SOS Reference Manual

Terric Abella, Sitong Feng, G Pershing, Sheron Wang

February 25, 2021

Contents

1 Introduction

The SOS (Shape Open System) Language is an imperative language with some functional elements designed to render 2D images, especially mathematically interesting images. This manual describes the SOS syntax and the meaning of SOS statements. It is intended to be a complete description of the language features at time of writing.

2 Lexical Conventions

A program in SOS (Sad Oblique Shapes) is first interpreted by parsing it as a string of *tokens*. The following sections describe exactly what tokens are allowed in SOS.

2.1 Identifiers

Most tokens in SOS are identifiers, which refer to types, variables, and functions. An identifier is a string containing only letters, numbers, and underscores, beginning with a letter. The following identifiers are predefined and cannot be overwritten:

```
int    float   bool    array
void   point   vector  affine
```

In addition, the following strings are keywords that can never be used for identifiers:

```
if   then   else   true   false   alias   struct
```

Keywords, like identifiers, are treated as one token.

2.2 Constants

Any string consisting of only numbers and at most one decimal point is a numeric constant. Constants are interpreted as a single token. In addition, either `e` or `E` may be used to indicate an integer exponent for a number represented in scientific notation.

2.3 Comments

`/*`, `*/`, and `//` are special tokens that are used to indicate comments. They are not passed as tokens by the scanner, but rather denote a segment of text to be ignored.

2.4 Symbols

All other characters are interpreted as symbols, like operators and delimiters. With some exceptions, these are parsed individually. The following is the complete list of symbolic tokens:

```
+   -   *   /   %   ^
!   ||  &&  =   ==  !=
<   >   <=  >=  ,   :
(   )   [   ]   {   }
;
```

3 Types

In SOS (Stylistically Observed Structures), every variable is strictly typed. There are some options for casting between types. This section outlines what types exist and what casts are possible.

3.1 Basic Types

SOS has three basic types: `bool`, `int`, and `float`. `bool` is a single bit representing either `true` or `false`. `int` represents a 32-bit integer. `float` represents a 32-bit floating point number. All other types are built using these three types. In addition, the `void` type can be seen as the fourth basic type. Assignments to `void` do nothing, but are sometimes required to form a complete expression.

All the basic types are stored as values and passed by value.

3.2 Derived Types

A new type can be created in three ways: as an `array`, as a `struct`, or as a function. An array is a block of memory with many variables of a specific type, along with an integer giving the length of the array. A struct consists of some number of *fields*, each with their own identifier and type. This allows associated information to be stored and processed together. A function is a stored expression with one or more *arguments* that can be used within the expression.

All the derived types are stored as pointers and passed as pointers. A simple assignment will replace the initial pointer, instead of writing to the location of the pointer. Depending on the type, there are ways to write the data directly.

3.3 Predefined Types

There are three predefined types in SOS: `point`, `vector`, and `affine`. These are all `struct`s that have additional built-in features, for convenience. The `point` and `vector` types are both structs with an x and y field, both floats. The `affine` type is a nine-tuple struct representing a 3 by 3 matrix.

3.4 Type Conversions

Many inbuilt types will automatically be converted to a different type when required. This will always happen as late in any calculation as possible. In each of the following sections, all the possible conversions from a given type will be listed, as well as their meanings. User-defined structs and arrays are never converted. In most cases, such as variable assignment or function application, one statement will expect a certain type, and the required cast will be clear. The only exception is comparisons, where in general `float` is preferred over `int` which is preferred over `bool`, and `point` is preferred over `vector`.

3.4.1 bool

A `bool` can only be cast to an `int`, by the map `false` $\rightarrow 0$ and `true` $\rightarrow 1$.

3.4.2 int

An `int` can be cast to a `bool`, where 0 maps to `false` and all other values map to `true`.

An `int` can also be cast to a `float` by the injection $x \rightarrow x$.

3.4.3 float

A `float` can be cast to a `int` by truncation. This is not to be confused with the floor operation. For instance, 0.5 and -0.5 both truncate to 0.

3.4.4 point and vector

A `point` casts trivially to a `vector` and vise versa, although this is generally discouraged. In most cases, code is more readable and understandable by making a clear distinction between points and vectors. The underlying function of such a cast is that the memory is simply re-interpreted.

3.4.5 void

All types can be cast to `void`, meaning that their types are simply discarded. `void` cannot be cast to any type besides itself.

3.5 Type Identifiers

The names `bool`, `int`, and `float` are all the identifiers for their respective types. Similarly, `point` is the identifier for a point, likewise for all the other derived types.

A `struct` type is referred to by the name given in its struct definition, and `struct` itself is not used as an identifier.

The string `array` *type-name* is the identifier for an `array` of a certain type. This can be nested, for example `array array int` is an array of integer arrays.

A function type is represented by `(`*arg-type-1* `, ...) ->` *return-type*, which is the type of a function with argument(s) of the specified type(s) and the specified return type. This notation is not commonly used, but is required to use a function as the argument in another function, for instance. The arguments can also optionally be named using the format *arg-type-1 arg-name-1* `, ...`.

4 Syntax and Expressions

An SOS (Shapes of Sorrow) program is formed by a series of *statements*. This section outlines all possible SOS statements and their meanings.

4.1 Statements

There are two forms of statements in SOS: *type definitions* and *starting expressions*. The distinction is that a type definition is a purely structural statement, whereas an expression has a value. Many expressions can contain other expressions, while type definitions are always concrete. Only a starting expression may form a statement, all other expressions can only be used within a starting expression.

Throughout this section, the production rules for the SOS grammar will be listed in the following format:

symbol:
    production ...

Italic characters represent another symbol, typewriter strings and symbols represent specific tokens. Here is the production rule for a statement:

statement:
    type-definition
    starting-expression

In addition, the following production rule is important:

expression:
  starting-expression

Meaning any starting expression may be used in any place that calls for an expression.

4.2 Type Definition

The only type of statement that is not an expression is a type definition. As such, these statements do not include any expressions and cannot be included in any other expressions.

There are two ways to make a new type identifier:

type-definition:
  `alias` *new-type-name* `=` *old-type-name*
  `struct` *struct-name* `=` { *prop-type-1 prop-name* `,` `...`}

The first format defines a new type name that is an alias for an existing type name. Internally, this means that every instance of the new type name will simply be treated as the old type name. It is possible for *old-type-name* to be a combination of types, like an `array` or a function type.

The second format defines a new `struct` type with the given fields.

```
    alias year = int
    struct person = {int identifier,int age}
```

4.3 Declarations and Assignments

In SOS, there is no such thing as a declaration without an assignment. That is to say, when a variable is introduced it must also have a value attached. Any declaration defines a *name* by which a variable can be referred to, the *type* of the variable, and the *value* of the variable. Value here is used quite loosely, variables can be functions or other complicated objects that may not have a "value" per se.

Declarations take one of two forms:

starting-expression:
  type-name variable-name = *expression*
  type-name function-name (*arg-type-1 arg-name-1*, . . .) = *expression*

These statements should be seen as essentially the same, with the second differing more as a convenience of notation than for any structural reason. Importantly, while in the first statement, the specified variable will have the specified type, the second variable will have type (*arg-type-1*, . . .) $\rightarrow$ *type-name*, that is to say a functional type. In both cases, the expression itself will have the same type and value as the variable that is defined. The inner expression must have the same type or a type that can be converted to the specified type.

If a variable or function has already been declared, it can be re-assigned as follows:

starting-expression:
    variable-name = *expression*

Which acts the same as the first type of declaration. A function may be assigned this way, where the arguments and argument names remain the same as the first declaration. This is generally discouraged as it will not be immediately clear what these arguments refer to.

```
    int var1 = 3
    int pow (int n, int x) = n^x
    var1 = pow(3,2)
```

4.4 Operators

The main tools for building meaningful expressions from other expressions are operators. The type of an operator expression depends on the types of the expression(s) it acts upon, which will be specified later. All operator expressions are of one of two forms:

expression:
    unary-operator expression
    expression binary-operator expression

4.4.1 Logical Operators

The logical operators are `!`, `||`, and `&&`. `!` is a unary operator which takes a `bool` and returns its negation. `||` and `&&` are binary operators on two `bool`s representing logical OR and AND, respectively. They are both left associative.

4.4.2 Comparison Operators

The comparison operators are `==`, `!=`, `<`, `>`, `<=`, and `>=`. All are binary operators that take two expressions of the same type that return a `bool`. `==` returns `true` if the expressions have the same value, `false` otherwise; `!=` does the opposite. On `int`s and `float`s, the other four operators represent less than, greater than, less then or equal, and greater than or equal.

4.4.3 Mathematical Operators

The mathematical operators are `+`, `-`, `*`, `/`, `%` and `^`; their meanings depend on the types given (all unshown combinations are invalid in SOS):

`int` (op) `int`	basic arithmetic: in order, addition, subtraction, multiplication, division, modulo, and exponentiation. Returns an `int`.
`float` (op) `float`	same as `int`, except for `%`, which is undefined. Returns a `float`.
`point` (op) `point`	`+` and `-` are component-wise addition and subtraction, which gives a `vector`, and `*` is the dot product, which gives a `float`.
`vector` (op) `vector`	same as `point`.
`point` (op) `vector`	same as `point`, but returns a `point` for addition and subtraction. Likewise if the arguments are given in the other order.
`float * point`, `vector`	uniform scale.
[`point` or `vector`] [`*` or `/`] `float`	uniform scale.
`affine * point`, `vector`	affine application. This is the main case where `point` and `vector` give very different meanings.
`affine * affine`	matrix multiplication.

In addition, `-` can be a unary operator, which returns the negation of an `int`, `float`, `point`, or `vector`. All the mathematical operators are left associative. If one of the arguments to a mathematical operator is an array, the operation is automatically applied element-wise on the list.

4.4.4 Sequencing

`;` is a special binary operator that allows for sequencing. Both expressions are evaluated, and the value of the second expression is kept as the value of this expression. This is left associative.

```
    int x = 2345; 0
```

4.5 Function Application

Perhaps the most important expression is function application. This comes in two forms:

starting-expression:
  function-name `(` *expression-1* `,` *expression-2*, ...`)`
  function-name `(` *paramater-name*`:`*expression*, ...`)`

With the first syntax, the arguments are applied to the parameters in the order specified in the original declaration. With the second syntax, the arguments are applied by name, in any order. The two styles cannot be mixed. Not all arguments need to be specified; the type of this expression will depend on exactly which arguments were specified.

```
    int pow (int n, int x) = n^x
    int var1 = pow(2,3)
    int var2 = pow(n:2,x:3)
```

4.6 Conditionals

The conditional expression is formated as follows:

starting-expression:
`if` *expression-1* `then` *expression-2* `else` *expression-3*

Where *expression-1* must resolve to type `bool`, and *expression-2* and *expression-3* must resolve to the same type. The whole expression will have this second type, and it will have the value of *expression-2* if *expresion-1* is `true`, and the value of *expression-3* otherwise.

```
    int x = 100
    int var3 = if x>0 then 1 else 0
```

4.7 Construction

An array can be created using the array construction expression:

expression:
`[` *expression-1* `,` *expression-2* `, ...]`

Where each expression must resolve to the same type. The whole expression will then be a list of that type.

A struct can be created using either struct construction expression:

expression:
`struct-name`{*expression-1* `, ...`}
{*expression-1* `, ...`}

The first version will always work and create a struct of the given name. The second version will determine the type it needs to be based on context. Whenever this is ambiguous, the first version is required.

```
    array arr = [1, 3+5, 7, 9]
    struct person = {int id, int age}
    person tom = {12345, 45}
```

4.8 Variable Reference

Variable reference is always a terminal expression. It is notated simply as:

expression:
    variable-name

And has the same type as the specified variable. In addition, the fields of a struct can be referenced with dot notation:

expression:
    struct-name `.` *field-name*

An element of an array can be accessed by index, where index 0 represents the first element. The syntax is as follows:

expression:
    array-name `[`*expression*`]`

Both struct access and array access can also be used in place of *variable-name* in an assignment expression.
starting-expression:
    struct-name `.` *field-name* `=` *expression*
    array-name `[`*expression*`]` = *expression*

4.9 Literals

Numeric literals are terminal expressions. They are expressed in base 10. Without a decimal point or scientific notation exponent, they will be interpreted as `int`s, otherwise they will be interpreted as `float`s. Commas cannot be used in any way, either for separating thousands or for the decimal point.
expression:
    numeric-literal

4.10 Parentheses

Parentheses can be used to clarify or alter the order of operations. A parenthetical expression is as follows:

expression:
    `(` *expression* `)`

The whole expression has the same type as the inner expression.

4.11 Scope

Scope specifies where identifiers are visible inside an application. The variable can be seen and referred to once they are declared within the program. But the parameters that are declared as a part of a function, can only be used within the expression that follows which defines the function, and lose visibility once the function declaration ends. The variables n in the following block should not be identified as the same as the parameter n inside the parentheses:

```
    int pow (int n, int x) = n^x
    int n = pow (3,2)
```

4.12 Importing Libraries

SOS has only one preprocessor directive: a line with the form

```
    import filename.extension
```

is replaced by the file *filename.extension*. The characters in the name of *filename* must not include newline or `/*`. Before scanning the files in OS, SOS will first try to find if there is any standard library with the *filename*. Even though the file imported is usually a **.sos* file, extension is still required.

As we are going to replace the line, declaring a new variable that is already in imported file is prohibited, but you could override it when needed. Also, the import graph cannot contain a cycle, for example, import A in file B and import B in file A is prohibited.

4.13 Comments

Any text written between the symbols `/*` and `*/` will be ignored. Comments can be nested this way. Additionally, any text between the symbol `//` and the next newline will be ignored.

```
    // a single line comment
    /* a multi line /* nested */ comment */
```

5 SOS Standard Library

The library functions are written with the SOS (SOS Object System) language in separate files which can be used with `import`. Certain library functions employ the external OpenGL library for its graphics utilities, and support extensive graphical operations. This section is not an exhaustive list of the library, but can be taken as a holistic sample of all library functions. All of the following files can be imported at once with the statement `import stdlib`.

5.1 Math

The file `math.sos` contains useful mathematical functions.

`sqrt : float -> float`
Computes the square root of a number.

`sin, cos, tan : float -> float`
Compute the trigonometric functions sine, cosine, and tangent for an angle in radians.

`asin, acos, atan : float -> float`
Compute the inverse trigonometric functions, returning an angle in radians.

`toradians : float -> float`
Converts an angle from degrees to radians.

5.2 Point

The file `point.sos` contains functions for dealing with points and vectors.

`distance : vector -> float`
Determines the distance of a vector (or distance of a point to the origin).

`project : vector * vector -> vector`
Projects one vector onto another.

5.3 Shape

The file `shape.sos` contains functions for dealing with collections of points, such as lines, curves, and shapes.

```
alias path = point array
alias shape = point array
```

`append : path * path -> path`
Appends one path onto another.

`reverse : path -> path`
Reverses a given path.

5.4 Color

The file `color.sos` contains functions for dealing with color.

`struct color = {float r, float g, float b, float a}`

`hsv : float * float * float -> color`
Creates a color with the given hue, saturation, and value.

5.5 Affine

The file `affine.sos` contains functions for manipulating affine transformations.
`scale : float -> affine`
Returns an affine represent a uniform scale by the given factor.

`trans : float * float -> affine, vector -> affine`
Returns an affine representing a translation by the given vector.

`rotate : float -> affine`
Returns an affine representing a counterclockwise rotation, with the angle given in radians.

5.6 Renderer

The file `renderer.sos` contains the main functions for interfacing with OpenGL.
`render : path -> (vector -> color) -> void`
Renders a path using the given color map.

6 Sample Program

```
// renders the dragon curve fractal
import stdlib

// set up affine transformations
affine A = scale(sqrt(2.0)) * rotate(45)
affine B = trans(0.5, 0.5)*scale(sqrt(2.0))*rotate
   (135)*trans(-1, 0)

// recursively defines the dragon curve
curve dragon (int n) =
  if n == 0
  then [{0, 0}, {1, 0}]
  else d = dragon (n-1) ;
/* affine application pointwise to a curve */
  append(A * d, B * reverse(d))

// interpolates a color along the line a-b
```

```
color rainbow (point a, point b, vector position) =
     // hsv color picker
  hsv (position * (b-a) / (b-a) * (b-a), 1, 1)

render (translate (60, 200) * scale(360) * dragon (10)
   , rainbow((0, 240), (480, 240)))
```