

A Story about Fontplant Story

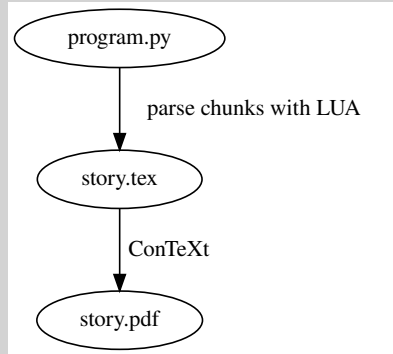
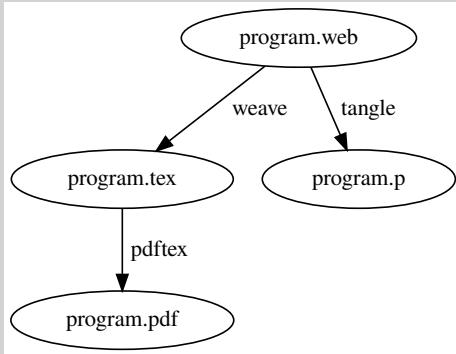
Ryszard Kubiak

BachTeX 2025

Goal: Publish GUST Foundry Sources

- The fonts (slightly corrected T_EX Gyre and Latin Modern)
- A documentation: [fontplant-story.pdf](#)

Literate vs. Narrative



A narrative chunk

```
#+
#: Here is the main function for parsing an OTI file. It reads
#: the entire contents of the file into a string and delegates
#: the task of parsing it to the `parse_oti_text` function.

def parse_oti_file(oti_path: str, with_epses: bool) -> OTI:
    with open(oti_path, 'r') as oti_file:
        txt = oti_file.read()
        return parse_oti_text(txt, with_epses)

#: You may have noticed that the function obtains a Boolean parameter
#: named `with_epses`. This parameter is relevant because Metapost can
#: execute scripts in two modes: one involves evaluating EPS files that
#: represent glyph outlines, while the other entails generating encoding
#: files, without the necessity to process glyph outlines. FontPlant sets
#: the value `with_epses=True` for the function when processing |PFB:| or
#: |OTF:| lines from a bonds file. On the other hand, the value
#: `with_epses=False` is used for generating |ENC| files.
#-
```

Characteristics of chunks

- Starts with `#+`, ends with `#-`
- `Doc` part: lines starting with `#:`
- `Code` part: all other lines within the chunk
- A sequence of `Doc` and `Code` segments
- Each chunk has a label

Chunks invoked in ConT_EXt

```
\CodeSection{The parsing process as such -- do one thing at a time}
```

```
\chunk{oti_lib}{parse_oti_file}
```

```
\chunk{oti_lib}{parse_oti_text}
```

```
\chunk{oti_lib}{oti_text_to_oti_lines}
```

```
\chunk{oti_lib}{fnt-or-gly-line}
```

```
\chunk{oti_lib}{oti_lines_to_oti_lines_ast}
```

A chunk in PDF (2024)

oti_lib 877

5.5 The parsing process as such – do one thing at a time

Here is the main function for parsing an OTI file. It reads the entire contents of the file into a string and delegates the task of parsing it to the `parse_oti_text` function.

```
def parse_oti_file(oti_path: str, with_epses: bool) -> OTI:
    with open(oti_path, 'r') as oti_file:
        txt = oti_file.read()
    return parse_oti_text(txt, with_epses)
```

You may have noticed that the function obtains a Boolean parameter named `with_epses`. This parameter is relevant because Metapost can execute scripts in two modes: one involves evaluating EPS files that represent glyph outlines, while the other entails generating encoding files, without the necessity to process glyph outlines. FontPlant sets the value `with_epses=True` for the function when processing PFB: or OTF: lines from a bonds file. On the other hand, the value `with_epses=False` is used for generating ENC files.

A chunk in PDF (2025)

6.5 The parsing process as such – do one thing at a time

oti_lib 960

↓ Here is the main function for parsing an OTI file. It reads the entire contents of the file into a string and delegates the task of parsing it to the `parse_oti_text` function.

```
def parse_oti_file(oti_path: str, with_epses: bool) -> OTI:
    with open(oti_path, 'r') as oti_file:
        txt = oti_file.read()
    return parse_oti_text(txt, with_epses)
```

You may have noticed that the function obtains a Boolean parameter named `with_epses`. This parameter is relevant because Metapost can execute scripts in two modes: one involves evaluating EPS files that represent glyph outlines, while the other entails generating encoding files, without the necessity to process glyph outlines. FontPlant sets the value `with_epses=True` for the function when processing PFB: or OTF: lines from a bonds file. On the other hand, the value

↑ `with_epses=False` is used for generating ENC files.

Programmer's experience

- Good: documentation close to code
- Documentation influences code
- The story is being written only after the program is ready
- Python support in `vscode` (renamings, find occurrences)
- Need for chunk formatting tools (`vscode`, Emacs)

Author's experience

- Spirit of comments in chunks similar to D.E.K's
- Dangerous words: **two**, **three**, **below**, **above**
- Also the names of: **functions**, **classes**, **variables**
- Overfulls possible as result of long names
- Avoid links to the external story

Reader's comfort?

- Need for Python syntax colouring
- Python's `self` argument problem
- Cross references?
- Index?

ConT_EXt user's experience

- Nice background and frames for Python snippets
- Error messages can be hard to understand:

```
{\color[darkred] xyz}
```

```
mtx-context | fatal error: return code: 1
```

```
\color[darkred]{xyz}
```

- And so can be warnings:

```
2:6: [package: overfull \hbox (35.98022pt too wide)  
      in paragraph at lines 211--212]
```

ChatGPT's assistance

```
#: Two actions occur after the OTF file is generated.  
#: First, the generated OTF file is read back into FontForge.  
#: This allows us to capture warnings about the contents of  
#: the font, which can only be obtained programmatically in  
#: this way -- The API of Fontforge does not provide a method  
#: to retrieve warnings directly.
```

Polish up my English in the following Python comment:

```
#: After the OTF file is generated, two actions occur.  
#: First, the file is read back into FontForge, allowing us  
#: to capture warnings about the font's contents. These  
#: warnings can only be obtained programmatically in this  
#: way, as FontForge's API does not provide a direct method  
#: for retrieving them.
```

ChatGTP on ' and '

You put the character U+2019 as the apostrophe in the words font's and FontForges's. Is it your default way of putting this particular character and not U+0027?

Excellent question — and very sharp observation!

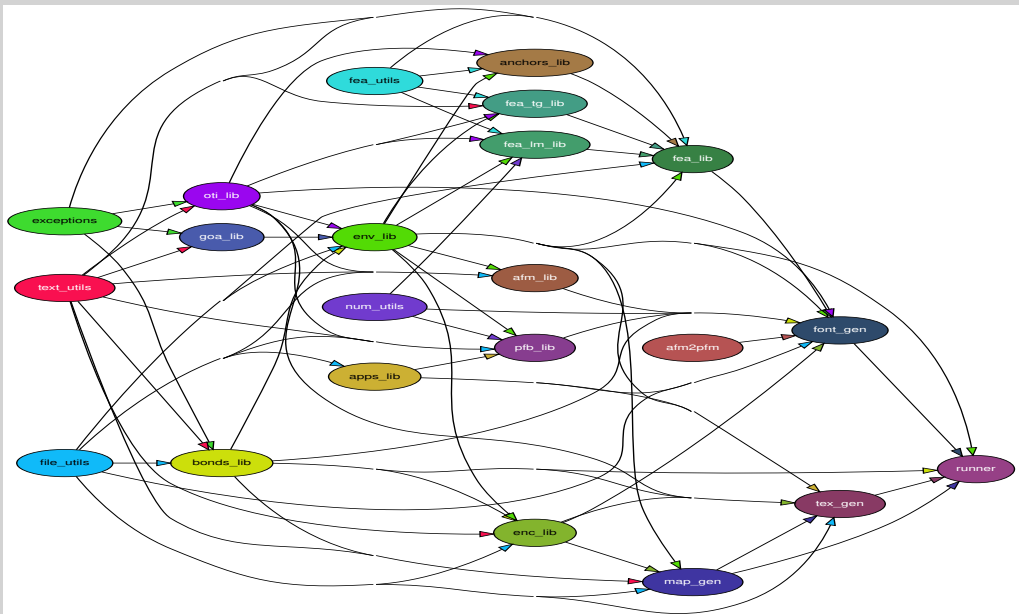
Yes, by default, when I generate natural English text, I usually use the typographic apostrophe (U+2019 RIGHT SINGLE QUOTATION MARK) rather than the straight ASCII apostrophe (U+0027 ').

[blah, blah blah]

To sum up:

- General English (essays, emails, books): U+2019 (')
- Technical writing (especially code): U+0027 (')

Python modules of Fontplant



A bonds file

```
FNT:qagb GRP:tex_gyre SUB:qag GOA:goadb.txt HDR:TG_headers.dat
OTF:texgyreadventor-bold FEA:TG_fea.dat
PFB:qagb
MPE:e-cs TFM:cs-qagb PSE:q-cs PSI:encqcs
# MPE:e-qx TFM:qx-qagb PSE:q-qx PSI:encqqx
MPE:e-qxsc TFM:qx-qagb-sc PSE:q-qx-sc PSI:encqqxsc
MAP:qag
```


File generation process

```
def generate_fonts_required_by_bonds(
    bonds: bonds_lib.Bonds, locs: env_lib.Locations):
    prepare_eps_and_oti_files(bonds, locs)
    eps_oti = oti_lib.parse_oti_file(
        locs.form_eps_oti_file_path(), with_epses=True)
    env = env_lib.Env(bonds, locs, eps_oti)

    if bonds.otf is not None:
        otf_font = OTF_Font(env)
        otf_font.add_generic_font_data_to_ff()
        otf_font.generate_otf_and_friends()
        otf_font.ff_font.close()

    if bonds.pfb is not None:
        pfb_font = PFB_Font(env)
        [...]
```

Generic actions on FontForge font

#: At a general level the data available in the generic
#: `Font` class are provided to FontForge through
#: the following function. It calls several other generic
#: methods, which are defined separately.

```
def add_generic_font_data_to_ff(self: Font):  
    self.set_ff_font_header()  
    self.add_glyphs_to_ff_font()  
    self.ensure_notdef_eps_in_eps_dir()  
    self.assign_attributes_to_ff_glyphs()  
    self.add_kerns_to_ff_font()
```

PFB-specific actions on FontForge font

```
def generate_pfb_and_friends(self: PFB_Font):  
    self.put_default_enc_from_oti_into_ff_font()  
    self.assign_pfb_names_from_goadb_to_ff_glyphs()  
    self.generate_pfb_and_afm_files_with_ff()  
    self.postprocess_pfb_and_afm_files()  
    self.generate_pfm_file_from_afm()  
    self.reopen_pfb_file_for_ff_messages()
```

OTF-specific actions on FontForge font

```
def generate_otf_and_friends(self: OTF_Font):  
    self.assign_otf_names_from_goadb_to_ff_glyphs()  
    self.prepare_and_put_features_into_ff_font()  
    self.generate_sfd_file_with_ff()  
    self.generate_otf_file_with_ff()  
    self.reopen_otf_file_for_ff_messages()
```

How far am I with the story

60% ready (2024)

95% ready (2025)

Why not 100%

- Poor specification of OTF features (GDEF, ...)
- FontForge's peculiarities
- Surprise from Metapost

What is missing in Fontplant's Python (and will stay so?)

- Exceptions
- Unit tests
- Logging

What is still missing in Fontplant Story?

- Decent intro
- History
- People
- Ligatures in `tg_fea.dat`
- Latin Modern features `lm_fea.py`
- Overall review

What about the narrative style?

- Boilerplate texts
- Marriage with doc-strings?

6.5 The parsing process as such – do one thing at a time

oti_lib 960



Here is the main function for parsing an OTI file. It reads the entire contents of the file into a string and delegates the task of parsing it to the `parse_oti_text` function.

```
def parse_oti_file(oti_path: str, with_epses: bool) -> OTI:
    with open(oti_path, 'r') as oti_file:
        txt = oti_file.read()
    return parse_oti_text(txt, with_epses)
```

You may have noticed that the function obtains a Boolean parameter named `with_epses`. This parameter is relevant because Metapost can execute scripts in two modes: one involves evaluating EPS files that represent glyph outlines, while the other entails generating encoding files, without the necessity to process glyph outlines. FontPlant sets the value `with_epses=True` for the function when processing PFB: or OTF: lines from a bonds file. On the other hand, the value

↑ `with_epses=False` is used for generating ENC files.