The BPL Grammar And Syntax

Quick Lexical Overview

We proceed to give a quick rundown of the BPL lexical analysis. Detailed usage and grammar analysis will be explained afterwards. This is meant to be used as a quick reference point.

- **start:** defines the start of our code and must always be present.
- **end:** defines the end of any multiline structure/block including *start.*
- **basic commands:** there is a list of basic commands/tasks to be used in a BPMN graph. For the time being, they are only displayed on the graph in a basic box shape but they can be stylized accordingly in the future (especially in a fully developed GUI). We restrict the user to specific commands so that there is a common standard.
- **process:** used to define a process. A process is defined as a collection of steps, grouped together under a process name. The process named "main" will initiate the execution of the code. Processes can be nested to one another, defining them as subprocesses. Nested (or *called*) processes that have a name which starts with "_" are not defined as subprocesses but simply as grouped steps, a property that is really useful for reducing the repeatance of same steps .
- **users:** used to define the users related to a process.
- **basic divisions, departments and positions:** as with the basic commands, there is a list of possible divisions, departments and positions a user may have and they will all be defined alongside the user.
- **change users:** defines a structure that allows additions and removals of users within it (for the current process).
- **add:** is a step within the *change users* structure. It adds the corresponding user to the current list of users in the current process.
- **remove:** is a step within the *change users* structure. It removes the corresponding user from the current list of users in the current process.
- **parallel:** used to parallel steps. All steps within it will display their content parallel to one another. To keep some steps linear, we group them as a *process* (by creating a subprocess or a process with a name starting with "_").

- **try:** is a part of the conditional structure and groups the steps the *retry* command will loop back to. Its steps are executed and displayed normally when reached in the code.
- **check:** is a part of the conditional structure and defines the condition to be displayed within BPMN's diamond shape.
- **yes:** is a part of the conditional structure and groups the steps in the affirmative edge of the structure.
- **no:** is a part of the conditional structure and groups the steps in the negative edge of the structure.
- **retry:** is a optional step within *yes/no* of the conditional structure. It commands the graph to display a loopback edge to the beginning of the *try* step, making loopbacks possible in our graph.
- **abort:** is a optional step within *yes/no* of the conditional structure. It commands the graph to display an edge leading to an end/failure node.
- **continue:** used as an empty step and exists to better visualize an empty step instead of leaving it blank.
- **call:** used to reference an existing *process* by name. It basically inserts the called process into the the graph as a subprocess of the current running process.

Grammar Analysis

We will now analyze how every command/structure is used in BPL by giving basic examples with visual representations in the form of graphs and code.

The *start* structure

```
start
.
.
.
end
```

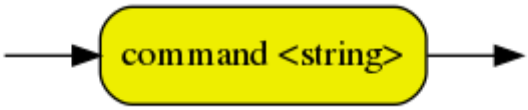
The *start structure* will and must wrap all of our code up and it fundamentally corresponds to the starting and ending nodes of a BPMN graph.

For the time being there are no other usages or initialization variables that the *start structure* accepts but we could potentially define a graph type of our choice, the language to be used and other global settings like those within it.

The *basic commands*

```
command <string>
```

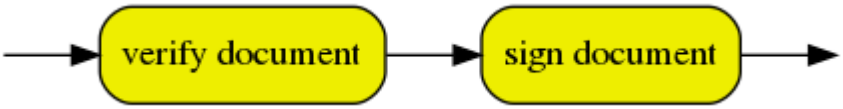
The *basic commands* is a list of tasks available in BPL. Said list is still in development, therefore we will not get into the specifics of it. The acceptable syntax is displayed above, with <*string*> meaning any phrase with alpharethmetic characters plus "_".



Let us give a more detailed example. Lets assume that our task list contains ***verify*** and ***sign*** as valid tasks and consequently as valid basic commands. We would like to initially verify a document and then sign it. Our code would look like this:

```
verify document
sign document
```

And our BPMN graph would be:



Graph A

Of course, in order for our code to run properly, we need to wrap the commands into a *process* and the process into a *start structure*. This is just a snapshot of our code for better understanding.

The *process* structure

```
process <string = name of process>
.
.
.
end
```

The *process* structure is used to categorize and group our code in different blocks and will visually create BPMN processes and subprocesses. All of our code must be a part of a process, excluding the *start* structure. Every process has a name defined as stated above. The process named “main” must always exist and it defines the starting point of our code compilation (as in many other programming languages).

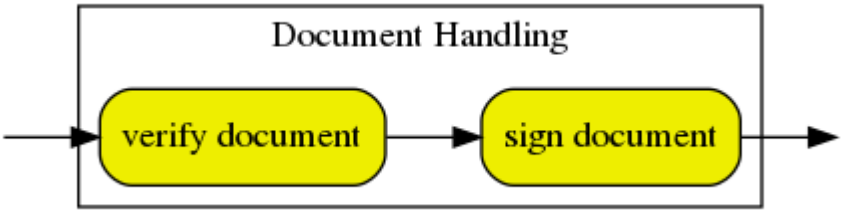
Let’s give an example based on the previous verify/sign document example. We will now give the correct code for creating the previous BPMN graph (*Graph A*):

```
start
    process main
        verify document
        sign document
    end
end
```

Processes can be nested to one another creating a subprocess. Subprocesses are important to organize tasks and beautify our code.

Let’s create a subprocess named “Document Handling” which describes the verification and signing of a document:

```
start
    process main
        process Document Handling
            verify document
            sign document
        end
    end
end
```



Graph B

As the exported graph suggests, subprocesses are visible in our BPMN. If we wanted to simply group tasks, without creating a subprocess we may have the process name begin with an “_” character:

```
...
process _Document Handling
...
```

This would result in *Graph A.* It might seem pointless for now, but it is useful in the *parallel* structure and in global processes which we are about to explain.

The examples above created a **local** subprocess to the “main” process as they are directly nested within it. Local processes are not held in memory, once they are executed they have no future usage and so, they can share names. **Global** processes are processes that are not nested to any other process, making them directly nested to the *start* structure. For example the “main” process is a global process. Global processes must have unique names and can be executed by using the *call* command:

```
call <string = global process name>
```

The code bellow will create *Graph B*, but now “Document Handling” is a global process and it can be reused whenever we want to:

```
start
    process Document Handling
        verify document
        sign document
    end

    process main
        call Document Handling
    end
end
```

Now, the usage of “_” should be obvious, as we might not want to create a subprocess every time we call a global process but simply we might just want to reuse tasks. Reminding that if we rename “Document Handling” to “_Document Handling” we will get *Graph A* and not *Graph B.*

The *users* structure

```
process ...
    users
        (division, department, position) <string=name>
        ...
    end
end
```

The users structure creates a block, in which we can define any number of users for the current *process.* It is an optional structure and if defined, it must exist directly under the *process* definition (else, we can use *change users* structure). A user is defined in a top down fashion, starting with their division then department, position and lastly their name.

Example:

```
    process main
         users
              (Marketing Division, Sales Department, Manager) John Smith
              (Marketing Division, Advertising Department, Manager) Smith Johnson
         end
    end
```

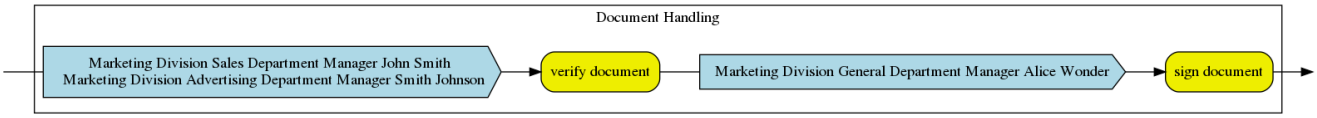
The *change users* structures

```
    change users
         add <user>
         remove <user>
    end
```

Similar to the *users* structure, the *change users* structure defines users. It can add and remove users within a process, defining a change of the personnel responsible for the upcoming tasks. Let’s modify *Graph B* so that John Smith and Smith Johnson verify the document but Alice Wonder signs it.

```
start
    process Document Handling
         users
              (Marketing Division, Sales Department, Manager) John Smith
              (Marketing Division, Advertising Department, Manager) Smith Johnson
         end
         verify document
         change users
              remove (Marketing Division, Sales Department, Manager) John Smith
              remove (Marketing Division, Advertising Department, Manager) Smith Johnson
              add (Marketing Division, General Department, Manager) Alice Wonder
         end
         sign document
    end

    process main
         call Document Handling
    end
end
```

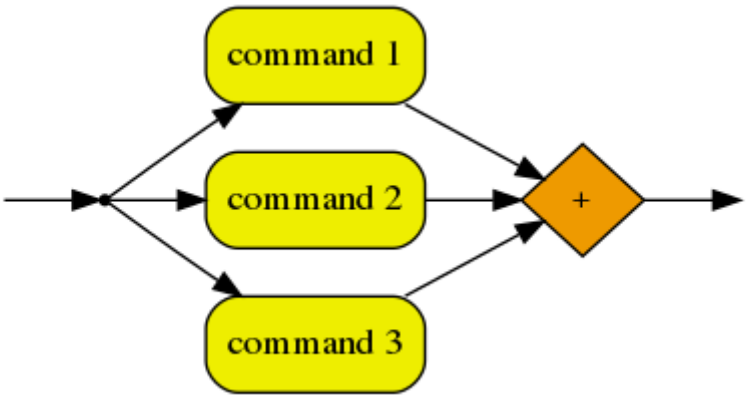


For the time being, we have to use the full user description for removal as homonymous personnel might exist.

The *parallel* structure

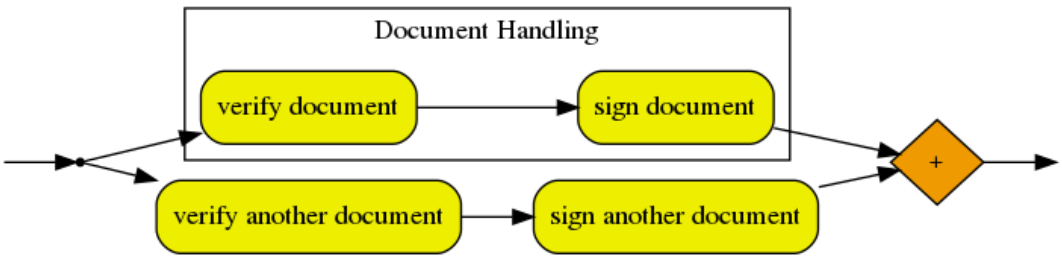
```
parallel
.
.
.
end
```

The parallel structure, as the name suggests, is responsible for paralleling steps/tasks. Any task inside its block will be parallel to the rest. For three tasks, the following graph indicates the output:



But, in most cases, we want to parallel multiple series of linear tasks to one another. That is possible by grouping them to processes. In this way, we “force” the user to use processes while paralleling multiple tasks which by extent “forces” them to keep the code organized. This enforcement is a positive property, especially for casual users as we want the code to be both easily writable and readable from them. Lets give an example by paralleling two document handling processes:

```
process main
    parallel
        process Document Handling
            verify document
            sign document
        end
        process _Document Handling
            verify another document
            sign another document
        end
    end
end
```



Now, based on the example above, we can see the usage of "_" in the *process* names and point another reason it is a part of the BPL grammar. Many a time, we will need to parallel processes but we won't necessarily need to create new visible subprocesses in the graph.

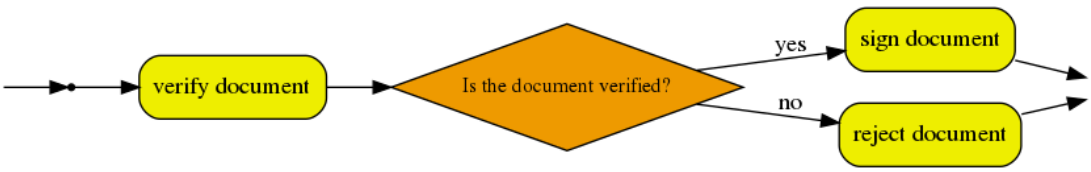
The *conditional* structure

```
try
.
.
.
check <string>
    yes
    .
    .
    .
    end
    no
    .
    .
    .
    end
end
```

The conditional structure is responsible for the conditional statements in a BPMN graph and it is arguably the most complex one but in reality it is not as complex as it looks.

Let us forget the *try* block for now and focus on the rest. The *check* command is accompanied by a string containing the condition and/or question. Then, we have two blocks within it, the *yes* and *no* blocks. Those are the two possible routes our graph may have and contains the tasks for the condition being true or false accordingly. Lets make an example based on the document handling. We now want to check if the verification was completed successfully and if it did we sign the document, else we reject it:

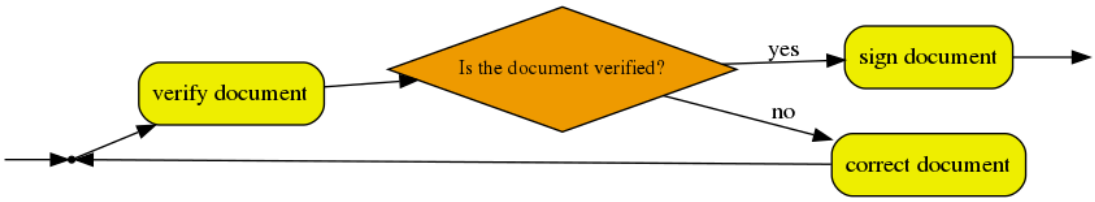
```
try
    verify document
check Is the document verified?
    yes
        sign document
    end
    no
        reject document
    end
end
```



We are including the *"verify document"* int the *try block*. The reason is not clear for now and on this example we could have left the *try block* empty (or use *continue*) while writing the task of verification right above the *conditional* structure. For the time being, we can say that any tasks that are closely related to the condition we want to check should be in the *try* block to keep the code organized.

Now, lets introduce the *retry* command of the *conditional* structure and fully understand the *try* block. So, based on the previous example, we now don't want to reject the document but correct it in case the verification was invalid. After we correct it though, we need to verify the corrected version and check the verification again. That's when *retry* comes in:

```
try
    verify document
check Is the document verified?
    yes
        sign document
    end
    no
        correct document
        retry
    end
end
```



The *retry* command creates a loopback and connects the current node to the tasks in the *try* block. Basically, the *try* block exist so that loopback is possible without goto and similar archaic programming commands.

There is also the *abort* command that signifies termination of the BPMN if reached:

```
try
    verify document
check Is the document verified?
    yes
        sign document
    end
    no
        abort
    end
end
```