Language Specifications for SANE Pascal Numerics

Apple Numerics

This document contains concise high–level specifications for integrating the Pascal language and the Standard Apple Numeric Environment (SANE), which includes *IEEE Standard (754) for Binary Floating-Point Arithmetic*. Language implementors can use this document with a SANE arithmetic engine to produce a Pascal which supports the Standard Apple Numeric Environment. Only aspects of Pascal pertaining to numerics and only the details required by language implementors are covered. SANE and the 68000 and 6502 SANE engines, which include basic floating point routines and elementary functions, are documented in the *Apple Numerics Manual*. 68000 and 6502 numeric scanners and formatters, available from the Apple Numerics Group, support compile–time scanning and run–time I/O.

Integer refers to the Pascal integer type or to the appropriate integer type if more than one are supported.

0. Background

SANE Pascal is ANSI Pascal, modified to exploit an arithmetic engine which includes an extended-precision implementation of the IEEE Standard. Much of SANE can be implemented in a library, which does not affect the Pascal language. However, to efficiently and effectively use the SANE extended type, and to handle IEEE NaNs and infinities, some extensions to Pascal are needed.

SANE engines (as will as the Intel 8087, Motorola 68881, and Zilog Z8070) evaluate arithmetic operations to the range and precision of an 80-bit *extended* type and deliver results to the extended format. Thus, the SANE extended type naturally replaces real as the basic arithmetic type for computing purposes. The types real (IEEE single), double, and comp serve as space–saving storage types. Installing extended in the role of real and adding three more floating point types are the most pervasive changes to Pascal specified by this document.

The IEEE Standard specifies two special representations for its floating point formats: NaNs (not–a–number) and infinities. This document expands the syntax for I/O to accommodate NaNs and infinities, and includes the treatment for NaNs in relationals which is required by the IEEE Standard.

Programs written for standard Pascal will run, without modification, under SANE Pascal.

SANE does not affect integer arithmetic.

1. Types

1.1 SANE Data Types

real	-- 4 byte (IEEE single type)
double	-- 8 byte (IEEE type)
comp	-- 8 byte (64–bit integer, with one NaN)
extended	-- 10 byte (IEEE type)

2. Constants

2.1 Simple Constants

2.1.1 Conversions. Conversion to binary for numeric constants in source text shall be done at compile time. SANE conversion routines shall be used for non-integer numeric constants.

2.1.2 Syntax. The syntax for simple numeric constants shall be that described in the ANSI standard, except that the values of digit-sequences shall not be constrained to lie in the interval 0 to maxint. This exception is for access to the wide precision of the extended type and to facilitate arithmetic with integral values in the comp type.

2.1.3 Typing. Constants which consist solely of an optional sign and a digit sequence and which can be represented exactly in integer format are of integer type. All other numeric constants are of type extended.

2.1.4 Predefined Constants. Predefined extended constants with hex values:

inf	$7FFF 0000 0000 0000 0000
pi	$4000 C90F DAA2 2168 C235

2.2 Constant Expressions

2.2.1 Typing. Numeric constant expressions shall be typed (either integer or extended) according to the general rules for expression types (see §4).

2.2.2 Evaluation. Expressions in constant definitions shall be evaluated at compile time. All constant expressions in executable statements must be evaluated as if at run time. (Constant expressions which do not signal SANE exceptions, with rounding precision set to single, can be done at compile time as an optimization.)

3. Operations

3.1 Operators and Precedence

These are as in standard Pascal.

3.2 Relationals

The operator <> shall mean *not equal* (i.e. less, greater, or unordered). (See §9, Implementation Notes.)

4. Expressions and Assignments

4.1 Typing

The type of an expression consisting solely of a numeric variable, constant, or function call is integer if the variable, constant, or function is of integer type, and is extended otherwise. An expression formed by subexpressions and an arithmetic operation is of integer type if all subexpressions are of integer type, and is extended otherwise -- except that SANE division (/) is always of extended type.

4.2 Evaluation

Integer (sub)expressions shall be evaluated using integer arithmetic and integer temporaries as needed. Extended (sub)expressions shall be evaluated using SANE arithmetic and extended-precision temporaries as needed. (See §9, Implementation Notes.)

4.3 Legal Assignments

Any numeric expression (whether integer or extended) may be assigned to any SANE type variable; conversions shall be via SANE conversion routines. An expression of extended type may not be directly assigned to an integer variable.

5. Input/Output

5.1 Read Syntax

In addition to the syntax recognized by standard Pascal, read shall recognize [+|-]<infinity> and [+|-]<NAN>:

```
<infinity>      ::= INF
<NAN>           ::= NAN[([<digits>])]
<digits>        ::= {0|1|2|3|4|5|6|7|8|9}
```

The numeric scanners available from the Apple Numerics Group recognize a superset of the required Pascal syntax. (See §9, Implementation Notes.)

5.2 Write Formatting

Numeric values from all SANE types, real, double, comp, and extended, shall be formatted in the manner of reals in ANSI Pascal, with the exceptions below. Routines available from the Apple Numerics Group support this formatting.

Infinities are written as [-]INF and NaNs as [-]NAN(ddd), where ddd is a decimal NaN code.

The exponent-digits field is always 4 wide (numbers from the extended type may require this); the first exponent digits is 0 only if the exponent is zero, and the field is padded on the right with blanks. This avoids the excess of leading 0's from the ANSI specification.

The ANSI restriction, FracDigits > 0, is removed to facilitate formatting comp values.

5.3 ASCII - Binary Conversions

The scanner which converts ASCII numeric input to SANE storage formats and the printer which converts SANE type values to ASCII output strings shall use the SANE binary-decimal conversion routines.

6. Procedures and Functions

6.1 Argument Types

An actual parameter passed by value to a SANE type formal parameters is an expression and hence is of extended or integer type; the compiler shall effect the conversion to the type of the formal parameter. An extended actual parameter may not be passed to an integer formal parameter.

6.2 Standard Pascal Numeric Functions

6.2.1 Sqr and Abs. These are generic functions of an integer or extended argument, returning an integer or extended result.

6.2.2 Round and Trunc. Round, unlike IEEE rounding to nearest, sends half-way cases away from zero. Round and trunc routines which correctly handle exceptions are available from the Apple Numerics Group.

6.2.3 Mathematical Functions. SANE directly supports sqrt, ln, exp, sin, cos, and arctan, each as an extended function of an extended argument.

7. Environment

7.1 Compile Time

IEEE defaults -- rounding to nearest, rounding to extended precision, and all halts disabled -- shall be in effect for compile–time calls to the SANE engine for constant conversions and constant–expression evaluations.

7.2 Run Time

An implementation may begin each program with the IEEE defaults -- rounding to nearest, rounding to extended precision, all exception flags clear, and all halts disabled. Alternatively, to more closely match the standard Pascal specification of error conditions, an implementation may begin each program with halts on invalid, overflow, and divide-by-zero enabled, and IEEE defaults otherwise. Environment changes are then dictated by the user's program.

7.3 Optimization

Optimizations must respect the floating point environment, including exception-flag setting which is a side-effect of arithmetic operations.

8. SANE Library Interface

In this sample interface, *integer* refers to 16-bit integers and *longint* refers to 32-bit integers.

```
      const

         DecStrLen  = 255;
         SigDigLen  = 20;      { for 68K;  use 28 for 6502 SANE }

{-------------------------------------------------------------------
*  Exceptions.
-------------------------------------------------------------------}
         Invalid        = 1;
         Underflow      = 2;
         Overflow       = 4;
         DivByZero      = 8;
         Inexact        = 16;

      type

{-------------------------------------------------------------------
*  Types for handling decimal representations.
-------------------------------------------------------------------}
         DecStr     = string[DecStrLen];

         CStrPtr    = ^char;

         Decimal    = record
                        sgn : 0..1;
                        exp : integer;
                        sig : string[SigDigLen]
                      end;

         DecForm    = record
                        style  : ( FloatDecimal, FixedDecimal );
                        digits : integer
                      end;

{-------------------------------------------------------------------
*  Ordering relations.
-------------------------------------------------------------------}

         RelOp      = ( GreaterThan, LessThan, EqualTo, Unordered );

{-------------------------------------------------------------------
*  Inquiry classes.
-------------------------------------------------------------------}
         NumClass   = ( SNaN, QNaN, Infinite, ZeroNum, NormalNum, DenormalNum );

{-------------------------------------------------------------------
*  Environmental control.
-------------------------------------------------------------------}
         Exception  = integer;

         RoundDir   = ( ToNearest, Upward, Downward, TowardZero );

         RoundPre   = ( ExtPrecision, DblPrecision, RealPrecision );
```

```
        Environment = integer;

{-------------------------------------------------------------------
*  Conversions between numeric binary types.
-------------------------------------------------------------------}
      function Num2Integer( x :  extended ) : integer;
      function Num2Longint( x : extended ) : longint; { if longint available }
      function Num2Extended( x : extended ) : extended;

      { If coercion functions are desired: }
      function Num2Real( x : extended ) : real;
      function Num2Double( x : extended ) : double;
      function Num2Comp( x : extended ) : comp;

{-------------------------------------------------------------------
*  Conversions between binary and decimal.
-------------------------------------------------------------------}
      procedure Num2Dec( f : DecForm;  x : extended;  var d : Decimal );
          { d <-- x according to format f }
      function Dec2Num( d : Decimal ) : extended;
          { Dec2Num <-- d }

      procedure Num2Str( f : DecForm;  x : extended;  var s : DecStr );
          { s <-- x according to format f }
      function Str2Num( s : DecStr ) : extended;
          { Str2Num <-- s }

{-------------------------------------------------------------------
*  Conversions between decimal formats.
-------------------------------------------------------------------}
      procedure Str2Dec( s : DecStr;  var Index : integer;  var d : Decimal;
                                                  var ValidPrefix : boolean );
          { On input Index is starting index into s, on output Index is one
            greater than index of last character of longest numeric substring;
            d <-- Decimal rep of  longest numeric substring;  ValidPrefix  <--
            "s, beginning at Index, contains valid numeric string or valid
            prefix of some numeric string" }
      procedure CStr2Dec( s : CStrPtr ;  var Index : integer;
                            var d : Decimal;  var ValidPrefix : boolean );
          { Str2Dec for character buffers or C strings instead of Pascal
            strings: the first argument is the address of a character buffer
            and ValidPrefix <-- "scanning ended with a null byte" }
      procedure Dec2Str( f : DecForm;  d : Decimal;  var s : DecStr );
          { s <-- d according to format f }

{-------------------------------------------------------------------
*  Arithmetic, auxiliary, and elementary functions.
-------------------------------------------------------------------}
      function Remainder( x, y : extended;  var quo : integer ) : extended;
          { Remainder <-- x rem y ;  quo <-- low-order seven bits of integer
            quotient x/y so that -127 < quo < 127 }
      function Rint( x : extended ) : extended;
          { round to integral value }
      function Scalb( n : integer;  x : extended ) : extended;
          { scale binary:  Scalb <-- x * 2^n }
      function Logb( x : extended ) : extended;
          { Logb <-- unbiased exponent of x }
      function CopySign( x, y : extended ) : extended;
          { CopySign <-- y with sign of x }
      function NextReal( x, y : real ) : real;
      function NextDouble( x, y : double ) : double;
      function NextExtended( x, y : extended ) : extended;
         { return next representable value from x toward y }

      function Log2( x : extended ) : extended;
         { base-2 log }
      function Ln1( x : extended ) : extended;
         { Ln1 <-- ln(1+x) }
      function Exp2( x : extended ) : extended;
         { base-2 exponential }
      function Exp1( x : extended ) : extended;
         { Exp1 <-- exp(x) - 1 }
      function XpwrI( x : extended;  i : integer ) : extended;
         { XpwrI <-- x^i }
      function XpwrY( x, y : extended ) : extended;
         { XpwrY <-- x^y }
      function Compound( r, n : extended ) : extended;
         { Compound <-- (1+r)^n }
      function Annuity( r, n : extended ) : extended;
         { Annuity <-- (1-(1+r)^(-n))/r }
      function Tan( x : extended ) : extended;
         { tangent }
      function  RandomX( var x : extended ) : extended;
         { returns next random number and updates argument;
            x integral, 1 <= x <= (2^31)-2 }


{-----------------------------------------------------------------
* Inquiry Routines.
-----------------------------------------------------------------}
      function ClassReal( x : real ) : NumClass;
      function ClassDouble( x : double ) : NumClass;
      function ClassComp( x : comp ) : NumClass;
      function ClassExtended( x : extended ) : NumClass;
         { return class of x }

      function SignNum( x : extended ) : integer;
         { 0 if sign bit clear, 1 if sign bit set }

{-----------------------------------------------------------------
*  NaN function.
-----------------------------------------------------------------}
      function NAN( i : integer ) : extended;
         { returns NaN with code i }

{-----------------------------------------------------------------
*  Environment access routines.
*     An exception variable encodes the exceptions whose sum is its value.
-----------------------------------------------------------------}
      procedure SetException( e : Exception;  b : boolean );
         { set e flags if b is true, clear e flags otherwise;  may cause halt }
      function TestException( e : Exception ) : boolean;
         { return true if any e flag is set, return false otherwise }
      procedure SetHalt( e : Exception;  b : boolean );
         { set e halt enables if b is true, clear e halt enables otherwise }
      function TestHalt( e : exception ) : boolean;
         { return true if any e halt is enabled, return false otherwise }
```

```
     procedure SetRound( r : RoundDir );
         { set rounding direction to r }
     function GetRound : RoundDir;
         { return rounding direction }
     procedure SetPrecision( p : RoundPre );
         { set rounding precision to p }
     function GetPrecision : RoundPre;
         { return rounding precision }
     procedure SetEnvironment( e : Environment );
         { set environment to e }
     procedure GetEnvironment( var e : Environment );
         { e <-- environment }
     procedure ProcEntry( var e : Environment );
         { e <-- environment, environment <-- IEEE default }
     procedure ProcExit( e : Environment );
         { temp <-- exceptions, environment <-- e, signal exceptions in temp }

     { If halt vector is to be made available to Pascal users: }
     function GetHaltVector : longint;
         { return halt vector }
     procedure SetHaltVector( v : longint );
         { halt vector <-- v }

{------------------------------------------------------------------
*  Comparison routine.
------------------------------------------------------------------}
     function Relation( x, y : extended ) : RelOp;
         { return Relation such that "x Relation y" is true }
```

9. Implementation Notes

9.1 Relationals

For the compiler writer there are two important points regarding IEEE style relationals:

- A SANE engine has two comparison operations, one (FCPX_) which signals invalid if its operands are unordered and one (FCMP_) which does not. For >, <=, <, and <= use FCPX_. For = and <> use FCMP_.

- When flipping the sense of a relational operator, consider the unordered possibility. For example, to branch if a < b is false, call FCPX_ to compare the operands and then use the SANE macro (FBUGE) which branches if the result of the comparison was unordered, greater, or equal.

9.2 Expression Evaluation

The following example illustrates SANE's extended-based expression evaluation. The assignment

```
r := i * d + c;
```

where the variables are declared by

```
var     r : real;
        i : integer;
        d : double;
        c : comp;
```

can be effected by the compiler as suggested by the sequence of assembly language macros:

```
PUSH     I_ADR
PUSH     T_ADR          ; t a compiler extended temporary
FI2X                    ; t <-- i
PUSH     D_ADR
PUSH     T_ADR
FMULD                   ; t <-- t * d
PUSH     C_ADR
PUSH     T_ADR
FADDC                   ; t <-- t + c
PUSH     T_ADR
PUSH     R_ADR
FX2S                    ; r <-- t
```

9.3 Scanning (2.1.2, 5.1)

The scanners available from the Apple Numerics Group recognize:

```
<ascii number>          ::= [{space | tab}] <left ascii>
<left ascii>            ::= [+|-] <unsigned ascii>
<unsigned ascii>        ::= <finite decimal> | <infinity> | <NAN>
<finite decimal>        ::= <decimal significand> [<exponent>]
<decimal significand>   ::= <integer> | <mixed>
<integer>               ::= <digits> [.]
<digits>                ::= {0|1|2|3|4|5|6|7|8|9}
<mixed>                 ::= [<digits>] . <digits>
<exponent>              ::= E [+|-] <digits>
<infinity>              ::= INF
<NAN>                   ::= NAN[([<digits>])]
```

(Square brackets enclose optional items, curly brackets enclose elements to be repeated at least once, and vertical bars separate alternative elements; letters that appear literally, like the 'E' marking the exponent field, may be either upper or lower case.)

Note that this a superset of the ANSI Pascal number syntax.

The scanner Str2Dec, or CStr2Dec, can be used for character-by-character input: repeatedly append characters to a string and call Str2Dec until the ValidPrefix parameter goes false; the Index parameter indicates overscanning.

9.4 Formatting

The following routines illustrate Pascal formatting using the formatter (Num2Str) available from the Apple Numeric Group. (These examples are written for clarity rather than efficiency.)

```
{-----------------------------------------------------------------
* Floating-Point Representation.
-----------------------------------------------------------------}
FloatFormat( x : extended;  TotalWidth : integer;  var s : DecStr );

    const  WMax    = 80; { max field width produced by formatter }

    var    f : DecForm;

    begin
       { Coerce TotalWidth to between 10 and WMax : }
       if TotalWidth > WMax then
          TotalWidth := WMax
       else if TotalWidth < 10 then
          TotalWidth := 10;
       { Call SANE formatter : }
       f.digits := TotalWidth - 8;
       f.style := FloatDecimal;
       Num2Str( f, x, s );
       { Pad blanks to make width of exponent digits field = 4 : }
       while length( s ) < TotalWidth do
          s := concat( s, ' ' );
    end;
```

```
{-----------------------------------------------------------------
* Fixed-Point Representation.
-----------------------------------------------------------------}
 procedure FixedFormat( x : extended ;  TotalWidth, FracDigits : integer;
                                var s : DecStr );

     const  WMax    = 80; { max field width produced by formatter }

     var    f : DecForm;

     begin
         { Call SANE formatter : }
         f.style := FixedDecimal;
         f.digits := FracDigits;
         Num2Str( f, x, s );
         { If too long, do in float format : }
         if s = '?' then
             FloatFormat( x, TotalWidth, s )
         { else right justify : }
         else begin
             if TotalWidth > WMax then TotalWidth := WMax;
             while length( s ) < TotalWidth do
                 s := concat( ' ', s );
         end;
     end;
```