The BST Tutorial

> *This document will guide you through the creation of a BST formatted story. You are welcomed to suggest additions and modifications here.*
>
> *If you wish to edit this document, **make sure you have the Ubuntu and Ubuntu Mono fonts installed**, as well as all their derivatives. They are NOT bundled inside this document due to LibreOffice bugs.*
> ***LibreOffice 6.0 is highly recommended for edition.***

Table of Contents

Read me!

> *You are reading the **version 8** of this document, **revision 2**. Each version revision aims to provide a better reading and learning experience while keeping up to date with the latest features.*
>
> *Latest changes:*
> ***Version 8** adds documentation for all the new features of BST 2.0, such as aliasing, XBF, and more.*
> ***Revision 3** adds additional information for version 2.0 beta 2, and fixes a whole lot of typos*
>
> *This document was made for the version **2.0 beta 2** of OpenBST. Features of the language may change between versions.*

Thank you for downloading this document. It will guide you through creating your very own story or text adventure game.

BST is the format used by OpenBST. In other words, OpenBST reads BST files.

OpenBST is available for download at the following location:
https://utybo.github.io/BST/#download

If any more recent version of this document is available, it is at the same location. Do note that the Tutorial and OpenBST do not share the same version code.

NOTE: A full action/checker reference table is available for download!

What is OpenBST? What is BST?

OpenBST is a program that allows you to read and create interactive fiction, also known as branching stories. Interactive fiction is fiction where you can interact with the story yourself.

BST is a language used to make interactive fiction, that was created specially for OpenBST. It was made to be an easy way of creating stories:

```
1:This is an amazing story that starts here.

Cool, huh?

:Wow, it's amazing!|2
:When will OpenBST be bug free?|3

2:Thanks!

3:That's a great question! I have no idea.
```

Recommended software: Atom

We recommend Atom for editing BST files. It is the only editor with official syntax highlighting support, and is overall nice to work with when you create BST stuff. Download and install Atom from https://atom.io and follow the steps below to add the BST extension.

The syntax extension for Atom is not available from the repositories, which means you have to manually download and register it.

Download it from https://github.com/utybo/language-bst (Clone or download > Download ZIP), unzip it and, inside the `language-bst-master` directory you just extracted, run the command

```
apm link
```

This will register the extension in Atom. To see reload packages, press Ctrl+Shift+F5, or simply close and reopen Atom.

To update the extension, just download the ZIP again using the same technique, and run the link command again in the new version's directory.

Using the built-in OpenBST editor

> ***The built-in editor is in beta.*** *Make sure you often save your files.*

> ***THE EDITOR DOES NOT SUPPORT NODE OR OPTION TAGS!*** *It is recommended that you only open in the editor files that you have created with the editor.*
>
> ***THE EDITOR DOES NOT KEEP FORMATTING OF LOGICAL NODES, AND DESTROYS ALL COMMENTS WHEN IMPORTING!***

OpenBST has a built-in BST editor that provides relatively easy creation of files. You will, however, need to know how to use options the regular way, as there is currently no way to add options with an easier GUI.

The editor is more of an experiment than a finished product. It is highly recommended that you only use the editor to get a feel of what OpenBST can do at a very basic level. We recommend using Atom for editing OpenBST files, as it can provide beautiful syntax highlighting for BST files simply by installing the extension. See the section above for more information.

Part 1: Static stories

So, you're here to learn how to create some story or text game stuff, right? Let's get started!

BST (Branching Story Tree) is a format that allows you to easily create branching stories, all based on text. It is quite powerful, as we'll see later on, as it has good scripting capacities.

For now, let's focus on creating our first, simple story. It will not have any kind of scripting, just to keep things simple.

A Simple Story

> *This will cover basic story creation, Text Nodes and basic usage of the OpenBST player. After reading it, you will be able to create branching stories without any problem.*

The syntax used by the BST is very simple. Create a text file, any editor will do, as long as your file is saved with the UTF-8 encoding.

The extension does not matter, but you cannot use LibreOffice or Micro$oft Office: you *have* to use a simple text editor. For instance, you can use Notepad or Notepad++ on Windows, or Gedit, Pluma or Mousepad on Linux, but there are many more available. If you use Notepad on Windows, make sure that, when saving your file, you select UTF-8 in the Encoding settings.

Nodes

In BST, every part of your story is called a "node". Let's take an example:

Jack is at the entrance of his house. He can either go upstairs, go to his kitchen, go to his bedroom or go to the toilets.

Here, every part of the house will be represented by a node: the entrance is a node, the kitchen is a node, the bedroom is a node… From the entrance node, you will be able to go to other nodes (other rooms) using options. Options are pretty self explanatory: the reader will be able to go to the next node using any of the options provided by the current node.

Let's create our first nodes. There are multiple types of nodes, but here we will just create a simple Text Node. We will see the other nodes later on, as they are quite advanced.

```
1:This is a first node
```

As you can see, a Text Node is a very simple node that only contains text and options. Save the file, open OpenBST, select "Open File" and choose the file we just saved. You will see

the text "This is a first node" shown, along with other information at the bottom. We will see about them later on. Don't worry about the buttons at the top, we will also see about them in another chapter.

What happened?

In BST, every node, regardless of its type, has a unique ID; the ID of the node we created here is 1.

We declared our node by using this simple syntax:

```
<id>:Text of the node
```

Every node is declared like this. Your BST story will always start on the node that has an ID of 1 by default.

But what if we want to create a node that has multiple lines of text? No problem! Just add your text like you normally would with any other editor. Here is an example:

```
1:This is

a

multi-line


node ;)
```

Now that we have created our first node, let's create another one. We just repeat the same pattern on a new line.

```
1:This is my first node
Very impressive right?

... okay maybe not.

2:This is my second node

Still not impressive I guess
```

As you can see, the same syntax is used to declare the node, only the content and the node's ID have changed. Again, every node must have a unique ID.

If you play the file again, you will notice that nothing has changed. Where is our second node?

The program cannot guess where the reader can go next: you have to tell it where we can go next using *options*.

Options

Options tell the program what are the nodes that the user can go to from the current node.

Let's take our previous example and add an option so that we can go from node 1 to node 2.

```
1:This is my first node
Very impressive right?

... okay maybe not.
:Let's make it more impressive|2

2:This is my second node

Still not impressive I guess
```

You can see that we added a line at the end of the first node. Launching the file with OpenBST will show you that you can have the option to go to node 2, and, if you click on it, the second node will be displayed.

You can of course have multiple options. Note that options cannot have multiple lines of text.

The syntax we just saw is:

```
:Option text that will be shown on the button|<id of the node this
will lead to>
```

Let's create a slightly more complex example:

```
1:First node of awesomeness
:Lame|2
:Cool|3

2:You are quite a rude person :(

3:Thank you!
:Actually no, it's lame|2
```

You can see that the node 2 has no options: such a node is called a final node. It represents one of the endings of your story, as the user cannot go anywhere else from there.

> *As you may have noticed, OpenBST gives you a small set of options for every final node, even if you did not request any. Said option are : 1) An unselectable "The End." button 2) An unselectable button that indicates the ID of the final node the user just landed on*

> *3) An option to exit OpenBST*
> *4) An option to restart the entire script.*
>
> *These options cannot be changed.*

Congratulations, you just created a branching story! That's all you need to know for creating your BST file!

Before we leave, do know that you can put comments inside BST files. Comments are lines that will be completely ignored by OpenBST, as if they never existed. Here is an example

```
1:An example
# that does not show this line
but shows this one
```

You'll see that only "An example but shows this one" (with a line break between example and but) will be displayed on screen, but that "that does not show this line" is not shown anywhere. Comments are lines that start with a `#`. They can be literally ANYWHERE inside the file; they just need to be on a distinct line. Do not worry if your BST file contains other occurrences of the `#` symbol, as comments only count if the *line* starts with it.

Now that we are all set and you know how to create basic nodes, let's have a look at how you can have more customized nodes and stories using tags.

Markup, colors, title, author, all that good stuff

> *This will cover tagging in all its forms*

Tags are simple ways to indicate to OpenBST that there is something special about the node, the story or an option.

There are two types of tags in BST:

- Top-level tags. Those have an impact on the entire story. They use the following syntax and are declared before any node has started: `tagName=Tag value`
- Secondary level tags. They only have an impact on nodes and options. For text nodes, they are declared after the content and before the options. For options, they are declared right after the option on a new line, before the next option.

  They use the following syntax: `::tagName=Tag value`

The following tags can be used and are recognized by OpenBST:

- `markup` : This tag can be used as a top level tag to specify the markup language in the entire document, or in a node to either override the top-level tag's value or specify a markup language that only is used in a specific node. The following values are recognized (case is ignored, so `markdown` and `MaRkDoWn` have the same effect)

    - `none` : Indicates that no markup language is used
    - `html` : The text is formatted in HTML. OpenBST supports full HTML5 through a WebKit engine since version 2.0 – for more information please read the chapter on the HTB module.
    - `markdown` : The text is formatted using Markdown
    - `md` : Same as "markdown"
- `author` : The name of the author(s). Top level tag only. Used for some parts in OpenBST, and it is in general better to know who made the story.
- `title` : The title of the story. Top level tag only. Used in OpenBST to provide better information.
- `color` : The color you want to use in a text node or an option. Secondary level tag, used in Nodes and Options. The following colors are available : `BLACK`, `BLUE`, `CYAN`, `DARK_GRAY`, `GRAY`, `GREEN`, `LIGHT_GRAY`, `MAGENTA`, `ORANGE`, `PINK`, `RED`, `WHITE` and `YELLOW`. You can also use any hex-formatted color (`#FFFFFF` for example. This will result in the text being white.)
- `font` : This tag allows you to choose between two fonts for your file : the Ubuntu font (with which this tutorial is written), with `font=ubuntu`, or Libre Baskerville, which is the default, with `font=libre_baskerville`
- `nsfw` : This should be set to “true” if your story is considered “not safe for work”. You know, mature stuff. Top-level only.
- `initialnode` : Stories start at node 1 by default ; if, for some reason, you not to start at another node, you can specify which using this tag. For example, if we want to start at, say, node 15, we would add this tag : `initialnode=15`
- `supertools` : In OpenBST, you have a tool bar that gives useful tools, but those can be a little bit overpowered and may be used for cheating. You can use this tag to restrict which tools can be accessed. (For more details on these buttons, see the chapter on OpenBST)
    - `all` : Show all the tools. This is the default value.
    - `hidecheat` : Hide the most powerful and cheating prone tools : Node ID indicator, Jump To Node button, and Variable Watcher
    - `savestate` : Hide everything except save states
    - `savestatenoio` : Show save states only, but exclude the Import and Export functions.
    - `none` : Hide everything.

    Do note that some buttons (mute, show background...) cannot be hidden. This tag may have different effects on other players.

Here is an example that uses most of these tags.

```
title=An Awesome Example
author=BST Team
markup=markdown

1:An awesome **pink** example!
::color=PINK
:I prefer red!|2
::color=RED
:I prefer cyan!|3
::color=CYAN

2:There you go, you red-lover
::color=RED

3: There you go, you cyan-lover
::color=CYAN
```

> *Logical Nodes and Virtual Nodes can also be tagged with secondary level tags.*

And there you have it! Next, we will see how to create scripts, dynamic options, Logical Nodes and Virtual Nodes!

Part 2: Dynamic stories (Scripting)

> *This will cover all there is to know about dynamic stories, except Virtual and Logical nodes*

Okay! Now that we're done talking about the basic, boring stories, let's begin working on what makes BST awesome: Scripting!

Since scripting is such a vast subject, we'll go step by step. First, let's create a very simple story that uses some basic scripting.

Basic scripting: options scripts, actions and variables

> *This will cover Script in Options, variables and variable inclusion in Text and Virtual Nodes.*

I have to admit something. I lied to you. The syntax for options is a little bit more complicated than what I showed. Do know that the one you saw earlier is perfectly fine, but there is another syntax that is slightly more advanced.

```
1:Example node
:Option 1|2|{incr:var}
```

```
2:Another example node
```

Try to run it! You'll see that you have an option that does not look different from what we saw in the first part of this tutorial. It's normal: scripts are silent. They only speak when an error occurs. The syntax is the same as the previous one, with the addition of `|{incr:var}`.

Here, the | is used to separate the ID of the next node from the rest of the option, which is a script.

A script is a succession of instructions. An instruction has an action name and a description. Here, the instruction is `{incr:var}`, with the action name being `incr` and the description being `var`. The `incr` action is a function that increments a variable.

A variable is a pretty much like a box. The box has a name on it, and we can put *something* inside it. Most of the instructions operate on a variable: here the `incr` action increments the variable `var` by one, where "var" is the name of the box, and the content of the box gets incremented by one. In BST, variables can contain either integers (both negative and positive) or strings (like `Abcdefg123 456 abba`).

You can put multiple instructions in an option. For example, we could have put

```
:Option 1|2|{incr:a}{incr:b}{incr:c}
```

...which would have incremented, in order, the variables "a", "b" and "c". Sometimes, you will have to be careful about the order of your instructions, so just make sure the order is logical.

If you only have one function, you can omit the curly brackets. This would have worked just like our previous example:

```
1:Example node
:Option 1|2|incr:var

2:Another example node
```

So that's it: when we choose the option "Option 1", the variable "var" gets incremented by 1. But can we actually see the content of the variable var in our story?

We can! It's not compulsory, but sometimes, you'll need to show the variable to the user. This can be useful when the variable is something like a health point counter, where the user would be interested in seeing how many points he has left.

Let's build an example that has a single node which shows the content of the variable "a" and an option that increments said variable.

```
1:a = ${a}
:a+1|1|{incr:a}
```

Also, I have not told you, but yes, you can add an option with the next node being the same node: here the option 1 will continue on the node 1. It's perfectly possible, but is useless when doing basic stories, as it only makes an infinite loop. Here, it will be very useful, since we get to see the result of incrementing the variable a.

As you may have seen, the syntax for including a variable's value inside a Text Node (or a Virtual Node, we will see about them later) is `${var}`, where "var" is the name of your variable.

Go on and run this story in OpenBST to see the result live! "a" starts at 0, then gets incremented by 1 every single time you click on the button!

And that's it! You just created your first dynamic story! Before we go deeper, here is a list of useful functions.

> *Full list available in the BST Reference. See the Read Me section for more information.*

- `incr:<variable>` : Increments the variable named `<variable>` by 1. Only works with numbers
- `decr:<variable>` : Decrements the variable named `<variable>` by 1. Only works with numbers.
- `set:<variable>,<value>` : Sets the variable `<variable>` to `<value>`. The value can be an integer or a string.
- `clone:<variable to clone>,<variable to set>` : Clones the value of `<variable to clone>` to `<variable to set>`. This is useful for duplicating variables.
- `<operation>:<putIn>,<a>,<b>` : Sets the variable `<putIn>` to the result of `<operation>` with `<a>` and `<b>`. The values can be normal numbers or variable names.
  `<operation>` can be :
  - `add` for an addition (a+b)
  - `sub` for a subtraction (a-b)
  - `mul` for a multiplication (a*b)
  - `div` for an division (a/b)
- `exit:` : Makes the program crash instantly. No other information is given, the program just quits. This may or may not work depending on your player.
- `input:<variable>,Text of the dialog prompt` : Shows a dialog box with a message (here, "Text of the dialog prompt") that asks for an input. Only works with strings.
- `rand:<variable>,<maximum>` OR `rand:<variable>,<minimum>,<maximum>` : Randomly pick a number between `<minimum>` and `<maximum>` (if `<minimum>` is not specified, it is assumed to be 0) and set the variable `<variable>` to this number.

> *A full list of all the available actions is available in the Reference*

So, now you know how to create *some* scripting stuff. But all that is quite limited: what if I want an option to go to a different node is a variable is, say, equal to one?

Next node definers (NND), conditional options

A very powerful tool for this is a Next Node Definer. By the way, you already know how to use one!

Okay, once again I lied to you. Let's take a very basic option.

```
:I am an option|2
```

The 2 is actually something called a Next Node Definer, also called NND because that's much shorter. Here, we use a Static NND, which never changes. There is another type of NND called "if-NND". An if-NND is a next node definer that determines what the next node is depending on whether a statement is true or not.

In short, an if-NND decides what the next node will be depending on if a condition is true or not.

But how can we define said conditions? We only saw functions that *do* things, not functions that *check* if something is true!

For this, we use a different type of functions called Checkers. Let's see how they work in an if-NND:

```
1:This is an example.

a = ${a}
:a+1|1|{incr:a}
:Go|2,3[equ:a,3]

2:a was equal to 3

3:a was not equal to 3
```

This is fairly simple. On our first node, we have a text that indicates the value of the variable "a". Then, we have an option to increment "a", and a Go option. The Go option will move us to node 2 is "a" is equal to 3, or node 3 if "a" is not equal to 3.

The syntax of an if-NND is:

```
<nodeIfTrue>,<nodeIfFalse>[checker]
```

Checkers follow the same syntax as the functions we saw previously, except that the symbols `{` and `}` are replaced by `[` and `]`.

What if we want to go to a node with the id of a variable? We can simply type the variables name. Here is an example:

```
1:This is an examples
:Increment a|1|{incr:a}
:Go to a|a
```

We simply have to put the variable's name instead of the node ID. This is called a variable NND. The variable name can of course have a variable whose name is longer than just one character.

Be careful about the variable's value! If it points to a node ID that does not exist, you will have an error! In our example, if we increase `a` more or less than once, it would point to a node that does not exist!

You can use variables wherever you can use regular node IDs, including on the basic NND we have seen before and on any NND we will see later on.

What if we want our option to be available only when we decide it can?

We use conditional options! Conditional options are options that have a checker. If the checker returns true, the option will be available, if it returns false, the option won't be shown.

Let's take our previous example and say that we cannot get over 3. For this, we just have to make our first option, the one which increments "a", be available only if `a` is less than 3.

```
1:This is an example.

a = ${a}
:a+1|1|[less:a,3]{incr:a}
:Go|2,3[equ:a,3]

2:a was equal to 3

3:a was not equal to 3
```

You'll see that you cannot go over 3 using this example.

The option has changed a little bit:

```
:a+1|1|[less:a,3]{incr:a}
```

You can see that we added a checker right before the `incr` function. The position of the checker does not matter, but there can only be *one* checker.

If the option only has ONE checker and nothing else (i.e. with no actions), you can omit the square brackets. For example, this is a perfectly valid option:

```
:I'm an option!|2|equ:a,5
```

You can of course combine an if-NND and the use of checkers or script functions inside an option.

Here is a list of useful checkers:

- `equ:<var1>,<var2>` : Checks equality between `<var1>` and `<var2>`. While `<var1>` HAS to be a variable, `<var2>` can either be a variable or a value.
- `not:<var1>,<var2>` : Does the opposite of `equ`, checking for inequality between `<var1>` and `<var2>`
- `greater:<var1>,<var2>` : Checks if `<var1>` is greater than `<var2>`. While `<var1>` HAS to be a variable, `<var2>` can either be a variable or a value.
- `greaterequ:<var1>,<var2>` : Checks if `<var1>` is greater than or equal to `<var2>`. While `<var1>` HAS to be a variable, `<var2>` can either be a variable or a value.
- `less:<var1>,<var2>` : Checks if `<var1>` is less than `<var2>`. While `<var1>` HAS to be a variable, `<var2>` can either be a variable or a value.
- `lessequ:<var1>,<var2>` : Checks if `<var1>` is less than or equal to `<var2>`. While `<var1>` HAS to be a variable, `<var2>` can either be a variable or a value.

> *For more information, a full list of checkers is available in the Reference*

Okay. Now we know how to do a lot of things, and we can make our stories dynamic. Next, we'll see what are the two other types of nodes: Virtual Nodes and Logical Nodes.

Part 3: Deeper inside scripting

> *This will cover Virtual and Logical nodes*

So, we are now able to do some stuff on one option; but let's be honest, while that is pretty useful for only one or two increments, it can get really messy when we get a lot of functions.

Enter the world of Logical nodes!

Logical Nodes

Logical nodes are scripts: pretty much what you used inside the options for scripting except way more powerful.

Simple Logical Nodes & NNDs in Logical Nodes

Let's have a look at a very simple logical node.

```
1:&
```

```
add:a,10
incr:a
incr:a
:2

2:Value of a is ${a}
:Do it again!|1
```

As you can see, a logical node is nothing more than a script in a dedicated node rather than all squashed in an option.

The first node of the example is a logical node, as indicated by the special beginning. Every node type have this beginning, with the node's id, a colon (`:`) and a special character that defines the type of the node. Here is a list of all the special characters:

- `&` : Logical node
- `>` : Virtual node (more details about them later)
- Anything else: Text node

Most types support having text right after that character: this is the case for Text Nodes as they do not have any special characters, and Virtual Nodes. Again, more details about them later.

Every line of a Logical Node is a function. Empty lines are ignored. They are executed in the order they are written: here, we first set the value of `a` to 10 and increment it twice.

A line that starts with a colon is a special line in Logical Nodes. Here, the user cannot choose where to go, it's up to the logical node to decide which node is next. Lines that start with a colon are Next Node Definers inside these Logical Nodes, meaning that when this line is reached, it will be used to see where to go next. You can use both static NNDs (as seen in the example) or if-NNDs, or any other NND. We also say that the next node's ID is "returned", that the Logical Node "returns" the next node's ID.

Let's take another example.

```
1:Press an option to start
:a = 1|10|{set:a,1}
:a = 2|10|{set:a,2}
:a = 3|10|{set:a,3}

2:a was 1

3:a was 2

4:a was 3

10:&
:2,-1[equ:a,1]
:3,-1[equ:a,2]
:4,1[equ:a,3]
```

```
# We return to 1 if for some reason a isn't 1, 2 or 3 (this shouldn't
happen normally, but you should always make sure your logical node
ends up somewhere)
```

In this example, we use a special feature of if-NNDs that is only available in Logical Nodes. By specifying -1 instead of an actual node, the program will just continue going through the Logical Node.

Here, once we reach node 10, we first check if a = 1; if yes, we go to node 2, if no, we continue. The same thing happens with the next NND; the last one is just here to make sure we have a node to go to.

If the node does not specify a node, OpenBST will show an error.

If statements

It is possible to have "if statements" inside a Logical Node. We do so by using the following syntax:

```
[checker]?{ifyes}{ifyes}...:{ifno}{ifno}...
```

We just use a checker as a condition, some instructions that will be executed if the checker says "true", a colon (`:`) and some instructions that will be executed if the checker says "false". Here is an example that tells if a number is positive, negative or equal to 0:

```
1:Press START to start
:START|2

2:&
set:a,10
[less:a,0]?{set:b,1}:{set:b,2}
[equ:a,0]?{set:b,3}
:3,-1[equ:b,1]
:4,5[equ:b,2]

3:The number ${a} is negative
4:The number ${a} is positive
5:The number ${a} is 0
```

If the number is strictly negative, we set b to 1. If it is not strictly negative, which means that it is either 0 or positive, we set b to 2. Now, if the number is 0, we set b to 3. After that, we go to the node 3 if b is 1, or continue if it has another value. At this point, its value can be either 2 or 3, so we just need one if-NND: if b is 2, we go to node 4; if it's not 2, it means that it has to be 3, we can jump to the node 5 right away.

Play around by changing the line `set:a,10` and you will see that the node after you press START will be correct.

As you can see, it is not necessary to have instructions executed if the checker returns "false".

```
[equ:a,0]?{set:b,3}
```

The colon can be omitted.

And that's all you need to know about Logical Nodes!

As you may have noticed, there are quite a lot of repetitions with the nodes 3, 4 and 5 in the example. There is a type of node specifically made to contain text that will be included at multiple places: Virtual nodes.

Virtual nodes

You can see virtual nodes as small pieces of paper that you can glue anywhere inside Text Nodes. Here is an example of a virtual node that shows the text "Hello" on top of every node:

```
1:${>10}
My name is Wilbert
:Nope, it's Robert|2

2:${>10}
My name is Robert
:Nope, it's Albert|3

3:${>10}
My name is Albert
:Nope, it's Wilbert|1

10:>Hello!
```

For the rest, virtual nodes are exactly like Text Nodes, except that they do not have options, and that the program will give an error if you try to go to one of them: they are not made for that. They are built to be included in other nodes. Virtual nodes can also contain other virtual nodes or variables; this is perfectly possible:

```
1:Virtual${>2}!

2:>node${>3}

3:>ception
```

As you can see, to include the text of a virtual node, you just need to do the same as if you wanted to include a variable, but use the node's ID instead of the variable's name and put a `>` between the `{` and the ID.

The above example will therefor result in the text `Virtualnodeception!`

Virtual nodes + Logical nodes = <3

You can also include a logical node inside a node; you just need to use `${&id}` instead of `${>id}`. The way it works is that the next node that the logical nodes returns will be used as a virtual node. For example, this works:

```
1:Text : ${&2}

2:&
:3

3:>Hello World
```

This does the same thing as if you put `${>3}` instead of `${&2}` as the Logical Node will always return the node 3.

Now that we have all the power of Logical Nodes and Virtual nodes, we can simplify this example:

```
1:Press START to start
:START|2

2:&
set:a,10
[less:a,0]?{set:b,1}:{set:b,2}
[equ:a,0]?{set:b,3}:
:3,-1[equ:b,1]
:4,5[equ:b,2]

3:The number ${a} is negative
4:The number ${a} is positive
5:The number ${a} is 0
```

… into something that does not contain any repetition:

```
1:Press START to start
:START|10

10:The number ${a} is ${&2}

2:&
set:a,10
[less:a,0]?{set:b,1}:{set:b,2}
[equ:a,0]?{set:b,3}:
:3,-1[equ:b,1]
:4,5[equ:b,2]

3:>negative
4:>positive
5:>0
```

And that's it! You now know everything about Logical and Virtual nodes! Here are some more examples if you need some:

```
1:${&2}, World!
:Hello|1|{set:a,0}
:Goodbye|1|{set:a,1}
:Farewell|1|{set:a,2}
:Screw you|1|{set:a,3}

2:&
:3,-1[equ:a,0]
:4,-1[equ:a,1]
:5,6[equ:a,2]

3:>Hello
4:>Goodbye
5:>Farewell
6:>Screw you
```

In this example, you can tell the world "Hello", "Goodbye", "Farewell" or "Screw you". We can also optimize it by making the options conditional, so that the user cannot select the text that is already selected for the node; for this, we just need to add a "not" checker to every node:

```
1:${&2}, World!
:Hello|1|[not:a,0]{set:a,0}
:Goodbye|1|[not:a,1]{set:a,1}
:Farewell|1|[not:a,2]{set:a,2}
:Screw you|1|[not:a,3]{set:a,3}

2:&
:3,-1[equ:a,0]
:4,-1[equ:a,1]
:5,6[equ:a,2]

3:>Hello
4:>Goodbye
5:>Farewell
6:>Screw you
```

Other actions

Here is a list of all the actions that were omitted from the previous list due to their complexity.

- `call:<logical node>` : Executes the logical node, then return to the current node. Sometimes, you will need to execute what is inside another logical node without actually going to it, as you need to continue execution in the *current* node. This method allows you to do just that. It is also useful if you wish to execute a logical

node in an option, like so : `:An awesome option|123|{call:1000}`, where 123 is a text node or a logical node and 1000 is a logical node.

You can use either a raw logical node ID, or a variable that contains the ID where you want to go.

Aliasing & Auto-ID

> ***This is a new feature that appeared in OpenBST 2.0!*** *Make sure you update OpenBST!*

Tired of remembering every single node number? Want to have something simpler? How about a simple name?

Aliasing allows you to give a name to some nodes. It's as simple as that.

```
1:An example of aliasing
:Go to aliased node|example

2:It works!
::alias=example
:Go back to 1|1
```

You can use the aliased name in any NND. The name may contain any non-accentuated character as well as underscores.

Because this makes the ID unnecessary, you can use the auto-ID mechanism to automatically attribute a unique ID to the node. To do this, simply use a `*` instead of an ID. Let's take the same example but use this auto-id mechanism instead of an actual ID (line in red).

```
1:An example of aliasing
:Go to aliased node|example

*:It works!
::alias=example
:Go back to 1|1
```

Aliases and auto-id mechanism also work on virtual and logical nodes.

Do note that wherever you can put a node number, you can also put an alias.

> ***Be careful when using both aliases and variables in your NNDs!***
> *The NNDs are solve in this order: first, we check if it's a number. If it is, go to the node with this ID. If not, check if a node is aliased with the text given. If there is, go to the node with this alias. If not, check if there are variables with the name given. If there is, and the variable contains text, try to go to the alias that matches the text contained in the variable.*

If the variable contains a number, try to go to the node with the ID contained in the variable. If the variable cannot be resolved, or if it does not exist, give an error.

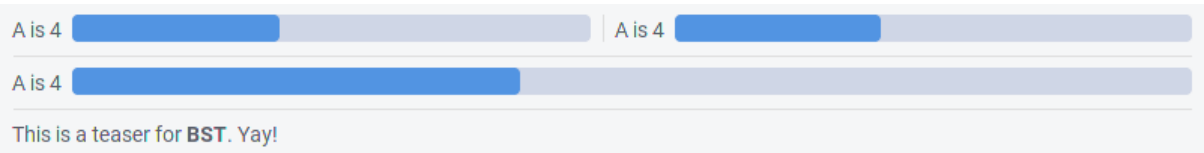
Modules & Miscellaneous documentation

> ***A foreword on modules:***
>
> *Modules are very useful parts of BST that were separated from the rest of the language due to their complexity or the fact that they can be tricky to add in a Player. All the modules described here work best in OpenBST – and while we will try to help other players to keep up to date, we cannot guarantee that these will work on them.*
>
> *All modules use a similar pattern for the tags, functions and checkers they use: they start using an identifier, an underscore, and the rest is pretty normal.*
>
> *Do note that modules tend to use variables to store internal data: as such, it is recommended that you avoid using variables that start with two underscores ("__"). Modules use the following pattern : "__modulename__..." with the rest being whatever the module needs.*
>
> *Players (such as OpenBST) are transparent for the BST file and do not normally modify your variables or tags unless you tell them to do so; they only read them otherwise. Players may also use variables that start with two underscores to keep track of internal data.*

Modules are very useful. They give you ways to go further with your story, and are one of the key elements for games. This part is dedicated to documenting everything that there is to know on these powerful tools.

Module: User Info Bar (or User Interface for BST)

UIB is a set of functions and tag that allow you to create a fully customizable bar that displays various information. In this chapter, we will see how to create the following bar.



Step 1: uib_layout

The first thing needed when using UIB is the `uib_layout`, a tag that contains all the information about the different components used with UIB.

Let's see the `uib_layout` used in the image.

```
uib_layout=tb,vs,tb,ln,hs,ln,tb,ln,hs,ln,t
```

All the components are separated by commas `,` (they can also be separated by `;`). Here is a complete list of all the components that exist:

- `tb` : a shortcut for `t,b` that is much more consistent and accurate visually.
- `vs` : a vertical separator
- `ln` : this indicates that the next component will be on a new line
- `hs` : a horizontal separator. Usually, this would be placed alone on a separate line.
- `t` : a text component. Its value can be formatted using the markup language specified in the top-level tag `markup`, even if you're not using a node.

  (Note that HTML support is extremely basic in this case)
- `b` : a bar component. It has the following properties :
  - `max` : the maximum value of the bar
  - `min` : the minimum value of the bar
- `gu` : a vertical gap that is a very useful replacement for `vs` in situations where a gap would be more appropriate. The letters mean "gap unrelated"

Components can have a value (for example, the value of a text component is the text, and the one of a bar component is a number) and/or properties.

The `uib_layout` tag can also contain much more advanced properties, but we'll have a look at them later. For now, this model works in most situations.

Step 2: Initialize UIB

UIB needs to be initialized before you touch anything else. Usually, you would do this on the first node which would then be a logical node, or on any other logical node. Just remember to initialize UIB before touching anything else or else very bad things will happen.

For this example, we make the first node a logical node. The function that initializes UIB is `uib_init:`, so we only need to add it to our logical node:

```
1:&
uib_init:
```

> *Under the hood, this command reads the UIB tag, interprets it, and translates it into actual visual components.*

Now that UIB is initialized, we need to set all the values and properties we need…

Step 3: Setting properties and values

This is fairly easy to do. Before we jump straight into setting all the values, you have to understand how ID attribution works in UIB.

Basically, every component that can have a value or a property is identified by the component's usual text representation (for a text component, this would be `t`) followed by a number. This number is unique to the component and is determined depending on the position of the component in the tag. For instance, if our layout tag was `t,t,tb` the first text component would have an ID of `t0`, the second one an ID of `t1`, the third one an ID of `t2`, while the bar would have an ID of `b0`. Do note that when you use the component `tb`, you're actually using *two* components: a `t` and a `b`, so you will have to refer to them as if were using `t,b` instead of `tb`.

> *The usage of tb instead of t,b adds various visual improvements that are very important in multiple situations. Just use this notation instead of the other, really.*

Now that this is done, we can continue to setting our values. Here is an example for the first `tb:`

```
uib_set:t0,>5
uib_setprop:b0,max,10
uib_set:b0,a
```

This may look complicated, but it really isn't. The `uib_set` function is used whenever you need to set something's value. The syntax is:

```
uib_set:<component id>,Value (that can contain commas)
```

The value depends on what you want to set:

- For a text component, the value can be text (`Hello World!`), a virtual node (`>nodeid`, here it is `>5`) or a logical node (`&nodeid`)
- For a bar component, this can be a number (`1` `2` `38` `920`) or a variable (`variablename`, here we use the variable `a`)

When you use a variable, a logical node or a virtual node, the value will be automatically updated whenever needed. It is not recommended to use a logical node that changes variables though, as it may be called at random times and result in strange behavior.

Here, we set the value of `t0` to the virtual node `5`, and the value of `b0` to the variable `a`.

Setting properties is done through the following syntax:

```
uib_setprop:<component id>,<property name>,<value>
```

The value here can be text or numbers, but it *cannot* be a variable or a node of any kind.

We repeat this pattern for all the other components and create a virtual node 5. Our BST file now looks like this:

```
uib_layout=tb,vs,tb,ln,hs,ln,tb,ln,hs,ln,t
markup=markdown

1:&
uib_init:

uib_set:t0,>5
uib_setprop:b0,max,10
uib_set:b0,a

uib_set:t1,>5
uib_setprop:b1,max,10
uib_set:b1,a

uib_set:t2,>5
uib_setprop:b2,max,10
uib_set:b2,a

uib_set:t3,This is a teaser for **BST**. Yay!

# Warning : this file will crash, as the logical node doesn't lead to
anywhere!

5:>A is ${a}
```

The `markup` tag was added so that the text `t3` can make use of it.

Step 4: Making the bar visible

By default, UIB is invisible. We just have to make it visible by using:

```
uib_setvisible:true
```

We can of course replace true by false if we want to hide UIB.

Now, we just have to add a node that allows us to control the variable `a`, and we're done! Here is the complete example:

```
uib_layout=tb,vs,tb,ln,hs,ln,tb,ln,hs,ln,t
markup=markdown

1:&
uib_init:
```

```
uib_set:t0,>5
uib_setprop:b0,max,10
uib_set:b0,a

uib_set:t1,>5
uib_setprop:b1,max,10
uib_set:b1,a

uib_set:t2,>5
uib_setprop:b2,max,10
uib_set:b2,a

uib_set:t3,This is a teaser for **BST**. Yay!

uib_setvisible:true
:2

2:Testtt ${a}
:a++|2|[less:a,10]{incr:a}
:a--|2|[greater:a,0]{decr:a}

5:>A is ${a}
```

And that's it! What we saw is just a very basic layout.

Step 5: Using advanced features

UIB supports a grid mode. Each component will be in a separate cell (except if you use the specific advanced options), and you can specify special properties for each column. Here is an example:

```
uib_grid=[][grow][][][grow]
uib_layout=tb,gu,tb,nl,tb,gu,tb
```

As you can see, each column is represented by square brackets []. If you wish to add properties to a column, you can put them inside the brackets (multiple properties can be separated by commas.

A list of all the properties is available at the following link:
http://www.miglayout.com/cheatsheet.html

Look at the section that describes column properties: everything described there can be used in UIB.

Advanced mode is another powerful tool where you can fully control what is displayed using MigLayout component properties. Here is an example :

```
uib_advanced=true
uib_layout=t:alignx right, growx;t:alignx left, growx
```

This is a layout with a text component aligned to the right and that fills up maximum space, and another text component aligned to the left (which is the default anyway, so specifying `alignx left` was not compulsory) and also fills up maximum space. The fact that we specified two `growx` will make the two text components have approximately the same size.

You can use all the component properties from the cheat-sheet linked above. You can also use both Advanced Mode and Grid Mode. However, this tutorial does not aim to give a complete description of it all, as it would be way too much work, due to the almost infinite possibilities of UIB.

Module: BRM, BST Resource Manager

BRM is a special kind of module. It does not do much by itself, but is very useful when used with other modules – and is even required by some.

When you will add customization to your BST file, you will soon wonder how you will be able to import images, sounds and more to your file.

The way to use images and sounds is using BRM. The BST Resource Manager is very easy to use.

Locate where your BST file is. In the same folder as the BST file, create a folder named `resources`. Inside this folder, you will have to create one folder for every module you wish to use that requires BRM. Let's say we have a module called `abcd`, we would have a folder called `abcd` inside the folder resources, and inside the folder `abcd` would be all our files to use with the module `abcd`.

The resources are loaded automatically.

> *The automatic loading was only introduced in OpenBST 2.0, if you need to support older versions, you have to load the files manually by using:*
>
> `brm_load:`

To access a file in modules, you need to use the file's name without the extension. For instance, `incrediblepicture.png` would be called `incrediblepicture` inside the BST file. The name without the extension is often referred to as "file name". If, for some reason, something does not work inside your BST file, make sure you're not adding the extensions by mistake!

Modules that require the use of BRM will have a small banner like the following:

> *This module requires BRM. Place your files in the folder "resources/example"*

Module: IMG, IMaGe tools

> *This module requires BRM. Place your files in the folder "resources/img"*
> *Note : the file name "none" will not be detected*

IMG is a fairly simple module that allows you to add a background image to your nodes. The background image will be adapted to the text area automatically, and whitened so that the text can still be read. A button also becomes available, allowing the user to view a non-modified version of your image.

Using IMG is as simple as specifying a tag in a node. Here is an example where we have an image called "example" in the appropriate folder

```
1:Hello world!
::img_background=example
:Go to 2|2

2:Goodbye world!
```

You will notice that, even if we changed node, the background didn't change. The tag *changes* the image, so the change stays across nodes. This is to avoid you having to add the tag to every single node, as you will then only need to put it when you change backgrounds.

You can change this behavior by specifying the following tag:

```
img_manual=true
```

This will turn "manual mode" on, which is just a fancy way of saying the image information is not kept across multiple nodes.

If you use the normal (non-manual) mode, you can go back to the normal background using

```
::img_background=none
```

> *Do note that you cannot change a node's background with a function – this is by design.*

Module: SSB, Sound System for BST

> *This module requires BRM. Place your files in the folder "resources/ssb"*

SSB, or Sound System for BST, is a very simple way to add sounds and background music to your BST file.

> *SSB supports .wav and .mp3 (encoded with PCM) files on all platforms. Other files may or may not work, but this is not guaranteed.*

SSB provides three functions that are pretty self explanatory:

```
ssb_play:soundeffect
ssb_ambient:backgroundmusic
ssb_stop:
```

The first one plays a sound effect once (a one-shot sound). The second one sets the background music, replacing the previous one (if any). The third one stops the background music.

> ***Why not use tags for the background music like in IMG?***
> *Because of the dynamic nature of music and the fact it can be stopped, SSB requires music to be set using the ambient function. This is by design.*

Module: BDF, BST Description File

> *This module requires BRM. Place your files in the folder "resources/bdf"*

The BDF module is here to make storing data easier. How?

Let's say you want to make a fighting game in BST. How would you store your variables? How would you store all the information on the weapons, the characters? Make a giant list at the beginning of your BST file? Something else? Whichever solution you decide to take has high chances of being downright painful.

BDF aims to give a simple way to store and dynamically load data. While this documentation will be very short and vague, you'll know when you'll need it. It is especially useful with video games, where you need to store NPCs, weapons, and all these kind of things.

BDF uses a stripped down syntax inspired from the one from BST. A BDF file is basically a file that can only contain comments and first level tags. Here is an example:

```
name=John
surname=Smith
age=24
```

Let's say we put this example in a file called `john.bdf`.

> *Use .df instead of .bdf if your file is detected as a font by your system. (Seems to happen on some Linux systems for some reason)*

We can now import all the values using the action `bdf_apply`. Here are all the existing syntaxes :

```
bdf_apply:<name>,<prefix>
bdf_apply:<name>
bdf_apply:!<variable>,<prefix>
bdf_apply:!<variable>
```

The name corresponds to the name of the BDF file, without the extension. Here, the name would be john. If the name of the file you want to load is stored in a variable (which can happen in a lot of cases, especially when dynamically loading BDF files), you have to put a `!` before the name of the variable.

> *This exclamation mark syntax is only applicable to bdf_apply – it may show up as a regular string rather than a variable on some syntax coloration tools, but it will be processed as one.*

This will import all the values from our BDF file into variables.

```
bdf_apply:john
# We would then have the following variables
# name ==> John
# surname ==> Smith
# age ==> 24
```

In many occasions, you'll have multiple values with the same names across multiple files. When we import all the values, we can require that a prefix needs to be prepended to the variable name.

```
bdf_apply:john,guy_
# The variables will then be :
# guy_name ==> John
# guy_surname ==> Smith
# guy_age ==> 24
```

And that's all you need to know about BDF!

Module: JSE, JavaScript Engine

The JSE module is rather simple, and allows you to interact with a JavaScript engine for more advanced mathematics

To interact with the JavaScript engine, you need to use the `jse_eval` action, which has the following syntax.

```
jse_eval:<variable where result is to be stored>,<JavaScript>
```

The variable will store anything that is the result of the JavaScript expression, translating it to simple characters first if it is not an integer. As numbers that are not integers are not supported, they are first truncated to become integers.

JSE internally creates a JavaScript engine, applying to it all the current values from BST. The creation process happens if the engine is not initialized when calling the `jse_eval` function. Each call of the `jse_eval` first updates the engine, making sure all the variables are up to date and coherent. Here is a complete example:

```
1:&
set:a,10
set:b,5
set:c,3
jse_eval:d,c+a*b
# At this point, d = 53
```

JSE can also be used as a checker. In this case, the syntax is simpler:

```
jse_eval:<javascript>
```

Do note that the concept of "booleans" does not exist in BST. If you wish to turn the string "true" or 1 into actual true and the string "false" or 0 into an actual false, use JavaScript's Boolean(variable):

```
1:&
set:a,1
set:b,0
:2,3[jse_eval:Boolean(a) && !Boolean(b)]

2:Hello! (the checker returns true, which means it leads to this
node)
3:This should never be reached
```

> ***Wanna know more about JavaScript operands? Mozilla has a nice list of them here:*** *https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_Operators*

> ***WARNING: CHANGES OF VARIABLES INSIDE THE ENGINE ARE NOT REFLECTED IN BST***
> *For instance, doing a++ will only increment "a"* ***inside the engine****, and the variable will* ***not*** *be incremented in BST!*

> *Previously, JSE had a "manual import" mechanism that is ignored as of OpenBST 2.0, meaning that whenever you call jse_eval the entire engine will be rebuilt and the variables will be automatically imported. It has been decided to remove it as it was just useless,*

modern JavaScript engines being extremely fast to import variables – though if there is popular demand we are not against putting it back in.

Module: XBF, Cross BST File

This module requires BRM. Place your files in the folder "resources/xbf"

XBF simply allows you to use multiple BST files along with your main file. All your files share the same variable registry (so the variables are the same with all the BST files). This is useful to have a more organized system for huge projects. They all share the same modules and BRM resources as well. Simply place your additional BST files inside the folder `resources/xbf`.

You can use XBF in two ways: you can use the XBF Next Node Definer to go to a node in another file, or use the `xbf_call` action which works exactly like the regular `call` action.

To use XBF's Next Node Definer, simply use this syntax:

```
:An Example of an option|xbf:anotherfile,1
```

This simply points to the node with ID 1, from the XBF file `anotherfile.bst` in the resources. This can be used in *any* of the file, may it be the main one or the additional one. However, this can only point to additional files. To point to a node inside the main file, use the following syntax, simply omitting the mention of an additional file.

```
:Option in additional file that points to the main file|xbf:1
```

You can use this with aliases or variables instead of a raw node ID, like any other NND.

The xbf_call action has the same syntax:

```
1:&
brm_load:
xbf_call:additional,1
:2

2:It works, with a being ${a}

Inside the resources/xbf/additional.bst file
1:&
set:a,10
```

This will result in `It works, with a being 10` being shown.

The `xbf_call` action has the same arguments as the XBF NND, with the same thing for calling a logical node from the main BST file.

Module: HTB, HTML Bridge

> *Parts of this module require BRM. Place your files in the folder "resources/htb"*

OpenBST 2.0 introduced full HTML5 support, and with it came the necessity to ease the interaction between the web engine and BST. HTB does just that. It has two NNDs and two actions for BST.

It goes without saying that you have to specify the HTML markup tag with the following:

```
markup=html
```

As with any top-level tag, it has to be specified before any node.

By default, **hrefs and JavaScript are disabled in OpenBST for security reasons.** The following NNDs allow you to enable them by prompting the user to enable them.

> ***Why use NNDs instead of actions?*** *For a simple reason : you can redirect the user to a specific node if they wish to leave JS or hrefs disabled. You could then provide a dead end to explain you need to have JS or hrefs enabled.*

You should always use these NNDs in logical nodes, as they will crash when used on text nodes. This is by design.

```
1:&
# You can of course use all the regular logical node stuff
# The following line requests JavaScript authorization
:htb_requestjs:2
# And this one requests href autorization
:htb_requesthref:3
# We just use a normal NND there
:4

2:I need JavaScript to function properly :(
3:I need HTML Anchors to function properly :(

4:It <a href="alert('It works even better!')">works!</a>
```

Launching this example will give you a nice link that launches an alert box. Cool, right? If you do not wish to use this redirection mechanism, you can just use the following:

```
:htb_requestjs:
:htb_requesthref:
```

And of course you do not have to enable both, only one is enough.

HTB also provides two methods for importing data, either as a raw string or as base64 encoded data.

Simply place any resource you wish in `resources/htb` and use one of the following:

```
htb_import:<resource>,<variable>
```

This will import the given resource (remember, you have to give the resource name without the file extension!) as a plain string in the variable.

```
htb_base64:<resource>,<variable>
```

And this one will do the same, except that the resource will be encoded with base64.

Module: XSF, eXtended Script Files

> *This module requires BRM. Place your files in the folder "resources/xsf"*

XSF just allows you to use Nashorn JavaScript files that can be used in Java. This makes scripting *way* easier, and allows you to use Java classes from inside JavaScript.

> *Nashorn, the library used inside OpenBST, has great interoperability (i.e compatibility) with Java classes. If you know a thing or two in Java, you are totally able to use them in JavaScript files for XSF! Have a look at the section 3 of this tutorial from Niko Köbler for more details on what Nashorn allows you to do.*

Put all your .js scripts in the resources XSF folder, and simply use this action to launch it:

```
xsf_exec:name,function,putTheResultIn
```

The arguments are:

- `name` : The name of the JavaScript file (without the .js at the end)
- `function` : What function to execute. The function should be not have any parameter.
- `putTheResultIn` : Optional, the value returned by the function will be placed in the variable specified here. Does nothing to the variable if the function returns null or has no return statement at all.

This system would be useless without a way to interact with BST: luckily for you, there is a way to do that with a bridge object.

Every time `xsf_exec` is executed, a new JavaScript engine is created and to it are applied a few things:

- A variable called `bst` that is bridge between the story and your script. More details on that later
- All the variables from the story.

> ***Changing the values of the variables from the story from within the script will not actually change them outside of the script.*** *For example, if you have set a variable named bananaCount in BST, and assign a different value to it in a script (e.g. bananaCount = 3), this will only change the value inside the engine, and the changes won't be registered outside of it, just like for the JSE module. You have to explicitly export any variable you wish to change (with the exception of the return value if the putTheResultIn parameter is specified)*

Here are all the functions available inside the `bst` object:

- `bst.get(varName)` : Retrieve a variable from BST. Only usable in very specific cases as all variables are automatically imported before running your function anyway.
- `bst.exec(action, parameters)` : Execute a BST action. For example, `add:a,b,c` would become `bst.exec("add", "a,b,c")`. This is very useful for executing methods on other modules
- `bst.export(varName, value)` : Export a value from the JavaScript engine to the BST engine. This is more efficient than doing `bst.exec("set", "varName," + value)` as it performs necessary conversions.

> *If you use an action that changes a value (such as "input") with bst.exec(), make sure you use bst.get() to retrieve the new variable value as they are not automatically synchronized!*

> *Please do not use networking features in scripts. OpenBST is designed to be ran offline, and BST files are expected to be ran offline. While you* can *do it, it doesn't mean that you* should *do it.*

> *Because these JavaScript thingies support calling Java code, you can actually dynamically create nodes using the OpenBST API. This API is unfortunately undocumented at the moment – while partial Javadocs can be extracted from source, no official well-made reference has been established, and this is not high on our priority list at the moment.*

OpenBST Guide for story creators

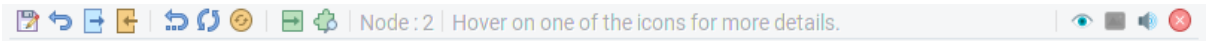
> *The screenshots in this section are outdated. While they may still be technically correct, the visuals have changed.*

While OpenBST is the default player designed with ease of use in mind, it is also a very powerful tool for story creators. Discover how to make the most of it in this section!

The tool-bar

The tool-bar is a bar with several buttons that you can find on top of the panel that holds your text node's text.

> ***LEGAL NOTICE:*** *ALL THE ICONS USED IN OPENBST AND THAT MAY BE ON THIS DOCUMENT WERE CREATED BY ICONS8 AND DISTRIBUTED UNDER THE CC-BY-ND LICENSE. MORE INFORMATION AVAILABLE ON THEIR WEBSITE, icons8.com*



We'll go through each section separately. All the tools that are on the left can be disabled using the `supertools` tag. Please refer to the Tags list for more information on this tag.

Save state options

The save state options are useful for everyone, not only story creators. There are four buttons in this section that allow you to manage save states. Do note these are quite limited in OpenBST, as you can only have one save state in memory.

- **Create a save state in memory**: Pressing this button will create a save state. The current save state is replaced by this one.
- **Restore the latest save state:** Restore the current save state in memory.
- **Export the latest save state** : Save the current save state in memory to a file. This file will be JSON formatted. It is not guaranteed that it will work with other players than OpenBST
- **Import a save state:** Import a save state previously saved using the **Export the latest save state** button.

Do note that Restore and Export buttons are only available when there is a save state in memory.

Reset and reload options

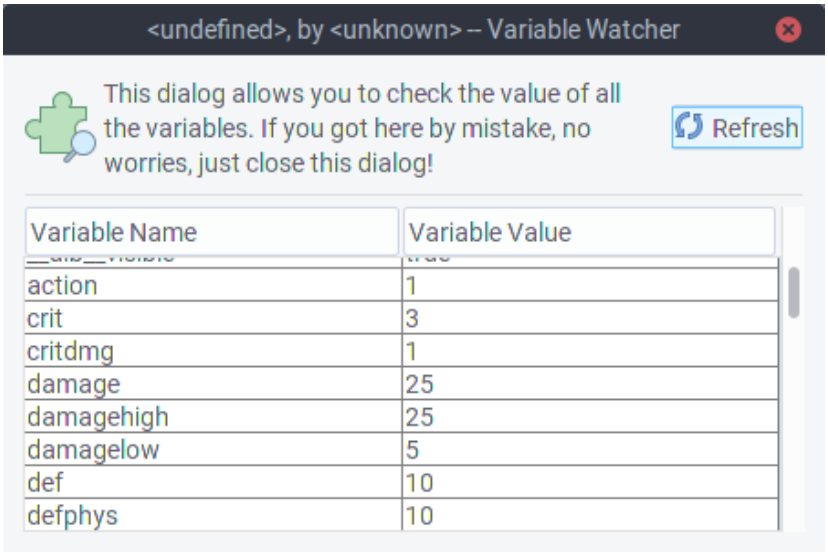
These options are useful for everyone, but the reload ones are very useful when editing your story.

- **Reset:** Restart the story from the beginning, cleaning all the variables saved and restarting from the initial node.
- **Reload (soft):** Reload the BST file, and restore the progress where you were. Do know that this can fail in many occasions, and is generally not recommended if you are using modules, as their behavior when reloading is generally poor. It is however fully functional when creating basic dynamic stories and static stories.
- **Reload (hard):** Reload the BST file, restarting fresh.

Jump to Node, Variable Watcher, Node ID

These three tools, which can be considered to be equivalent to cheating on many occasions, are extremely powerful for debugging.

- **Jump to Node:** Jump to a specific node. The node can be either a virtual node or a text node. A prompt will ask you to which node you wish to go.
- **Variable Watcher:** This very useful tool allows you to see what are the current variable values. You can still interact with OpenBST while the dialog is open. If you wish to update the variable information, click on the Refresh button.



- **Node ID:** There is a very simple Node ID indication to that simply tells you which node you currently are on.

Options for the user

These options are purely made for the user, and cannot be disabled using the `supertools` tag.

- **Background visible:** Sets whether the background is visible or invisible. The background comes from the IMG module.
- **View background:** Show the background in a separate pop-up in its original form. The pop-up can be resized, and the background is resized with it.
- **Mute:** Mutes (or unmutes) sound coming from this tab. Said sounds come from the SSB module.
- **Close tab:** Close the tab.

BSP: Package your BST files

Packaging a BST file allows you to distribute it easily. You may have noticed that your BST file requires a `resources` folder. This makes sharing your story rather hard: you always have to zip it yourself, tell other people they need to unzip, place in a folder…

All of this can get very annoying, but there is a solution for you: packaging! Packaging is a feature that allows you to turn your file (and resources) into a single, simple file that can be opened like any other file.

In a nutshell: `OpenBST menu (the blue bar at the top of the window) > Advanced Tools > Package a BST file`, then fill in the required fields. File to package is the BST file you want to package, while the output is where the BSP file will be stored. Click Start packaging, and you're done! Just wait for a bit, and that's it!

Your BSP file can be opened like any regular BST file!

OpenBST Experimental Features

> ***WARNING:*** *These features may change, disappear, or be broken at any given time. The following section describes experimental, buggy and/or obscure mechanism that are only available in OpenBST.*
>
> *Up to date with OpenBST 2.0*

OpenBST provides a number of experimental features, that are waiting for promotion to actual features.

> ***DO NOT DISTRIBUTE FILES THAT USE EXPERIMENTAL FEATURES.*** *OpenBST will warn you when using anything experimental.*

> *Inside OpenBST source files, experimental code is often marked with a $EXPERIMENTAL comment somewhere around it.*
>
> *Also, the actions provided here are also available inside the bst-java library; in that case, the experimental elements will have an @Experimental annotation.* ***Experimental classes and methods are not API.*** *Full disclosure in the @Experimental annotation javadoc.*

HTBex: HTB EXperimental

> *The HTBex module is part of HTB. Place any resource as you usually would with HTB.*

Use custom CSS files (htbex_css...)

> *These need more in-depth testing before release*

`htbex_cssapply:<resource>[,<resource>]...`

Apply the given CSS resources (inside the HTB folder) to the HTML5 panel. You can apply multiple CSS resources by separating them with commas.

`htbex_cssremove:<resource>[,<resource>]...`

Remove the given CSS resources from the HTML5 panel. You can remove multiple resources by separating them commas. Removing a resource that was not previously applied (or is already removed) has no effect.

`htbex_cssclear:`

Remove all currently applied CSS resources from the HTML5 panel.

Internal IMG backgrounds

The IMG module supports using the internal backgrounds seen in the Welcome screen of OpenBST. To use them, simply put the following tag in one of your nodes:

```
::img_background=$internal1
```

If you wish to see all the available backgrounds, go in the advanced tools > Included BST files > Internal backgrounds