The *incomplete* `swak4Foam` reference

Bernhard F.W. Gschaider

June 12, 2013

Contents

List of Figures

List of Tables

1 Introduction

This document gives an overview of the usage of `swak4Foam`. It explains the common paramters, expressions and usable parts. It is not intended as an introduction to `swak4Foam` (for that have a look at http://openfoamwiki.net/index.php/File:Swa

The structure of the document is

- the first part gives an overview of the parsers. The parser is the part that reads expressions entered by the user and interprets them. These parsers are central to `swak4Foam` and this chapter explains
  - the expressions
  - common settings and concepts
  - order of evaluations
- the next part describes the parts of `swak4foam` that are directly usable. This means
  - the utilities (including the popular `funkySetFields`)
  - boundary conditions (including `groovyBC`)

  - the collection of function objects
  - and the function plugins that can extend the parsers
- the last part gives a short descrition on how to use functionality of `swak4foam` in your own programs including a description of how to write your own plugin-function

1.1 Generating a printable version of this document

This document was written in `org-mode` (http://orgmode.org) an outliner mode for the text editor Emacs. This mode offers a number of ways to export the contents to nicely formatted `HTML` and `PDF`. Please don't try to 'improve' the document in any other text-editor as all the indentations etc have special meaning.

Some of the diagrams require external softaware packages (they are automatically called by `org-mode`):

Graphviz http://www.graphviz.org

Ditaa http://ditaa.sourceforge.net

PlantUML http://plantuml.sourceforge.net/

1.2 Authorship and license

This document is licensed under the *Creative Commons Attribution-ShareAlike 3.0 Unported* License (for the full text of the license see http://creativecommons.org/licen
sa/3.0/legalcode). As long as the terms of the license are met any use of this document is fine (commercial use is explicitly encouraged).

Authors of this document are:

Bernhard F.W. Gschaider original author and responsible for the strange English grammar. He is also the current maintainer of this document

(should you do substantial modifications to this document then add yourself to this list and push the changes to a repository where the maintainer can merge them to the main line)

2 The parsers (expression grammar)

The central concept in `swak4Foam` is the *parser*. A parser reads a string with an expression, interprets it according to a grammar and then evaluates the grammar yielding a result.

In principle each parser is composed of three elements pictured in figure 1:

- the `Lexer` which reads the tokens from the string (the lexers are generated by `flex` from a special description file)
- the `GrammarParser` which gets the token from the `lexer` and interprets them according to a specified grammer (the grammar specification is turned into a program by `bison`)
- the `Driver` is the part of the parser that is "seen" by the calling program. It
  - starts the grammar parser and the lexer
  - assists them in decisions like "is this symbol a variable?"
  - collects the results
  - implements concrete data generation actions (like reading fields, getting cell centers, etc)

In the following texts the term *Parser* will refer to this complex of three entities

Which parser is used depends on the entity the calculation is done on and determines the supported functionality (differential operators are for instance not available on `patch`). Figure 2 gives an overview of the available parser/driver and their relations. Parsers with a `« grammar »` in the name implement a parser/lexer pair which is used by the drivers derived from it. Drivers whose names are in *italics* are only abstract classes.

As all drivers are derived from *Common* there is a set of options that is available in all drivers/parser.

Usually the parser used is determined by the using entity (for instance `patch` is used by `groovyBC`) but sometimes (for instance the `swakExpression`-function object) the used parser can by selected by name. These names and a description of the entity the parser works on are given in table 1.

In principle new parsers for different entities can be implemented and selected at run-time (as for instance are the *FAM*-parsers which are located in a separate library that has to be loaded at run-time)

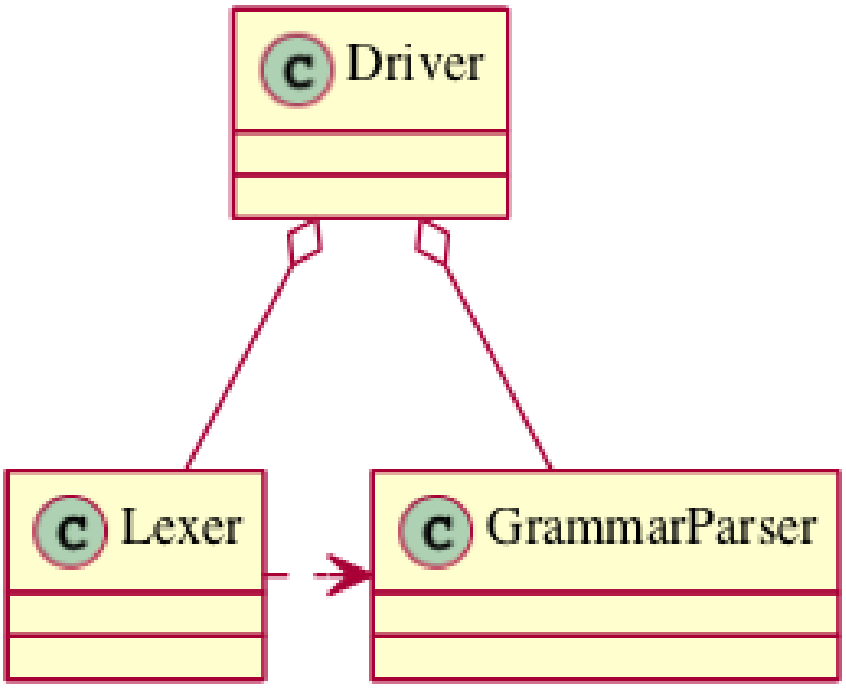

Figure 1: Relationship Driver/Lexer/Parser

2.1 Expressions

The basic syntax of the expressions is modelled after the syntax of expressions in OpenFOAM-programs. This means:

- the syntax is C++
  - the usual precedence rules apply
- if possible the same operators and function names as in OpenFOAM are used

The type of result of an exprerssion does not have to be declared. swak4Foam determines it from the expression. In certain cases the calling entity (BC, functionObject etc) expects a certain type and will complain **after** the evaluation has finished.

Available types are

scalar ordinary floating point expressions

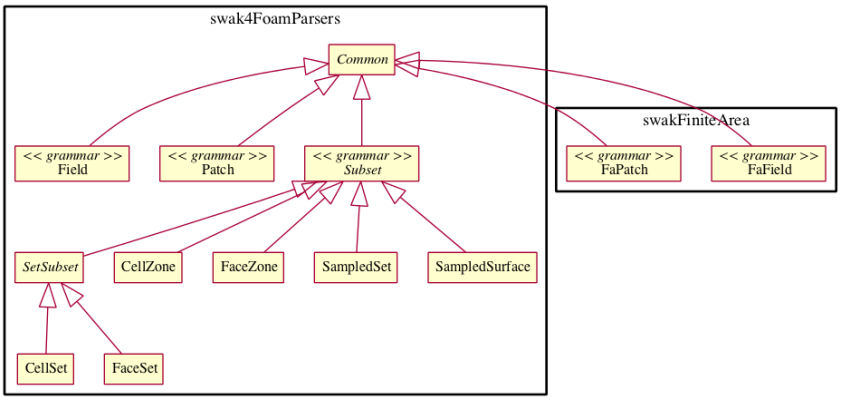

Figure 2: Inheritance relation of the Parsers

vector a three component vector (usually describing a position in space)

tensor a tensor with 3×3 components

symmTensor a 3×3 symmetric tensor (for the components $a_{ij} = a_{ji}$)

sphericalTensor Spherical tensor

boolean results of logical operations (can only be `true` or `false`). Certain parsers implement them with scalars being 0 or 1. If values other then 0 or 1 are found (which can happen for instance due to interpolation) they are interpreted as `true` (only exactly 0 is interpreted as `false`)

If the type of subexpressions for a certain operator/function is incompatible (for instance when trying to add a vector to a scalar) the parser will issue an error message during the evaluation.

Most parsers have two kinds of structures where calculations are performed:

1. the "native" structure of that parser. For instance for the `internalField`-parser this would be the values in the cells
2. the "secondary" structure of the parser. For the `internalField` this would be the value on the faces (`internalField` is special as it also has another *secondary* structure: the values on the vertexes)

Table 1: Selection names for the parsers

name	Description
`internalField`	Calculation on the internal values of a field
`patch`	Calculation on a boundary patch
`faceZone`	On a `faceZone` of the mesh
`faceSet`	On a `faceSet`
`cellZone`	Calculation on a `cellZone`
`cellSet`	Set of cells
`set`	Calculation on a `sampledSet`
`surface`	Calculation on a `sampledSurface`
`internalFaField`	Internal values of a FAM-field (`1.6-ext` only)
`faPatch`	Boundary patch of a FAM-field (`1.6-ext` only)

swak4Foam does **not** automatically convert values between these structures (as it usually involves an interpolation) but specific functions has to be used. The parser will complain if subexpressions of different structures are combined. This usually leads to confusion with constants which are defined on the *native* structure and have to be converted explicitly to the secondary structure if necessary (for instance `toPoint(1)` to use the constant `1` on the vertexes of a patch). Table 2 gives an overview of the structures.

The following sections describe the basic concepts of the expressions.

2.1.1 Constants and type building

This applies to all types of expressions.

Numeric constants can be written in any form they can be written in C++/OpenFOAM. Just a few examples: `42`, `3.1415`, `6.66e2` etc

The symbol `pi` is π.

Vector values can be constructed using the keyword `vector` and three scalar values (which can be constants or expressions that yield a scalar): for instance `vector(1,2,3)` or `vector(1,pos().x,0)`.

Tensors are constructed with the keyword `tensor` and 9 scalar values for the components.

Symmetric tensors are constructed using the keyword `symmTensor` and the 6 components a_{xx}, a_{xy}, a_{xz}, a_{yy}, a_{yz} and a_{zz}.

Table 2: Structures for the different parsers

Parser	*native* structure	secondary structure
`internalField`	Cell values	Face values and point values
`patch`	Face values	Point values
`faceZone`	Face values	none
`cellZone`	Cell values	none
`faceSet`	Face values	none
`cellSet`	Cell values	none
`set`	Values on sample points	none
`surface`	Values on the facets	vertices - not yet implemented
`internalFaField`	Area (face) values	Edge values
`faPatch`	Edge values	Point values

Spherical tensors are constructed using `sphericalTensor` and one scalar value.

If no field or variable with the name `I` exists then this gives the unit tensor.

The logical constants `true` and `false` are available

2.1.2 Operators

These operators are implemented for all the parsers (the usual precedence-rules apply):

`+ - * /` Arithmetic operations

`&` Inner product for vectors and tensors

`^` Cross product of two vectors

`%` Modulo operator. The implementation of this operator differs from the usual implementations: for an expression `a%b` the function is defined in the range $\frac{-b}{2} < x < \frac{b}{2}$ as x (not as usual in the range $0 < x < b$)

`&& ||` The logical *and* and *or* operators

`!` Logical negation

`< > >= <=` Comparisons

`== !=` Equality and inequality-operators

`? :` *if-then-else*-operator. An expression `a ? b : c` means "if the logical expression `a` is `true` the value of expression `b` is used. Otherwise the value of expression `c`"

In addition there are two unary operators:

`-` gives the negative of an expression

`-*` the *Hodge dual* of a tensor expression

- Component operator `.`
  For the data types with multiple components the single components can be accessed as scalar with the operator `.` and the number of the component after the expression (for instance `U.x` gives the `x`-component of the field `U`). Table 3 gives an overview of the components of the various types

Table 3: Component names for the data types

Data type	Components
Vector	x y z
Tensor	xx xy xz yx yy yz zx zy zz x y z
Symmetrical tensor	xx xy xz yy yz zz
Spherical tensor	ii

  For the tensor types there is also the "component" `T` that transposes the tensor (`A.T` gives the transposed tensor for `A`)

  `x`, `y` and `z` for tensors are the rows as vectors.

2.1.3 Mathematical functions available in all parsers

The mathematical functions described in the *Programmers Guide* are implemented in all parsers:

mag(x) Absolute value $|x|$. Implemented for all types. Yields a scalar

The following functions only work for scalars:

pow(x,y) Power x^y. Only implemented for scalars

exp(x) Exponential function e^x

log(x) Natural logarithm

log10(x) Logarithm with the base 10

sin, cos, tan Usual trigonometric functions

asin, acos, atan Inverse trigonometric functions

sinh, cosh, tanh Hyperbolic functions

asinh, acosh, atanh Inverse hyperbolic functions

sqr(x) Square x^2

magSqr(x) Square of the magnitude $|x|^2$

sqrt(x) Square root $\sqrt{x}$

erf(x) Error function

erfc(x) Complement error function

besselJ0, besselJ1, besselY0, besselY1 Bessel-functions

lgamma Logarithm gamma function

These functions depend on the sign of a scalar:

positive(x) 1 if $0 \leq x$. 0 otherwise

negative(x) 1 if $x < 0$. 0 otherwise

sign(x) 1 if x is positive. -1 if it is negative

These functions act on tensors:

diag returns a vector with the diagonal elements

tr Trace of the tensor

dev Deviatoric component

dev2 Deviatoric component times two

symm Symmetric component

twoSymm Symmetric component times two

skew Skew-symmetric component

det Determinant

cof Cofactors

inv Inverse

sph Spherical part of a tensor

eigenValues Return a vector with the eigenvalues of the tensor. Sorted by ascending magnitude

eigenVectors Return a tensor with the eigenvectors of the tensor in the rows. Sorted by ascending magnitude of the eigenvalue

These functions examine the whole fields (in parallel over all processors) and return a field which has one value anywhere:

max(x) maximum of the field (for types with multiple components it return the maximum of each component)

min(x) the minimum

maxPosition(x) Only defined for scalar expressions. A vector with the position where the maximum value is found

minPosition(x) Like `maxPosition` but with the minimum

sum the sum of all the field values

average the average of the field values

There are also binary forms:

min(x,y) Gives back a field that in each "cell" has the minimum of `x` and `y` in that cell

max(x,y) Same for the maximum

These functions build on the random numbers available in OpenFOAM:

rand A random number that is uniformly distributed in the range $[0, 1)$. It **can** take an integer argument that will act as a seed to the random function (if unset the seed 0 is used) but with the number of the current timestep added (so that the random distribution is different at each time-step but still reproducible)

randFixed Similar to `rand` but the distribution of the random numbers will stay the same for all time-steps

randNormal A Gauss-normal distributed random number (seed can be provided). Different at each time-step

randNormalFixed Like `randNormal` but fixed in time

These functions are always available. They are not "mathematical" but help identify certain entities:

id the identification number of an element (for instance the cell number for an `internalField`). This number is only unique on each processor

cpu The processor number an element on is for a parallel run

2.1.4 OpenFOAM-specific functions

The following functions are not available in all parsers. In the description in brackets there will be a shorthand description of the parsers in which it will be available (mind: for the subset parser this doesn't mean that all drivers actually support this function: for instance does the volume function `vol()` not make sense for face zones. Calling this function will result in an error message). Table 4 lists the short descriptions.

Table 4: Shorthand for the parsers

Parser	Shorthand
`internalField`	F
`patch`	P
`subset`	S
`faInternalField`	FF
`faPatch`	FP

- Information about the mesh
  These functions give information about the mesh and are used without arguments:

  pos() Position of the native structures of the parser (for instance cell centers for `internalField`) (F, P, S, FF, FP)

  vol() Cell volumes (F, S)

  area() Face area as a scalar (F, P, S, FF)

  pts() Positions of the vertices (F, P, S, FP)

  fpos() Positions of the faces/edges between cells (F, FF)

  fproj() surface field with the projection of the face onto the Cartesian coordinates (F, FF)

  face() Face vectors (F, FF)

  dist() Scalar field that gives the distance to the nearest wall (using `wallDist`) (F, P)

  nearDist() Scalar field that gives the distance to the nearest wall (using `nearWallDist`)(F)

  rdist() A field with the distances from a given vector (shorthand for `mag(pos()-v)`) (F, P, FF)

  length() Edge length (FF, FP)

  Sf() Surface vectors (P, S, FP)

  Cn() Neighbour cell center position (P)

  Fn() Neighbour face center position (FP)

  delta() Cell center to face center vector (P, FP)

  weights() Patch weighting factors (P, FP)

  normal() Normal vectors (P, S, FP)

  These functions are only available in the `internalField`-parser and identify cells, faces or points belonging to a certain group. Most of them take a name as an argument. The result is a boolean field:

  set(name) `True` for all cells in the cell-set `name`

  zone(name) `True` for all cells in the cell-zone `name`

  fset(name) `True` for all faces in the face-set `name`

  fzone(name) `True` for all faces in the face-zone `name`

pset(name) `True` for all points in the point-set `name`

pzone(name) `True` for all points in the point-zone `name`

onPatch(name) `True` for all faces on the patch `name`

internalFace() `True` for all faces which are **not** on a patch

This function is only implemented for the Subset-parser:

flip() For face-zones and face-Sets this gives the orientation of the face. 1 if the face is oriented in the "right" direction, -1 if not. Used to get consistent mass flows etc across these sets/zones

- Information about time
  Some special functions implemented in all parsers:

  oldTime(fieldName) value of a field at the last time

  deltaT() Scalar field with the current time-step size

  time() Scalar field with the current time

- Differential operators
  The differential operators are only available in the `internalField`-parser. They are available in various forms. In the following list an argument like `cellExpr` means "an expression of any type defined in a cell", an argument `faceScalar` means "only a scalar defined on a face is valid here"

  div(cellExpr) Divergence of tensor and vector fields

  div(faceScalar,cellExpr) Divergence with a "face flux"

  div(faceExpr) Divergence of a value defined on faces

  grad(cellExpr) Gradient

  curl(cellVector) Curl of a vector field

  magSqrGradGrad(cellScalar) Whatever the name says

  snGrad(cellExpr) Surface normal defined on the faces

  laplacian(faceScalar,cellExpr) Laplacian with an inhomogeneous constant defined on the faces

  laplacian(cellScalar,cellExpr) Laplacian with an inhomogeneous constant defined in the cells

  laplacian(cellExpr) Laplacian without a constant

ddt(cellFieldName) this only works for fields for which the last time-step is stored. Time derivative

d2dt2(cellFieldName) Second time derivative

meshPhi(cellVector) Additional flux by the mesh movement

meshPhi(cellScalar,cellVector) Additional flux

flux(faceScalar,cellExpr) Flux

These functions give the explicitly discretized form. For a more detailed explanation see the *Programmers Guide*.

The above functions are also implemented (if appropriate) in the `faInternalField`. Additionally these functions are implemented there:

lnGrad(areaExpr) Like `snGrad`

- Functions that interpolate
  These functions interpolate fields between the native and the secondary structure of a parser

interpolate(cellExpr) Interpolates to the faces (F, FF)

interpolateToPoint(cellExpr) Interpolates to points (F)

interpolateToCell(pointExpr) Interpolates to the cells (F)

toPoint(faceExpr) To the point values (P, S, FP)

toFace(pointExpr) To the cell values (P, S, FP)

These functions are not strictly interpolations, but are used to calculate a cell value from a face value. They are described in detail in the *Programmers Guide*:

integrate(faceExpr) Integrate over the faces(F, FF)

surfSum(faceExpr) Sum the values on the faces(F, FF)

faceAverage(faceExpr) Average of the face values(F, FF)

reconstruct(faceScalar) Reconstruct a vector field from the face fluxes (F)

These two functions are for quickly generating constant fields:

surf(scalar) Generate a constant face-field (no interpolation necessary) (F, FF)

point(scalar) Generate a constant point-field (F)

- Other fields
  These functions take a field name and return a field from another place. They are only available in the patch parser:

  internalField(fieldName) Get the value of the field on the neighbouring internal cells(P, FP)

  neighbourField(fieldName) For a coupled patch get the value of the internal field of the coupled patch (P, FP)

  These functions are only available if the patch has been defined as a `mappedPatch` (`directMappedPatch` in OpenFOAM before 2.0) or a subclass in the `boundary`-file:

  mapped(fieldName) For a mapped patch get the value of the field "on the other side" (P)

  mappedInternal(fieldName) Similar but get the value of the internal field "on the other side" (P)

  This function is the only "differential operator" defined on patches:

  snGrad(fieldName) Gradient of the field `name` in the surface normal direction (P, FP)

2.1.5 Valid names

Valid names in swak4Foam start with either a letter or _ and continue with any number of letters, digits or _.

OpenFOAM allows the definition of names that have other characters too (like : or -). In that case these fields can be accessed using the `aliases`.

2.1.6 Variables and fields

Names that are not functions specified in the grammar can be a number of things. It is tested for a number of other things (the first matching thing is used) and only when nothing of that name is found an error is raised:

1. The name of another mesh. This is only available in the Field-Parser and will be discussed below

2. A timeline. This is an object where a scalar is specified as a function of time. The current simulation time is used.

   For the specification see the discussion of the `timelines`-entry below

3. A lookup table. This works like a timeline but a scalar (that can be different in each "cell") has to be specified between `(` and `)`

   For details see the discussion of `lookuptables` below

4. A field or a variable. Fields are `GeometricFields` that are usually declared and used by the OpenFOAM-solver. Depending on the application they are either

   - looked up in memory
   - looked up on disc and read in (in this case they **may** be cached in memory)

   Variables are intermediate values that have been assigned a name and are stored in memory (more on the declaration of those below.)

   The usual lookup order rules are (but you shouldn't rely on them anyway and give variables etc names that do not "shadow" regular fields):

   (a) Variable of same name and type is found before a field
   (b) Data types are searched in this order: scalar, vector, tensor, symmetrical tensor, spherical tensor
   (c) Native structure before secondary structure

   Before looking for a field the `aliases` table is checked and if the current name is found there instead the *real name* defined for that alias is searched. This allows accessing fields that have names with characters that are not valid for swak-names.

5. Names of plugin-functions. The concept of plugin-functions is described below

- Fields from other meshes
  If another mesh named `other` has been specified in the field parser (how to specify that see below) then the expression `other(field)`

tries to find `field` on the other mesh and uses the values in the expression (if necessary it interpolates the field to the local mesh. All the usual problems associated with interpolation may occur).

This mechanism does **not** allow the specification of an arbitrary expression on the other mesh. That would be possible with a (yet unwritten) plugin-function.

- Types of variables
  Once a variable has been set for a parser subsequent evaluations can access its value. The variable can be set multiple times during a timestep. At the end of a timestep the value is lost (so the variable has to be set before it can be used).

  There are two special flavors of variables that have to be specified beforehand and change the value that is read:

  stored variables these variables keep their value to the next timestep so they can be used **before** they are set. An initial value for that variable has to be provided.

  delayed variables If this variable is used at a time t then the value which that variable had at the time $t - t_{offset}$ will be used. If that time is before the start-time then a default value is used.

  If a variable sequence is evaluated multiple times during a timestep (for instance because there is a sub-iteration cycle in the solver and a boundary condition is evaluated multiple times) then these variables behave each time as if this was the first time during the timestep and only keep the last value they were assigned for the next time-step. This makes it for instance possible to accumulate things like a mass-flow in a stored variable without bothering how many sub-iterations the non-orthogonal corrector did.

  There are two additional flavors of variables for advanced usage. They only make sense for global variables and the types have to be specified before they are first used:

  StackExpressionResult this variable starts with a size of 0. If a value is assigned than the **uniform** value is appended to this variable (making it grow from a size of N to $N + 1$). The purpose of this variable is collecting multiple values. At the end of a time-step the size of the variable is reset to 0

StoredStackExpressionResult like `StackExpressionResult` but the value is not erased between time-steps. Purpose of this variable is collecting a timeline of a single value (for instance to check convergence)

- Global variables
  There is also the possibility to access global variables. These variables are organized in *scopes* which are a collection of variables. Scopes are only accessed if specified so in the parser. This avoids reading unneeded global variables.There are function objects that can set the values of global variables.

2.1.7 Plugin functions

Plugin functions are functions that can be added to the parsers by loading a dynamic library. They are added to a dynamic lookup-table and treated similar to the builtin functions. The difference in the behavior is that they are **not** polymorphic: that means that the type of the arguments and the return value are fixed. While for instance the function `mag(x)` works for various types of `x` (scalar, vector, tensor ...) for a plugin function `foo(x)` the type of `x` is fixed.

There are two basic types for arguments:

primitive types these are constant values (no expressions possible) of simple types that can be parsed by the usual `Istream`-mechanism in OpenFOAM. The possible primitive types are

word simple names

string character strings enclosed by “”

scalar real values

bool `true` or `false`

label integer values

vector three values enclosed by `()`

parsed values these are values returned by a swak-parser (it does not necessarily have to be the same parser type as the calling one. For instance a plugin-function for a patch-parser can have an argument that is the result of an expression on the internal field)

The first time a parser of a specific type (the field parser for instance) is used and there are plugin-functions registered for that parser then a list of the available functions and there arguments are printed to the standard output. The information given for each function is

- the name
- type of the return value
- the arguments with type and a name that should give a hint on their meaning. The type consists of
  - the name of the parser (or `primitive` if a primitive value is expected) as given in table 1
  - the type expected from that parser

separated by a `/`.

One example is the following output:

```
"Loaded plugin functions for 'FieldValueExpressionDriver':"
  lcFaceMaximum:
    "volScalarField lcFaceMaximum(internalField/surfaceScalarField faceField)"
 psiChem_RR:
   "volScalarField psiChem_RR(primitive/word speciesName)"
```

This means that there is a function `lcFaceMaximum` that returns a `volScalarField` and takes a value of type `surfaceScalarField` as the argument. The function `psiChem_RR` takes the name of a species as the argument.

If the evaluation of parameter expression fails the location in this expression will be given. Also the location in the expression that called the plugin-function (in fact the whole stack if this expression is part of another plugin-function call)

2.2 Parameters

Usually parsers are getting their configuration parameters from an OpenFOAM dictionary (the only exceptions that a non-programming user will encounter are the utilities). For the most commonly used cases these are:

groovyBC the sub-dictionary that has the boundary condition specification (rule of thumb: the one that the `type` is specified in)

function objects the sub-dictionary that specifies the details of the function object (also the one with `type` in it)

Some of the parameters are required, some are optional.

Note: parameters like `expression` are **not** part of the parser specification but are part of the item using the parser. The parser "only" evaluates them.

Description of the parameters are split in two parts:

- parameters common to all parsers. This holds the majority of the parameters including variable specification
- special parameters for concrete parsers

If in the following descriptions a default value for a parameter is specified then the parameter is **not** required.

2.2.1 Common parameters

Parameters for debugging the parser are:

debugCommonDriver Writes debugging information of the `Common` driver like variable evaluations etc. Makes output very verbose. Type: integer. Default: `0`

traceScanning Makes the machine-generated (by `flex`) lexer-code output debugging information. Type: Boolean. Default: `false`

traceParsing Makes the machine-generated (by `bison`) parser-code output debugging information. Type: Boolean. Default: `false`

This option allows switching of warnings that point to a probable problem:

variableNameIdenticalToField if a variable is set to a name that is identical to the name of a that is already present in the current mesh then a warning is issued because this usually indicates a mix-up. If this option is set to `true` then no warning is given. Default: `false`

These settings change the behavior of where fields are looked for by the parser. They may be overridden by the using application (for instance for `groovyBC` searching files on disk is counterproductive. For `funkySetFields` it is necessary):

searchOnDisc Search fields on the disc. Type: Boolean. Default: `false`

searchInMemory Look for files in memory. Either this or `searchOnDisc` has to be set. Type: Boolean. Default: `true`

cacheReadFields If `searchOnDisc` is set and a file has been read from disc it is stored in memory to avoid disc access on subsequent read. Type: Boolean. Default: `false`

This parameter defines the behavior of the `oldTime`-function:

prevIterIsOldTime If for a field no old-time value is stored, but one from a previous iteration then this is used. Type: Boolean. Default: `false`

These parameters are optional and are used for specifying timelines and lookup tables to be used in expressions. The only difference between them is how they are used but the specification syntax is the same:

timelines Single time-dependent values (for instance an in-flow velocity). The format of this is "a list of dictionaries". There is only one entry in that dictionary that is "swak-specific":

name name of the timeline. The timeline will be accessed under that name in expressions. The other parameters depend on the `interpolationTable`-class of OpenFOAM:

fileName The name of the data file

outOfBounds How to behave if an argument outside of the specified data is given (for instance fail with an error)

readerType Type of the reader. Currently only two types are supported:

openFoam the regular OpenFOAM-format which is a list of value pairs: time and value

csv Comma separated values format. This format requires addition parameters.

The default value is `openFOAM`

The following options are only required for the `csv`-format

hasHeaderLine Whether the file has a header line that should be skipped before the actual data begins

timeColumn number of the column of the data that holds the time. Note: the first column has the number 0 (C-convention)

valueColumns List with the column numbers that hold the actual data. Length of the list has to be the number of components in the data type (scalar: 1, vector: 3, tensor: 9)

separator Character that separates the data values in a line. Default: a comma

lookuptables Single values that depend on another variables (for instance a temperature-dependent thermal conductivity). Specified exactly like `timelines` but when used a scalar expression has to be provided.

This optional parameter can be used to define aliases for field and set names:

aliases This is a dictionary that has the information which *real* field name belongs to an alias name. Alias names got to conform to the standard for swak-names. Real names are according to the OpenFOAM-standard (which allows more characters)

- General variable specification
  Variables are specified by the parameter `variables`. If this parameter is not set then no variables are accessible. The value of the parameter can have two forms: either a single string or a list of strings (which is just syntactic sugar to make the variable list more readable). Inside the strings single variable specifications are separated by ; (semicolons). **Note**: the last variable specification also has to be terminated by a semicolon!

  The variables will be evaluated in the order they are declared. A variable can be assigned a value more than once.

  The regular variable assignment is of the form

  ```
  varName=expression;
  ```

  which assigns the result of the `expression` to the variable `varName`. The evaluation of `expression` happens with the current parser and the whole (probably inhomogeneous) solution is saved for further evaluations.

  But variables can also be evaluated on other entities and their value can be used in the *local* parser. This evaluation of *external expressions* is triggered by `{}` after the variable name like this:

```
varName{parserType'name/regionName}=expression
```

This means that `expression` is evaluated with the parser specified between `{}`. The form given above is the most general form. The specification of the `regionName` is only needed in multi-mesh cases if another mesh should be accessed. If omitted the current mesh is used. The `parserTypes` can be one of the parsers specified in table 1 and `name` selects the concrete entity the parser should work on (for instance the patch name or the name of the cell set). If the `parserType` is `patch` then it can be omitted and the specification of the patch name is sufficient:

```
varName{patchName}=expression
```

evaluates the `expression` on patch `patchName`.

In the general case it is only possible to use external expressions if the expression yields a uniform value (for instance a sum) as a general way to interpolate from any entity to any other entity (for instance from a cell set to a patch) in a predictable, logical way is not possible. So if the expression yields a non-uniform value then a warning is issued and the average is used.

The only exception currently implemented is if the current patch is a `mapped` patch and the external expression is evaluated on the "partner patch". In this case the non-uniform result will be mapped to the local patch.

- Special variables specifications
  The two optional values `storedVariables` and `delayedVariables` give swak a hint which variables should be treated special (for an explanation on how these variables work see above)

  `storedVariables` is a list of dictionaries that specify which variables should be stored. The two entries in that dictionary are

  name the name of the variable. If a variable of that name is encountered during the evaluation of expressions or being assigned to then it is treated as a stored variable (which will keep its value until the next timestep)

  initialValue if the variable is accessed before it has been set, then this value is used

In addition swak writes an additional entry (which is used for restarting) if the variables are written out (for instance in a `groovyBC`):

value the current value of the stored variable as a dictionary. Entries in that dictionary are (although they rarely have to be edited) are

valueType word describing the value (for instance `scalar` meaning that the value is a list of scalars)

isPoint whether this value is defined on the *native structure* or the points

singleValue a boolean. If `true` the value is the same for the whole list and therefor only a single value is stored

value list with the actual values (type according to the `valueType`)

The optional list `delayedVariables` holds the information about those. The dictionaries hold the following information:

name the name of the delayed variable

delay how much the value is "delayed" between writing and reading

startupValue value to use if time is smaller that `delay` (and therefor no values can be in the "pipeline")

storeInterval Interval in which values are actually stored (the used delayed values will be linearly interpolated between these values)

And again:

value holds the current value for restarting purposes

- Specification of global variables
  The optional entry `globalScopes` gives a list with the names of the global namespaces that are searched for global variables. These namespaces are searched in the order they are specified in this list

- Specification of the mesh region
  If the case is a multi-region case then the mesh region for this parser can be specified. Otherwise the used region is context-dependent (usually the default mesh is used):

  region Name of the mesh to be used

2.2.2 Parser-specific parameters

Certain drivers/parsers have additional parameters.

- Additional parameters of the field-parser
  This has only one additional parameter:

  dimensions physical dimensions of the result. Depending on the application this parameter may or may not be used. Optional (otherwise the result is dimensionless)

- Additional parameter of the patch-driver
  The only additional parameter here is

  mappingInterpolation A sub-dictionary with the interpolation schemes to be used if this is a mapped patch and mapping with interpolation is used. Optional. If unset this is an empty dictionary

  Also instances of this driver where it is not obvious from the context (for a `groovyBC` it is) a parameter to specify the name of the patch is needed:

  patchName the name of the patch the parser works on

- Additional parameters for the subset drivers
  The additional (optional) parameters for this class of drivers is concerned with what is happening if a field is undefined on the native structure:

  autoInterpolate If this variable is `true` and for instance the parser works on faces and a field is **not** defined as a face-field but is defined as a volume-field then the driver will automatically interpolate the field to the faces. If the variable is `false` then the evaluation will fail. Default value: `false`

  warnAutoInterpolate if this is `true` and `autoInterpolate` is `true` then every time a field is automatically interpolated a warning is issued. Default: `true`

- Additional parameter for `cellSet` and `faceSet` drivers
  To specify which set the driver is working on one parameter is needed:

  setName name of the cell or face-set

- Additional parameter for `cellZone` and `faceZone` drivers
  To specify which zone the driver is working on one parameter is needed:

  zoneName name of the cell or face-zone

- Additional parameters for sampled set and sampled surfaces
  These two drivers have two parameters that determine how field values are mapped to them:

  interpolate if this is `true` then the field values are interpolated to the sample. Otherwise the field is "only" sampled (the value of the nearest cell is used). Default: `false`

  interpolationType This parameter is only read if `interpolate` is `true`. This parameter determines how the interpolation should take place. There is no default value for this.

  Also there are parameters for each of the parsers that are used to look up the surface or the set in a repository (a database that swak has for these structures).

  surfaceName name of the sampled surface the sampled driver should work on

  setName name of the sampled set to work with

  Adding sets and surfaces to the repositories can be done with appropriate function objects. If no surface with the name given by `surfaceName` is present then the specification of the surface is looked for:

  surface a sub-dictionary with the specification of the sampled surface (for details see the OpenFOAM-documentation). This surface is added to the repository under the name `surfaceName`

  A missing set `setName` is treated in the same way: The specification is looked for

  set Specification of the sampled set

  For sampled surfaces two optional entries exist:

  writeSurfaceOnConstruction if set to `true` the surface will be written when it is constructed at the current time in a subfolder `surfaceRepository`

autoWriteSurface if set to `true` the surface is written at every write-time in a subfolder `surfaceRepository`

If one of the above options is set then the following option has to be set:

surfaceFormat format in which the surface should be written

Similar optional entries exist for sampled sets:

writeSetOnConstruction if set to `true` the set will be written when it is constructed at the current time in a subfolder `setRepository`

autoWriteSet if set to `true` the set is written at every write-time in a subfolder `setRepository`

If one of the above options is set then the following option has to be set:

setFormat format in which the set should be written

- Additional parameters for the finite area (FAM) drivers
  The `faInternalField` driver adds the same parameter as the field-driver:

  dimensions physical dimensions of the result

  The `faPatch` driver adds a parameter to determine the name of the patch:

  faPatchName the name of the patch

2.3 Information written for restarting

Certain features of the parsers (especially stored and delayed variables) need to write information to allow an exact restart. For boundary conditions this is the standard behavior and there (for instance in `groovyBC`) that information is written to the field-file.

For other items (especially function objects) no such facility exists automatically. If such a driver has data to write (but only then) it creates at write-time in the current time-folder a sub-folder `swak4Foam` in which it saves a dictionary whose file name is composed of the name of the function object and the type name of the driver. During a restart these files are read and stored and delayed variables are restored to the state they had

at write them. If this is not the desired behavior these files can be deleted before restart.

3 Usable parts

3.1 Utilities

3.2 Boundary conditions

3.3 Function objects

3.4 Function plugins

3.5 Data entry

The main library introduces a subtype of `DataEntry` that is selected under the name `swak` wherever data entries lie `constant`, `polynomial` etc are used. After that a dictionary with additional parameters is required. An example entry would look like this:

```
flowRateProfile swak {
    expression "exp(-t)";
    independentVariableName t;
    valueType patch;
    patchName top;
    integrationIntervalls 100;
};
```

Required entries in the dictionary are

expression the expression to be evaluated

independentVariableName the name of the independent variable that was passed during evaluation (usually this is the time)

valueType this determines the type of parser that is used. Additional parameters for the initialization may be needed and the usual entries like `variables` are of course possible

Only for integrations an additional parameter is needed

integrationIntervalls number of intervals the integration range is divided into.

4 Programming

4.1 Writing plugin-functions

4.2 Adding new parsers