Using natural language for test specification, is that really wise?

An introduction to bbt

Agenda

Part 1 - Introduction to bbt

Part 2 - Partial Parsing

Part 3 - Surviving an ambiguous world

Conclusion

Part 1 - Introduction to bbt

Feel free to install bbt and test what is presented

- with Alire

```
alr install bbt
```

- by other means, including AppImage
  https://github.com/LionelDraghi/bbt#installation

And should you leave now, here is the starting point

- `bbt help tutorial`

Lionel Draghi

- 15 years as Ada software developer
- Retired from software dev in 2007
- Author of Archicheck and smk (https://github.com/LionelDraghi), brilliant apps that no one uses
- but whose merit is being the origin of the creation of bbt

What is bbt?

- `bbt` is a dead-simple tool for *black box* testing your command line apps (⚠️ it's **not** a unit testing tool)
- It targets Apps reading some input and writing some output, like for example `grep` or `gcc`
- `bbt` inputs are markdown files, embedding descriptions of an expected behavior in Gherkin format, **with the steps in natural language**
- From a practical point of view, you really run the doc : `bbt my_spec.md`

Live demo - Getting started

Where does bbt come from?

- Initial black box testing is easy to do within a Makefile
- Become complex and error-prone when adding the documentation generation
- More general discrepancies problem due to overlapping info from readme, code, tests...
- How do you ensure that all examples in the user guide still work as expected?

```
48  test2: ../../obj/archicheck
49          @ #--------------------------------------------------------------------
50          @ ${TR} start "Mixt recursive and non-recursive file identification test"
51          @ ${TR} cmt
52          @ ${TR} cmt "if sources :"
53          @ ${TR} cmt "> ./dira/a.ads  "
54          @ ${TR} cmt "> ./dira/dira1/c-d.ads  "
55          @ ${TR} cmt "> ./dirb/b.ads  "
56          @ ${TR} cmt "> ./dirb/dirb1/c.ads  "
57          @ ${TR} cmt
58          @ ${TR} cmt "then"
59          @ ${TR} cmt
60          @ ${TR} cmt "> archicheck -I dira -Ir dirb --list_files"
61          @ ${TR} cmt
62          @ ${TR} cmt "should put:"
63
64          @ ${TR} cmt '```'
65          @ cat expected_file_list.2 >> testrec.md
66          @ ${TR} cmt '```'
67          @ ${TR} cmt ""
68
69          @ ${CREATE_PKG} A spec   -in dira -with B
70          @ ${CREATE_PKG} B spec   -in dirb -with K
71          @ ${CREATE_PKG} C spec   -in dirb/dirb1 -with A
72          @ ${CREATE_PKG} C.D spec -in dira/dira1 -with B
73
74          @ ../../obj/archicheck -I dira -Ir dirb --list_files | sort > file_list.2
75          @ ${TR} assert true /usr/bin/sdiff "-sbB expected_file_list.2 file_list.2"
76
77          @ ${TR} end
78
```

Design principles and attributes

- **Simple and obvious**
- **The documentation is the *single source of truth***
  - *Never repeat yourself*
  - Tests comes from documentation, not the other way around
  - If it says that the app is run with that command line, then it is!
- **Whatever the documentation**
  Specifications, tests, user guide, slides, readme... all are valid source of truth

Live demo 2 - Let's create a runnable README file

Part 2 - Partial Parsing

A word on Partial parsing

In the NLP field, **partial parsing**, also known as **shallow parsing**, **light parsing** or **chunking**, occupies a position between simple tokenization and full syntactic parsing.

- Deep parsing and understanding can be complex and is not always necessary
- Partial parsing is simpler and faster
- However, it may be ambiguous and lack precision

Partial parsing is based on word spotting

Example: *Eliza* (1966), the famous psychotherapist emulator

- The logic behind may be as simple as

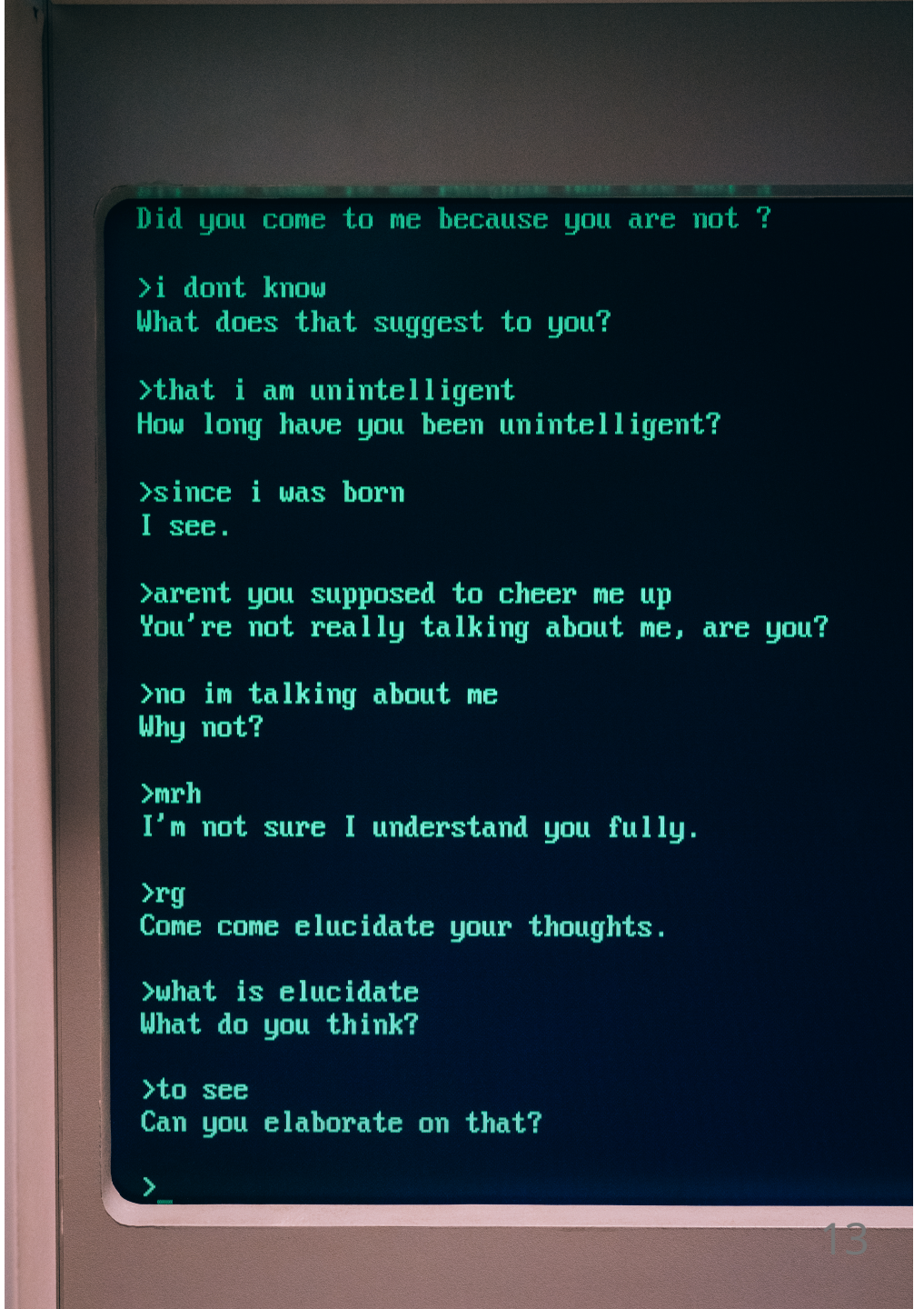
```
if Answer.Contains ("you") then
   Ask ("You're not really talking about me, are you?");

elsif Answer.Start_With ("no") then
   Ask ("Why not?");
...

else
   Ask ("I see.");
```

- But nevertheless very wise

```
elsif Answer.Contains ("rust") then
   Ask ("What's your problem with Ada???");
```



Consider those bbt steps:

```
- Given there is no existing `.config` file
- When I run `my_app --init`
- Then there is a `.config` file
- And the file `.config` contains `autosave = true`
```

It may seem complex at first glance, but is it?

The same steps :

```
- Given   there                  is no      existing `.config` file
- When    I                      run        `my_app --init`
- Then    there                  is         a `.config` file
- And     the file `.config`     contains   `autosave = true`
```

- All sentences have the same simple structure, in the same order
  preposition + subject phrase + verb phrase + object phrase
- The Markdown syntax is helping : parameters are between backticks, no possible confusion with keywords
- A small vocabulary: about 20 keywords to build all the possible sentences
- **Ultimately, the complexity seems manageable**

bbt implementation (1/4) : Tokenization

Consider the step :

```
- Given there is no existing `.config` file
```

Tokenization

Given	there	is	no	existing	`.config`	file
keyword	-	keyword	keyword	-	parameter	keyword

bbt implementation (2/4) : Chunking

Before the verb, it's the subject chunk, after the verb it's the object chunk.

And if it's a Markdown code span (or a code block), it's a parameter.

Preposition	**Subject phrase**	***Subject parameter***	**Verb phrase**	**object phrase**	***Object Parameter***
Given			*Is_No*	*File_Name*	*.config*

bbt implementation (3/4) : Grammar

The Grammar is a table of actions indexed by (preposition, Subject, Verb, Object...)

For example here :

```
Grammar (Preposition => Given, Verb => Is_No, Obj_Attrib => File, ...) := Setup_No_File;
```

- Note : you can display the grammar with `bbt lg` (or `bbt list_grammar`)

bbt implementation (4/4) : Actions

- The action and the parameters are stored in a Tree that represent a bbt document (that is a list of features containing a list of scenarios, containing etc.)
- When all documents are parsed, a runner walk through and run actions in sequence. Could be:

```
Setup_No_File (Subject_Param => "", Object_Param => ".config");
```

- Want to know what will be done?
  There is a dry run mode: `bbt ex` (or `bbt explain`)

- Current State:
  - Token: less than 40
  - Grammar definition: about 50 lines
  - Total lines of code for lexing and parsing: 640

- WIP, The code could be simpler / smaller / easier to read, and more robust...

- But the goal is already achieved, as `bbt` can process sentences like

```
- Then the resulting `log.txt` file does not contain any `Error:`
```

Part 3 - Surviving an ambiguous world

- Example of a detected Ambiguity

```
- given there is no `config` file in the current directory
```

  - 💣 In the Object chunk, there is both `file` and `directory` keywords...
  - May be detected because both word are in bbt's vocabulary

- Example of an undetected ambiguity, where bbt understand the opposite of what is said!

```
- then the output never contains `Error`
```

  - 💣 `never` is not a keyword, this will indeed check that the output contains `error`
  - Can't be detected because `never` is ignored by bbt

In practice

- **If you use "test first", it's impossible to miss that kind of errors**
- In any case, it is recommended to:
  - Stick to the usual best practices for writing specifications: be clear and **concise**, avoid double negatives, don't hesitate to repeat yourself 😄 ...
  - Put complex comments on separate lines (and keep steps simple)
  - Uses `bbt explain` in case of doubt
- A feature to rewrite the steps in a standardized way (`bbt rewrite scenario.md`) would be easy to implement, but doesn't seems useful at this stage

Special thanks

- to AdaCore
  - ❤️ For awarding bbt **Ada Crate of the year 2024** (https://blog.adacore.com/)

- to the early adopters and contributors
  - ❤️ Paul (https://github.com/pyjarrett)
  - ❤️ Manuel (https://github.com/mgrojo/coap_spark)
  - ❤️ Simon (https://github.com/simonjwright/ada_caser)

Conclusion

Ideas and features are welcome https://github.com/LionelDraghi/bbt/discussions

Slides made with Marp available here
https://github.com/LionelDraghi/bbt/blob/main/docs/AEiC_2025_presentation/

Q & A