathbb{Z};$
$LAST\_OP_0,\ LAST\_OP : \mathbb{Z};$
$IN\_NUMBER : BOOLEAN;$
$O : UNARY$

$$IN\_NUMBER = FALSE;$$
$$ACCUMULATOR = ACCUMULATOR_0;$$
$$LAST\_OP = LAST\_OP_0;$$
$$O = BASICSoCHANGE\_SIGN \Rightarrow DISPLAY = {\sim}DISPLAY_0;$$

$$O = BASICSoFACTORIAL \land DISPLAY_0 \geq 0 \Rightarrow DISPLAY = fact\ DISPLAY_0;$$
$$O = BASICSoSQUARE\_ROOT \land DISPLAY_0 \geq 0 \Rightarrow$$
$$DISPLAY ** 2 \leq DISPLAY_0 < (DISPLAY + 1) ** 2$$

The binary operations are specified by the following schema.

z

DO_BINARY_OPERATION

$$ACCUMULATOR_0,\ ACCUMULATOR : \mathbb{Z};$$
$$DISPLAY_0,\ DISPLAY : \mathbb{Z};$$
$$LAST\_OP_0,\ LAST\_OP : \mathbb{Z};$$
$$IN\_NUMBER : BOOLEAN;$$
$$O : BINARY$$

$$IN\_NUMBER = FALSE;$$
$$DISPLAY = ACCUMULATOR;$$
$$LAST\_OP = O;$$
$$LAST\_OP_0 = BASICSoEQUALS \Rightarrow$$
$$ACCUMULATOR = DISPLAY_0;$$
$$LAST\_OP_0 = BASICSoPLUS \Rightarrow$$
$$ACCUMULATOR = ACCUMULATOR_0 + DISPLAY_0;$$
$$LAST\_OP_0 = BASICSoMINUS \Rightarrow$$
$$ACCUMULATOR = ACCUMULATOR_0 - DISPLAY_0;$$
$$LAST\_OP_0 = BASICSoTIMES \Rightarrow$$
$$ACCUMULATOR = ACCUMULATOR_0 * DISPLAY_0$$

The disjunction of the schemas for the unary and binary operations is then what is needed to define the response to pressing an arbitrary button press.

z

$$DO\_OPERATION \mathrel{\widehat{=}} DO\_UNARY\_OPERATION \lor DO\_BINARY\_OPERATION$$

5.2 The SPARK Package

We will now use the schemas of the previous section to define the package *OPERATIONS*. First we set up the script in which to develop the package.

SML

```
new_script1 {name="OPERATIONS", unit_type="spec", library_theories=["preliminaries"]};
```

Since we used the short forms of the SPARK names in the previous section, we need to rename the schema signature variables to prefix them with the appropriate package names.

Compliance Notation

```
with BASICS, STATE;
package OPERATIONS is
procedure DIGIT_BUTTON (D : in BASICS.DIGIT)
        Δ STATEoDISPLAY, STATEoIN_NUMBER [
         DO_DIGIT [
           STATEoDISPLAY0/DISPLAY0,  STATEoDISPLAY/DISPLAY,
           STATEoIN_NUMBER0/IN_NUMBER0, STATEoIN_NUMBER/IN_NUMBER,
           D/D] ] ;
procedure OPERATION_BUTTON (O : in BASICS.OPERATION)
        Δ STATEoACCUMULATOR, STATEoDISPLAY,
               STATEoIN_NUMBER, STATEoLAST_OP [
         DO_OPERATION[
          STATEoACCUMULATOR0/ACCUMULATOR0,
          STATEoACCUMULATOR/ACCUMULATOR,
          STATEoDISPLAY0/DISPLAY0,  STATEoDISPLAY/DISPLAY,
          STATEoLAST_OP0/LAST_OP0,  STATEoLAST_OP/LAST_OP,
          STATEoIN_NUMBER0/IN_NUMBER0, STATEoIN_NUMBER/IN_NUMBER,
          D/D] ] ;
end OPERATIONS;
```

SML

```
output_ada_program{script="−", out_file="wrk507b.ada"};
output_hypertext_edit_script{out_file="wrk507b.ex"};
```

5.3 Package Implementation

5.3.1 Package Body

The following specification of the package body is derived from the package specification in the obvious way. We leave a k-slot for any extra declarations we may need.

SML

```
new_script {name="OPERATIONS", unit_type="body"};
```

Compliance Notation

```
$references BASICS, STATE;
package body OPERATIONS is
procedure DIGIT_BUTTON (D : in BASICS.DIGIT)
        Δ STATEoDISPLAY, STATEoIN_NUMBER [
         DO_DIGIT [
           STATEoDISPLAY_0/DISPLAY_0, STATEoDISPLAY/DISPLAY,
           STATEoIN_NUMBER_0/IN_NUMBER_0, STATEoIN_NUMBER/IN_NUMBER,
           D/D] ]
    is begin
        Δ STATEoDISPLAY, STATEoIN_NUMBER [
         DO_DIGIT [ STATEoDISPLAY_0/DISPLAY_0, STATEoDISPLAY/DISPLAY,
           STATEoIN_NUMBER_0/IN_NUMBER_0, STATEoIN_NUMBER/IN_NUMBER,
           D/D] ]                              (3001)
    end DIGIT_BUTTON;
procedure OPERATION_BUTTON (O : in BASICS.OPERATION)
        Δ STATEoACCUMULATOR, STATEoDISPLAY,
              STATEoIN_NUMBER, STATEoLAST_OP [
         DO_OPERATION[
          STATEoACCUMULATOR_0/ACCUMULATOR_0,
          STATEoACCUMULATOR/ACCUMULATOR,
          STATEoDISPLAY_0/DISPLAY_0, STATEoDISPLAY/DISPLAY,
          STATEoLAST_OP_0/LAST_OP_0, STATEoLAST_OP/LAST_OP,
          STATEoIN_NUMBER_0/IN_NUMBER_0, STATEoIN_NUMBER/IN_NUMBER,
          D/D] ]
    is
      ⟨ Extra Declarations ⟩                   ( 500 )
    begin
        Δ STATEoACCUMULATOR, STATEoDISPLAY,
              STATEoIN_NUMBER, STATEoLAST_OP [
         DO_OPERATION[ STATEoACCUMULATOR_0/ACCUMULATOR_0,
          STATEoACCUMULATOR/ACCUMULATOR,
          STATEoDISPLAY_0/DISPLAY_0, STATEoDISPLAY/DISPLAY,
          STATEoLAST_OP_0/LAST_OP_0, STATEoLAST_OP/LAST_OP,
```

```
        STATEoIN_NUMBER0/IN_NUMBER0, STATEoIN_NUMBER/IN_NUMBER,
        D/D] ]                        (3002)
    end OPERATION_BUTTON;
end OPERATIONS;
```

Introducing the package body gives us 8 very trivial VCs to prove:

SML

```
open_theory "cn";
set_pc"cn";
open_theory "OPERATIONS'body";
set_goal([], get_conjecture"-""vcOPERATIONS_1");
a(REPEAT strip_tac);
val _ = save_pop_thm "vcOPERATIONS_1";
```

SML

```
set_goal([], get_conjecture"-""vcOPERATIONS_2");
a(REPEAT strip_tac);
val _ = save_pop_thm "vcOPERATIONS_2";
```

SML

```
set_goal([], get_conjecture"-""vcOPERATIONS_3");
a(REPEAT strip_tac);
val _ = save_pop_thm "vcOPERATIONS_3";
```

SML

```
set_goal([], get_conjecture"-""vcOPERATIONS_4");
a(REPEAT strip_tac);
val _ = save_pop_thm "vcOPERATIONS_4";
```

SML

```
open_theory "OPERATIONSoDIGIT_BUTTON'proc";
set_goal([], get_conjecture"-""vcOPERATIONSoDIGIT_BUTTON_1");
a(REPEAT strip_tac);
val _ = save_pop_thm "vcOPERATIONSoDIGIT_BUTTON_1";
```

SML

```
set_goal([], get_conjecture"-""vcOPERATIONSoDIGIT_BUTTON_2");
a(REPEAT strip_tac);
val _ = save_pop_thm "vcOPERATIONSoDIGIT_BUTTON_2";
```

SML

```
open_theory "OPERATIONSoOPERATION_BUTTON'proc";
set_goal([], get_conjecture"-""vcOPERATIONSoOPERATION_BUTTON_1");
a(REPEAT strip_tac);
val _ = save_pop_thm "vcOPERATIONSoOPERATION_BUTTON_1";
```

SML

```
set_goal([], get_conjecture"-""vcOPERATIONSoOPERATION_BUTTON_2");
a(REPEAT strip_tac);
val _ = save_pop_thm "vcOPERATIONSoOPERATION_BUTTON_2";
```

5.3.2 Supporting Functions

We choose to separate out the computation of factorials and square roots into separate functions which replace the k-slot labelled 500. In both cases, we prepare for the necessary algorithms. Our approach for both functions is to introduce and initialise appropriately a variable called *RESULT*, demand that this be set to the desired function return value and return that value.

SML

```
open_scope "OPERATIONS.OPERATION_BUTTON";
```

Compliance Notation

```
(500) ≡
   function FACT (M : NATURAL) return NATURAL
      Ξ [ FACT(M) = fact(M) ]
   is
      RESULT : NATURAL;
   begin
      RESULT := 1;
      Δ RESULT [M ≥ 0 ∧ RESULT = 1, RESULT = fact M ]        (1001)
      return RESULT;
   end FACT;

   function SQRT (M : NATURAL) return NATURAL
      Ξ [SQRT(M) ** 2 ≤ M < (SQRT(M) + 1) ** 2]
   is
      RESULT : NATURAL;
     ⟨ other local vars ⟩              (2)
   begin
     RESULT := 0;
     Δ RESULT [RESULT = 0, RESULT ** 2 ≤ M < (RESULT + 1) ** 2](2001)
    return RESULT;
   end SQRT;
```

The above results in a number of VCs to show that the function bodies achieve what is demanded in the function specification. We now prove these VCs, some of which require the following lemma about SPARK natural numbers.

SML
```
open_theory "preliminaries";
set_goal([], ⌜∀m : NATURAL• m ≥ 0⌝);
a(rewrite_tac[z_get_spec⌜NATURAL⌝] THEN REPEAT strip_tac);
val natural_thm = save_pop_thm"natural_thm";
open_scope "OPERATIONS.OPERATION_BUTTON.FACT";
```

SML
```
set_goal([], get_conjecture"−""vcOPERATIONSoOPERATION_BUTTONoFACT_1");
a(REPEAT strip_tac THEN all_fc_tac[natural_thm]);
val _ = save_pop_thm "vcOPERATIONSoOPERATION_BUTTONoFACT_1";
```

SML
```
set_goal([], get_conjecture"−""vcOPERATIONSoOPERATION_BUTTONoFACT_2");
a(REPEAT strip_tac THEN all_var_elim_asm_tac1);
val _ = save_pop_thm "vcOPERATIONSoOPERATION_BUTTONoFACT_2";
```

SML
```
open_scope "OPERATIONS.OPERATION_BUTTON.SQRT";
set_goal([], get_conjecture"−""vcOPERATIONSoOPERATION_BUTTONoSQRT_1");
a(REPEAT strip_tac);
val _ = save_pop_thm "vcOPERATIONSoOPERATION_BUTTONoSQRT_1";
```

SML
```
set_goal([], get_conjecture"−""vcOPERATIONSoOPERATION_BUTTONoSQRT_2");
a(REPEAT strip_tac THEN all_var_elim_asm_tac1);
val _ = save_pop_thm "vcOPERATIONSoOPERATION_BUTTONoSQRT_2";
```

SML
```
open_scope "OPERATIONS";
```

5.3.3 Algorithm for Factorial

Factorial is implemented by a for-loop with loop-counter J and an invariant requiring that as J steps from *2* up to M, *RESULT* is kept equal to the factorial of J:

SML
```
open_scope "OPERATIONS.OPERATION_BUTTON.FACT";
```

Compliance Notation
```
 (1001) ⊑
   for J in INTEGER range 2 .. M
   loop
       Δ RESULT [J ≥ 1 ∧ RESULT = fact (J−1), RESULT = fact J] (1002)
   end loop;
```

This produces 4 VCs, which we proceed to prove, beginning with a lemma about the first two values of factorial (needed because our algorithm avoids the unnecessary pass through the loop with $J = 1$).

SML

```
set_goal([], ⓩfact 0 = 1 ∧ fact 1 = 1⌝);
a(rewrite_tac[z_get_specⓩfact⌝,
        (rewrite_rule[z_get_specⓩfact⌝] o z_∀_elimⓩ0⌝ o
                    ∧_right_elim o ∧_right_elim o z_get_spec)ⓩfact⌝
]);
val fact_thm  = save_pop_thm"fact_thm";
```

SML

```
set_goal([], get_conjecture"−""vc1001_1");
a(REPEAT strip_tac THEN asm_rewrite_tac[fact_thm]);
val _ = save_pop_thm "vc1001_1";
```

SML

```
set_goal([], get_conjecture"−""vc1001_2");
a(REPEAT strip_tac THEN all_var_elim_asm_tac1);
a(lemma_tacⓩM = 0 ∨ M = 1⌝);
(∗ ∗∗∗ Goal "1" ∗∗∗ ∗)
a(PC_T1 "z_lin_arith" asm_prove_tac[]);
(∗ ∗∗∗ Goal "2" ∗∗∗ ∗)
a(asm_rewrite_tac[fact_thm]);
(∗ ∗∗∗ Goal "3" ∗∗∗ ∗)
a(asm_rewrite_tac[fact_thm]);
val _ = save_pop_thm "vc1001_2";
```

SML

```
set_goal([], get_conjecture"−""vc1001_3");
a(REPEAT strip_tac);
(∗ ∗∗∗ Goal "1" ∗∗∗ ∗)
a(asm_ante_tacⓩ2 ≤ J⌝ THEN PC_T1 "z_lin_arith" prove_tac[]);
(∗ ∗∗∗ Goal "2" ∗∗∗ ∗)
a(asm_rewrite_tac[z_plus_assoc_thm]);
val _ = save_pop_thm "vc1001_3";
```

SML

```
set_goal([], get_conjecture"−""vc1001_4");
a(REPEAT strip_tac THEN asm_rewrite_tac[]);
val _ = save_pop_thm "vc1001_4";
```

Now we can complete the implementation of the factorial function by providing the loop body:

Compliance Notation

```
(1002) ⊑
      RESULT := J * RESULT;
```

Again this gives rise to a VC which we prove immediately, completing the implementation and verification of the factorial function:

SML

```
set_goal([], get_conjecture"−""vc1002_1");
a(REPEAT strip_tac THEN all_var_elim_asm_tac1);
a(lemma_tac⌜Z∃K:𝕌• K + 1 = J⌝);
(* *** Goal "1" *** *)
a(z_∃_tac⌜Z J − 1⌝ THEN PC_T1 "z_lin_arith" prove_tac[]);
(* *** Goal "2" *** *)
a(all_var_elim_asm_tac1);
a(rewrite_tac[z_plus_assoc_thm]);
a(ALL_FC_T rewrite_tac[z_get_spec⌜Z fact⌝]);
val _ = save_pop_thm "vc1002_1";
```

5.3.4 Algorithm for Square Root

For square root, we need two extra variables to implement a binary search for the square root.

SML

```
open_scope"OPERATIONS.OPERATION_BUTTON.SQRT";
```

Compliance Notation

```
(2) ≡
    MID, HI : INTEGER;
```

The following just says that we propose to achieve the desired effect on *RESULT* using *MID* and *HI* as well.

Compliance Notation

```
(2001) ⊑
      Δ RESULT, MID, HI
          [RESULT = 0, RESULT ** 2 ≤ M < (RESULT + 1) ** 2] (2002)
```

This produces two very trivial VCs:

SML

```
set_goal([], get_conjecture "−" "vc2001_1");
a(REPEAT strip_tac);
val _ = save_pop_thm "vc2001_1";
```

SML

```
set_goal([], get_conjecture "−" "vc2001_2");
a(REPEAT strip_tac);
val _ = save_pop_thm "vc2001_2";
```

Now we give the initialisation for *HI* and describe the loop which will find the square root:

Compliance Notation

```
(2002) ⊑
        HI := M + 1;
        $till ⟦RESULT ** 2 ≤ M < (RESULT + 1) ** 2⟧
        loop
            Δ RESULT, MID, HI
                [RESULT ** 2 ≤ M < HI ** 2, RESULT ** 2 ≤ M < HI ** 2] (2003)
        end loop;
```

This gives us 3 more VCs to prove, which depend on a few mathematical facts about the exponentiation operator:

SML

```
set_goal([], ⓩ∀x: ℤ• x ** 1 = x⌝);
a(REPEAT strip_tac);
a(rewrite_tac[rewrite_rule[](
    z_∀_elimⓩ(x ≙ x, y ≙ 0)⌝ (∧_right_elim(z_get_specⓩ(_**_)⌝)))]);
val star_star_1_thm = pop_thm();
```

SML

```
set_goal([], ⓩ∀x: ℤ• x ** 2 = x * x⌝);
a(REPEAT strip_tac);
a(rewrite_tac[star_star_1_thm, rewrite_rule[](
    z_∀_elimⓩ(x ≙ x, y ≙ 1)⌝ (∧_right_elim(z_get_specⓩ(_**_)⌝)))]);
val star_star_2_thm = pop_thm();
```

SML

```
set_goal([], get_conjecture "−" "vc2002_1");
a(REPEAT strip_tac THEN all_fc_tac[natural_thm]);
(* *** Goal "1" *** *)
a(asm_rewrite_tac[star_star_1_thm, star_star_2_thm]);
(* *** Goal "2" *** *)
a(POP_ASM_T ante_tac THEN DROP_ASMS_T discard_tac THEN strip_tac);
a(z_≤_induction_tacⓩM⌝);
(* *** Goal "2.1" *** *)
a(rewrite_tac[star_star_1_thm, star_star_2_thm]);
(* *** Goal "2.2" *** *)
```

```
a(POP_ASM_T ante_tac);
a(rewrite_tac[star_star_2_thm]);
a(PC_T1 "z_lin_arith" asm_prove_tac[]);
val _ = save_pop_thm "vc2002_1";
```

SML
```
set_goal([], get_conjecture "-" "vc2002_2");
a(REPEAT strip_tac);
val _ = save_pop_thm "vc2002_2";
```

SML
```
set_goal([], get_conjecture "-" "vc2002_3");
a(REPEAT strip_tac);
val _ = save_pop_thm "vc2002_3";
```

Now we implement the exit for the loop and specify the next step:

Compliance Notation
```
(2003) ⊑
        exit when RESULT + 1 = HI;
        Δ RESULT, MID, HI
            [RESULT ** 2 ≤ M < HI ** 2, RESULT ** 2 ≤ M < HI ** 2] (2004)
```

Again we get VCs which we now prove:

SML
```
set_goal([], get_conjecture "-" "vc2003_1");
a(rewrite_tac[]);
a(REPEAT strip_tac);
a(all_var_elim_asm_tac1);
val _ = save_pop_thm "vc2003_1";
```

SML
```
set_goal([], get_conjecture "-" "vc2003_2");
a(REPEAT strip_tac);
val _ = save_pop_thm "vc2003_2";
```

SML
```
set_goal([], get_conjecture "-" "vc2003_3");
a(REPEAT strip_tac);
val _ = save_pop_thm "vc2003_3";
```

Now we can fill in the last part of the loop:

Compliance Notation

```
(2004)    ⊑
    MID := (RESULT + HI + 1) / 2;
    if      MID ** 2 > M
    then    HI := MID;
    else    RESULT := MID;
    end if;
```

We now prove the 2 VCs produced, which completes the implementation and verification of the square root function.

SML

```
set_goal([], get_conjecture "−" "vc2004_1");
a(rewrite_tac[star_star_2_thm]);
a(REPEAT strip_tac);
val _ = save_pop_thm "vc2004_1";
```

SML

```
set_goal([], get_conjecture "−" "vc2004_2");
a(rewrite_tac[star_star_2_thm]);
a(REPEAT strip_tac);
val _ = save_pop_thm "vc2004_2";
```

5.3.5 Digit Button Algorithm

We now continue with the body of the digit button procedure. An if-statement handling the two cases for updating the display, followed by an assignment to the flag should meet the bill here.

SML

```
open_scope"OPERATIONS.DIGIT_BUTTON";
```

Compliance Notation

```
(3001) ⊑
    if    STATE.IN_NUMBER
    then  STATE.DISPLAY := STATE.DISPLAY * BASICS.BASE + D;
    else  STATE.DISPLAY := D;
    end if;
    STATE.IN_NUMBER := true;
```

This produces 2 VCs corresponding to the two branches of the if-statement. Both are easy to prove:

SML

```
set_goal([], get_conjecture"−""vc3001_1");
a(REPEAT strip_tac);
a(asm_rewrite_tac[z_get_specⓩDO_DIGIT⌝]);
a(REPEAT strip_tac);
val _ = save_pop_thm "vc3001_1";
```

SML

```
set_goal([], get_conjecture"-""vc3001_2");
a(REPEAT strip_tac);
a(asm_rewrite_tac[z_get_spec⌜DO_DIGIT⌝]);
val _ = save_pop_thm "vc3001_2";
```

5.3.6 Operation Button Algorithm

We now complete the implementation and verification of the package *OPERATIONS* by giving the body of the procedure for handling the operation buttons.

SML

```
open_scope "OPERATIONS.OPERATION_BUTTON";
```

Compliance Notation

```
(3002) ⊑
   if     O = BASICS.CHANGE_SIGN
   then    STATE.DISPLAY := -STATE.DISPLAY;
   elsif  O = BASICS.FACTORIAL
   then    STATE.DISPLAY := FACT(STATE.DISPLAY);
   elsif  O = BASICS.SQUARE_ROOT
   then    STATE.DISPLAY := SQRT(STATE.DISPLAY);
   else   if      STATE.LAST_OP = BASICS.EQUALS
          then    STATE.ACCUMULATOR := STATE.DISPLAY;
          elsif   STATE.LAST_OP = BASICS.PLUS
          then    STATE.ACCUMULATOR := STATE.ACCUMULATOR + STATE.DISPLAY;
          elsif   STATE.LAST_OP = BASICS.MINUS
          then    STATE.ACCUMULATOR := STATE.ACCUMULATOR - STATE.DISPLAY;
          elsif   STATE.LAST_OP = BASICS.TIMES
          then    STATE.ACCUMULATOR := STATE.ACCUMULATOR * STATE.DISPLAY;
          end if;
          STATE.DISPLAY := STATE.ACCUMULATOR;
          STATE.LAST_OP := O;
   end if;
   STATE.IN_NUMBER := false;
```

SML

```
open_theory "preliminaries";
val basics_defs = map z_get_spec(get_consts"BASICS'spec");
val op_defs = map z_get_spec(flat(
        map get_consts ["preliminaries", "OPERATIONS'body", "OPERATIONS'spec"]));
```

The first three VCs are concerned with the unary operations.

SML

```
open_scope "OPERATIONS.OPERATION_BUTTON";
set_goal([], get_conjecture"−""vc3002_1");
a(rewrite_tac op_defs);
a(z_∀_tac THEN ⇒_tac THEN asm_rewrite_tac basics_defs);
val _ = save_pop_thm "vc3002_1";
```

For the next two VCs, it is necessary to make the (reasonable) assumption that a non-negative number of the precision handled by the calculator will fit in a SPARK *NATURAL*. This amounts to the following axiom:

Z

$$BASICSoMAX\_NUMBER \leq INTEGERvLAST$$

SML

```
val number_ax = snd(hd(get_axioms"−"));
set_goal([], get_conjecture"−""vc3002_2");
a(rewrite_tac op_defs);
a(z_∀_tac THEN ⇒_tac THEN asm_rewrite_tac basics_defs);
a(all_var_elim_asm_tac1 THEN strip_tac);
a(lemma_tac ⓩSTATEoDISPLAY ∈ NATURAL⌝);
(* *** Goal "1" *** *)
a(DROP_NTH_ASM_T 5 ante_tac);
a(ante_tac number_ax);
a(asm_rewrite_tac(z_get_specⓩNATURAL⌝ :: basics_defs));
a(PC_T1 "z_lin_arith" prove_tac[]);
(* *** Goal "2" *** *)
a(ALL_FC_T rewrite_tac[z_get_specⓩFACT⌝]);
val _ = save_pop_thm "vc3002_2";
```

SML

```
set_goal([], get_conjecture"−""vc3002_3");
a(rewrite_tac op_defs);
a(z_∀_tac THEN ⇒_tac THEN asm_rewrite_tac basics_defs);
a(all_var_elim_asm_tac1 THEN strip_tac);
a(lemma_tac ⓩSTATEoDISPLAY ∈ NATURAL⌝);
(* *** Goal "1" *** *)
a(DROP_NTH_ASM_T 6 ante_tac);
a(ante_tac number_ax);
a(asm_rewrite_tac(z_get_specⓩNATURAL⌝ :: basics_defs));
a(PC_T1 "z_lin_arith" prove_tac[]);
(* *** Goal "2" *** *)
a(all_fc_tac[z_get_specⓩSQRT⌝]);
a(REPEAT strip_tac);
val _ = save_pop_thm "vc3002_3";
```

Because the binary operations only involve built-in arithmetic operators, they are a little easier to verify than the unary ones.

SML
```
set_goal([], get_conjecture"-""vc3002_4");
a(rewrite_tac op_defs);
a(z_∀_tac THEN ⇒_tac THEN asm_rewrite_tac basics_defs);
val _ = save_pop_thm "vc3002_4";
```

SML
```
set_goal([], get_conjecture"-""vc3002_5");
a(rewrite_tac op_defs);
a(z_∀_tac THEN ⇒_tac THEN asm_rewrite_tac basics_defs);
val _ = save_pop_thm "vc3002_5";
```

SML
```
set_goal([], get_conjecture"-""vc3002_6");
a(rewrite_tac op_defs);
a(z_∀_tac THEN ⇒_tac THEN asm_rewrite_tac basics_defs);
val _ = save_pop_thm "vc3002_6";
```

SML
```
set_goal([], get_conjecture"-""vc3002_7");
a(rewrite_tac op_defs);
a(z_∀_tac THEN ⇒_tac THEN asm_rewrite_tac basics_defs);
val _ = save_pop_thm "vc3002_7";
```

SML
```
set_goal([], get_conjecture"-""vc3002_8");
a(rewrite_tac op_defs);
a(z_∀_tac THEN ⇒_tac THEN asm_rewrite_tac basics_defs);
val _ = save_pop_thm "vc3002_8";
```

That completes the formal verification of the calculator packages.

SML
```
output_ada_program{script="OPERATIONS'body", out_file="wrk507c.ada"};
output_hypertext_edit_script{out_file="wrk507c.ex"};
```

6 EPILOGUE

The following ProofPower-ML commands produce the various parts of the Z document and then print out a message for use when this script is used as part of the Compliance Tool test suite.

SML

```
output_z_document{script="BASICS'spec", out_file="wrk507.zdoc"};
output_z_document{script="STATE'spec", out_file="wrk507a.zdoc"};
output_z_document{script="OPERATIONS'spec", out_file="wrk507b.zdoc"};
output_z_document{script="OPERATIONS'body", out_file="wrk507c.zdoc"};
```

The following commands check that all the VCs have been proved.

SML

```
val thys = get_descendants "cn" less "cn";
val unproved =
map (fn thy => (open_theory thy; (thy, get_unproved_conjectures thy))) thys drop (is_nil o snd);
val _ =
        if      is_nil unproved
        then    diag_line "All module tests passed"
        else    diag_line "Some VCs have not been proved";
```