

Updated for Flaka v1.2.2.

Flaka 1.2
häfelingen IT

2/54

1 Introduction

Writing a project's build script is serious business. And so it is when using Ant. Ant does not provide you with any abstraction how the project needs to be build. There is no underlying logic. In fact all what Ant provides are a lot of small work units, calls tasks, that need to be glued together to implement the desired logic. Ant is therefore quite similar to writing a Shell script where you can utilize all those fine masterpieces like `cp`, `mkdir` and `find`. However, Ant lacks any decent control structures. There is no `if-then-else` and there is no `while` ¹. What's more, you have to use a rather unfriendly, sometimes even hostile XML syntax. On the bright side however, consider this Ant snippet:

```
<copy todir="${destdir}">
  <fileset dir="${srcdir}">
    <include name="*.jar" />
  </fileset>
</copy>
```

It's its expressiveness which makes me list this example on the bright side as it is easy to grasp what will happen: files ending in `.jar` are copied from one folder into another folder. Now assume that you have to include some further logic in that snippet above:

- *if a certain folder exists, copy files matching `*.zip`, otherwise stick with `*.jar`.*

When staying with Ant, you define a property and use it like

```
<include name="${jar_or_zip}"/>
```

The problem is that you have to define that property elsewhere thus you start to miss part of the logic. This is where I believe that Ant has a true deficit and this is where Flaka kicks in. Flaka provides, amongst other things, with an expression language (EL). Thus you could write

```
<include name=" *.{ mydir.exists ? 'zip' : 'jar' } " />
```

However, even having such a powerful extension like EL I am missing the full power of a programming language's control structures. Yes, I want to have conditionals, repetitive constructs and a decent exception handling. Furthermore, I want to have variables which I can set or remove for pleasure. I don't want to be restricted that such variables may carry strings only. Any data object must be allowed. In summary, this is what Flaka currently is all about:

- Expression Language
- Well known control structures

This pillars are Flakas approach to simplify the process of writing a build script with Ant. You are by no means forced to use all or any of those pillars. You can for example just use the control structure tasks with or without making use of EL or vice versa.

¹ `if` and `while` can be implemented in terms of calling targets depending on whether a property exists or not. Rather awkward ..

1.1 Where to go from here?

- Download ² Flaka and read the [installation page](#).
- Make sure to consult chapter [EL](#). It contains a lot of information on this enormous useful extension.
- Have a closer look in the reference part of this manual for all the gory details.
- Start writing build scripts using Flaka and give feedback ³.

1.2 Conventions

Ant build file examples show a mix of tasks provided by Flaka and by Ant. Ant task do not require a namespace while those provided by Flaka do. Flaka's namespace is

```
antlib:it.haefelinger.flaka
```

and within this manual, the abbreviation `c` will be used for this namespace. Therefore it becomes easy to see who is the provider of a task:

```
<echo> Ant </echo>  
<c:echo> Flaka </c:echo>
```

Thus all build file snippets shown assume that the build file contains the following XML namespace declaration:

```
<project xmlns:c="antlib:it.haefelinger.flaka" ..>  
  <!-- build script example -->  
</project>
```

² See <http://code.google.com/p/flaka/downloads/list>

³ flaka@haefelinger.it

2 Installation

Download ⁴ latest version of Flaka and `drop ant-flaka-x.y.z.jar` into your local Ant installation. All that needs to be done is to put this jar into Java's classpath when running Ant. There are various techniques how to do so. Please read-on what exactly needs to be done.

2.1 Ready, ..

The following **requirements** must be satisfied before you start:

- Flaka requires Java 1.5 or newer ⁵ or newer. You can change the version by setting environment variable `JAVA_HOME`. Have also a look at Ant's manual ⁶ for other environment variables to utilize.
- Ant, version 1.7.0 or newer ⁷.

2.2 Charge, ..

The most primitive technique is to save `ant-flaka-x.y.z.jar` into Ant's library folder `lib`. If you have no clue where Ant is installed, try

```
$ ant -diagnostics | grep ant.home  
ant.home: /opt/ant/1.7.1
```

Saving something in Ant's library folder may not work if you lack required permissions. There is also the disadvantage that when switching to another installation of Ant, you need to reinstall Flaka again. Therefore consider to use option `-lib`:

```
$ ant -lib ant-flaka-x.y.z.jar
```

A pretty nice feature of option `-lib` is, that if the argument is a folder, that folder is scanned for jar files. Therefore you may want to do something like this:

```
$ mkdir $HOME/lib/ant  
$ cp ant-flaka-x.y.z.jar $HOME/lib/ant  
$ ant -lib $HOME/lib/ant
```

This approach has the nice advantage that you simply can drop other jar files into folder `$HOME/lib/ant` to make them reachable without touching the original Ant installation. As already mentioned, this will get handy when you have multiple Ant installations. Notice that option `-lib` can be applied more than once if your jars reside in various folders. Finally notice, that folders are *not* recursively scanned.

When working from the command line, it is rather annoying to provide option `-lib` for each and every call. Fortunately, Ant recognizes environment variable `ANT_ARGS` which can be used to let `-lib` disappear:

⁴ see either <http://download.haefelinger.it/flaka> or use <http://code.google.com/p/flaka/downloads>

⁵ <http://java.sun.com/javase/downloads>

⁶ <http://ant.apache.org/manual>

⁷ <http://ant.apache.org>

```
$ ANT_ARGS="-lib $HOME/lib/ant"
$ export ANT_ARGS
```

The drawback with this technique is that you need to make sure that this variable is set in every environment you start up Ant. This sounds perhaps easier than actually done. Luckily again, Ant reads file `$HOME/.antrc` and `$HOME/.ant/ant.conf` on each and every start-up. It is therefore recommended to set variable `ANT_ARGS` in one of this files⁸ to make just every plain call to `ant` is aware of Flaka. For example:

```
$ cat $HOME/.antrc
ANT_ARGS="-lib $HOME/lib/ant"
```

Finally it should be mentioned, that Ant scans for additional jars in folder `$HOME/.ant/lib`. Thus the following two invocations of Ant are identical

```
$ ant
$ ant -lib $HOME/.ant/lib
```

2.3 Fire!

To check whether your setup is proper, create a small build script and try to execute it. For example, try this

```
$ cat > build.xml << EOF
<project xmlns:c="antlib:it.haefeling.flaka">
  <c:logo>
    Hello, #{property['ant.file'].tofile.name}
  </c:logo>
</project>
^D
```

Then, when created, try to execute it with your Flaka equipped Ant. You should see something like

```
$ ant
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::                      Hello , build.xml                    ::
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
BUILD SUCCESSFUL
[. .]
```

if everything is well setup.

⁸ when doing so, you dont need to *export* that variable

The *Unified Expression Language* ⁹, further in this document abbreviated as *EL*, is a special purpose programming language typically used for embedding expressions in web applications. While EL is part of the JSP Specification, it does not depend on JSP and can therefore be used in a variety of other contexts. One such context is Ant. Consider the following example ¹⁰:

```
<echo>
  Modified #{ format('%tD', file(project).mtime) }
<echo>
```

The EL expression here, a function call with two arguments, utilizes two standard of Flaka's standard functions, namely `format()` for creating a string and `file()` for turning something into a file object. Function `format()` accepts an arbitrary number of arguments. Two arguments are provided in this example. A string argument as the first argument while the second argument is the object returned by querying property `mtime` on the file object returned `file(project)`. Argument `project` is one of Flaka's implicit objects. In particular, `project` represents the current Ant project. The evaluation of `file(project)` is the project's base directory as file object. Such a file object has various properties. One property is the file's last modification time available as property `mtime`. The whole EL expression is embedded in the textual context of Ant's standard `echo` task. The embedment is done using `#{` and `}` respectively. When executed, this snippet produces something like

```
[echo] Modified 10/29/10
```

Here are some further examples of EL expressions:

```
7 * (5.0+x) >= 0           ; (1)
a and not (b || false)    ; (2)
empty L ? null : L[0]     ; (3)
list('a', 'b')            ; (4)
split('a,b', ',', '')    ; (5)
project.name               ; (6)
size(file('.').list)      ; (7)
```

The first expression (1) shows a algebraic equation. Notice the usage of 5.0 being of type float, 7 being an integral type and furthermore x as variable. A Boolean expression is shown in the second example using operators `and`, `not` and operator `||`. The same expression could also have been written like `a && !(b or false)`. Example (3) shows operator `empty` and conditional operator `?:`. The expression could be read like: check whether a (list) object is empty. If empty, return `null`, otherwise return the list's first item. The fourth and fifth expression shows two list generating functions - `list()` just collects all arguments into a list while `split()` breaks a string apart based on a regular expression. The project's name is queried in example (6) while example (7), calculates the number of files and folders in the current working directory.

⁹ http://en.wikipedia.org/wiki/Unified_Expression_Language

¹⁰ To make this example work, one need to globally enable EL for all string contexts. By default, EL is *not* globally enabled. To enable it globally, use Flaka's task `<c:install-property-handler/>`

3.1 EL References

It turned out that it is a good idea to have a clear distinction between properties and EL expressions. Therefore, an EL expression must be enclosed by `{..}` rather than by `$. .}`. An enclosed EL expression is called a reference to an EL expression. Similarly is `$. .}` named a reference to an Ant property. Consider:

```
<echo>
  ${3 * 4} ; Property reference
  #{3 * 4} ; EL expression reference
</echo>
```

Ant is by default not aware of EL. Thus the EL reference `#{3 * 4}` would not be evaluated and would be passed as is. To let Ant evaluate EL expression references, EL must be enabled ¹¹. Once enabled, EL references can be used wherever Ant property references can be used. Consider:

```
<echo>
  #{3 * 4} ; still #{3 * 4}
</echo>
<c:install-property-handler />
<property name="twelve" value="#{ 3 * 4 }" />
<echo>
  #{3 * 4} ; 12
  ${twelve} ; 12
</echo>
```

3.1.1 Enabling

To enable EL references in addition to Ant property references, Ant's standard property helper engine must be exchanged. This is best done using Flaka's [install-property-handler](#) task. This task can be used everywhere. It is however recommended to (a) call it only once (b) to call it as early as possible.

Starting with version 1.8 Ant provides a task for switching property helpers. This task can also be used to change the characteristics how Ant property references are handled. To support this new property API, Flaka provides a second Antlib in addition to the standard one. Consider:

```
<project xmlns:prop="antlib:it.haefelinger.flaka.prop">
  <propertyhelper>
    <prop:elreferences />
  </propertyhelper>
  <echo>
    #{ 3 * 4 } ; 12
  </echo>
</project>
```

For those who want to stay more in-line with Ant property references, Flaka ships with a property evaluator. When this evaluator is plugged in, `$. .}` will be evaluated as EL reference. Consider:

¹¹ see task [install-property-handler](#) for how to do this


```
<project xmlns:prop="antlib:it.haefelinger.flaka.prop">
  <propertyhelper>
    <prop:elevaluator />
  </propertyhelper>
  <echo>
    ${ 3 * 4 } ; 12
  </echo>
</project>
```

3.1.2 Conversion

Ant properties are strings, thus Ant properties are used in string contexts and consequently the same applies for EL references. When an EL reference is seen, the EL expression is firstly evaluated into an object of some type T . Then, in a second step, that object value is coerced¹² into a string value.

3.1.3 Nested References

Nested references are *not* supported. The following example, which tries to do some sort of meta-programming on implicit object project, is therefore illegal

```
<c:echo>
  #{ project.#{ property } } ; illegal
</c:echo>
```

3.1.4 Nested Properties

Ant property references $\${. .}$ are resolved before any EL reference is resolved. Consider:

```
<property name="pname" value="basedir" />
<c:echo>
  #{ project.#{pname} } ; project.basedir
  #{ project.#{ 'basedir' }} ; no way
</c:echo>
```

3.1.5 The Great Escape

How about getting a character sequence like $\#{ 3 * 4}$?. In other words, how to prevent a EL reference from being evaluated? There are basically two techniques available. Consider:

```
<echo>
  #{ '#{3 * 4}' } ; #{ 3 * 4 }
  \#{ 3 * 4 } ; #{ 3 * 4 }
</echo>
```

The first line shows the recommended technique, i.e. embed the character sequence $\#{$ in a EL string. The second line demonstrates the usage of $\backslash\#{$ as escape sequence. Using this escape sequence is counter intuitive. Consider:

¹² implicitly type converted

```
<echo>
    #{ 3 * 4 }      ;    12
    \#{ 3 * 4 }    ;    \#{ 3 * 4 }
    \\#{ 3 * 4 }   ;    \\#{ 3 * 4 }
    \\#{ 3 * 4 }   ;    \\#{ 3 * 4 }
    \\\#{ 3 * 4 }  ;    \\\#{ 3 * 4 }
    \\\#{ 3 * 4 }  ;    \\\#{ 3 * 4 }
</echo>
```

In other words, when character sequence `\#{` is seen, then character `\` is consumed and `#{` does not start an EL reference, i.e. the sequence is taken literally. It has to be noted, that character `\` alone does not have any meaning and is especially not a general escape character. Thus any number of contiguous backslash characters before an EL reference successfully disables its evaluation.

Given this behaviour, how is it then possible to create backslash characters before an evaluated EL reference? As an example, assume that we have a EL variable named `host` and our task is to generate a UNC ¹³ notation. Consider:

```
<echo>
    \#{ host }      ;    \#{ host }
    #{ format( '\\\\%s', host ) } ;    \\myhostname
</echo>
```

Please notice that escaping a property reference is different than escaping a EL reference. A property reference is taken literally if prefixed by character `$`. In addition, every double dollar sequence `$$` is reduced into a single dollar character. Consider:

```
<echo>
    ${foo}          ; bar
    $$${foo}        ; ${foo}
    $$$${foo}       ; $bar
    $$$$${foo}      ; $$${foo}
    foo $ bar       ; foo $ bar
    foo $$ bar      ; foo $ bar
</echo>
```

3.1.6 Whitespace

Trailing and leading whitespace is meaningless within EL references. This is different from Ant properties where whitespace is meaning full. Consider:

```
<property name=" p " value="weired name ' p '" />
<property name="p" value="regular name 'p'" />
<c:echo>
    ${p}                ; regular name 'p'
    ${ p }              ; weired name ' p '
    #{ p }              ; regular name 'p'
    #{ property[ ' p ' ] } ; weired name ' p '
</c:echo>
```

¹³ Uniform Naming Convention, see also [http://en.wikipedia.org/wiki/Path_\(computing\)#Uniform_Naming_Convention](http://en.wikipedia.org/wiki/Path_(computing)#Uniform_Naming_Convention)

3.1.7 Syntax Errors

A syntax error in a EL expression will be silently ignored. Furthermore, the EL engine will try to recover by ignoring all characters until either character } or <eof>, the end of input, is seen. Consider:

```
0#{ 3 * 3      ; 0
0#{ 3 * }3     ; 03
0#{ 3 * 4 }3   ; 0123
```

3.1.8 EL Awareness

Flaka is naturally EL aware. Regardless of whether EL has been enabled or not, all Flaka tasks understand EL references #{. .} without any additional action.

In addition, attributes of tasks provided by Flaka can be *typed*. A argument provided to a *typed* attribute is expected to be an EL expression. Evaluating that expression will be done in the context of an expected *type*. Consider attribute `test` of Flaka's `when` task:

```
<c:when test=" file(path).isdir ">
  <!-- do something with that dir folder .. -->
</c:when>
```

Attribute `test` is typed and it's expected type is `boolean`. The argument must be a an EL expression, here `file(path).isdir`. That EL expression will be evaluated into some object and then, in a second step coerced into a boolean value.

Notice that Ant and EL references can be used in typed attributes. After all, a typed attribute is a regular attribute as far as Ant is concerned. It is just Flaka who is evaluating the expression as EL. Thus consider this kind of meta-programming , that all attributes are handled by the currently installed property handler. Thus even for attribute `test` the normal attribute rules apply. Consider this kind of meta-programming where by default an object is queried for property `isdir` subject to whether variable `prop` is empty or not. If not empty, then the value is queried for property `prop`:

```
<c:when test=" obj.#{ empty prop ? 'isdir' : prop} ">
  ..
</c:when>
```

The following sections will handle some advanced issues regarding EL references and especially their relation with Ant property references.

3.2 Data Types

The following types can be used when writing EL expressions:

- *null* to represent the absence of any data
- *integer* for integer values
- *float* for any floating point values
- *boolean* to express Boolean logic

- *string* to represent char sequences
- *list* for iterable types
- *map* for dictionary like types
- *file* representing file objects
- *project* representing Ant project instances
- *object* for all other data values

The question is how to get a element of those? EL defines literals denoting elements of type `null`, `integer`, `float`, `boolean`, and `string`. Furthermore there are implicit objects and functions, see below.

3.2.1 Null Literals

The `null` type contains only one data element also called `null`. From a semantic point of view it is used to represent the absence of any data. Within EL, `null` has an interesting characteristic: it can be asked whether it has a certain property and the answer will always be `null` again. Consider:

```
null['any property'] ; null
null.mtime           ; null
```

This rather different from other languages where asking `null` for a property is asking for trouble, i.e. `null` pointer exception and the like. Notice also the following differences between operator `empty` and function `nullp()` when working with value `null`:

```
empty null           ; true
empty list()         ; true
empty ''             ; true
nullp(null)          ; true
nullp(list())        ; false
nullp('')            ; false
```

3.2.2 String Literals

A string literal starts and ends with the same quotation character. Quotation characters are either the single quote `'` or the double quote `"` character. If the quotation character is needed within the literal, then the escape character `\` must be used. The escape character must also be used if the escape character itself is to be expressed in the literal. The escape character can't be used to escape other characters than the quotation character and the escape character.

```
"abc"               ; abc
'abc'               ; abc
'abc"               ; illegal
"a'c"               ; a'c
'a"c'               ; a"c
'a\'c'              ; a'c
'a\"c'              ; a\"c
"a\"c"              ; a"c
```

```
'a\bc'      ; a\bc  
'a\\bc'     ; a\bc  
'ab\'       ; illegal  
'ab\\\'     ; ab\
```

3.2.3 Object Literals

Well, EL has no notation for *object* literals like literals for type integer and boolean. So, how to get an object in the first place? There are two possibilities:

- use an implicit object; and
- use a function

Once you have an object, you can in addition * use an object's property to retrieve another object. When does a property exist and how to retrieve it? This and other questions are answered in section [EL Properties](#) while section [EL Implicit Object](#) lists available implicit objects and section [EL Functions](#) is about functions to be used.

Notice that EL does also not provide a notation for arrays or list objects nor is there a `list` data type. Nevertheless, Flaka provides a `list()` function to create a collection of arbitrary objects. There is also task `for` able to iterate over collection types and there is function `size()` which returns the number of items in a collection. How does this work? EL uses a concept called *duck typing* where a object's type is not given by a class but rather by it's properties: *I call every object that walks, swims and quacks like a duck, a duck*. Thus properties are looked up during runtime and a object provides all required properties, the object is applied.

3.3 Variables

The EL language does not allow you to create variables and must thus be created by other means ¹⁴. Nevertheless, variables can be used within expressions. When a *name* is evaluated, then *name* is looked up

- as implicit object; or
- as entry in EL's variable dictionary; or
- as Ant property name

in this particular order, consider:

```
<property name="p" value="property" />  
<property name="q" value="property" />  
<c:let>  
  q = 'variable'  
</c:let>  
<echo>  
  ${p}           ; 'property'  
  #{p}           ; 'property'  
  ${q}           ; 'property'  
  #{q}           ; 'variable'  
  #{property.q } ; 'property'  
</echo>
```

¹⁴ See task [let](#) for creating EL variables, properties and overriding properties

If no entry can be associated with *name*, then `null` will be the look-up's result. This has the interesting consequence, that EL references are *always* disappearing, i.e. can always be resolved. Whereas unknown properties remain as property references. Consider:

```
<echo>
  ${not_a_name} ; ${not_a_name}
  #{not_a_name} ; ''
</echo>
```

3.4 Operators

The following operators are defined in [EL](#):

- operator `empty` checks whether a variable is empty or not and returns either `true` or `false`. It is important to understand that `null` is considered empty.
- condition operator `?:` can be used for branching in expressions. The expression `cond ? a : b` evaluates expression `cond` in a Boolean context. If `eval(cond)` returns `true` then the result of the expression is `eval(a)` and otherwise `eval(b)`.
- operators `.` and `[]` are used to query properties on objects. See also section [Properties_](#).
- logical operators `not`, `and` and `or`
- relational operators `==`, `!=`, `<`, `>`, `<=` and `>=` (resp. `eq`, `ne`, `lt`, `gt`, `le` and `ge`).
- usual arithmetic operators like `+`, `-`, `*`, `/`, `mod` and `div` etc.

3.5 Implicit Objects

Flaka provides implicit objects that can be utilized writing [EL](#) expressions:

| Implicit Object | Description |
|-----------------------|---|
| <code>project</code> | The current Ant project as object. To query the project's default target, base folder and other things (see also project properties and natural properties). |
| <code>property</code> | Use this object to query project properties. |
| <code>e</code> | The mathematical number <code>e</code> , also known as Euler's number. |
| <code>pi</code> | The number PI |

The following implicit objects are deprecated:

| Implicit Object | Alternative |
|------------------------|---------------------------------|
| <code>reference</code> | <code>project.references</code> |
| <code>var</code> | <code>project.references</code> |
| <code>target</code> | <code>project.targets</code> |
| <code>taskdef</code> | <code>project.taskdefs</code> |
| <code>macrodefs</code> | <code>project.macrodefs</code> |
| <code>tasks</code> | <code>project.tasks</code> |
| <code>filter</code> | <code>project.filters</code> |

3.6 Functions

This sections presents functions defined by Flaka and which are available without any further action. Notice that providing own functions is currently not possible. A note about the conventions used in the notation of function signatures:

- T is a placeholder meaning any type
- $T..$ means that a variable list of arguments of type T can be used
- $name:T$ is used to give a parameter a name which is then used in the follow up explanation of this function.

3.6.1 Generic Functions

`typeof(T):string` A function to determine the object's type:

```
typeof(null)           ; 'null'
typeof('')            ; 'string'
typeof(3)              ; 'integer'
typeof(pi)             ; 'float'
typeof(true)          ; 'boolean'
typeof(list())        ; 'list'
typeof(file('.','))   ; 'file'
typeof(project)       ; 'project'
typeof(project.properties) ; 'map'
typeof(other)         ; 'object'
```

`nativetype(T):string` Use this function to determine native type, the type of the underlying implementation, of the given argument.

```
nativetype(null)      ; ''
nativetype('')        ; 'java.lang.String'
nativetype(3)         ; 'java.lang.Long'
nativetype(pi)        ; 'java.lang.Double'
nativetype(true)     ; 'java.lang.Boolean'
nativetype(list())   ; 'java.util.ArrayList'
nativetype(file(project)) ; 'java.io.File'
nativetype(project)  ; 'org.apache.ant.tools. ↵
    Project'
nativetype(project.properties) ; 'java.util.Hashtable'
nativetype(file(project).mtime) ; 'java.util.Date'
```

`size(T):integer` The size of the object is given by the number of entities it contains. This is 0 (zero) for all primitive types like integer, null, float, boolean. Otherwise the object's size is determined via a `size` or `length` property.

```
size(null)           ; 0
size(3)              ; 0
size(pi)             ; 0
size(true)          ; 0
size('abc')          ; 3
size(list(1,2))     ; 2
```

```
size(file(..))      ; see below
size(project)      ; 0
size(object)       ; object.size or object.length or 0
```

If the argument of `size()` is of type *file* and the argument denotes an existing and accessible directory *d*, then `size(d)` returns the amount of files and folders in *d*. Otherwise, if the argument denotes an existing and accessible file, then the length of that file is returned. Otherwise, `size()` will return 0.

`nullp(T):boolean` Evaluates to `true` if object is the null entity and `false` otherwise. Compare this function with operator `empty` which returns `true` if either the object in question does not exist or if literally empty, for example the empty list or the empty string.

```
nullp(null)        ; true
nullp(list())      ; false
nullp('')          ; false
empty null         ; true
empty list()       ; true
empty ''           ; true
```

3.6.2 File and Folder Functions

`file(T.):file` A function to create a file object. If `file()` is called without argument, then the current working directory is returned. Otherwise, if the function is called with one argument and that argument is already a file, the argument is simply returned. Otherwise, if the argument's type is `project`, then the base directory of that Ant project is returned. If the argument is a `list` type, then a file is constructed based on the list's elements. Otherwise the argument is stringized. If the stringized argument consists only of whitespace, then the current working directory is returned while otherwise that string is taken as the file's path name. If `file()` is called with two or more arguments, then the behaviour is the same as if `file()` would have been called with one list argument where the list consists of the function's arguments.

```
file()              ; your JVM current work directory
file(' ')          ; file()
file(file(arg))    ; file(arg)
file(project)      ; project.basedir
file(list('/',a))  ; file('/',file(a))
file(other)        ; file(format('%s',other))
file(a,b,c)        ; file(list(a,b,c))
file(a,list(b,c),d) ; file(list(a,b,c,d))
```

3.6.3 String Functions

`concat(T.):string` Creates a string by concatenating all *stringized* objects. If no object is provided, the empty string is returned. Thus to create the string `foobar`, try

```
concat('foo','bar') ; 'foobar'
```


`format(string,T.):string` This function is a Swiss army knife for creating a string based on existing objects. The function expects a format string as first argument followed by any number of arguments. The optional arguments are used to construct the result string based on format instructions embedded in the first argument. Some examples:

```
format('foobar')           ; 'foobar'
format('foo%s', 'bar')     ; 'foobar'
format('%s%S', 'foo', 'bar') ; 'fooBAR'
format('%s', list('a',2))  ; '[a, 2]'
```

The number of format options to be used are almost infinite ¹⁵.

`replace(string, subst:T, regex:T):string` Create a new string by replacing substrings. Substrings to be replaced are described via regular expressions. If no substitute string is given, the empty string is used. The default regular expression is `\s*,\s*` which means, that all commas - including leading and trailing whitespace - are replaced. Arguments are stringized before used.

```
replace('a, b')           ; 'ab'
replace('a, b', '')       ; 'ab'
replace('a, b', '', '\s*,\s*') ; 'ab'
replace(true, 'false', 'true') ; 'false'
```

The behaviour of this function is undefined if called without arguments.

`split(string, regex:T):list` A function to tokenize a string into a list of strings. Tokens are separated from each other by text matching a given regular expression. Arguments are stringized before used. If no regular expression is given, then `\s*,\s*` is used. The behaviour is undefined if no arguments are given.

```
split('a,b')              ; list('a','b')
split('a:b', ':')        ; list('a','b')
```

`trim(string):string` A convenience function to remove leading and trailing whitespace from a string (stringized object). This function can be expressed in terms of function `replace()` like

```
trim(s)                   ; replace(s, '', '^\s*/\s*$')
```

`ltrim(string):string` Similar to function `trim()` above but only leading whitespace is being removed.

```
ltrim(s)                  ; replace(s, '', '^ \s*')
```

`rtrim(string):string` Similar to function `trim()` above but only trailing whitespace is being removed.

```
rtrim(s)                  ; replace(s, '', '\s*$')
```

¹⁵ Compare <http://download.oracle.com/javase/1.5.0/docs/api/java/util/Formatter.html>

`matches(text:object:regex:object):boolean` A function to match a character sequence against a regular expression. If the regular expression matches a subsequence of text, the function returns `true`. If the regular expression is invalid, the function's behaviour is undefined.

```
matches('aZc','aZc')      ; true
matches('aZc','Z')        ; true (!)
matches('aZc','^Z$')      ; false
matches('aZc','^\w+')     ; true
matches('aZc','[~ab[')    ; undef.
```

Be aware that `matches()` works on subsequences whereas Java's String method `matches()` works on the entire input sequence. However, it is rather easy to let `matches()` return `true` only if the entire input sequence is covered by the regular expression: wrap the original regex using anchors:

```
matches(concat('^',myregex,'$'),...)
```

`glob(text:object:regex:object):boolean` Similar to function `matches()` described in the previous section: the input is matched against a glob expression. A glob expression may contain wildcard characters like `*` for any number of characters, `?` for exactly one character and `[.]` for a set of known characters. Furthermore wildcard characters can be escaped using character backslash `\`. Function `glob()` always takes the whole input sequence into account, thus

```
glob('', '')                ; true
glob('a', 'a')              ; true
glob('a', '\w')             ; false, '\w' means just 'w'
glob('w', '\w')             ; true
glob('abc', 'ab')           ; false
glob('abc', 'ab*')          ; true
glob('abc', 'ab?')          ; true
glob('abc', 'ab[c]')        ; true
glob('ab*', 'ab\*')         ; true
```

3.6.4 List Functions

This section lists EL functions operating on lists where a list is a synonym for any collection of elements. In case you are missing a function to retrieve the n-th list element, then try

```
list('a','b','c')[1]      ; 'b'
```

`list(T..):list` A function taking an arbitrary number of elements to create a list object. Returns the empty list when called without arguments.

```
list()                    ; []
list('a',2)               ; ['a',2]
list(list('a',2))         ; [['a',2]]
```

`append(T..):list` This function is similar to `list` by creating a list based on given arguments. However, each argument being a list is treated in a special way by appending the list elements rather the list itself.

```
append()                ; list()
append('a',2)           ; list('a',2)
append(1,list('a',2),true) ; list(1,'a',2,true)
```

`join(string,list):string` This functions creates a string by joining elements in `list` with the first argument. The first argument is stringized. If the second argument not given, the empty string is returned. Otherwise, if the second argument is not a list, then the stringized second argument is returned. The behaviour is undefined if called without arguments.

```
join(':')                ; ''
join(':',5)              ; '5'
join(':',list('a',2))   ; 'a:2'
```

If this function is called with more then two arguments, then all arguments but the first are collected into a list object and then processed like described above.

```
join(':', 'a', 2)        ; join(':', list('a', 2))
```

3.6.5 Mathematical Functions

| Function | Description |
|---|---|
| <code>sin(double):double</code> | The mathematical sine function |
| <code>cos(double):double</code> | The mathematical cosine function |
| <code>tan(double):double</code> | The mathematical tangent function |
| <code>exp(double):double</code> | The mathematical exponential function, <code>e</code> raised to the power of the given argument |
| <code>log(double):double</code> | The mathematical logarithm function of base <code>e</code> |
| <code>pow(double,double):double</code> | Returns the value of the first argument raised to the power of the second argument. |
| <code>sqrt(double):double</code> | Returns the correctly rounded positive square root of a double value. |
| <code>abs(double):double</code> | Returns the absolute value of a double value. |
| <code>min(double, double):double</code> | Returns the smaller of two double values. |
| <code>max(double, double):double</code> | Returns the larger of two double values. |
| <code>rand():double</code> | Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0. |

3.7 Properties

It's best to introduce properties with three simple examples. It's about to ask a string about his uppercase variant, about it's length and about to create a folder with the string's value.

```
'abc'.toupper           ; 'ABC'  
'abc'.length           ; 3  
'abc')['tofile'].mkdir ; true/false
```

The specification of EL provides only a notation to query an object for a property. EL does not specify which properties must exist nor does it require that an object must have any properties. Each implementation is free to define properties according to the underlying implementation and usage domain. What is specified however is, how to query a property. Consider:

```
obj.name  
obj['name.with.dot']
```

The first variation is the standard notation to look-up property `name` on object `obj`. Here `name` can be composed of almost any character but character `.` itself. After all, `.` is the look-up operator here. This limitation can cause problems in certain domains. Therefore, an alternative look-up operator `[]` has been defined by the specification.

Another important point to keep in mind is about looking up a property on the `null` object and what is the result of asking for a property which does not exist? The perhaps surprising answer is that both case do not cause an error or worse but are perfectly legal and well defined. The result is in both cases the `null` object. Consider:

```
null.someproperty      ; null  
obj.notexisting         ; null
```

Now it's about time to tell, which properties are available.

3.7.1 Natural Properties

A *natural* property `x` exists if the underlying Java object has a public getter method with the same name as `x` and where the names are compared case-insensitively. Assume that we have on object `f` of type `java.io.File`¹⁶. The following listing shows two natural properties on object `f` and how `f` will be used by the underlying EL implementation:

```
f.name                 ; f.getName()  
f['parentfile']       ; f.getParentFile()
```

3.7.2 Primitive Type Properties

Primitive data types (integer, float, boolean and null) have no properties.

¹⁶ see <http://download.oracle.com/javase/1.5.0/docs/api/java/io/File.html>

3.7.3 List and Array Properties

Besides *natural* properties can lists and arrays be queried with an *index* returning the element at that position or `null` if the index is out of range. Consider:

```
list('a', 'b')[1]    ; 'b'
list('a', 'b')[-1]   ; null
list('a', 'b')[2]    ; null
```

3.7.4 String Properties

Table above lists properties that can be queried besides natural properties:

| Property | Type | Description |
|----------|--------|--|
| length | int | number of characters in this string |
| size | int | same as property length |
| tolower | string | return this string in lowercase characters only |
| toupper | string | return this string in uppercase characters only |
| trim | string | remove leading and trailing whitespace characters |
| tofile | file | create a file based on this string; the so created will be relative to the current build file's base folder if the string's value does not denote a absolute path. Furthermore, the empty string will create a file object denoting the project's base folder (i.e. the folder containing the build script currently executed). Notice that <code>.</code> and <code>..</code> denote absolute paths, not relative ones. |

3.7.5 File Properties

Files and folders is Ant's bread and butter. A couple of properties are defined on file objects to simplify scripting (see below). Consider:

| Property | Type | Description |
|---------------|---------|--|
| absoluteFile | file | The absolute form of this abstract pathname |
| absolutePath | string | The absolute form of this abstract pathname |
| canonicalFile | file | The canonical form of this abstract pathname |
| canonicalPath | string | The canonical form of this abstract pathname |
| delete | boolean | deletes the file or folder (true); false otherwise |
| exists | boolean | check whether file or folder exists |
| isdir | boolean | check whether a folder (directory) |
| isfile | boolean | check whether a file |
| ishidden | boolean | check whether a hidden file or folder |
| isread | boolean | check whether a file or folder is readable |
| iswrite | boolean | check whether a file or folder is writable |
| length | integer | same as size |
| list | list | array of files in folder |

| Property | Type | Description |
|---------------------|---------|---|
| <code>mkdir</code> | boolean | creates the folder (and intermediate) folders (true); false otherwise |
| <code>mtime</code> | Date | last modification date |
| <code>name</code> | string | The basename |
| <code>parent</code> | file | parent of file or folder as file object |
| <code>path</code> | string | abstract pathname into a pathname string. |
| <code>size</code> | integer | number of bytes in a (existing) file; 0 otherwise |
| <code>toabs</code> | file | file or folder as absolute file object |
| <code>tostr</code> | string | file name as string object |
| <code>touri</code> | URI | file as URI object |
| <code>tourl</code> | URL | file as URL object |

3.7.6 Matcher Properties

A *matcher object* is created by task `switch` if a regular expression matches a input value. Such a matcher object contains details of the match like the start and end position, the pattern used to match and it allows to explore details of capturing groups (also known as *marked* subexpression).

| Property | Type | Description |
|----------------------|---------|--|
| <code>start</code> | int | The position within the input where the match starts. |
| <code>s</code> | int | Same as <code>start</code> |
| <code>end</code> | int | The position within the input where the match ends (the character at <code>end</code> is the last matching character) |
| <code>e</code> | int | Same as <code>end</code> |
| <code>groups</code> | int | The number of capturing groups in the (regular) expression. |
| <code>size</code> | int | Same as <code>groups</code> |
| <code>length</code> | int | Same as <code>groups</code> |
| <code>n</code> | int | Same as <code>groups</code> |
| <code>pattern</code> | string | The regular expression that was used for this match. Notice that glob expressions are translated into regular expressions. |
| <code>p</code> | string | Same as <code>pattern</code> |
| <code>i</code> | matcher | The matcher object for <i>i</i> -th capturing group. See task <code>switch</code> for examples. |

The following example shows how to utilize matcher properties:

```
<c:switch value="foobar">
  <re expr="^(.*) (b.*)$" var="m">
    <echo>
      #{ m          } ; 'foobar'
      #{ m[0]       } ; 'foobar'
      #{ m.start    } ; 0
      #{ m.end      } ; 5
      #{ m.groups   } ; 2
      #{ m.pattern  } ; '^(.*) (b.*)$'
      #{ m[1]       } ; 'foo'
```

```
    #{ m[2]      } ; 'bar'  
    #{ m[1].end  } ; 2  
    #{ m[2].start } ; 3  
    <echo>  
  </re>  
</c:switch>
```

3.7.7 Project Properties

This section lists additional properties that can be queried on an object of type `Project`, i.e. of an Ant project. For natural properties, checkout the Javadoc of class `org.apache.tools.ant.Project` ¹⁷.

| Property | Type | Description |
|------------------------|------|--|
| <code>basedir</code> | file | The project's base directory as file object. |
| <code>targets</code> | list | A list of all target names |
| <code>tasks</code> | list | A list of all taskdef and macrodef names |
| <code>taskdefs</code> | list | A list of all taskdef names |
| <code>macrodefs</code> | list | A list of all macrodef names |

3.8 Type Conversion

Every EL expression is evaluated in the context of an *expected* type. When a evaluated expression does not match it's expected type, implicit type conversion takes place. The following sections list the rules which apply.

3.8.1 Type boolean

The following table describes the conversion of object `obj` into an boolean value:

| Type | Result |
|---------|---|
| null | false |
| string | false if obj is "", otherwise <code>Boolean.valueOf(obj)</code> |
| boolean | obj |
| file | true if the file described by obj exists |
| other | false |

¹⁷ for example <http://javadoc.haefelinger.it/org.apache.ant/1.7.1/org/apache/tools/ant/-Project.html>

4 Property Tasks

4.1 properties

This task essentially allows one to move the contents of a Ant property file ¹⁸ into a build file.

```
<c:properties>
  ; this is a comment

  ; regular properties
  p0 = hello
  p1 : ${p0}, \
      world!

  ; EL references enabled by default
  p2 = hello, #{ 'world' }!
</c:properties>
```

This task is quite similar to [let](#). The major difference is that this task is about creating properties in the spirit of Ant while [let](#) on the other side, supports in addition the creation of variables, overriding and deleting of properties.

4.1.1 Attributes

| Attribute | Type | Default | Description |
|-----------|---------|---------|---|
| cs | string | ; | The sequence starting a comment-line |
| ics | string | ; | The sequence starting a inline-comment |
| cl | boolean | true | Whether continuation lines shall be supported |

Use an empty attribute value ("") to turn comments off.

4.1.2 Elements

This task accepts a implicit text element.

4.1.3 Behaviour

The format to define properties within a this task is a subset of the format supported by Ant. Consider:

- Ant uses character # to start a comment while Flaka uses ; ¹⁹
- Ant does not support inline comments ²⁰
- Ant comment lines do not support continuation lines, this task does.

¹⁸ see <http://ant.apache.org/manual/Tasks/property.html>

¹⁹ Use attribute cs to change the comment-line character

²⁰ Use attribute ics to change the comment-line character or to turn off this feature

- This task requires that either = or : separates the key from it's value.
- Leading and trailing (!) whitespace characters are ignored. This is different from standard Ant where trailing whitespace is significant ²¹.
- EL-references #{..} are resolved regardless of whether EL has been enabled or not.

4.2 unset

The unset statement allows the removal of properties ²². Use this task with care as properties are not meant to be changed during execution of a project.

```
<c:unset>
  ; remove property p1 ..
  p1
  ; .. and p2
  p\
  2
</c:unset>
<fail if="p1" />
<fail if="p2" />
```

You may also use the more universal task [let](#) for property removal.

4.2.1 Attributes

| Attribute | Type | Default | Description |
|-----------|---------|---------|--|
| debug | boolean | false | Turn extra debug information on |
| comment | string | ; | The character that starts a comment line |

4.2.2 Elements

This element accepts implicit text.

4.2.3 Behaviour

Each non comment line defines a property name to be removed. Comment lines and empty lines are ignored. Continuation lines are accumulated before being processed. EL-references are resolved. Consider:

```
<c:unset>
  ; property 'foo bar', not 'foo' and 'bar'
  foo bar

  ; EL references are resolve, so this will remove
  ; property p15.
  p#{3*5}
</c:unset>
```

²¹ .. and responsible for unexpected and hard to track script behaviour

²² Removal of user-properties is currently not supported

5 Assignment Tasks

5.1 let

XML is not particular easy to read for humans. When assigning a couple of variables and properties, this becomes obvious. This elementary task allows to set multiple variables and properties in one go. In addition, comments and continuation lines are allowed for additional readability and comfort. For example:

```
<c:let>
  f = 'my/path'
  ; turn f into a file object
  f = file(f)
  b = f.isdir ? true : false
  ; assign a *property*
  p := 'hello world'
  ; override a property if you dare
  p ::= "HELLO \
WORLD"
</c:let>
```

In this example, variable `f` is first assigned to be a string object (`'my/path'`). The comment line, the one starting with character `;`, tells what the next line is going to do: turn `f` into a file object which can then be used for further processing. Here we create variable `b` which becomes `true` if `f` is a directory and `false` otherwise.

Character `=` is used to create or set a variable while `:=` indicates to create a property instead. If such a property already exists, it will not be changed in accordance with Ant's standard behaviour. If you however dare to override a property, then you may do so using `::=` as the last example shows.

Notice that the right side of `=`, `:=` and `::=` is evaluated as EL expression while the left side is expected to constitute a valid identifier for a variable or property.

5.1.1 Attributes

| Attribute | Type | Default | Description |
|------------------|---------|-------------------|---|
| <code>cs</code> | string | <code>;</code> | The sequence starting a comment-line |
| <code>ics</code> | string | <code>;</code> | The sequence starting a inline-comment |
| <code>cl</code> | boolean | <code>true</code> | Whether continuation lines shall be supported |

Use an empty attribute value (`""`) to turn comments off.

5.1.2 Elements

This task accepts an implicit text element.

5.1.3 Behaviour

In a first step, physical lines are processed into logical lines by taking continuation lines into account. Then, any comments and lines containing only whitespace characters are stripped from input.

Then, each line is taken apart into a key and value by using the first occurrence of `=`, `:=` or `::=` as breaking point. Leading and trailing whitespace characters are removed from key and value. Then, property and EL references are resolved. Eventually the value part is evaluated as EL expression and stored as variable or property. When stored as property, the evaluated value will be coerced into a string.

5.1.4 Then meaning of `null` and `void`

This task can also be used to *remove* variables and properties. The following table gives an overview of the meaning of `null` and `void`²³ on the right side of an assignment:

| Assignment Type | Evaluated Value | Description |
|------------------|-------------------|--|
| <code>=</code> | <code>null</code> | Remove variable if existing, else do nothing |
| <code>=</code> | <code>void</code> | Same as <code>= null</code> |
| <code>:=</code> | <code>null</code> | Nothing |
| <code>:=</code> | <code>void</code> | Same as <code>:= null</code> |
| <code>::=</code> | <code>null</code> | Remove property |
| <code>::=</code> | <code>void</code> | Same as <code>::= null</code> |

5.1.5 Examples

To start with, some basic examples.

```
<c:let>
  ; a comment line
  ; variable x will hold integer 12
  x   = 3 * 4
  ; property p will hold string '12'
  p   := 3 * 4
  ; ignored, cause property p exists already
  p   := 'twelve'
  ; override property p to hold string value '12'
  p   ::= 'twelve'
</c:let>
```

It is perfectly legal to use property or variable references. Variables and properties created within `let` can also be used. Be aware that property references are resolved *before* EL references. Consider:

```
<property name="a" value="a" />
<c:let>
  ; variable v becomes the string object 'bar' and property p
  ; becomes 'foobar'
  v   = 'b${a}r'
```

²³ `void` means that the absence of any characters

```

p := 'foo#{v}'
; property- and El-references can be used on both sides. ←
  Here we
; override property 'a' to become string value 'bar'
${a} ::= '#{v}'
; This will not work as planned cause there is no property
; with that weired name '#{v}'.
f = ${#{v}}
</c:let>

```

The following kind of meta programming will *not* work for let cause property and EL references are resolved after the logical line has been broken up in a key and value part:

```

<c:let>
; This example will NOT work as planned
assigntype := condition ? '=' : ':='
key ${assigntype} el-expr
</c:let>

```

5.2 list

A elementary task to create a variable containing a *list* of objects.

```

<c:list var="mylist">
; each line is a EL expression
3 * 5
; each line defines a list element
list('a',1,''.tofile)
</c:list>

```

5.2.1 Attributes

| Attribute | Type | Default | EL | Meaning |
|-----------|--------|--|--------------------------------------|--|
| var | string | | r | The name of the variable to be assigned. |
| el | bool | true | no | Enable evaluation as EL expression |
| cs | string | ; | The sequence starting a comment-line | ics |
| string | ; | The sequence starting a inline-comment | c1 | boolean |

Use an empty attribute value ("") to turn comments off.

5.2.2 Elements

This task may contain an implicit text element.

5.2.3 Behaviour

This task creates and assigns in any case a (possible) empty list, especially if no text element is present. The name of the variable is given by attribute `var`. This attribute may contain references to EL expressions.

If given text element is parsed on a line by line basis, honouring comments and continuation lines. Each line will be evaluated as EL expression after having resolved `#{. .}` and `#{. .}` references. An illegal EL expression will be discarded while the evaluation of lines continues. Turn on extra debug information in case of problems.

The evaluation of a valid EL expression results in an object. Each such object will be added to a list in the order imposed by the lines.

A single line can't have more than one EL expressions. Thus the following example is invalid:

```
<c:list var="mylist">  
  ;; not working  
  3 * 5 'hello, world'  
</c:list>
```

Use attribute `e1` to disable the interpretation of a line as EL expression:

```
<c:list var="mystrings" e1="false">  
  3 * 5  
  ;; assume that variable message has (string) value 'world'  
  hello, #{message}  
</c:list>
```

This creates a list variable `mystrings` containing two elements. The first element will be string `3 * 5` and the second element will be string `hello, world`. Notice that even if EL evaluation has been turned off, EL references can still be used.

6 Reporting Tasks

6.1 echo

Task `<c:echo/>` is an extension of Ant's standard `echo` task ²⁴ to generate nicely formatted output. Ant's standard `echo` task is rather fragile when it comes to reformatting XML. Here is a simple example where the output dumped is shown on the right side:

```
| <echo>foobar</echo>           ; '[echo] foobar'
```

Then, one day you reformat your XML input ²⁵ ending up in ²⁶

```
| <echo>                           ; '[echo]'  
| ... foobar                       ; '[echo] ... foobar'  
| </echo>                          ; '[echo]'
```

All of a sudden there are three more space characters in front of `foobar` and, to make matters worse, there are two additional empty lines printed. Compare this output with Flaka's `echo`

```
| <c:echo>                           ; '[c:echo] foobar'  
|   foobar                          ;  
| </c:echo>
```

which I believe is what you had in mind. Besides being invariant regarding formatting, `echo` supports other features in helping to generate pretty output.

6.1.1 Attributes

This task supports all attributes inherited from Ant's `echo` task. In addition, further supported attributes are:

| Attribute | Type | Default | Description |
|--------------------|---------|-------------------|---|
| <code>cs</code> | string | <code>;</code> | The sequence starting a comment-line |
| <code>ics</code> | string | <code>;</code> | The sequence starting a inline-comment |
| <code>cl</code> | boolean | <code>true</code> | Whether continuation lines shall be supported |
| <code>ws</code> | boolean | <code>true</code> | Whether whitespace shall be stripped off |
| <code>shift</code> | string | <code>``</code> | Prefix for each output line |

6.1.2 Elements

This task optionally accepts implicit text.

²⁴ see <http://ant.apache.org/manual/Tasks/echo.html>

²⁵ `xmlint` is a good choice

²⁶ In order to visualize space characters, character dot (`.`) is going to be used within this chapter

6.1.3 Behaviour

In an initial step, continuation lines are merged into logical lines. Then any comments are stripped from input. Afterwards, unwanted whitespace characters are removed. The process is illustrated ²⁷ using the introduction example used above:

```
<c:echo>
..foo           ; 'foo'
...bar         ; '.bar'
.baz           ; 'baz'
</c:echo>
```

The whitespace sequence before the first non-whitespace character is analyzed: if this sequence does not contain any line-ending characters, no whitespace handling will occur. Otherwise, the number of whitespace characters before the first non-whitespace character and after the last line-ending character is determined. This will be 2 in the example above. Then, the whole printable content is being left shifted by this amount. Eventually, the content is trimmed: leading whitespace up and inclusive the first line-ending character as well as the last line-ending character followed by trailing whitespace is removed. This gives then the output shown as above. Notice that `foo` and `baz` are now on the same indentation level by this left-shift. It is possible to perform a right-shift of content *after* the automatic left shift:

```
<c:echo shift="5">
..foo           ; '.....foo'
</c:echo>
```

Attribute `shift` expects a unsigned integral number followed by an optional arbitrary sequence of characters. This allows for a different *shift* character sequence as show next:

```
<c:echo shift="5> ">
..foobar       ; '> > > > >foobar'
</c:echo>
```

6.1.4 Examples

Create a single output line consisting of 20 times character %:

```
<c:echo shift="20%" /> ; '%%%%%%%%%%%%%%%%%%%%%%%%%'
```

Type a comment in echo to document what will be echoed:

```
<c:echo>
; make sure that property p is defined here
p : ${p}           ; 'p : ..'
</c:echo>
```

Follow the guideline that lines should not exceed 80 characters by using continuation lines if necessary

²⁷ Again character dot `.` is used to illustrate a whitespace character with the exception of line ending characters

```
<c:echo>
  a very long \      ; 'a very long line'
  line
</c:echo>
```

EL is enabled by default:

```
<c:echo>
  #{3 * 4}      ; '12'
</c:echo>
```

The comment character sequence can be changed or disabled:

```
<c:echo cl="#" ic="">
  #{3 * 4}      you cannot see me
  not ; any longer an inline comment!
</c:echo>
```

You can use \ as start of an escape sequence while however \\ does not escape \ from it's meaning. Consider:

```
<c:echo>
  \#{3 * 4}      ; '#{3 * 4}'
  \\#{3 * 4}     ; '\\#{3 * 4}'
  #{ '\\ ' }#{3 * 4} ; '\\12'
  #{ concat( '\\ ',3*4) } ; '\\12'
  ; comment      ;
  \; not a comment ; ';' not a comment'
  \\; not a comment ; '\\; not a comment'
</c:echo>
```

6.2 logo

A small handy task to create a kind of *framed* text like

```
.....
::          Text          ::
::      more text      ::
.....
```

This kind of *logo* is easily created and dumped on standard output stream like

```
<c:logo chr="::" width="20">
  ;; here is my text
  Text
  more text
</c:logo>
```

where *chr* defines the *wrap character* - here `::` and *width* defines the overall length of one line - here 20 characters.

6.2.1 Attributes

| Attributes | Type | Default | Description |
|--------------------|---------|-------------------|---|
| <code>chr</code> | string | : | The wrapping character .. |
| <code>width</code> | integer | 80 | Width of one line in terms of regular characters. |
| <code>cs</code> | string | ; | The sequence starting a comment-line |
| <code>ics</code> | string | ; | The sequence starting a inline-comment |
| <code>c1</code> | boolean | <code>true</code> | Whether continuation lines shall be supported |

Use an empty attribute value ("") to turn comments off.

6.2.2 Elements

This task accepts an implicit text field. This text field may contain comments (;) and leading whitespace is ignored. Thus the same rules as for task [echo](#) are applying here.

6.2.3 Behaviour

Contents of text element is read line by line. Comment lines are ignored. Leading whitespace is ignored. Leading whitespace is determined by the first non-whitespace character. See also task [echo](#) for details.

The contents of each line is centered. Leading and trailing space is filled up with *blanks_* in order to reach the given line width. If a line is longer than `width`, then all contents after `width` characters is silently skipped.

7 Conditional Tasks

With standard Ant, task `condition` is used to set a property if a condition is given. Then a macro, task or target can be conditionally executed by checking the existence or absence of that property (using standard attributes `if` or `unless`). Flaka defines a couple of control structures to handle conditionals in a simpler way.

Task `when` evaluates an `EL` expression. If the evaluation gives `true`, the sequence of tasks are executed. Nothing else happens in case of `false`.

```
<c:when test=" expr ">
  ; executed if expr evaluates to true
</c:when>
```

The logical negation of `when` is task `unless` which executes the sequence of tasks only in case the evaluation of `expr` returns `false`.

```
<c:unless test=" expr ">
  ; executed if expr evaluates to false
</c:unless>
```

The body of `when` and `unset` may contain any sequence of tasks or macros (or a combination of both).

Task `choose` tests each `when` condition in turn until an `expr` evaluates to `true`. It executes then the body of that `when` condition. Subsequent `whens` are then not further tested (nor executed). If all expressions evaluate to `false`, an optional `catch-all` clause gets executed.

```
<c:choose>
  <when test=" expr_1 ">
    ; body_1
  </when>
  ..
  <otherwise> ; optional_
    ; catch all body
  </otherwise>
</c:choose>
```

A programming task often seen is to check whether a (string) value matches a given (string) value. If so, a particular action shall be carried out. This can be done via a series of `when` statements. The nasty thing is to keep track of whether a value matched already. Flaka provides a handy task for this common scenario, the `switch` task:

```
<c:switch value=" 'some string' ">
  <matches re=" regular expression or pattern " >
    ; body_1
  </case>
  ..
  <otherwise> ; optional
    ; catch all body
  </otherwise>
</c:switch>
```

Each case is tried in turn *to match* the string value (given as `EL` expression). If a case

matches, the appropriate case body is executed. If it happens that no case matches, then the optional default body is executed. To be of greater value, a regular expression or pattern expression can be used in a case condition.

7.1 when

Task `when` represents a else-less if statement. The following example dumps the content of a file to stdout via Ant's `echo` task if the file exists.

```
<c:when test=" 'path/to/file'.tofile.isfile" >
  <c:let var="fname" property="true" value=" f " />
  <loadfile property="__z__" srcFile="{fname}" />
  <echo message="{__z__}" />
</c:when>
```

Note that the example is bit artificial cause Ant's `loadfile` task is sufficient.

7.1.1 Attributes

| Attribute | Type | Default | EL | Description |
|-------------------|--------|---------|------|---|
| <code>test</code> | string | false | expr | A EL expression that must evaluate to true in order to execute the body of this if statement. |

7.1.2 Elements

- Any tasks or macro instances.

7.2 unless

This task is the logical opposite of task `when`. It's body is only executed if the condition evaluates to false. See `when` for details. This example shows how to create a folder named `libdir` if such a folder does not already exist.

```
<c:unless test=" 'libdir'.tofile.isdir ">
  <mkdir dir="libdir" />
</c:unless>
```

7.3 choose

A task implementing a series of *ifelse* statements, i.e. a generalized *if-then-else* statement.

7.3.1 Attributes

| Attribute | Type | Default | EL | Description |
|-----------|------|---------|----|-------------|
|-----------|------|---------|----|-------------|

| | | | | |
|--------------------------|---------|-------|---|---|
| <code>when.test</code> | string | false | = | A EL condition. When true corresponding clause will be executed. |
| <code>unless.test</code> | string | true | = | A EL condition. When false corresponding clause will be executed. |
| <code>debug</code> | boolean | false | = | Turn on extra debug information. |

7.3.2 Elements

| Element | Cardinality | Description |
|------------------------|-------------|---|
| <code>when</code> | infinite | To be executed if condition evaluates to true |
| <code>unless</code> | infinite | To be executed if condition evaluates to false |
| <code>otherwise</code> | [0,1] | To be executed if no <code>when</code> or <code>unless</code> clause got executed |
| <code>default</code> | [0,1] | Synonym for <code>otherwise</code> |

7.3.3 Behaviour

Each `when` and `unless` clause's conditions are evaluated in order given until a clause gets executed. Then, further processing stops ignoring all further elements not taken into account so far. If no `when` or `unless` clause got executed, then a present `otherwise` or `default` clause gets executed.

The shortest possible `choose` statement is

```
<c:choose />
```

It's useless and does nothing, it's completely harmless.

The following example would execute all macros or tasks listed in the `otherwise` clause cause no `when` or `unless` clause got executed.

```
<c:choose>
  <otherwise>
    <!-- macros/tasks -->
  </otherwise>
</c:choose>
```

This would execute all macros and tasks listed in the `otherwise` clause since no `when` clause got executed.

```
<c:choose>
  <when test=" true == false" >
    <echo>new boolean logic detected ..</echo>
  </when>
  <unless test=" 'mydir'.tofile.isdir ">
    <echo> directory mydir exists already </echo>
  </when>
  <otherwise>
    <echo> Hello,</echo>
    <echo>World</echo>
```

```

    </otherwise>
</c:choose>

```

7.4 switch

Task switch provides text based pattern matching. An illustrative example:

```

<c:switch value="${value}">
  <re expr="(+|-)?(0|[1-9]\d*)" var="g">           (1)
    <c:echo>
      integer with absolute value: #{g[2]}
    </c:echo>
  </re>
  <glob expr "*.jar">                               (2)
    <echo>It is a -- jar!</echo>
  </glob>
  <cmp eq="foo" lt="foo">                           (3)
    <!--  ${val} is less or equals "foo" -->
  </cmp>
  <otherwise>
    <!-- none of the above clauses matched -->
  </otherwise>
</c:switch>

```

This example demonstrates, how a given textual value can be compared using (1) regular expressions, (2) using glob expression or (3) compared for equality or alphabetical order. Why providing alternatives for regular expressions? Cause the biggest drawback of regular expressions is their complexity. Compare this two variations to check whether a string value ends in .jar:

```

<re="(.*)\.jar" var="g">
  <c:echo>
    basename is #{g[1]}.
  </c:echo>
</re>

<glob="*.jar" var="g">
  <c:echo>
    filename is #{g[0]}.
  </c:echo>
</glob>

```

The latter one, the glob expression, is much easier to grasp. There, * just stands for a sequence of arbitrary characters. In most pattern recognition tasks, this is all what is needed to get going. If more power is required, then regular expressions are the tool to be applied. Eventually cmp allows for simple text comparison without any meta-characters. It also allows to check whether the input string is before or after a given test string.

7.4.1 Attributes

| Attribute | Type | Default | Description |
|-----------|------|---------|-------------|
|-----------|------|---------|-------------|

| | | | |
|------------------|---------|-------|---|
| value | string | "" | The string value that needs to be matched against. |
| var | string | - | Save details of this match as <code>matching</code> object. |
| not | boolean | false | Whether to invert the test result or not. |
| find | boolean | false | Whether to match the input value partially (true) or as a whole (false) |
| debug | boolean | false | Whether to turn on extra debug information |
| literally | boolean | false | Whether to take an invalid expression literally or not. |
| ignorecase | boolean | false | Whether to enable case-insensitive matching. |
| comments | boolean | false | Whether to allow whitespace and comments in an <i>regex</i> . |
| dotall | boolean | false | Whether literal <code>.</code> matches <i>any</i> character. |
| unixlines | boolean | false | Whether <i>only</i> <code>\n</code> is accepted as line terminator |
| multiline | boolean | false | Whether <code>^</code> and <code>\$</code> shall <i>only</i> match begin and end of input |
| <i>re.expr</i> | string | | Element matches: Specify a matching pattern as regular expression. |
| <i>glob.expr</i> | string | | Element matches: Specify a matching pattern as glob expression |
| <i>cmp.lt</i> | string | - | <code><cmp/></code> : Specify a matching pattern as glob expression |
| <i>cmp.eq</i> | string | - | <code><cmp/></code> : Specify a matching pattern as glob expression |
| <i>cmp.gt</i> | string | - | <code><cmp/></code> : Specify a matching pattern as glob expression |

Attribute **value** defines the test (string) value. EL [EL](#) references can be used. Leading and trailing whitespace is always removed. A test of a clause will always evaluate to false if this attribute is not set. It is legal however to have a switch without this attribute.

The default settings of attributes **var**, ..., **debug** have been shown in table above. A default setting can be changed by using the attribute on switch element level. Those default settings are inherited by each clause-element - `<re/>`, `<glob/>` or `<cmp/>`. Each attribute **var**,..., **debug** can be applied on clause-elements where they override their inherited setting:

```
<c:switch .. not="true">
  ..
  <cmp eq="foo" lt="foo" not="false">
    <!-- still less than "foo" or equal "foo" -->
  </cmp>
</c:switch>
```

Use attribute **var** to specify the name of an EL variable to hold match details like the number of capturing groups, the value of the first capturing group and the like. See [matcher properties](#) for a list of available properties; see also below for examples. The attribute value may contain EL references.

Attribute **not** can be used to invert the result of a test match.

```
<c:switch ..>
  <cmp eq="foo" lt="foo" not="true">
    <!-- greater than "foo" -->
  </cmp>
</c:switch>
```

When applying a regular or glob expression on a test value, then by default, the expression must describe the whole test value for a successful match. Attribute **find** can be used to change this behaviour. Set to **true**, then a successful match is given if the expression describes a part of the input string:

```
<c:switch value="foobar">
  <glob find="true" expr="foo">
    <!-- matches foobar -->
  </glob>
</c:switch>
```

Attribute **find** does not apply on clause `cmp`.

Use attribute **debug** to enable extra output of debug information on your attached standard error stream, bypassing Ant's logging mechanics.

The semantics of attribute **literally** varies depending on it's context. A build failure is thrown if an illegal regular expression pattern - like `*` - is seen. If **literally** is enabled, the regular expression string is taken literally instead, no exception is thrown while a warning message is reported. If attribute **literally** is enabled in context of a glob expression, then any meta characters are taken as regular characters. Attribute **literally** has no meaning in the context of clause `cmp`.

The remaining part of this section describes attributes **ignorecase**, ..., **multiline**. This attributes can be used to change the characteristics of the underlying regular expression engine. Modern regular expression engines also allow to their characteristics on the fly by using embedded flags. Embedded flag alternatives are also listed.

Attribute **ignorecase** can be used to enable case-insensitive matching. This attribute also applies to clause `cmp`. Case-insensitive matching is enabled on the US-ASCII charset only. Unicode-aware matching can not be enabled. The embedded flag expression equivalent flag of this attribute is `(?i)`.

Use attribute **comment** to enrich regular expressions by comments. When set, whitespace is ignored and embedded comments starting with `#` are ignored until the end of a line. Comments mode can also be enabled via the embedded flag expression `(?x)`.

Attribute **dotall** can be used to change the characteristics of meta-character `.`, the dot. By default all characters but line terminators are matched by this meta-character. When set, all characters are matched. The embedded equivalent is `(?s)`.

Use attribute **unixlines** to let the regular expression engine accept `\n` as line termination character. The embedded flag expression is `(?d)`.

Attribute **multiline** changes the characteristics of meta-character `^` and `$`. By default they match at the beginning and the end of the entire input sequence. When set, they match just after and just before a line termination character. Multiline mode can also be enabled via the embedded flag expression `(?m)`.

| Element | Cardinality | Description |
|------------------|-------------|---|
| re | arbitrary | Regular expression based test clause. |
| glob | arbitrary | Glob expression based test clause. |
| cmp | arbitrary | Clause for basic equality and ordering. |
| otherwise | arbitrary | Task container, executed if all tests failed. |
| matches | arbitrary | Legacy test clause (deprecated). |

Element **re** is a task container. Embedded tasks are conditionally carried out. The condition is satisfied if the regular expression expressed in attribute **expr** matches the given input value *switch.value*. If attribute **expr** is not used, then the condition is satisfied. Use attribute **not** to negate the condition. Be aware of the following pathological case:

```
<c:switch value=..>
  <re>
    <!-- always carried out -->
  </re>
<c:case>
```

Element **glob** is a task container similar to element **re**. The only difference is, that attribute **expr** is interpreted as glob expression instead of a regular expression as in **re**.

Element **cmp** is a task container similar to element **re** and **glob**. This element supports attributes **lt**, **eq** and **gt** in addition to inherited attributes. Embedded tasks are carried out if each comparison in each used attribute evaluates to true. If none of the attributes is used, then the condition is also satisfied.

Element **otherwise** does not express a test clause and does not support any attributes. This element is a task container. Embedded tasks are carried out if attribute *switch.value* has been set and if none of the test-clauses matched. This element can be present more than once. Elements are carried out in syntactical order.

Element **default** is an alias name for element **otherwise**.

7.4.3 Behaviour

String attribute *value* is applied against a series of *case*-clauses in their syntactical order. Elements **re**, **glob**, **cmp** and **matches** are *case*-clauses. If a clause matches, then tasks associated with the matching clause are carried out and no further clauses are will be tested. If no *case*-clause matches, then all optional available **otherwise** elements are carried out in their syntactical order.

7.4.4 Examples

Tests based on regular expressions

```
<c:switch value="foobar">
  <re expr="^foobar$" />      ; matches
  <re expr="foobar" />       ; matches
  <re expr="foo" />          ; no
  <re expr="foo" find="true" ; matches
```



```
<re expr="#{'foobar'}" ; matches
</c:switch>
```

The following example shows how to utilize matcher properties:

```
<c:switch value="foobar">
  <re expr="^(.*) (b.*)$" var="m">
    <echo>
      #{ m          } ; 'foobar'
      #{ m[0]       } ; 'foobar'
      #{ m.start    } ; 0
      #{ m.end      } ; 5
      #{ m.groups   } ; 2
      #{ m.pattern  } ; '^(.*) (b.*)$'
      #{ m[1]       } ; 'foo'
      #{ m[2]       } ; 'bar'
      #{ m[1].end   } ; 2
      #{ m[2].start } ; 3
    <echo>
  </re>
</c:switch>
```

Tests using glob expressions

```
<c:switch value="foobar">
  <glob expr="foobar" /> ; matches
  <glob expr="f*r" /> ; matches
  <glob expr="foo" /> ; no
  <glob expr="foo" find="true" /> ; matches
  <glob expr="#{'foobar'}" /> ; matches
</c:switch>
```

Matcher properties can also be used when using glob expressions. However, since glob expressions do not have enough semantics to express capturing groups, their use is limited. Consider:

```
<c:switch value="foobar">
  <glob expr="f*r" var="m">
    <echo>
      #{ m          } ; 'foobar'
      #{ m[0]       } ; 'foobar'
      #{ m.start    } ; 0
      #{ m.end      } ; 5
      #{ m.groups   } ; 0
      #{ m.pattern  } ; 'f.*b'
      #{ m[1]       } ; ''
    <echo>
  </glob>
</c:switch>
```

Finally use element `cmp` to do text based comparison in absence of any meta characters. Since element `cmp` is not based on a regular expression engine, matcher properties are not available. Consider

```
<c:switch value="foobar">
  <cmp eq="foobar" /> ; matches
```

```
<cmp eq="f*r" /> ; no  
<cmp eq="foo" find="true"/> ; no  
</c:switch>
```

Element `cmp` can be used to sort text in alphabetical order. Consider:

```
<c:switch value="b">  
  <cmp eq="b" /> ; true  
  <cmp lt="c" /> ; true  
  <cmp gt="a" /> ; true  
  <cmp lt="a" eq="b" /> ; true  
</c:switch>
```

7.4.5 Pathological

Switch has been designed to be rather tolerant. No elements are required in which case of course nothing happens.

```
<c:switch />  
<c:switch value="foobar" />
```

An `otherwise` element is carried out if not `case`-clause matches, especially in the case of absent `case`-clauses:

```
<c:switch>  
  <otherwise>  
    <echo>executed, cause nothing matched.</echo>  
  </otherwise>  
</c:switch>
```

8 Repetative Tasks

Flaka has a looping statement. Use task `for` to iterate over a *list* of items. Use `break` and `continue` to terminate the loop or to continue the loop with the next item.

```
<c:for var=" name " in=" ''.tofile.list ">
  ; sequence of task or macros
  ; used <c:continue /> to continue ; and
  ; <c:break /> to stop looping
  ; use #{name} to refer to current item (as shown below)
  <c:echo>#{name}</c:echo>
</c:for>
```

Attribute `in` will be evaluated as [EL](#) expression. In the example above, that [EL](#) expression is `''.tofile.list` which, when evaluated, creates a list of all files in the folder containing the current build script. To understand the expression, have a look at [properties](#) of a string and [properties](#) of a file.

8.1 while

A task using a while loop:

```
<c:let>
  countdown = 3
</c:let>
<c:while test=" countdown >= 0 ">
  <c:echo>
    #{countdown > 0 ? countdown : 'bang!' }..
  </c:echo>
  <c:let>
    countdown = countdown - 1
  </c:let>
</c:while>
```

8.1.1 Attributes

| Attribute | Type | Default | EL | Description |
|-------------------|--------|---------|------|--|
| <code>test</code> | string | false | expr | The condition for looping as EL expression |

8.1.2 Elements

The body of this task may contain an arbitrary number of tasks or macros.

8.1.3 Behaviour

All tasks listed as elements are executed as long as the [EL](#) expression evaluates to true.

8.2 for

A task that implements a loop statement. Iterating over a list of *objects*:

```
<c:for var="x" in=" list('a', 2, 'src'.tofile, typeof(list ↵
  ())) ">
  <c:echo>
    #{x}
  </c:echo>
</c:for>
```

8.2.1 Attributes

| Attribute | Type | Default | EL | Description |
|-----------|--------|---------|------|---|
| var | string | | #{} | The variable holding each loop item. This variable can be referenced within the body like #{var} where var is the string used in this attribute. If not used, then no iteration takes place and no warning is issued. Notice that you can use #{. .} only in EL enabled tasks. |
| in | string | | expr | The items to be iterated over as EL expression. A iteration takes place except if null is the evaluation result. Otherwise, if the evaluation result is <i>not iterable object</i> , a temporary list containing that object is created on the fly. Iteration takes then place over that list and otherwise over the iterable collection. |

8.2.2 Elements

The body of this task may contain an arbitrary number of tasks or macros.

8.2.3 Behaviour

This is the shortest possible for statement. It's legal albeit completely useless.

```
<c:for />
```

8.3 break

A task mirroring a break statement. When used within a [for](#)-loop, the loop will be terminated. If this task is used outside of a [for](#)-loop, a build exception will be thrown.

```
<c:for var="i" in=" list(1,2,3,4,5,6) ">
```

```
<c:echo>i = #{i}</echo>
  <c:when test=" i == 3 ">
    <c:break />
  </c:when>
</c:for>
```

Being executed, the following will be dumped on stdout:

```
[c:echo] i = 1
[c:echo] i = 2
[c:echo] i = 3
```

8.3.1 Attributes

| Attribute | Type | Default | EL | Description |
|-----------|--------|---------|------|---|
| test | string | - | expr | Terminate loop when EL expression evaluates to true |
| if | string | - | #{} | Terminate if property exists |
| unless | string | - | #{} | Terminate if property does not exist |

8.3.2 Behaviour

When used without any attributes, the surrounding `for` or `while` loop will terminate at once. When used with attributes, then the loop will terminate if at least one attribute evaluates to true. Otherwise, the loop will not be terminated and continues as usual.

The example given above can thus be shortened to

```
<c:for var="i" in=" list(1,2,3,4,5,6) ">
  <c:echo>i = #{i}</echo>
  <c:break test=" i == 3 " />
</c:for>
```

8.4 continue

A task mirroring a continue statement. When used within a `for`-loop, the loop will be continued with the next loop item (i.e. any statements after task continue are ignored). If this task is used outside of a `for`-loop, a build exception will be thrown.

```
<c:for var="i" in=" list(1,2,3,4,5,6) ">
  <c:when test=" i > 3 ">
    <c:continue />
  </c:when>
  <c:echo>i = #{i}</echo>
</c:for>
```

This would print:

```
[c:echo] i = 1
[c:echo] i = 2
[c:echo] i = 3
```

8.4.1 Attributes

| Attribute | Type | Default | EL | Description |
|-----------|--------|---------|------|--|
| test | string | - | expr | Continue loop when EL expression evaluates to true |
| if | string | - | #{} | Continue if property exists |
| unless | string | - | #{} | Continue if property does not exist |

8.4.2 Behaviour

When used without any attributes, the surrounding `for` or `while` be continued while following tasks or macros are ignored in the current iteration step. When used with attributes, then the loop will be continued if at least one attribute evaluates to `true`. Otherwise, the subsequent tasks or macros are executed.

The example given above can thus be shortened to

```
<c:for var="i" in=" list(1,2,3,4,5,6) ">  
  <c:continue test=" i > 3 " />  
  <c:echo>i = #{i}</echo>  
</c:for>
```

9 Exceptional Tasks

Flaka has been charged with exception handling tasks.

9.1 trycatch

A task mirroring try-catch exception handling found in modern programming languages. The following snippet demonstrates the usage of trycatch:

```
<c:trycatch property="reason">
  <try>
    <echo>one ..</echo>
    <echo>two ..</echo>
    <fail message="bang" />
    <echo>three ..</echo>
  </try>
  <catch>
    <echo>..caught : ${reason}</echo>
  </catch>
  <else>
    <echo>nothing exceptionell</echo>
  </else>
  <finally>
    <echo>..finally</echo>
  </finally>
</c:trycatch>
```

When executed, the following will be reported by Ant:

```
[echo] one ..
[echo] two ..
[echo] ..caught : bang
[echo] ..finally
```

9.1.1 Attributes

| Attribute | Type | Default | Description |
|-------------|--------|------------------|---|
| property | string | | Property that should hold the exception message |
| reference | string | trycatch.object | Reference to hold the exception object |
| catch.type | string | *.BuildException | A glob pattern to match the exception's type |
| catch.match | string | * | A glob pattern to match the exception's message |

9.1.2 Elements

| Element | Description |
|---------|---|
| try | A task container to hold tasks and macros to be given a try |

| | |
|---------|---|
| catch | A task container to be executed if an exception gets thrown |
| else | Task container executed if no exception is thrown in the <code>try</code> element |
| finally | A task container to be executed in any case |

All elements are optional. There can be an arbitrary number of catch-clauses.

9.1.3 Behaviour

If a try-clause exists, then tasks within that clause are executed. If an exception is thrown by one of them, then ..

- If a property name is given, then that property will hold the message of the exception thrown.
- If a reference name is given, then that reference will hold the exception object thrown.
- All catch-clauses are tried in syntactical order whether a clause matches.
- If there is a *matching* catch clause, then all tasks in that matching clause are executed and all further catch-clauses are ignored.

If no exception is thrown within the try-clause, then an optional else-clause will be executed.

An optional finally-clause is always executed, regardless of whether an exception gets thrown in a try-, else- or within a catch-clause. A finally-clause is especially executed in the absence of a try-clause.

An exception thrown in a try-clause for which a matching catch-clause exists, is assumed to be handled by the matching clause and therefore not propagated. An exception for which no matching clause exists, an exception thrown in an else-clause or in a matching catch-clause is propagated to an optional existing finally-clause. If no finally-clause exists, then the incoming exception will be propagated to the caller. The finally-clause propagates it's incoming exception to the caller.

An exception thrown in a catch-clause, in an else-clause or in a finally-clause overrides any previously existing exception when propagating. However, an exception thrown in a catch-, else- or finally-clause will neither update nor assign the property or reference.

A catch-clause can be decorated with a *type* or *match* argument to handle a specific exception class. Both arguments expect a glob expression. A catch-clauses matches if both, the type-argument and the match-argument do match. The type-argument is matched against exception's native type name. The match-argument is matched against the exception's message. Both default values ensure that a build exception thrown by Ant is caught while an implementation dependent exception passes. Consider:

```
<c:trycatch>
  <try>
    <fail message="houston, we have a problem" />
  </try>
  <catch match="*houston*">
    ; handle exceptions where the exception's message
    ; contains the word 'houston'
```



```
</catch>
<catch type="java.lang.*">
    ; handle Java runtime exceptions
</catch>
<catch>
    ; handle standard build exceptions
</catch>
</c:trycatch>
```

When matching against the error message, be aware that the actual error message might be slightly different from the actual message given cause usually the error message contains also information about where the exception got thrown. It is therefore wise to accept any leading and trailing space.

9.2 throw

A task to re-throw a previously thrown exception.

```
<c:trycatch>
  <try>
    <fail message="houston,.."/>
  </try>
  <catch>
    <echo level="DEBUG">*** exception seen ***</echo>
    <c:throw />
  </c:catch>
</c:trycatch>
```

9.2.1 Attributes

| Attribute | Type | Default | Description |
|-----------|--------|----------------------|---|
| reference | string | trycatch.o- bject | The name of the reference holding the previously thrown exception |
| var | string | trycatch.o- bject | An alias for attribute reference |

9.2.2 Behaviour

When this task is executed, it looks-up whether the current project contains a specific reference. By default, the name of that reference is `trycatch.object`. The name of this reference can be changed by using attribute `reference`. If that reference exists and an exception is attached with that reference, then that exception is thrown. Otherwise, a standard build exception is generated and thrown. In that case, `throw` acts like Ant's standard task `fail`.

Note that `throw` works in any context. Consider

```
<c:trycatch>
  <try>
    <fail message="houston,.. " />
  </try>
  ; ignores build exception
```

```
<catch />
</c:trycatch>
..
; very much later
..
; throws build exception 'houston,...'
<c:throw />
```

9.3 fail

This task has been derived from Ant's core task `fail` ²⁸. In addition, EL-references can be used regardless whether EL has been enabled or not. Furthermore, attribute `test` has been added. The value of `test` will be evaluated as EL expression in a boolean context. Thus it is possible to throw a build exception not only based on whether a property exists or not but also whether an EL-expression is evaluated as `true`.

```
<c:fail message="negative number #{ i } seen" test=" i lt 0 ↵
" />
```

9.3.1 Attributes

| Attribute | Type | Description |
|-------------------|--------|-----------------------------|
| <code>test</code> | string | An arbitrary EL-expression. |

²⁸ see <http://ant.apache.org/manual/CoreTasks/fail.htm>

10 Other Tasks

10.1 install-property-handler

A task to install Flaka's property handler. When installed, Ant *understands* EL references like `#{..}` in addition to standard property references `${..}`. Consider:

```
<echo>
  #{3 * 4}
</echo>
<c:install-property-handler />
<echo>
  #{3 * 4}
</echo>
```

This is the output of above's snippet:

```
[echo] [1] #{3 * 4}
[echo] [2] 12
```

10.1.1 Attributes

| Attributes | Type | Default | EL | Description |
|------------|--------|---------|------------------|--|
| type | string | elonly | <code>#{}</code> | Install handler with certain additional features enabled (see below) |

10.1.2 Behaviour

If `type` is `elonly` (exactly as written), then the new handler will only handle `#{..}` in addition. If `type` is `remove`, then unresolved property references are discarded.

10.2 rescue

Task `rescue` is essentially a container for an arbitrary number of tasks. In addition, it allows to rescue variables and properties.

```
<c:rescue>
  <vars>
    foo
  </vars>
  <properties>
    bar
  </properties>
  task_1
  ..
  task_N
</c:rescue>
```

No matter what will happen with property `var` and variable `foo` within `sequential`, this will be unnoticable outside of `rescue` cause the values (or better: state) will be restored after having executed all embedded tasks. This will of course also work in case an exception is thrown by one of the tasks.

10.2.1 Attributes

This task does not define attributes.

10.2.2 Elements

| Name | Cardinality | Description |
|-------------------------|-------------|---|
| <code>vars</code> | 0..1 | Defines a list of variable names. Attributes and behaviour is that of task list except that interpretation of lines as EL expressions are disabled. |
| <code>properties</code> | 0..1 | Defines a list of property names. Attributes and behaviour is that of task list except that interpretation of lines as EL expressions are disabled. |
| <code>task</code> | arbitrary | A (arbitrary) task or macro to be executed |

10.2.3 Behaviour

Executes all embedded tasks. Variables and properties listed in `vars` and `properties` are restored to their previous state, i.e. if not existing before the execution, they will not exist afterwards. If existed, their value will be restored.

11 Flaka Glossary

A compilation of words and their meaning in Flaka.

11.1 Whitespace

Whitespace characters are the blank character (*SP*, ` `), the horizontal tab character (*_HT_*, `\t`) and the form feed character (*FF*, `\f`). End-of-line characters are not considered whitespace in this document.

11.2 End-of-Line

A line-ending character sequence is either a carriage-return character followed by a line-feed character or the single line-feed character.

11.3 Continuation Lines

A continuation line is a sequence of characters with a non-escaped backslash character as last character before the end-of-line sequence. Tasks supporting continuation lines will accumulate the content of such a line with the *accumulated* content of the following physical line. The continuation character, the line-ending character sequence and all immediate whitespace characters are not accumulated.

```
a\  
b\\\br/>c
```

Defines two logical lines: (1) a b\\ and (2) c.

11.4 Comment Lines

A comment line is a sequence of character where the first non-whitespace character is the comment character - usually the character `;`:

```
    ; a comment line  
no ; comment line  
 \; not a comment line
```

Here, only the first line defines a comment line. The second line contains an inline-comment. When processed, comment-line get entirely removed from input up and including the end-of-line character sequence. In case a line must start with the comment-line character, then sequence `\;` can be used to do so. It must be noted, that escape character `\` has no other meaning, i.e. `\\` will not produce single character `\` but rather `\\`.

11.5 Inline Comments

A inline-comment is a sequence of characters starting with the inline-comment character and ending - while not including - the end-of-line sequence or end of input. Inline comments are discarded when processed. In case character ; shall keep it's usual meaning, then \; can be used to do so.

11.6 Property Reference

A reference to a [property](#) is written as `${. .}`. Property references are handled by the Ant property handler. If not changed, then `${x}` will be replaced by the value of property x if such a property exists. Otherwise, the reference will be left as is.

11.7 Expression Reference

A reference to an [EL](#) expression is written as `#{. .}`. [EL](#) is not part of Ant and can thus only be handled by certain tasks. References may appear in attribute values or in text elements. Not all attributes can handle EL references and neither all text elements. If a attribute or text element can handle EL references, it is specifically mentioned.

11.8 Base Folder

Relative files are usually meant to be relative to the current working directory. Not so in Ant, where a file is relative to the folder containing the build script of the current project. This folder is called the base directory or base folder. Ant defines property `basedir` to contain the (absolute) path name of this folder. When using [EL](#) expressions you can use the empty string to create the base folder as file object, like in ``.tofile``. See also [built-in-props](#) for a list of standard Ant properties.

12 Colophon

Flaka 1.2
häfelinger IT

55/54

This document got written in AsciiDoc markup and translated into DocBook by using the `asciidoc` command. From DocBook it got translated into \LaTeX using `dblatex` and from \LaTeX eventually into PDF by using $X_{\text{}}\TeX$.