Language description

This page describes the **Sunset Language**, a simplified programming language used for engineering calculations.

Features:

- Automatic handling of units
- Reporting in Markdown with LaTeX mathematical formatting

Comments

Comments start with `//` and cause the remainder of a line to be ignored.

```
// This is a comment, the contents of which is ignored by the interpreter/compiler.
x = 35 {mm}

y = 50 {N} // This is also a comment
```

The preferred style is to have all comments on new lines with a space between the `//` character and the beginning of the comment.

Units

Units are physical units of measurement. The standard unit abbreviations can be used and are enclosed in square brackets `{unit}`, such as:

- `{m}` for metres
- `{mm}` for millimetres
- `{s}` for seconds
- `{MPa}` for megapascals

Derived units are described using expressions containing the below operators. Standard order of operations applies.

- `*`, `.` or ``(at least one space) for multiplication
- `/` for division
- `^` for exponents

For example, `{mm^2}`, `{mm*mm}`, `{mm mm}`, `{mm.mm}` are all equivalent, but not equal to `{mmmm}`.

Units with different dimensions can be mixed using these operators. For example `{kN/m}` and `{kN m^-1}` are both equivalent.

Parentheses can be used to group together operations. For example, `{kg m/s^2}` is not equivalent to `{kg (m/s)^2}`, as the latter will resolve to `{kg m^2 / s^2}`.

Values

Values are numbers that are optionally followed by a unit. The following are valid values:

```
40          // This is a 'unitless' value
12 {mm}
35 {kN m}
345,850.598 {kPa}
```

A space is not required between the number and the unit, but is preferred.

```
12 {mm}     // This is valid and preferred
35{kN m}    // This is also valid but not preferred
```

All commas are ignored when processing numbers, and `en` or `En` may be used to raise any number to the `n`th exponent.

```
12e5       // Equal to 1,200,000
12e-2      // Equal to 0.12
14,32.12   // Equal to 1,432.12 as all commas are ignored.
```

Variables

Variables are defined by assigning a variable or expression to a **name**. Names must start with a letter or underscore, and can contain any combination of letters, numbers and underscores.

```
// Valid variable names
length = 35 {mm}
_duration = 2.5 {s}
```

```
// Invalid variable names
3length = 35 {mm}       // Starts with a number
my variable = 68 {s}    // Name contains a space
my@variable = 45 {kg}   // Name contains a non-alphanumeric character
```

Variable metadata

Metadata describing variables can be provided in the lines following a variable by starting the line with one of the below letters and a colon `:`. Each letter is for a specific piece of metadata.

- `s:` for symbols used in reporting calculations in LaTeX format. E.g. `\phi M_{sx}` as the symbol ϕM_{sx}
- `d:` for a description of the variable. E.g. `The bending section capacity of the plate in the x axis.`
- `r:` for a code reference of the variable. E.g. `AS4100 Cl. 4.3.2`
- `l`: for a label to be used when the variable is included in a user interface. E.g. `Bending capacity`

For example, the above variable may be annotated with metadata as follows:

```
bendingCapacity = 1500 {kNm}
    s: \phi M_{sx}
    d: The bending section capacity of the plate in the x axis.
    r: AS4100 Cl. 4.3.2
```

Tabs are recommended before the metadata descriptors for readability.

Symbol shorthand

Angle brackets `<symbol>` may be used as shorthand for the definition of a symbol following the declaration of a variable.

```
// The following two definitions are equivalent, with the second using the symbol definition shorthand
width = 150 {mm}
    s: b
    d: The width of the plate.

width <b> = 150 {mm}
    d: The width of the plate.
```

If the symbol doesn't contain any spaces, it may be used as an alternative to the name in the following calculations.

```
width <b> = 150 {mm}
    d: The width of the plate.

thickness <t> = 10 {mm}
    d: The thickness of the plate.

// The following two expressions are equivalent
area <A> = b * t
area <A> = width * thickness
```

If a symbol can just be used as a name (i.e. it doesn't contain any invalid characters), it may be defined as a name by starting the name with `@`.

```
// Given a variable in which the name is not required:
width <b> = 150 {mm}

// Using just the symbol but starting with @ makes both the symbol and name equal
@b = 150 {mm}

// This cannot be used for any variable where the symbol cannot be later used as a name
// For example, for the following variable

bending_capacity <\phi M> = 0.9

// This cannot be defined as `@\phi M` as it is not a valid name for later calculations due to the space in the symbol
```

Description shorthand

Double quotation marks `"` may be used as shorthand for the description following the declaration of a variable.

The order of the description and the reference do not matter, however convention is for the reference to be placed first.

```
// The following two definitions are equivalent,
// with the second using the description definition shorthand

capacity_factor = 250 {MPa} "The yield strength of steel."
    s: \phi
    r: AS4100 Cl. 2.5.4
```

Calculations

To perform calculations, simply use the following operators and functions to form expressions assigned to a variable.

- `+` for addition
- `-` for subtraction
- `*` for multiplication
- `/` for division
- `^` for exponents
- `sqrt(x)` for square roots
- `sin(x)`, `cos(x)`, `tan(x)` for standard trigonometric functions.
- `asin(x)`, `acos(x)`, `atan(x)` for standard inverse trigonometric functions

```
// Variable definitions with values

width <b> = 150 {mm}
    d: The width of the plate.

thickness <t> = 10 {mm}
    d: The thickness of the plate.

// Calculations are performed by assigning an expression to a variable

area <A> = width * thickness "The cross sectional area of the plate."

// More variables may be defined with values after a calculation is performed

capacityFactor <\phi> = 0.9
yieldStrength <f_y> = 250 {MPa}

// Calculations may be performed based on the results of previous calculations

bendingCapacity <\phi M_sx> = capacity_Factor * f_y * A
d: The axial capacity of the section
r: AS4100 Cl. 4.5.3
```

Reassigning units

Occasionally, units must be reassigned after the calculation. This tends to occur when empirical formulae are used. This can be done by assigning units to a variable using `{unit}`, which converts the value back to a unitless number, then using `{unit}` again to convert the result to the desired unit.

```
compressiveStrength <f^'_c> = 32 {MPa}

# The following empirical formula will result in flexuralStrength being assigned the units {MPa^0.5} or {kg^0.5 / (m^0.5 s^1)}
flexuralStrength <f^'_{ct.f}> = 0.6 * sqrt(compressiveStrength)     # Results in 3.39 {kg^0.5 / (m^0.5 s^1)}
```

Instead, the compressive strength must be converted to a unitless value before the square root:

```
compressiveStrengthUnitless = compressiveStrength {MPa}             # Results in 32 with no units

# Note that it is important to get the units right here, as the following would be an error:
compressiveStrengthUnitless = compressiveStrength {kPa}             # Results in 32,000 with no units

# To avoid confusion, we recommend that _unit is used for any variables that are converted to a unitless quantity
compressiveStrength_MPa = compressiveStrength {MPa}
```

Once this conversion is done, the flexural strength can be calculated in a unitless form then converted back to the correct units:

```
flexuralStrength_MPa = 0.6 * sqrt(compressiveStrength_MPa)          # Results in 3.39 with no units
flexuralStrength <f^'_{ct.f}> = flexuralStrength_MPa {MPa}          # Results in 3.39 {MPa}
```

This can be shortened to a single expression to avoid creating the additional `compressiveStrength_MPa` and `flexuralStrength_MPa` variables.

```
compressiveStrength <f^'_c> = 32 {MPa}
flexuralStrength <f^'_{ct.f}> = 0.6 * sqrt(compressiveStrength {MPa}) {MPa}
```

Reporting

Performing calculations will result in a report being generated in the format of choice. The most common format is Markdown, which can be used to then output PDF reports.

Text

Comments with a single `#` are not included in the report. If `##` is used to start a comment, it is included in the report. Standard Markdown can be used to style the comment.

Calculations

Variables are reported if a symbol is defined for that variable, but are not reported if a symbol is not defined. If a reference is defined it will also be added to the report next to the calculation.

All variables with a description will be printed at the end of the calculation with their description.

```
## #### Plate section modulus
# The two "##"s at the beginning of the comment above is used to signal that it will be included in the report.
# The "#### " following it means that a level 4 heading will be added as per standard Markdown.
# This line and the two lines above will not be included in the report as they begin with only a single #.

## Calculate the **plastic** section modulus of the plate.
# The **plastic** is Markdown for "make 'plastic' bold".

@b = 150 {mm}
    d: Width of the plate.
@t = 10 {mm}
    d: Thickness of the plate.

plasticDenominator = 4                      # This variable will not be reported as it does not have a symbol defined

@Z_p {Example reference} = b * t ^ 2 / plasticDenominator
    d: Plastic section modulus.
```

This will result in the following report:

> ### Calculation of the plastic section modulus
>
> Calculate the plastic section modulus of the plate.
>
> $$ > b = 150 \text{ mm}$$
> $$ > d = 10 \text{ mm}$$
> $$ > Z_p = 3,750 \text{ mm}^2 \quad \text{(Example reference)} >$$
>
> $$ > \$\$Where: -\$b\$: Width of the plate. -\$d\$: Thickness of the plate. -\$Z_p\$: Plastic section m$$

Types of variables

A variable can take on any of the following types:

- Constants
- Expressions
- Elements
- Conditionals
- Lists
- Dictionaries

Constants

A constant can be a single value or an expression that evaluates to a number of constants.

```
# Single value constant
YieldStrength <f_y> = 250 {MPa} "The yield strength of steel."

# Expression containing multiple constants
Area <A> = 100 {mm} * 30 {mm} "Cross-sectional area of plate."

# Expression that evaluates to a constant
AxialCapacity <N> = f_y * A "Axial capacity of plate."
```

Expressions

Expressions are described in the section above on calculations.

Elements as variables in other elements

Elements can also be used as variables in other elements.

If we wanted to calculate the elastic capacity of a section, we might define some elements as below:

```
Section:
    inputs:
        Width <w> = 10 {mm}
        Depth <d> = 100 {mm}

    Area <A> = w * d
    @I_xx = w * d^3 / 12

IsotropicMaterial:
    inputs:
        YieldStrength <f_y> = 300 {MPa}
        Density <\rho> = 7800 {kg / m^3}

Beam:
    inputs:
        Section = Section()
        Material = IsotropicMaterial(YieldStrength: 250 {MPa})

    AxialCapacity <N> = Section.Area * Material.YieldStrength
    BendingCapacity <M> = Section.I_xx * Material.YieldStrength
    Weight <w> = Section.Area * Material.Density
```

Note that for elements used as variables, they do not need to define a symbol as there is no straightforward way of printing them to the screen.

> [!NOTE] Consider whether they should be provided with a symbol or some kind of name for the purpose of reporting and UI generation.

The variables within an element can be accessed with the `.` modifier.

Roadmap

Still to be described

- [x] Branching element behaviour
- [x] Conditionals
- [ ] Arrays and dictionaries
- [ ] Collection functions
- [ ] Options
- [ ] Error handling
- [ ] Comparisons

Implementation

- [ ] Variables and expressions
- [ ] Elements
- [ ] Conditionals
- [ ] Arrays
- [ ] Collection functions
- [ ] Dictionaries
- [ ] Options

Elements

Defining elements

Elements are groups of expressions. Their definition consists of a name, one or many input variables and their default values and a series of expressions. The inputs are defined in an `inputs:` section and the calculations are defined in a `calculations:` section.

The tabs are included for readability but are not strictly required.

For example, a `PadFooting` element may be as below.

```
PadFooting:
    inputs:
        Width <w> = 1199 {mm} "Width of the footing"
        Length <l> = 1599 {mm} "Length of the footing"
        Depth <d> = 799 {mm} "Depth of the footing"

    calculations:
        BearingArea <A_bearing> = w * l
            d: Bearing area of the footing on the ground

        Volume <V> = w * l * d
            d: Volume of the footing
```

The default value of an input variable must be a constant or an element instantiated with constant parameters. As all elements have default values, all instantiated elements can be treated as constants.

Instantiating elements

Elements may be instantiated using default values only, with all parameters entered or with named parameters only and the remaining values as default.

```
PadFootingDefault = PadFooting() # 1199x1600x800 footing
PadFootingAll = PadFooting(1399 {mm}, 2400 {mm}, 900 {mm}) # 1400x2400x900 footing
PadFootingNamed = PadFooting(Width: 1499 {mm}) # 1500x1600x800 footing
```

Conditional execution of element calculations

To conditionally execute calculations with an element, `branch` elements can be created.

This allows elements to dynamically recast themselves to a child element based on certain parameters within them.

Examples:

- Shear behaviour of beam sections
-

> [!NOTE] This may cause quite a lot of unexpected behaviour. Consider the difference between `branch` elements and inherited elements. If using inherited elements only, it may be necessary to prevent overrides of functions and the creation of new inputs in inherited elements (could result in too many rules for the user). This may be described by the Liskov Substitution Principle?
>
> One behaviour we have already used is with `if` statements, where the definition of functions changes depending on the output of another function. Can something similar be used for larger branching behaviour that allows code to be separated into different elements and files?

```
ElementA {
  inputs {
      X = 35 {mm}
  }

  calculations {
      Y = 45 {mm}
  }
}
```

> [!NOTE] Consider whether we should allow for conditional branches of execution within an element. For example, if a certain thing is true, do all of these calculations and if not don't do them.
>
> There may be some benefit to confining this for the purpose of readability and type checking, and using some form of sub-element behaviour if the behaviour is particularly hard to manage.
>
> As an example, think of slender vs. stocky concrete columns. Perhaps a `this` keyword could be used to reroute a particular element down to an inherited element if there are a lot of calculations that don't apply? Then some type checking can be done if necessary.

```
Column:
  inputs:
      Slenderness = 20

  this = if (Slenderness > 20: StockyColumn) else (SlenderColumn)
```

> Essentially use an `if` statement, where the element itself can be assigned as a child element.

Consider how this might work with overridden behaviours - this may be starting to become needlessly complex. Perhaps there is something to be said about partial classes that continue on after the first class is reassigned? Sometimes you want to have common behaviour that uses the results of a particular class. Perhaps the best thing to do then is to be able to pass all of the inputs into another class at once

```
Column:
  inputs:
      Slenderness = 20

  calculations:
      ColumnBehaviour = if (Slenderness > 20): StockyColumn(inputs)
else: SlenderColumn(inputs)
```

This would only work if `StockyColumn` derives from `Column` only and doesn't introduce any additional inputs or other behaviours (otherwise you would end up with some uncontrolled default values). I think we just need to call this a partial element. You would almost never want to pass in

```
Column branch StockyColumn:
  A =
```

What about something like CHS vs UB shear behaviour in steel? There may just be a need for including partial behaviour into an element.

```
Beam:
  ...
  match (Section == CHSSection): CHSBeam

Beam branch CHSBeam:
  # Just continues on calculations from the previous beam
```

Perhaps some of it is just:

```
match (Section):
  is CHSSection:
      # Calculations for CHS beam
      X = CHSCalculationResultX
      Y = CHSCalculationResultY
  is UBSection:
      # Calculations for UB beam
```

```
        X = UBCalculationResultX
        Y = UBCalculationResultY
```

And the compiler picks up if there are any dependencies on if branches that aren't common between all branches of the if statement. This is equivalent to: X = if (Section is CHSSection): CHSCalculationResultX else if (Section is UBSection): UBCalculationResultX Y = if (Section is CHSSection): CHSCalculationResultY else if (Section is UBSection): UBCalculationResultY

A private modifier may be required here such that stronger type checking is imposed - i.e. all public calculations must be duplicated between the different match sections.

The ability of an element to dynamically cast itself to a different element makes some sense - this way a `Column` instantiated with stocky properties is equivalent to a StockyColumn. That said there's not much point in doing this as one wouldn't want to instantiate a StockyColumn with regular `SlenderColumn` properties and create a logical error.

Perhaps the best thing to do here is to treat branches instantiated with different behaviours as abstract types using a `branch` keyword as above. This should resolve itself to a `match` statement which then resolves to multiple `if` statements. Type checking is undertaken.

This precludes the use of multiple inheritance, perhaps we should just copy the C# way of doing things and use single inheritance with multiple interfaces.

Anonymous elements

Anonymous elements group variables with dynamically generated inputs.

To do this, create a new variable with an unused element name and the `.` operator. This will create an anonymous element that is nested within the current element.